Journal of Parallel and Distributed Computing **57**, 166–187 (1999)
Article ID jpdc.1998.1525, available online at http://www.idealibrary.com on IDEAL®

# Communication-Efficient Sorting Algorithms on Reconfigurable Array of Processors With Slotted Optical Buses[1]

Mounir Hamdi,*,[2] Chunming Qiao,† Yi Pan,‡ and J. Tong*

*Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong;
†Department of Electrical and Computer Engineering, State University of New York at Buffalo, Amherst, New York 14260;
‡Department of Computer Science, University of Dayton, Dayton, Ohio 45469-2160

Received February 21, 1997; accepted December 9, 1998

The reconfigurable array with slotted optical buses (RASOB) has recently received a lot of attention from the research community. In this paper, we first discuss the reconfiguration methods and communication capabilities of the RASOB architecture. Then, we use this architecture for the implementation of efficient sorting algorithms on the 1D RASOB and the 2D RASOB. Our parallel sorting algorithm on the 1D RASOB is based on an efficient divide-and-conquer scheme. It sorts $N$ data items using $N$ processors in $O(k)$ communication cycles where k is the size of the data items to be sorted in bits. We further develop a parallel sorting algorithm on the 2D RASOB based on the sorting algorithm on the 1D RASOB in conjunction with the well known Rotatesort algorithm. Similarly, this algorithm sorts $N$ data items on a 2D RASOB of size $N$ in $O(k)$ communication cycles. These sorting algorithms are much more efficient than state-of-the-art sorting algorithms on reconfigurable arrays of processors with *electronic* buses using the same number of processors. 

*Key Words:* parallel computing; sorting; optical communication; reconfigurable networks.

## 1. INTRODUCTION

Optical interconnects can offer many advantages over their electronic counterparts including high connection density and relaxed bandwidth-distance product. As a result, they will soon be a viable alternative for multiprocessor interconnections [7, 8, 13, 17]. Recently, a large body of research has been devoted to the

[1] This research work was supported in part by the Hong Kong Research Grant Council under the Grant RGC/HKUST 100/92E.

[2] Corresponding author. E-mail: hamdi@cs.ust.hk.



0743-7315/99 $30.00

technologies [6, 22, 40], architectures [9, 10, 29, 30, 34], and algorithm development [11, 27, 28, 31–33, 35, 36] for optically interconnected parallel computer systems. In particular, the *reconfigurable array with slotted optical buses* (RASOB) architecture [34, 35] has attracted the attention of many researchers in the recent past owing to its promise in superior performance. In this paper, we first describe the RASOB architecture. Then, we present the efficient implementation of several sorting algorithms on RASOB.

RASOB possesses many desirable features which include rich connectivities and low hardware and control complexities. It adopts a two-dimensional structure in order to maintain a reasonable optical path length even when scaled to a large number of nodes. Unlike ordinary 2D structures with row and column buses, RASOB can establish an all-optical path between two nodes that are neither at the same row nor at the same column. RASOB is also unique in that it uses only a single $2 \times 2$ switch at each intersection of a row bus and a column bus, resulting in a much lower hardware complexity than other reconfigurable meshes such as those in [10, 30]. More importantly, all the switches in RASOB are set to the *same* state at the *same* time, and hence, switch control is extremely simple. However, RASOB has one constraint: no more than one processor, at the same row, may send messages to the processors at the same column in one communication cycle (to be defined later). Consequently, it makes it challenging, yet interesting, to develop efficient algorithms on RASOB.

Of particular interest to us in this paper is the implementation of parallel sorting algorithms on RASOB. Because of its fundamental importance, sorting is one of the most extensively studied computing problems. Many researchers have developed various parallel algorithms to speed up sorting on different parallel computation models [1, 19]. In particular, fast state-of-the-art sorting algorithms were presented recently for various models of processor arrays with reconfigurable electronic buses [21]. Wang et al. proposed a constant time algorithms using $O(N^3)$ processors [39]. Using the *Columnsort* technique proposed by Leighton [18], Ben-Asher *et al.* proposed an $O(4^t)$ time sorting algorithm using $O(N^{1+2\times(2/3)^t})$ processors for $t \geqslant 2$ [4]; but Jang *et al.* proposed a constant time sorting algorithm using $O(N^2)$ processors [12]. Recently, Nigam and Sahni proposed two simpler constant time sorting algorithms when compared to that of Jang *et al.* using $O(N^2)$ processors [26]. Finally Kao *et al.* proposed a constant time sorting algorithm using $O(N^{5/3})$ processors under the assumption of a very wide data bus between the processors [14].

Although very fast, all these algorithms require the number of processors to be larger than $N$ to achieve that speed. However, when the size of data items to be sorted, $N$, is equal to the number of processors, it was shown that the reconfigurable array of processors with electronic buses cannot sort in better than $O(N)$ time [21, 25].

The sorting algorithms presented in this paper for RASOB use the same number of processors as the number data items to be sorted. Yet they are almost as fast as the state-of-the-art sorting algorithms which employ a much larger number of processors. This is an indication of the superiority of RASOB as compared to array of processors with reconfigurable electronic buses.

The rest of the paper is organized as follows. Section 2 describes the RASOB architecture. In Section 3, we give the detailed design and analysis of our sorting algorithms on a 1D RASOB which is based on an efficient divide-and-conquer technique. Then, we use this sorting algorithm in conjunction with the well known Rotatesort algorithm to implement our second routing algorithm on a 2D RASOB. Finally, we conclude the paper in Section 4.

## 2. ARCHITECTURAL MODEL

Figure 1 illustrates the architecture of a 2D RASOB. As shown in Fig. 1a, there are n folded row buses and n folded column buses interconnecting the processor array. Each processor has a transmitting interface to the upper segment of a row bus, and two receiving interfaces to the lower segment of the row bus and the right segment of a column bus, respectively. Hereafter, we refer to each row or column of a 2D RASOB as a 1D RASOB, and unless specified otherwise, use the term RASOB to denote a 2D RASOB.

An important architectural feature of the RASOB is that a $2 \times 2$ electro-optical switch is placed at the intersection of a row and a column bus, as shown in Fig. 1b. When the switch is set to "straight," a message arriving along a row bus will continue propagating on that row bus; otherwise, the message will be *switched* to the column bus instead. During a specific period, all the switches at a given row are set to straight and messages propagate only on a row bus. As a result, processors at a row communicate with each other at the same row. This type of communications is referred to as "*row communications*" and the period during which row communications is accomplished is referred to as a *row communication cycle*. A processor may also communicate with a processor at a different row, which may or may not be at a different column. This type of communications is referred to as "*column communications*" and is accomplished by *switching* the message from a row bus to the desired column bus during a period called *column communication cycle*. In doing so, the switches are set to "*cross*" for the duration of the message and then changed back to the "*straight*" state. Due to the high switching speed (tens to hundreds of picoseconds) [2], the control overhead incurred when changing from one phase to another is almost negligible.

An interesting feature of RASOB is that all the switches are set to the same state at the same time. This greatly simplifies the switch control, which is often done in slow electronics. Another interesting feature of RASOB is that the same architecture may support both SIMD and MIMD modes of parallel computing. In an SIMD mode, each processor knows whom and when to transmit or to receive from. As a result, desired communications can be accomplished by letting the source send a message at a specific point in time and by letting the destination receive the message at another specific point in time. To support MIMD mode of computation in an RASOB, an all-optical addressing mechanism, termed coincident pulse addressing [6, 20], can be used. However, it makes the architecture of RASOB slightly more complex. We note that as in all optical bus based architectures employing *time division multiplexing* (or slotted) communications, synchronization is a crucial issue in RASOB. The larger fan-out, less power loss, better noise immunity and other

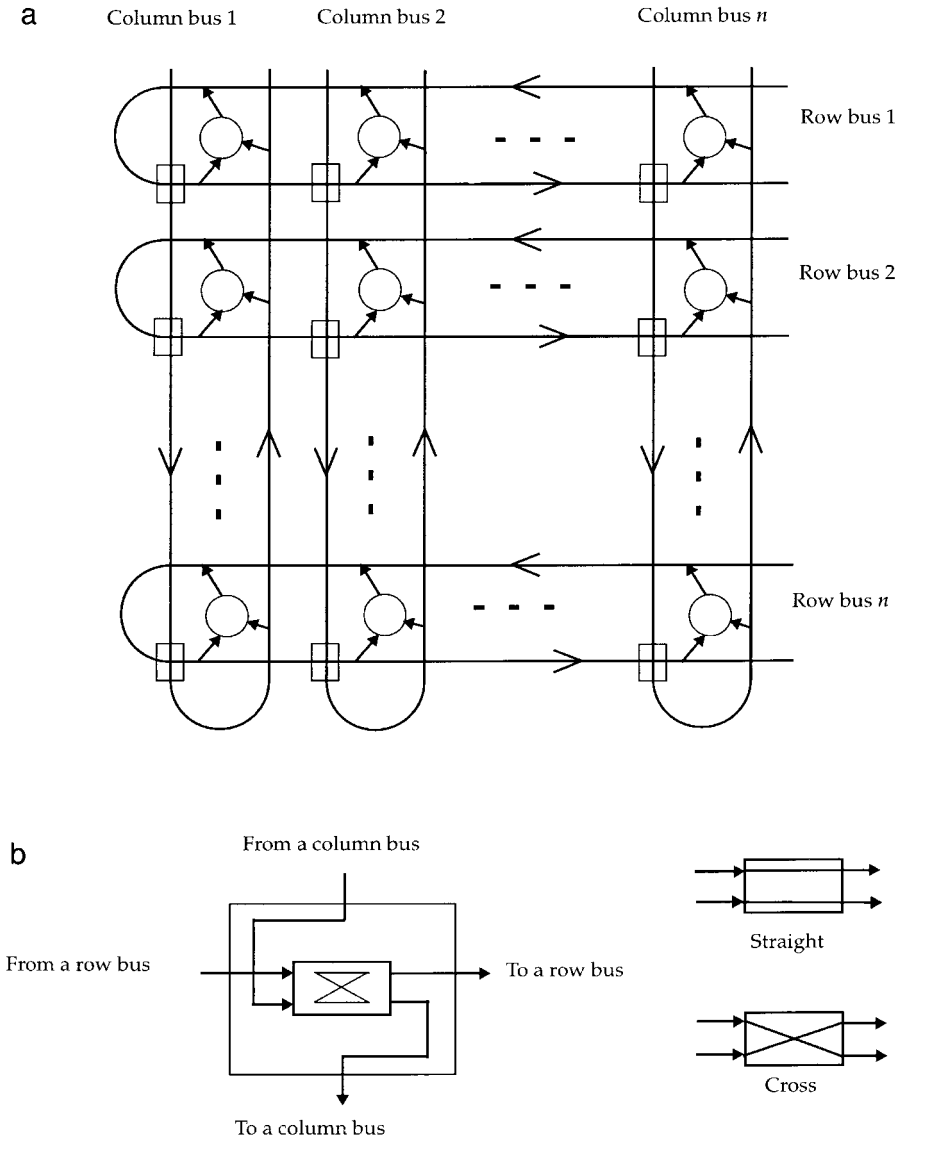


**FIG. 1.** (a) The architecture of RASOB, and (b) A switch interconnecting a row and a column bus.

desirable features of optics makes it possible to distribute a high-frequency global clock to hundreds of receiver modules with little clock skew [15, 38]. In addition, optical data signals encounter small jitters when propagating in single-mode fibers for less than one kilometer even at a speed as high as 2.5 Gbps [3, 37]. These jitters can be tolerated by using narrow (e.g., 10 bits) guard bands at each end of a time slot (up to few hundreds of bits).

In the following subsections, we first illustrate row and column communications, and then discuss both hardware and control complexities of an RASOB as well as its connectivities.

### 2.1. Row Communication

In a row communication cycle, each row bus operates independently from the others so it is sufficient to describe just one row bus (e.g., row bus $r$), as shown in Fig. 2. In the following presentation, we will denote the processors at row $r$ from left to right by $p(r, 1)$, $p(r, 2)$,..., and $p(r, n)$, respectively.

There are two important optical transmission properties, namely, *unidirectional proagation*and *predictable propagation delay* of the optical signals, that make concurrent access of an optical bus possible. More specifically, with an appropriate spatial separation between the neighboring processors, message collision can be avoided even when the processors are transmitting messages concurrently [9, 10, 11, 24, 30, 34]. In the following discussions, we assume that each processor on a row bus is separated in time by $D = bw + \delta$ (seconds) from its neighbors, where $b$ is the maximal length of a packet in bits, $w$ is the optical pulse width (or bit duration) in seconds, and $\delta > 0$ is used as *guard bands* to tolerate synchronization error to a certain degree. This temporal separation can be achieved by separating the two neighboring *transmitter interfaces* on the upper segment as well as the *receiver interfaces* on the lower segment of a row bus with a fiber length $D \times c$, where $c$ is the speed of light in the fiber, as shown in Fig. 2. Without loss of generality, we assume that the length of the folded part, which is the separation of the transmitter and receiver interface of $p(r, 1)$, is also made equivalent to $D$.

We may use the train *loading/unloading* model to describe the operations in a row communication cycle (*Slotted communication*). Let us imagine that at the beginning of a row communication cycle, a *train* (or motorcade) of $n$ *cars* (communications slots) is originated at the rightmost end of the upper segment of the row bus. Each *car* can be regarded as an empty packet slot with a duration of $D$ and is numbered 1 through $n$ from left to right. During a row communication cycle, the switches that connect the row bus with column buses are in the "*straight*" state so that the *train* will run through the lower segment of the row bus. A simple assignment of the *cars* is to let processor $p(r, 1)$ use *car* 1 for sending its packet, let $p(r, 2)$ use *car* 2 for sending its packet and so on.

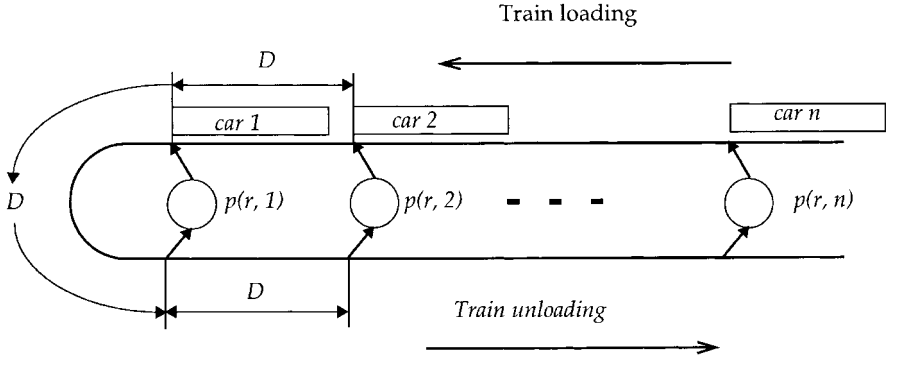


**FIG. 2.** Train loading/unloading on a row bus.

With this assignment of the *cars*, the time when $p(r, i)$ may transmit its packet, relative to the beginning of the row communication cycle, is given by

$$\text{RowSend}[(r, i)] = (i-1)D + (n-i)D = (n-1)D. \tag{1}$$

As a result, all processors will be transmitting simultaneously because the transmitting time does not depend on $i$. In addition, a receiving processor can determine the exact time when the *car* carrying the packet will arrive at its receiver interface. More specifically, if processor $p(r, i)$ is expecting a packet sent by $p(r, j)$, it can calculate the time it should pick up the packet as below,

$$\text{RowRec}[(r, i) \leftarrow (r, j)] = (n-1)D + (i+j-1)D = (n+i+j-2)D. \tag{2}$$

By placing all the processors under a synchronized control and letting each processor send and receive at specific points in time as in Eqs. (1) and (2), the row bus can be reconfigured into a variety of interconnection patterns.

### 2.2. Column Communication

If a processor needs to communicate with another processor at a different row, it has to send a packet in a column communication cycle. The *train loading/unloading* model used previously is also useful in illustrating the principles involved in column communications. More specifically, we let *car* 1 of the *train* make a *turn*, from the lower segment of a row bus, onto column bus $n$, *car* 2 make a turn onto column bus $(n-1)$, and so on. For simplicity, we assume that the switches are placed near the receiver interfaces so that the propagation delay between a switch and its nearby receiver is almost negligible. This also implies that the switches are placed $D$ apart from each other.

Similar to Eq. (2), we can determine the time that *car* $k$ arrives at switch $(n-k+1)$ to be

$$\text{SwitchArvl}[(r, n-k+1) \leftarrow (r, k)] = (2n-1)D. \tag{3}$$

Since the right side of the equation does not contain $k$, every *car* arrives at its turning point at the same time. Therefore, one may set the switches on a row bus to "*cross*" simultaneously and by doing this, the $n$ packets in the *train* are switched onto their respective destination columns, one packet per each column. This arrangement implies that during a column communication cycle, two or more processors at the same row cannot send packets destined to the same column.

If $p(i, j)$ needs to communicate with $p(r, k)$, where $r \neq i$, $p(i, j)$ have to transmit (or load) a packet into $car(n-k+1)$. We can determine the time for $p(i, j)$ to transmit its packet to be

$$\text{ColSend}[(i, j) \rightarrow (r, k)] = (n-k)D + (n-j)D = (2n-j-k)D. \tag{4}$$

By separating the adjacent row buses by $D$, a column bus will look like a row bus that is turned 90° counterclockwise after the packets are switched. More specifically, that every row bus switches a packet onto a column bus at the same time is similar to the case where packets are transmitted in a row bus simultaneously. With as much as $D$ separation between every two row buses, there will be again a *train* of $n$ *cars*, each carrying a packet, formed on the left segment of a column bus. Hence, we can determine the time for $p(r, k)$ to pick up the packet at the its receiver interface on the column bus, which is sent by $p(i, j)$, to be

$$\text{ColRec}[(r, k) \leftarrow (i, j)] = (2n + i + r - 2)D. \tag{5}$$

### 2.3. Connectivity and Complexity

Software reconfiguration can be performed with little control overhead because each of Eqs. (1) to (5) involves simple arithmetic calculations. In addition, the hardware complexity of the proposed architecture is low because each processor uses only one two-state $2 \times 2$ switch and has only one transmitter. Although two receiver interfaces are needed by each processor, a single high-speed electronic receiving circuit may be shared among these two interfaces. As a comparison, most mesh-based reconfigurable architectures would require at least an equal number of switches having four or more states and four I/O interfaces per each processor [21, 25].

Despite the low control and hardware complexities, the RASOB provides strong connectivities due to the following characteristics: First, a direct connection between any two processors can be established. The existence of such a direct "*all-optical*" path is important, because conversions between optical and electronic signals required for buffering and address decoding at intermediate nodes are costly. Second, reconfiguration is flexible as one may *interleave* row and column communication cycles in many ways to provide the communication band-width required by an application. Finally, since only a portion of optical power is tapped off at each receiver interface, *multicasting* can be supported simply by programming multiple receivers to receive at different points in time during the same communication cycle. Noting that the one-to-one and broadcast are special cases of multicasting, we may summarize the communication capabilities of the RASOB below:

- All processors at row $i$ can multicast to the processors at the same row at the same time; and such a row-to-row multicasting can be performed on all $n$ rows simultaneously.
- All processors at column $j$ can multicast to the processors at the same column at the same time; and such a column-to-column multicasting can be performed on all $n$ columns simultaneously.
- $p(i, j)$ can multicast to several processors at any column $k$ and such a processor-to-column multicasting can be performed by all the $n^2$ processors at the same time, with a *restriction* being that two or more processors at the same row $i$ cannot multicast to the same column $k$ at the same time.

Note that while the first two items on the list, by themselves, mean that the RASOB is at least as powerful as any mesh with row and column buses [21], the third item clearly shows that the RASOB has a stronger connectivity than other meshes with row and column buses. In addition, we note that the RASOB has many other capabilities that are not included in this list [9, 30, 34] but may be deducted from its basic features described in the previous two subsections. For example, *simulcasting*, in which a processor sends different messages to different processors simultaneously can be supported by assigning more than one car to a source processor in a given communication cycle.

## 3. ALGORITHM DEVELOPMENT

Although the RASOB has a strong connectivity, it, like many practically scalable architectures, has a weaker connectivity than a completely connected network. The capabilities as well as restrictions of the architecture makes it an interesting yet challenging task to design efficient algorithms for the RASOB. In designing algorithms for the RASOB, one may use the idea proposed in [10, 34] to partition the set of connections required by an application into subsets such that the connections in each subset can be established in a row communication cycle or a column communication cycle. However, such a partition may not result in optimal (i.e., minimal) number of communication cycles and therefore a customized design may be necessary. As an example of how one can take advantage of the capabilities while overcoming the restrictions of the RASOB architecture, we develop efficient sorting algorithms for the 1D RASOB and the 2D RASOB which outperform state-of-the-art sorting algorithms on the various models of arrays of processors with reconfigurable *electronic* buses.

### 3.1. Sorting on a Linear RASOB ($N = P$)

The sorting problem can be defined as the rearrangement of $N$ data items so that they are in ascending or descending order. Given a sequence $SQ = \{s_0, s_1, ..., s_{N-1}\}$ of $N$ data items, a linear ordering "$<$" is defined in $SQ$ and $N$ is an integer. Initially, the data items of $SQ$ are permuted in a random order. The purpose of sorting is to arrange the data items of $SQ$ into a new sequence $SQ' = \{s'_0, s'_1, ..., s'_{N-1}\}$ such that $s'_i < s'_{i+1}$ for $i = 0, 1, ..., N-2$. If two data items $s_i$ and $s_j$ are equal, then $s_i$ is taken to be the larger of the two data items if $i > j$; otherwise $s_j$ is the larger data item.

Our sorting algorithm on a 1D RASOB, which sorts $N$ data items on $P$ processors, where $N = P$, uses a divide-and-conquer approach to sort the data items as follows. First, we divide all the data items into two groups $SQ_S$ and $SQ_L$, where $SQ_S = \{s_{0_S}, s_{1_S}, ..., s_{l_S}\}$ and $SQ_L = \{s_{0_l}, s_{1_l}, ..., s_{m_l}\}$ such that each data element of $SQ_S$, $s_{i_s}$ is smaller than each data element of $SQ_L$, $s_{j_l}$. However, the data elements of $SQ_S$ and the data elements of $SQ_L$ may not be sorted yet. In the next step of our division process, we divide the data elements of $SQ_S$ into two subgroups such that each data element of the first subgroup is smaller than each data element of the second subgroup. We do the same thing for the data elements of $SQ_L$. We continue this division

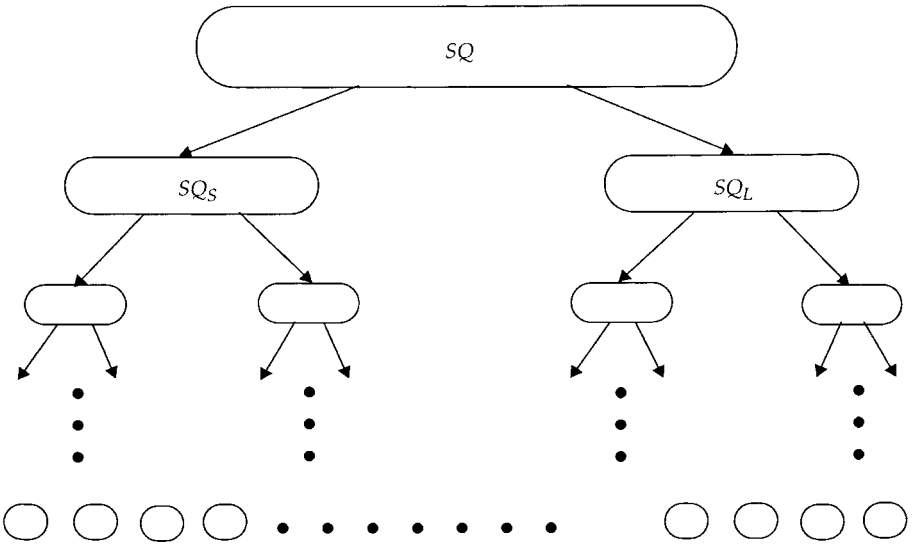


**FIG. 3.** The general idea of the divide-and-conquer sorting algorithm on a 1D RASOB.

process until the size of each subgroup is equal to 1, in which case all the data elements will be sorted in the 1D RASOB. Figure 3 further illustrates this division scheme.

The detailed steps of our recursive sorting algorithm on a 1D RASOB are described next.

Given $N$ data items, let each data item be represented by $k$ binary bits and distributed one data item per processor in a 1D RASOB. Hereafter, two processors are said to be in the same group as the data items they hold belong to the same group. Each processor also holds two variables, namely START and END, which represent the index of the left-most processor and the index of the right-most processor within the same group, respectively. At the beginning of the sorting algorithm (i.e., before any division of the data elements), all processors belong to a single group, and the index of the leftmost processor is 1 (i.e., START = 1). Similarly, at the beginning of the sorting algorithm, the variable END is equal to $N$ which is the index of the rightmost processor of the group. During the sorting algorithm, each processor maintains and updates the values of these two variables so that it knows the members of its own group. Members of the same group must have the same values of START and END. We denote $\text{START}_i$ and $\text{END}_i$ to be the values of the variables START and END of processor $p(i)$.

Our sorting algorithm performs $k$ iterations, where each iteration performs the following eight steps:

PROCEDURE SORT.
During iteration $I$ ($I = 1$ to $k$):

1. Each processor $p(i)$ broadcasts the data item it holds to $p(\text{START}_i)$, $p(\text{START}_{i+1})$, ..., $p(\text{END}_i)$, including itself, if the $I$th most significant bit of its data item is equal to 0. After this step, the RASOB processors will contain a variable number of data items in their respective receiving buffers.

2. Each processor $p(i)$ checks its receiving buffer if it contains an $(i-\text{START}_i+1)$th data item. If so, it marks the data down and does a *type I replacement* which is described in step 5 of the algorithm. Moreover, it clears its receiving buffer. The value of $(i-\text{START}_i+1)$ represents the position of $p(i)$ in its subgroup, starting from left to right. Processor $p(i)$ gets the $(i-\text{START}_i+1)$th data item in order to let the processors on its left get data items of smaller values since the $I$th significant bit is 0. As a result, the processors on its left side can form their own subgroup later.
3. This is analogous to Step 1 above. Each processor $p(i)$ broadcasts the data item it is holding to $p(\text{START}_i)$, $p(\text{START}_i+1)$, ..., $p(\text{END}_i)$, including itself, if the $I$th most significant bit of its data item is equal to 1. After this step, the RASOB processors will contain a variable number of data items in their respective receiving buffers.
4. Each processor $p(i)$ checks its receiving buffer if it contains an $(\text{END}_i-1+1)$th data item. If so, it marks the data down and does a *type II replacement* which is described in step 5 of the algorithm. Moreover, it clears its receiving buffer. The value of $(\text{END}_i-i+1)$ represents the position of $p(i)$ in its subgroup, starting from right to left. Processor $p(i)$ gets the $(\text{END}_i-i+1)$th data item in order to let the processors on its right get data items of larger values since the $I$th significant bit is 1. As a result, the processors on its right side can form their own subgroup later.
5. Each processor does a *type I replacement* or a *type II replacement* by replacing the data item it is holding with the marked data item.
6. Each processor $p(i)$ sends a message to $p(i-1)$ only if $i-1 \geqslant \text{START}_i$ and sends a message to $p(i+1)$ only if $i+1 \leqslant \text{END}_i$ to find out what type of *replacement* they have performed. In other words, processor $p(i)$ finds out whether the processor on its immediate left and the processor on its immediate right belong to the same group or not since $p(\text{START}_i)$ has no processor on its immediate left belonging to the same group and $p(\text{END}_i)$ has no processor on its immediate right belonging to the same group.
7. For each processor $p(i)$, if it has performed a *type I replacement* and finds that $i+1 \leqslant \text{END}_i$ and $p(i+1)$ has performed a *type II replacement*, then $p(i)$ sends a message to $p(\text{START}_i)$, $p(\text{START}_i+1)$, ..., $p(i)$ informing them to change their variable END to be equal to $i$.
8. For each processor $p(i)$, if it has performed a *type II replacement* and finds that $i-1 \geqslant \text{START}_i$ and $p(i-1)$ has performed a *type I replacement*, then $p(i)$ sends a message to $p(i)$, $p(i+1)$, ..., $p(\text{END}_i)$ informing them to change their variable START to be equal to $i$.

**End** {Procedure SORT}

We note here that Step 7 and Step 8 are used to divide a group of data items into two subgroups such that the value of each data element of the first subgroup is smaller than the value of each data element in the second subgroup. Further, within one iteration, say iteration $x$, of the above algorithm, each processor performs just

one type of *replacement* (i.e., *type I* or *type II*) since its $x$th most significant bit can be either 0 or 1 but not both at the same time. In the Appendix of this paper, we illustrate a step-by-step example of the application of the above algorithm to sort 10 data items on a 1D RASOB.

Now, we are ready to present the following theorem.

THEOREM 1. *The* SORT procedure *can be computed in* $O(k)$ *row communication cycles on a* 1D RASOB, *where* $k$ *is the size of the data elements to be sorted in bits.*

*Proof.* Step 1 and Step 3 of Procedure SORT procedure are simple broadcast operations on a 1D RASOB, where the corresponding processors load their respective data items into the appropriate *car* of the transmission *train*n of slots. Each of these *cars* (slots) will be read by all processors in a 1D RASOB in a single row communication cycle. Consequently, Step 1 and Step 3 of *Procedure* SORT take $O(1)$ communication cycles. Step 2, Step 4, and Step 5 of *Procedure* SORT each takes $O(1)$ time since they simply involve accessing the receiving buffers of the processors and the *replacement* of the data items they are holding. Step 6 is a simple routing pattern between neighboring processors which can be accomplished in a single row communication cycle. Finally, Step 7 and Step 8 of *Procedure* SORT involve a broadcasting operation which also takes a single row communication cycle. Hence, each iteration of *Procedure* SORT takes $O(1)$ row communication cycles. Consequently, the whole sorting algorithm on a 1D RASOB takes $O(k)$ row communication cycles since the number of iterations of *Procedure* SORT is $k$. ∎

We also note here that under most practical situations $k$ rarely exceeds 32. As a result, the communication time complexity of this sorting algorithm on a 1D RASOB is *almost* constant. Furthermore, it was argued in [30] that for reasonable size RASOBs (say up to 10,000), the duration of a communication cycle is comparable to the time for a CPU operation and may be assumed constant. Finally, we have to realize that no PRAM algorithm can achieve the above performance, since the lower bound theorem of [18] implies that sorting of $n$ bits on the CRCW PRAM will need $\Omega(\log n/\log\log n)$ time, given only a polynominal number of processors.

One possible drawback of the above *Procedure* SORT is that it requires each processor of the 1D RASOB to have a receiving buffer of size $N$, where $N$ is the size of the data items to be sorted. However, the above drawback can be overcome by simply installing a counter at each processor and requiring the size of the receiving buffers be equal to just 1 (e.g., they can hold just a single data item). During each iteration of *Procedure* SORT, each processor is required to replace the data item it is holding either by the $(i-\text{START}_i+1)$th data item it receives in its buffer in Step 2, or by the $(\text{END}_i-i+11)$th data item it receives in its buffer in Step 4. We can use these counters to keep track of the data item that the appropriate processor is interested in. More specifically, we first set the counter of each processor to 0 before the execution of Step 2 and before the execution of Step 4 of *Procedure* SORT. Then, we increment the counter by 1 for each received data item in the buffer of the processor. Since the receiving buffers of each processor can hold only a single data item, every new incoming data item will replace the old one, until the counter is equal to $i-\text{START}_i+1$ if Step 2 is being executed or the counter is

equal to $\text{END}_i - i + 1$ if Step 4 is being executed. Afterwards, each processor $p(i)$ stops accepting any more messages in its receiving buffer for the whole duration of either Step 2 or Step 4. Consequently, right after Step 2 or Step 4 of *Procedure* SORT, the data item inside the receiving buffer of each processor is the one needed to perform the *replacement* in Step 5.

### 3.2. *Sorting on a Linear RASOB* ($P < N$).

In the previous subsection, we have presented the implementation of a sorting algorithm on a 1D RASOB, where the number of data items to be sorted, $N$, is equal to the number of processors, $P$. At first glance, it may seem that such an algorithm is very limited. For example, what if we want to sort $N$ data items on a $P$-processor 1D RASOB, where $P < N$? Fortunately, there are several good methods for converting an algorithm that was designed for a $P_1$-processor network so that it can run on a $P_2$-processor network (where $P_2 < P_1$) with minimum slowdown [1, 21]. The only requirement is that the processors of $G_2$ be *coarser-grained* than the processors of $G_1$. For example, in order to sort $N$ data items on a $P$-processor 1D RASOB, where $P < N$, each processor would have to store at least $N/P$ data items for the RASOB to be able to sort $N$ items. As a result, the *granularity* of the processors should be large. The method consists of having each processor of $G_2$ simulate $\lceil P_1/P_2 \rceil$ processors of $G_1$. Typically this will induce a slowdown of $\lceil P_1/P_2 \rceil$, which is to be expected since we are using a factor of $P_1/P_2$ fewer processors. As an example, consider the problem of simulating a $P_1$-processor 1D RASOB $G_1$ on a $P_2$-processor 1D RASOB $G_2$. There are many ways of performing the simulation, but the simplest is to assign the tasks of processors $\lceil P_1/P_2 \rceil (i-1)+1$, $\lceil P_1/P_2 \rceil (i-1)+2$, ..., $\lceil P_1/P_2 \rceil i$ of $G_1$ to processor i of $G_2$ for $1 \leqslant i \leqslant P_2$. For example, see Fig. 4. Each step of $G_1$ can then be simulated in $\lceil P_1/P_2 \rceil$ steps on $G_2$. Hence, any algorithm that runs in $T$ steps on $G_1$ can be run in $T \lceil P_1 P_2 \rceil$ on $G_2$.

By applying the above technique, we can sort $N$ data items on a $P$-processor 1D RASOB, where $P < N$, in $O(kN/P)$ steps using *Procedure SORT* described in the previous subsection.

The preceding method provides a good answer to the question of what to do when $P < N$. In fact, the same technique will work for virtually all types of algorithms designed for RASOB where the input data is equal to the number of processor. As a consequence, the algorithms designed in this paper will focus our attention on a RASOB architecture that grows with the size of the input without losing generality, secure in the knowledge that the resulting algorithms can later be scaled to any smaller size RASOB.

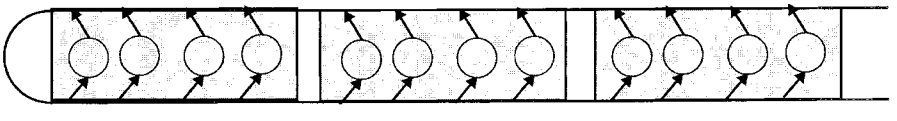

**FIG. 4.** Simulating a 12-processor 1D RASOB $G_1$ on a 3-processor 1D RASOB $G_2$. Each processor in $G_2$ is responsible for simulating 4 processors of $G_1$. Hence, every step of $G_1$ takes 4 steps in $G_2$, and any algorithm that runs in $T$ steps on $G_1$ can be made to run in 4T steps in $G_2$.

### 3.3. Sorting on a 2D RASOB

*Procedure* SORT on the 1D RASOB shown in the previous section can be easily extended to be implemented on a 2D RASOB while retaining the same communication time complexity, $O(k)$ cycles. One way to extend *Procedure* SORT for its implementation on a 2D RASOB is to use Leighton's *Columnsort* algorithm [18] or to use Maberg and Gafni *Rotatesort* algorithm [23]. In our implementation of *Procedure* SORT on the 2D RASOB, we use the *Rotatesort* algorithm. However, the *Columnsort* implementation can be done in a similar way.

The *Rotatesort* algorithm is a row–column sorting technique proposed originally for the two-dimensional mesh processor arrays [23]. As this paper is not particularly concerned with 2D mesh arrays, the main interest is in *Rotatesort* as it applies to a 2D arrays of data items to be sorted. More precisely, the fact that *Rotatesort* partitions a set of $N$ data items into subsets (rows and columns), which can be sorted independently and efficiently on a 2D RASOB using the 1D RASOB SORT *procedure*.

Given $N = RS$ data elements arranged as a 2D $R \times S$ array, the *Rotatesort* technique [23] sorts the $N$ data elements by alternately transforming the rows and columns of the array. The number of row and column communication cycles will be constant (14 or 16 communication cycles) and will be shown later. Each transformation phase consists of either performing a circular shift operation on the elements of each row or each column.

During *Rotatesort*, the $R \times S$ array of data elements is partitioned as shown in Fig. 5. A *vertical strip* (or, *horizontal strip*) is an $R \times S^{1/2}$ (or, $S^{1/2} \times S$) subarray of data elements. Also, a *block* is a $S^{1/2} \times S$ subarray of data elements. The algorithm as presented in [23] assumes that $R = 2^r$ and $S = 2^s$, where $s$ is an even integer and $r \geqslant s/2$. However, other values of $R$ and $S$ can be used with little modification to the algorithm. The algorithm description can be facilitated by defining the macros (operations):

- Macro BALANCE applies to a subarray of size $u \times v$ and consists of three steps:
  a. Sort all columns downward.
  b. Rotate each row $i$ rightward by $(i \bmod v)$ positions.
  c. Sort all columns downward.
- Macro UNBLOCK distributes the data elements of each block among all columns. It consists of two steps:
  a. Rotate each row $i$ rightward by $(iS^{1/2} \bmod S)$ positions.
  b. Sort all columns downward.
- Macro SHEAR: equivalent to performing one iteration of the *shear-sort* algorithm [1]. It consists of two steps:

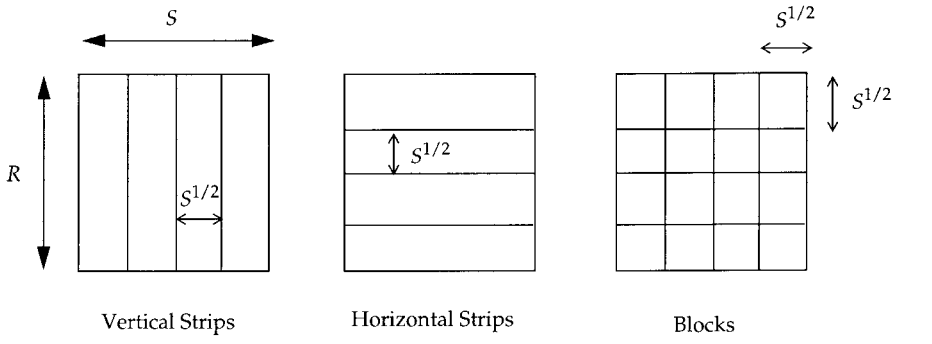


**FIG. 5.** Partitions of the $R \times S$ array.

a. Sort all even-numbered rows rightward and all odd-numbered rows leftward.

b. Sort all columns downward.

The *Rotatesort* algorithm can be described in terms of these macros as

PROCEDURE Rotatesort.

(1) Perform BALANCE on each vertical slice.

(2) Perform UNBLOCK on the entire array.

(3) Perform BALANCE on each horizontal slice.

(4) Perform UNBLOCK on the entire array.

(5) Perform three iterations of SHEAR on the array.

(6) Sort all rows rightward.

Following the applications of the above macros, the $R \times S$ array will be sorted in row-major order. It is clear that the above algorithm uses eight column communication cycles and nine row communication cycles, for a total of 17 communication cycles. However, since Step 3c) (i.e., step c of the second BALANCE operation) and Step 4a) both involve row transformations, the two steps can be combined into one row transformation, thus reducing the total number of communication cycles to 16. Furthermore, when $R \leqslant S^{1/2}$ then the number of communication cycles can be reduced to 14.

Now, let us examine the time complexity of this algorithm when implemented on a 2D RASOB. The BALANCE macro consists of two sorting steps along the columns of the 2D RASOB. That is, we need to perform two sorting steps on a 1D RASOB using *Procedure* SORT shown in the previous section. Thus, the two sorting steps of the BALANCE macro can be performed on a 2D RASOB in $O(k)$ row communication cycles. The second step needed in the BALANCE macro is the rotation of each row $i$ by $(i \bmod v)$ positions. This can be easily accomplished in a single row communication cycle on the 2D RASOB using the *loading/unloading train* model. Hence, the BALANCE macro can be executed on a 2D RASOB in $O(k)$ communication cycles. The UNBLOCK macro consists of one sorting step

along the columns which takes $O(k)$ column communication cycles as shown above, and one rotation step of all the rows rightward which takes a single row communication cycle. Therefore, the UNBLOCK macro can be executed on a 2D RASOB in $O(k)$ communication cycles. Finally the SHEAR macro which consists of two sorting steps along the rows and along the columns also can also be executed on a 2D RASOB in $O(k)$ communication cycles using *Procedure* SORT of the 1D RASOB. Consequently, the whole *Rotatesort procedure* can be executed on the 2D RASOB in $O(k)$ communication cycles, where $k$ is the size of data elements to be sorted in bits.

As mentioned previously, under most practical situations $k$ rarely exceeds 32. Note that, the total number of cycles is 10's of $k$; therefore, the constant in $O(k)$ is in the 10's. Given $k \leqslant 32$, our sorting algorithm on a 2D RASOB is almost a constant-time sorting. In fact, given RASOB's capability of supporting communications between processors at different rows and different columns, the number of cycles needed can be further reduced by eliminating the rotation on each row in the macros BALANCE and UNBLOCK before sorting each column. A step-by-step application of the *Rotatesort procedure* is illustrated in the Appendix of this paper.

This leads to the following theorem.

THEOREM 2. *Sorting N data items can be computed in $O(k)$ communication cycles on a* 2D *RASOB size N, where k is the size of the data items to be sorted in bits.*

## 4. CONCLUSION

The *reconfigurable array with slotted optical buses* (RASOB) architecture has recently been proposed as an alternative to reconfigurable arrays with *electronic* buses. It takes advantage of the unique properties of optical transmission to achieve flexible reconfiguration and strong connectivities with low hardware and control complexity. In this paper, we used this novel architecture for the implementation of two efficient sorting algorithms. The first sorting algorithm has been implemented on a 1D RASOB. This sorting algorithm, which is based on a divide-and-conquer approach, has a communication time complexity of $O(k)$ cycles, where $k$ is the size of the data elements to be sorted in bits. The second algorithm has been implemented on a 2D RASOB. It uses the sorting algorithms implemented for the 1D RASOB, in conjunction with the well-known *Rotatesort* algorithm to achieve an $O(k)$ communication time complexity. Both of these algorithms assume that the size of the data elements to be sorted is equal to the size of the number of processors. However, they can be extended easily to the case where the size of the data elements is larger than the number of processors. Moreover, the developed algorithms are more efficient that state-of-the-art sorting algorithms which have been recently implemented on various models of array of processors with reconfigurable electronic buses.

## APPENDIX A

A step-by-step example of the application of Procedure SORT of Section 3.1 to sort 10 data items on a 1D RASOB is given by the following Figs. a–m:

| | ① | ② | ③ | ④ | ⑤ | ⑥ | ⑦ | ⑧ | ⑨ | ⑩ |
|---|---|---|---|---|---|---|---|---|---|---|
| *START* | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| *END* | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
| Input | 1101 | 0010 | 1111 | 1000 | 0101 | 1010 | 0111 | 0001 | 1001 | 1110 |

**FIG. a.** The initial configuration. The values of *START* and *END* are given in decimals and the values of Input are represented in binary.

| | ① | ② | ③ | ④ | ⑤ | ⑥ | ⑦ | ⑧ | ⑨ | ⑩ |
|---|---|---|---|---|---|---|---|---|---|---|
| Buffer 1 | 0010* | 0010 | 0010 | 0010 | 0010 | 0010 | 0010 | 0010 | 0010 | 0010 |
| Buffer 2 | 0101 | 0101* | 0101 | 0101 | 0101 | 0101 | 0101 | 0101 | 0101 | 0101 |
| Buffer 3 | 0111 | 0111 | 0111* | 0111 | 0111 | 0111 | 0111 | 0111 | 0111 | 0111 |
| Buffer 4 | 0001 | 0001 | 0001 | 0001* | 0001 | 0001 | 0001 | 0001 | 0001 | 0111 |
| Buffer 5 | — | — | — | — | — | — | — | — | — | — |

**FIG. b.** After step 1 of the algorithm. This iteration deals with the first bit of each input number; *signifies the data item that is doing the *replacement*.

| | ① | ② | ③ | ④ | ⑤ | ⑥ | ⑦ | ⑧ | ⑨ | ⑩ |
|---|---|---|---|---|---|---|---|---|---|---|
| Buffer 1 | 1101 | 1101 | 1101 | 1101 | 1101 | 1101 | 1101 | 1101 | 1101 | 1101* |
| Buffer 2 | 1111 | 1111 | 1111 | 1111 | 1111 | 1111 | 1111 | 1111 | 1111* | 1111 |
| Buffer 3 | 1000 | 1000 | 1000 | 1000 | 1000 | 1000 | 1000 | 1000* | 1000 | 1000 |
| Buffer 4 | 1010 | 1010 | 1010 | 1010 | 1010 | 1010 | 1010* | 1010 | 1010 | 1010 |
| Buffer 5 | 1001 | 1001 | 1001 | 1001 | 1001 | 1001* | 1001 | 1001 | 1001 | 1001 |
| Buffer 6 | 1110 | 1110 | 1110 | 1110 | 1110* | 1110 | 1110 | 1110 | 1110 | 1110 |
| Buffer 7 | — | — | — | — | — | — | — | — | — | — |

**FIG. c.** The state of the processors' receiving buffers after step 3 of the algorithm.

| | ① | ② | ③ | ④ | ⑤ | ⑥ | ⑦ | ⑧ | ⑨ | ⑩ |
|---|---|---|---|---|---|---|---|---|---|---|
| *START* | 1 | 1 | 1 | 1 | 5 | 5 | 5 | 5 | 5 | 5 |
| *END* | 4 | 4 | 4 | 4 | 10 | 10 | 10 | 10 | 10 | 10 |
| Input | 0010 | 0101 | 0111 | 0001 | 1110 | 1001 | 1010 | 1000 | 1111 | 1101 |

**FIG. d.** The placement of the data elements after step 8 of the algorithm, and the associated values for START and END.

| | ① | ② | ③ | ④ | ⑤ | ⑥ | ⑦ | ⑧ | ⑨ | ⑩ |
|---|---|---|---|---|---|---|---|---|---|---|
| Buffer 1 | 0010* | 0010 | 0010 | 0010 | 1001* | 1001 | 1001 | 1001 | 1001 | 1001 |
| Buffer 2 | 0001 | 0001* | 0001 | 0001 | 1010 | 1010* | 1010 | 1010 | 1010 | 1010 |
| Buffer 3 | — | — | — | — | 1000 | 1000 | 1000* | 1000 | 1000 | 1000 |
| Buffer 4 | — | — | — | — | — | — | — | — | — | — |

**FIG. e.** The state of the processors' receiving buffers after step 1 of the second iteration which deals with the second bit of each input number.

| | ① | ② | ③ | ④ | ⑤ | ⑥ | ⑦ | ⑧ | ⑨ | ⑩ |
|---|---|---|---|---|---|---|---|---|---|---|
| Buffer 1 | 0101 | 0101 | 0101 | 0101* | 1110 | 1110 | 1110 | 1110 | 1110 | 1110* |
| Buffer 2 | 0111 | 0111 | 0111* | 0111 | 1111 | 1111 | 1111 | 1111 | 1111* | 1111 |
| Buffer 3 | — | — | — | — | 1101 | 1101 | 1101 | 1101* | 1101 | 1101 |
| Buffer 4 | — | — | — | — | — | — | — | — | — | — |

**FIG. f.** The state of the processors' receiving buffers after step 3 of the second iteration.

| | ① | ② | ③ | ④ | ⑤ | ⑥ | ⑦ | ⑧ | ⑨ | ⑩ |
|---|---|---|---|---|---|---|---|---|---|---|
| *START* | 1 | 1 | 3 | 3 | 5 | 5 | 5 | 8 | 8 | 8 |
| *END* | 2 | 2 | 4 | 4 | 7 | 7 | 7 | 10 | 10 | 10 |
| Input | 0010 | 0001 | 0111 | 0101 | 1001 | 1010 | 1000 | 1101 | 1111 | 1110 |

**FIG. g.** The placement of the data elements after step 8 of the second iteration of the algorithm, and the associated values for *START* and *END*.

| | ① | ② | ③ | ④ | ⑤ | ⑥ | ⑦ | ⑧ | ⑨ | ⑩ |
|---|---|---|---|---|---|---|---|---|---|---|
| Buffer 1 | 0001* | 0001 | 0101* | 0101 | 1001* | 1001 | 1001 | 1101* | 1101 | 1101 |
| Buffer 2 | — | — | — | — | 1000 | 1000* | 1000 | — | — | — |
| Buffer 3 | — | — | — | — | — | — | — | — | — | — |

**FIG. h.** The state of the processors' receiving buffers right after step 1 of the third iteration of the algorithm.

| | ① | ② | ③ | ④ | ⑤ | ⑥ | ⑦ | ⑧ | ⑨ | ⑩ |
|---|---|---|---|---|---|---|---|---|---|---|
| Buffer 1 | 0010 | 0010* | 0111 | 0111* | 1010 | 1010 | 1010* | 1111 | 1111 | 1111* |
| Buffer 2 | — | — | — | — | — | — | — | 1110 | 1110* | 1110 |
| Buffer 3 | — | — | — | — | — | — | — | — | — | — |

**FIG. i.** The state of the processors' receiving buffers right after step 3 of the third iteration of the algorithm.

| | ① | ② | ③ | ④ | ⑤ | ⑥ | ⑦ | ⑧ | ⑨ | ⑩ |
|---|---|---|---|---|---|---|---|---|---|---|
| *START* | 1 | 2 | 3 | 4 | 5 | 5 | 7 | 8 | 9 | 9 |
| *END* | 1 | 2 | 3 | 4 | 6 | 6 | 7 | 8 | 10 | 10 |
| Input | 0001 | 0010 | 0101 | 0111 | 1001 | 1000 | 1010 | 1101 | 1110 | 1111 |

**FIG. j.** The placement of the data elements after step 8 of the third iteration of the algorithm and the associated values for *START* and *END*.

| | ① | ② | ③ | ④ | ⑤ | ⑥ | ⑦ | ⑧ | ⑨ | ⑩ |
|---|---|---|---|---|---|---|---|---|---|---|
| Buffer 1 | — | 0010* | — | — | 1000* | 1000 | 1010* | — | 1110* | 1110 |
| Buffer 2 | — | — | — | — | — | — | — | — | — | — |

**FIG. k.** The state of the processors' receiving buffers right after step 1 of the fourth iteration of the algorithm.

| | ① | ② | ③ | ④ | ⑤ | ⑥ | ⑦ | ⑧ | ⑨ | ⑩ |
|---|---|---|---|---|---|---|---|---|---|---|
| Buffer 1 | 0001 | — | 0101* | 0111* | 1001 | 1001* | — | 1101* | 1111 | 1111* |
| Buffer 2 | — | — | — | — | — | — | — | — | — | — |

**FIG. l.** The state of the processors' receiving buffers right after step 3 of the fourth iteration of the algorithm.

| | ① | ② | ③ | ④ | ⑤ | ⑥ | ⑦ | ⑧ | ⑨ | ⑩ |
|---|---|---|---|---|---|---|---|---|---|---|
| *START* | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| *END* | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Input | 0001 | 0010 | 0101 | 0111 | 1000 | 1001 | 1010 | 1101 | 1110 | 1111 |

**FIG. m.** The placement of the data elements after step 8 of the fourth iteration of the algorithm, and the associated values for *START* and *END*. This is the end of the algorithm.

## APPENDIX B

A step-by-step application of the *Rotatesort Procedure* on a $4 \times 4$ 2D RASOB is given by the following Figs. a–i:

| | | | |
|---|---|---|---|
| 10 | 9 | 14 | 2 |
| 4 | 15 | 11 | 12 |
| 6 | 1 | 5 | 13 |
| 8 | 3 | 7 | 0 |

(a)

| | | | |
|---|---|---|---|
| 4 | 2 | 5 | 0 |
| 6 | 3 | 7 | 2 |
| 8 | 9 | 11 | 12 |
| 10 | 15 | 14 | 13 |

(b)

| | | | |
|---|---|---|---|
| 0 | 1 | 4 | 5 |
| 2 | 3 | 6 | 7 |
| 8 | 9 | 11 | 12 |
| 10 | 13 | 14 | 15 |

(c)

| | | | |
|---|---|---|---|
| 0 | 1 | 4 | 5 |
| 6 | 7 | 2 | 3 |
| 8 | 9 | 11 | 12 |
| 14 | 15 | 10 | 13 |

(d)

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 6 | 7 | 4 | 5 |
| 8 | 9 | 10 | 12 |
| 14 | 15 | 11 | 13 |

(e)

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 5 | 6 | 7 | 4 |
| 10 | 11 | 8 | 9 |
| 15 | 12 | 13 | 14 |

(f)

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 7 | 4 | 5 | 6 |
| 10 | 11 | 8 | 9 |
| 13 | 14 | 15 | 13 |

(g)

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 7 | 6 | 5 | 4 |
| 8 | 9 | 10 | 11 |
| 15 | 14 | 13 | 12 |

(h)

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |

(i)

## REFERENCES


1. S. G. Akl, "The Design and Analysis of Parallel Algorithms," Prentice–Hall, Englewood Cliffs, NJ, 1989.
2. R. Alferness, L. Buhl, S. Korothy, and R. Tucker, High-speed *b*-reversal directional coupler switch, *Photonic Switching*, *OSA Technical Digest* **13** (1987), 77–78.
3. R. Ballart and Y. C. Chin, SONET: Now it's the standard optical network, *IEEE Commun. Mag.* **27**, 3 (Mar. 1989), 8–15.
4. Y. Ben-Asher, D. Pelg, R. Ramaswami, and A. Shuster, The power of reconfiguration, *J. Parallel Distrib. Comput.* **13** (1991), 139–153.
5. A. Benner, H. Jordan, and V. Heuring, Optically switched lithium niobate directional couplers for digital optical computing, *in* "SPIE Proceedings, Digital Optical Computing II," Vol. 1215, pp. 343–352, SPIE, Bellingham, WA, 1990.
6. D. M. Charulli, S. P. Levitan, R. G. Melhem, M. Bidnurkar, R. Ditmore, G. Gravenstreter, Z. Guo, C. Qiao, M. F. Sakr, and J. P. Teza, Optoelectronic buses for high-performance computing, *Proc. IEEE* **82**, 11 (Nov. 1994), 1702–1710.

7. R. E. Floren *et al*., Optical interconnects in the Touchstone supercomputer program, *in* "SPIE Proc. Intergrated Optoelectronics for Communication and Processing, October 1991," Vol. 1582, pp. 46–54.
8. A. Guha, J. Bristow, C. Sullivan, and A. Husain, Optical interconnections for massively parallel architectures, *Appl. Opt*. **29**, 8 (1980).
9. Z. Guo, R. Melhem, R. W. Hall, C. Chiarulli, and S. P. Levitan, Array processors with pipelined optical busses, *J. Parallel Distrib. Comput*. **12**, 3 (1991), 269–282.
10. Z. Guo, Optically interconnected processor arrays with switching capability, *J. Parallel and Distributed Computing* **23** (1994), 314–329.
11. M. Hamdi and Y. Pan, Efficient parallel algorithms on optically interconnected arrays of processors, *IEE Proceedings-Computers and Digital Techniques*, Vol. 142, pp. 87–92, March 1995.
12. J. Jang and V. K. Prasanna, An optimal sorting algorithm on a reconfigurable mesh, *in* "Proc. Int. Parallel Processing Symp.," pp. 130–137, 1992.
13. B. O. Kahle, E. C. Parish, T. A. Lane, and J. A. Quam, Optical interconnects for interprocessor communications in the connection machine, *in* "IEEE Conf. on Computer Design," Oct. 1989.
14. T. W. Kao, S. J. Horng, Y. L. Wang, and H. R. Tasi, Designing efficient parallel algorithms on CRAP, *IEEE Trans. Parallel Distrib*. (1995), 554–560.
15. D. Kiefer and V. Swanson, Implementation of optical clock distribution in a supercomputer, Cray Research Inc., internal report, 1995.
16. V. P. Kumar and C. S. Raghavendra, Array processors with multiple broadcasting, *Journal of Parallel and Distributed Computing* **2** (1987), 173–190.
17. P. Lalwaney, L. Zenou, A. Ganz, and I. Koren, Optical interconnects for multiprocessors: Cost performance analysis, *in* "Proc. on Frontiers of Mass. Para. Comp.," pp. 278–285, Oct. 1992.
18. F. T. Leighton, Tight bounds on the complexity of parallel sorting, *IEEE Trans. Comput*., (1985), 344–354.
19. F. T. Leighton, "Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes," pp. 622–624, Morgan Kaufmann, San Mateo, CA, 1992.
20. S. P. Levitan, D. M. Chiarulli, and R. G. Melhem, Coincident pulse addressing technique for multiprocessor interconnections structures, *Appl. Opt*. **29**, No. 14 (May 1990), 2024–2033.
21. H. Li and Q. F. Stout (Eds.), "Reconfigurable Massively Parallel Computers," Prentice–Hall, 1991.
22. Y. S. Liu *et al*., Polymer optical interconnect technology (POINT) optoelectronic packaging and iterconnect for board and backplane applications, *in* "Proceedings 46th Electronic Components and Technology Conference," pp. 308–315, 1996.
23. J. M. Maberg and E. Gafni, Sorting in constant number of row and column phases on a mesh, *Algorithmica* (1988), pp. 561–572.
24. R. Melhem, D. Chiarulli, and S. Levitan, Space multiplexing of waveguides in optically interconnected multiprocessor systems, *The Computer Journal* **32** (4) (1989), 362–369.
25. R. Miller, V. K. P. Kumar, D. I. Reisis, and Q. F. Stout, Parallel computations on reconfigurable meshes, *IEEE Trans. Comp*. **42** (1993), 678–692.
26. M. Nigam and S. Sahni, Sorting $n$ numbers on $n \times n$ reconfigurable meshes with buses, *Journal of Parallel and Distributed Computing* **23** (1994), pp. 37–48.
27. Y. Pan, Order statistics on optically interconnected multiprocessor systems, *Opt. Laser Technol*. **26**, 4 (Aug. 1994), pp. 281–287.
28. Y. Pan and M. Hamdi, Singular value decomposition on processor arrays with pipe-lined bus systems, *Journal of Network and Computer Applications* **19**, No. 3 (July 1996), 235–248.
29. Y. Pan and K. Li, Linear array with a reconfigurable pipelined bus system: Concepts and applications, *in* "Proceedings 1996 International Conference on Parallel and Distributed Processing Techniques," 1996.
30. S. Pavel and S. G. Akl, "On the Power of Arrays with Reconfigurable Optical Buses," Technical Report 95-374, Department of Computing and Information Science, Queen's University, 1995.

31. S. Pavel and S. G. Akl, Efficient algorithms for the hough transform on arrays with reconfigurable optical buses, *in* "Proceedings of the International Parallel Processing Symposium," pp. 697–701, 1996.

32. S. Pavel and S. G. Akl, Matrix operations using arrays with reconfigurable optical buses, *Journal of Parallel Algorithms and Applications*, to appear.

33. C. Qiao, R. G. Melhem, D. M. Chiarulli, and S. P. Levitan, Optical multicasting in linear arrays, *International Journal of Optical Computing* **2**, No. 1 (April 1991), 31–48.

34. C. Qiao and R. Melhem, Time-division optical communications in multiprocessor arrays, *IEEE Transactions on Computers* **42**, (5) (May 1993), 577–590.

35. C. Qiao, On designing communication-intensive algorithms for a spanning optical bus based array, *Parallel Processing Letters* **5**, No. 5 (Sept. 1995), 499–511.

36. S. Rajsekaran and S. Sahni, "Sorting, Selection and Routing on the Array with Reconfigurable Optical Buses," Technical Report, Department of Computer and Information Science, University of Florida, 1996.

37. D. Sarrazin, H. Jordan, and V. Heuring, Fiber optic delay line memory, *Applied Optics* **29**, No. 5 (Feb. 1990), 627–637.

38. S. Tang *et al*., 1-GHZ clock signal distribution for multiprocessor supercomputer, *Proc. Massively Parallel Processing with Optical Interconnects '96* (1996), 186–191.

39. B. F. Wang, G. H. Chen, and F. C. Lin, Constant time sorting on a processor array with a reconfigurable bus system, *Information Processing Letters* (1990), 187–192.

40. C. Zhao, T-h Oh, and R. Chen, General purpose bi-directional optical backplane: high-performance bus for multiprocessor systems, *in* "Proc. Massively Parallel Processing with Optical Interconnects," pp. 188–194, 1995.

---

MOUNIR HAMDI received the B.Sc. degree with distinction in Electrical Engineering (Computer Engineering) from the University of Southwestern Louisiana in 1985, and M.Sc. and Ph.D. degrees in Electrical Engineering from the University of Pittsburgh in 1987 and 1991, respectively. While at the University of Pittsburgh, he was a research fellow involved with various research projects on interconnection networks, high-speed communication, parallel algorithms, switching theory, and computer vision. In 1991 he joined the Computer Science Department at Hong Kong University of Science and Technology as an Assistant Professor. He is now Associate Professor of Computer Science and Director of the Computer Engineering Programme. His main areas of research are parallel computing, ATM packet switching architectures, high-speed networks, and wireless networking. Dr. Hamdi has published over 80 papers on these areas in various journals, conference proceedings, and book chapters. He was guest editor of a special issue of *Informatica* on "optical parallel computing". He co-founded and co-chaired the International Workshop on High-Speed Network Computing, is on the editorial board of the *IEEE Communicmations Magazine* and *Parallel Computing*, and has been on the program committee of various international conferences. Dr. Hamdi received the best paper award at the 12th International Conference on Information Networking. Dr. Hamdi is a member of IEEE and ACM.

CHUNMING QIAO got his B.S. degree in Computer Science and Engineering in 1985 from the University of Science and Technology of China (USTC) in Hefei, People's Republic of China. He received the Andrew-Mellon Distinguished doctoral fellowship award from the University of Pittsburgh and later got his Ph.D. degree in computer science there in 1993. Since then, he joined SUNY at Buffalo as an assistant professor in the ECE department and received a NSF Research Initiation Award (RIA) in 1994 for his research on fiber-optic interconnection networks. Dr. Qiao's research interests cover the two converging areas of computers and communications. They include parallel and distributed computing and systems, interconnection networks, and photonic switching. Currently, he is conducting NSF funded research on optical burst switching (OBS) for fiber-optic wavelength division multiplexed (WDM) networks and internetworks (e.g., IP over WDM). Dr. Qiao has published more than 50 papers in *IEEE Trans. on Computers*, *IEEE Trans. on Parallel and Distributed Systems*, *IEEE Trans. on Communications*, and *IEEE Trans. on Networking*, as well as other fine journals and conference proceedings. He has served as a co-chair for both the 1997 and 1998 All-optical Networking Conferences, a Program vice

co-chair for the 1998 International Conference on Computer Communications and Networks (IC3N), a session organizer/chair at the IEEE MILCOM'96 (the Conference on Military Communications), and program committee member and session chairs in several other conferences and workshops. He is also an editor of the *Journal on High-Speed Networks (JHSN)* and a member of the IEEE Computer Society and the IEEE Communications Society.

YI PAN was born in Jiangsu, China. He entered Tsinghua University in March 1978 with the highest college entrance examination score among all 1977 high school graduates in Jiangsu Province. He received his B.Eng. degree in computer engineering from Tsinghua University, China in 1982 and his Ph.D. degree in computer science from the University of Pittsburgh in 1991. Dr. Pan joined the Department of Computer Science at the University of Dayton, Ohio, in 1991 and has been an associate professor since 1996. His research interests include parallel algorithms and architectures, optical communication and computing, distributed computing, task scheduling, and networking. He is an author of more than 60 research papers, a co-editor of four conference proceedings and a new book entitled *Parallel Computing Using Optical Interconnections*, published by Kluwer Academic in 1998, and a contributor of several book chapters. He has received several awards including the Japan Society for the Promotion of Science Fellowship (1998), AFOSR Summer Faculty Fellowship (1997), NSF Research Opportunity Award (1994, 1996), the best paper award from PDPTA '96, and Andrew Mellon Fellowship from Mellon Foundation (1990). His research has been supported by NSF, AFOSR, U.S. Air Force, and the state of Ohio. Dr. Pan is currently on the editorial board of the *Journal of Supercomputing* and the *Journal of Parallel and Distributed Computing Practices*, and he is an associate editor of the *Internation Journal of Parallel and Distributed Systems and Networks*. He has served as a guest editor of special issues for four international journals: *Information Sciences*, *Parallel Processing Letters*, *Informatica*, *and International Journal of Parallel and Distributed Systems and Networks*. He is the program chair of the 10th International Conference on Parallel and Distributed Computing and Systems in 1998, the conference co-chair of the Fourth International Conference on Computer Science & Informatics in 1998, and the program chair of the 1999 IPPS Workshop on Optics and Computer Science. He has also served as vice program chair, publicity chair, session chair, or committee member for more than 15 international conferences. Dr. Pan is a senior member of IEEE and a member of the IEEE Computer Society. He is listed in Men of Achievement and Marquis' *Who's Who in the Midwest*.

J. TONG received his B.Sc. degree from the Computer Science Department at Hong Kong University of Science and Technology with honor. He participated in this research as part of his final-year thesis. His areas of interest are algorithms design and analysis and high-performance computing.