Fast Sorting Algorithms on Reconfigurable Array of Processors With Optical Buses

Mounir Hamdi, J. Tong, and C. W. Kin
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

Abstract

The Reconfigurable Array with Spanning Optical Buses (RASOB) has recently received a lot of attention from the research community. By taking advantage of the unique properties of optical transmission, the RASOB provides flexible reconfiguration and strong connectivities with low hardware and control complexities. In this paper, we use this architecture for the design of efficient sorting algorithms on the 1-D RASOB and the 2-D RASOB. Our parallel sorting algorithm on the 1-D RASOB, which sorts N data items using N processors in O(k) time where k is the size of the data items to be in bits, is based on a novel divide-and-conquer scheme. On the other hand, our parallel sorting algorithm on the 2-D RASOB is based on the sorting algorithm on the 1-D RASOB in conjunction with the well known Rotatesort algorithm. This algorithm sorts N data items on a 2-D RASOB of size N in O(k) time. These sorting algorithms outperform state-of-the-art sorting algorithms on reconfigurable arrays of processors with electronic buses.

1. Introduction

Reconfigurable architectures are attractive because they provide alternatives to completely connected systems at lower implementation costs. Since optical interconnects can offer many advantages over its electronic counterpart, they will soon be a viable alternative for *multiprocessor interconnections* [3]. This paper describes the *Reconfigurable Array with Spanning Optical Buses* (*RASOB*) architecture which provides flexible reconfiguration as well as rich connectivities at low hardware and control complexities [2, 11]. Then we use this architecture for the efficient implementation of sorting algorithms which outperform state-of-the-art sorting algorithms on arrays of processors with *electronic* buses.

A *unique* feature of the *RASOB* architecture that distinguishes it from other array of processors with either optical or electronic buses [3, 19] is that *there is a direct connection between any two processors*. More specifically, in a *RASOB*, a processor at row i and column j can send a message, without buffering at any intermediate processor, to a processor at row k and column l, even if $i \neq k$ and $j \neq l$. Such a direct connection between these two processors at different rows and different columns can be established by setting an electro-optical switch [23] that interconnects the row i and column l. We will refer to the operation of setting switches as *hardware reconfiguration* in an *RASOB*.

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

The *RASOB* architecture also takes advantage of two important properties of the *optical transmissions*, namely, unidirectional propagation and predictable unit propagation delay. Hence, the processors can be programmed to send and receive messages under synchronized control, such that a connection between a source and a destination is established by letting the source send a message at a specific point in time and letting the destination receive the message at another specific point in time [3, 8, 23]. We refer to this type of reconfiguration as *software reconfiguration*.

Because some of the reconfiguration is done in software, the complexity of both hardware and control required for the reconfiguration in an *RASOB* can be kept low. However, despite its low control and hardware complexity, the proposed *RASOB* architecture provides flexible reconfiguration that leverages the high communication bandwidths available in optical interconnects. As a result, the *RASOB* is a very promising architecture for the efficient parallel implementation of many communication intensive algorithms.

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 1-D *RASOB* and on a 2-D *RASOB*. Finally, we conclude in section 4.

2. Architectural Model

The *RASOB* architecture is similar to the array structure described in [2, 10] and in particular to the array structures described in [11]. A main difference is that in the proposed architecture, messages are sent and received according to specific timing requirements. This makes the proposed architecture suitable for SIMD applications. On the other hand, the structure in [11] employs an addressing mechanism which supports MIMD applications at higher hardware and control complexities. Figure 1 illustrates the architecture of a 2-D *RASOB*. As shown in Figure 1a, there are n folded row buses and n folded column buses. 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. We denote the row of a 2-D *RASOB* or the column of a 2-D *RASOB* as a 1-D *RASOB*. Further, the term

0-8186-7267-6/96 $05.00 © 1996 IEEE

RASOB is used to denote a 2-D *RASOB*.

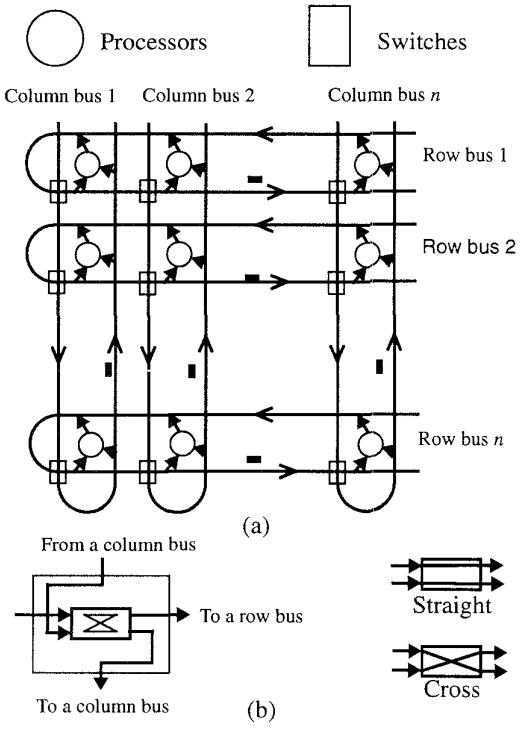

Figure 1. (a) The architecture of *RASOB*, and (b) A switch interconnecting a row and a column bus.

A distinct architectural feature of the *RASOB* is that a 2 × 2 electro-optical switch is placed at the intersection of a row and a column bus, as shown in Figure 1.b. When the switch is set to "*straight*", a message arriving along a row bus will continue propagating; otherwise, the message will be *switched* to the column bus. During a specific period, all the switches at a given row are set to *straight*. 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 phase*. 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 phase*. In doing so, the switches are set to "*cross*" for the duration of the message and then changed back to the straight state.

2.1 Software Reconfiguration

In a row phase, 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 Figure 2. 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,

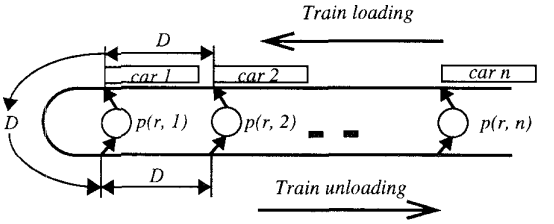

Figure 2. Train loading/unloading on a row bus.

unidirectional propagation and *predictable propagation delay* of the optical signals, that make concurrent access of an optical bus possible. With an appropriate spatial separation between the neighboring PEs, message collision can be avoided even when the PEs are transmitting messages concurrently [2, 3, 8, 11]. Hence, we assume that each processor on a row bus is separated in time by $D = bw + \delta$ 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 errors. 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 Figure 2.

We may use the *train loading/unloading* model to describe the operations in a row phase. Let us imagine that at the beginning of a row phase, a *train* of n *cars* is originated at the right-most end of the upper segment of the row bus. Each *car* can be regarded as an empty packet slot with a duration of D. During a row phase, 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.

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

$$RowSend[(r,i)] = (i-1)D + (n-i)D = (n-1)D$$

As a result, all processors will be transmitting simultaneously. 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,

$$RowRec[(r,i) \leftarrow (r,j)] = (n-1)D + (i+j-1)D \quad (2)$$
$$= (n+i+j-2)D$$

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

2.2 Hardware Reconfiguration

If a processor needs to communicate with another pro-

cessor at a different row, it has to send a packet in a column phase. The *train loading/unloading* model is also useful in illustrating the principles involved in column communications. 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.

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

$$SwitchArvl[(r, n-k+1) \leftarrow (r, k)] = (2n-1)D$$

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 phase, *two or more processors at the same row can not 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 a packet into *car* (n - k + 1). We can determine the time for $p(i, j)$ to transmit its packet to be

$$ColSend[(i,j) \rightarrow (r,k)] = (n-k)D + (n-j)D + (2n-j-k)D$$

By separating the adjacent row buses by D, a column bus will look like a row bus that is turned 90^0 anti-clockwise after the packets are switched. 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

$$ColRec[(r,k) \leftarrow (i,j)] = (2n+i+r-2)D$$

2.3 Connectivity and Complexity

Software reconfiguration can be performed with little control overhead because each of the above Equations (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 × 2 switch and has only one transmitter. 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. Second, reconfiguration is flexible as one may *interleave* Row and Column phases in many ways to provide the communication bandwidth 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 of time during the same phase.

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 [2, 11] 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/Column phase. However, such a partition may not result in optimal number of communication phases 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 1-D *RASOB* and the 2-D *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*

Because of its fundamental importance, sorting is one of the most extensively studied computing problems. 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 $SQ = \{s_0, s_1, \ldots, s_{N-1}\}$ 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 $SQ' = \{s'_0, s'_1, \ldots, s'_{N-1}\}$ such that $s'_i < s'_{i+1}$ for $i = 0, 1, \ldots, N-2$. 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.

Many researchers have developed various parallel algorithms to speed-up sorting on different parallel computation models. In particular, fast state-of-the-art sorting algorithms were presented recently for various models of processor arrays with reconfigurable *electronic* buses [3]. Wang *et al.* proposed a constant time algorithms using $O(N^3)$ processors [12]. Using the *Columnsort* technique proposed by Leighton [10], Ben-Asher *et al.* proposed an $O(4^t)$ sorting algorithm using $O\left(N^{1+2\times(2/3)^t}\right)$ PEs, for $t \geq 2$ [1]; but Jang *et al.* proposed a constant time sorting algorithm using $O(N^2)$ processors [4]. Recently, Nigam and Sahni proposed two simpler constant time sorting algorithms using $O(N^2)$ processors [9]. 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 [5].

Although very fast, all these algorithms require an excessive number of processors to achieve that speed. However, when the size of data items to be sorted, N, is equal to the number of PEs, it was shown that the reconfigurable array of processors cannot sort in better than $O(N)$ time [9].

The sorting algorithms presented in this paper use the same number of processors as the number data items to be sorted. Yet there are almost as fast as the state-of-the-art

sorting algorithms which employ a much larger number of processors.

Our sorting algorithm on a 1-D *RASOB* 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 $SQ_S = \{s_{0_s}, ..., s_{l_s}\}$ and $SQ_L = \{s_{0_l}, ..., s_{m_l}\}$ 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 SQ_L may not be sorted yet. Next, we divide the data elements of SQ_S into two sub-groups such that each data element of the first sub-group is smaller than each data element of the second sub-group. We do the same thing for the data elements of SQ_L. We continue this division process until the size of each subgroup is equal to 1, in which case all the data elements will be sorted in the 1-D *RASOB*. Figure 3 further illustrates this division scheme.

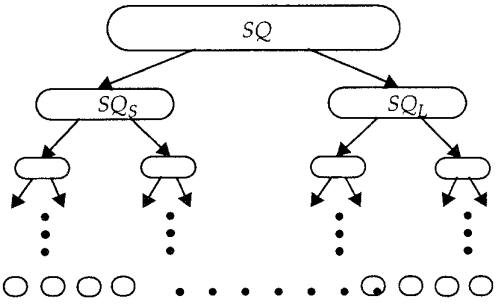

Figure 3. The general idea of the divide-and-conquer sorting algorithm on a 1-D RASOB.

Given N data items, where each data item is represented by k binary bits, and are distributed one data item per processor in a 1-D *RASOB*. Initially, each processor holds a single data item and two variables namely $START = 1$ and $END = N$. The variable *START* represents the index of the left-most processor within a group. Thus, at the beginning of the sorting algorithm, all processors belong to a single group, and the index of the left-most processor is 1 (i.e., $START = 1$). Similarly, the variable *END* represents the index of the right-most processor within a group. Consequently, at the beginning of the sorting algorithm, the variable *END* is equal to N which the index of the right-most processor of the group. During the sorting algorithm, each processor must maintain and update 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 $START_i$ and 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 corresponds to a bit position of the data elements to be sorted, of the following 8 steps:

Procedure SORT

During iteration I:

1. Each processor $p(i)$ broadcasts the data item it holds to $p(START_i)$, $p(START_i + 1)$, ..., $p(END_i)$, including itself, if the Ith 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 $(i - START_i + 1)th$ data item. If that is the case, then it marks it 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 $(i - START_i + 1)$ represents the position of $p(i)$ in its subgroup, starting from left to right. Processor $p(i)$ gets the $(i - START_i + 1)th$ data item in order to let the processors on its left get data items of smaller values since the Ith significant bit is 0. Hence, the processors on its left side can form their own subgroup later.
3. This analogous to Step 1 above. Each processor $p(i)$ broadcasts the data item it is holding to $p(START_i)$, $p(START_i + 1)$, ..., $p(END_i)$, including itself, if the Ith 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.
4. Each processor $p(i)$ checks its receiving buffer if it contains an $(END_i - i + 1)th$ data item. If that is the case, then it marks it 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 $(END_i - i + 1)$ represents the position of $p(i)$ in its subgroup, starting from right to left. Processor $p(i)$ gets the $(END_i - i + 1)th$ data item in order to let the processors on its right get data items of larger values since the Ith significant bit is 1. Hence, 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 $i - 1 \ START_i$ and sends a message to $p(i + 1)$ only if $i + 1 \leq END_i$ 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 $p(START_i)$ has no processor on its immediate left belonging to the same group and $p(END_i)$ 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 $i+1 \leq END_i$ and $p(i+1)$ has performed a *type II replacement*, then $p(i)$ sends a message to $p(START_i)$, $p(START_i + 1)$, ..., $p(i)$ so that they 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 $i - 1 \ START_i$ and $p(i-1)$ has performed a *type I replacement*, then $p(i)$ sends a message to

$p(i), p(i+1), ..., p(END_i)$ informing them to change their variable *START* to be equal to i.

End {Procedure SORT}

We can note here that Step 7 and Step 8 are used to divide a group of data items into two sub-groups such that the value of each data element of the first subgroup is smaller than the value of each data element in the second sub-group. Further, within one whole iteration, say iteration x, of the above algorithm, each processor can perform just one type or *replacement* (i.e., *type I* or *type II*) since its xth most significant bit can be either 0 or 1 but not both at the same time.

Theorem 1: The *SORT procedure* can be computed in $O(k)$ time on a 1-D *RASOB* where k is the size of the data elements to be sorted in bits.

Proof: Step 1 and Step 3 of the *SORT procedure* are simple broadcast operations on a 1-D *RASOB* where the corresponding processors load their respective data items into the appropriate *car* of the transmission *train* of slots. each of these *cars* (slots) will be read by all processors in a 1-D *RASOB* in a single row communication phase. Consequently, Step 1 and Step 3 of the *procedure SORT* take $O(1)$ time. Step 2, Step 4 and Step 5 of the *procedure SORT* obviously 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 communications phase. Finally, Step 7 and Step 8 of the *procedure SORT* involve a broadcasting operation which also takes a single row communication phase. Hence, each iteration of the *procedure SORT* takes $O(1)$ time. Consequently, the whole sorting algorithm on a 1-D *RASOB* take $O(k)$ time since the number of iterations of the *procedure SORT* is k where k is the size of the data items to be sorted in binary bits. ■

We have to note here that under most practical situations k rarely exceeds 20. As a result, the time complexity of this sorting algorithm on a 1-D *RASOB* is *almost* constant.

One drawback of the above *SORT procedure* is that it requires each processor of the 1-D *RASOB* to have a receiving buffer of size N, where N is the size of the data items to be sorted. Fortunately, the above drawback can be solved by simply installing a counter at each processor and requiring the size of the receiving buffers be equal to just 1. During each iteration of the *SORT procedure*, each processor is required to replace the data item it is holding either by the $(i - START_i + 1)th$ data item it receives in its buffer in Step 2, or by the $(END_i - i + 1)th$ data item it receives in its buffer in Step 4. Hence, we can use these counters to keep track of the data item that the appropriate processor is interested in. First, we have to set the counter of each processor to 0 before the execution of Step 2 and before the execution of Step 4 of the *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 $i - START_i + 1$ if Step 2 is being executed or the counter is equal to $END_i - i + 1$ is Step 4 is being executed. Afterwards, each processor $p(i)$ stops accepting any more messages for the whole duration of either Step 2 or Step 4. Consequently, right after Step 2 or Step 4 of the *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 2-D *RASOB*

The *procedure SORT* on the 1-D *RASOB* can be easily extended to a 2-D *RASOB* while retaining the same time complexity, $O(k)$. One way to extend the *procedure SORT* is to use Leighton's *Columnsort* algorithm [9] or to use Maberg and Gafni *Rotatesort* algorithm [7]. In our implementation of the *procedure SORT* on the 2-D *RASOB*, we use the *Rotatesort* algorithm.

The *Rotatesort* algorithm is a row-column sorting technique proposed originally for the 2-Dl mesh [7]. As this paper is not particularly concerned with 2-D mesh, the main interest is in *Rotatesort* as it applies to a 2-D 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 on a 2-D *RASOB* while using just the 1-D *RASOB SORT procedure*.

Given $N = RS$ data elements arranged as a 2-D $R \times S$ array, the *Rotatesort* technique [7] sorts the N data elements by alternately transforming the rows and columns of the array. The number of row and column phases is constant (14 or 16 phases). 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 Figure 4. A *vertical strip* (respectively, *horizontal strip*) is an $R \times S^{1/2}$ (respectively, $S^{1/2} \times S$) subarray of data elements. Also, a *block* is a $S^{1/2} \times S$ subarray of data elements. The algorithm as presented in [7] assumes that $R = 2^r$ and $S = 2^s$, where s is an even integer and $r \geq s/2$. However, other values of R and S can be used with little modification. The algorithm description can be facilitated by defining the following macros:

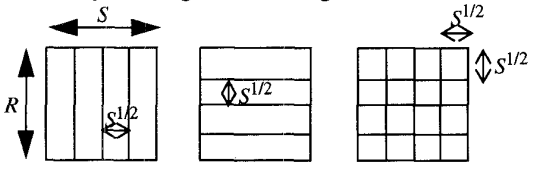

Figure 4. Partitions of the $R \times S$ array.

- Macro BALANCE: applies to a subarray of size $u \times v$ and consists of the following 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 the following 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. It consists of the following two steps:
  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 follows:

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 phases and nine row phases, for a total of 17 phases. However, since Step 3c) and Step 4a) both involve row transformations, the two steps can be combined into one row transformations, thus reducing the total number of phases to 16. Further, when $R \leq S^{1/2}$, then the number of phases can be reduced to 14.

Now, let us examine the time complexity of this algorithm when implemented on a 2-D *RASOB*. The BALANCE macro consists of two sorting steps along the columns of the 2-D *RASOB*. That is, we need to perform two sorting steps on a 1-D *RASOB* using the *procedure SORT* shown the previous Section. Thus, the two sorting steps of the BALANCE macro can be performed on a 2-D *RASOB* in $O(k)$ time. 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 phase on the 2-D *RASOB* using the *loading/unloading train* model. Hence, the BALANCE macro can be executed on a 2-D *RASOB* in $O(1)$ time. The UNBLOCK macro consists of one sorting step along the columns which takes $O(k)$ time as shown above, and one rotation step of all the rows rightward which takes a single row communication phase. Therefore, the UNBLOCK macro can be executed on a 2-D *RASOB* in $O(k)$ time. Finally the SHEAR macro which consists of 2 sorting steps along the rows and along the columns also can also be executed on a 2-D *RASOB* in $O(k)$ time using the *procedure SORT* of the 1-D *RASOB*. Thus, the whole *Rotatesort Procedure* can be executed on the 2-D *RASOB* in $O(k)$ time where k is the size of data elements to be sorted in bits. As mentioned previously, under most practical situations k rarely exceeds 20 which render the complexity of our sorting algorithm on a 2-D *RASOB* almost as efficient as constant time sorting.

Theorem 2: Sorting N data items can be computed in $O(k)$ time on a 2-D *RASOB* of size N where k is the size of the data items to be sorted in bits.

4 Conclusion

The *RASOB* architecture has recently received a lot of attention from the research community. 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 1-D *RASOB*. It which is based on a divide-and-conquer approach, has a time complexity of $O(k)$ where k is the size of the data elements to be sorted in bits. The second algorithm has been implemented on a 2-D *RASOB*. It uses the sorting algorithms implemented for the 1-D *RASOB* in conjunction with the well known *Rotatesort* algorithm to achieve an $O(k)$ time complexity. Both of these algorithms are more efficient that state-of-the-art sorting algorithms on various models of array of processors with reconfigurable electronic buses. Hence, the *RASOB* seem to be a very promising architecture for future massively parallel computing.

References

[1] Y. Ben-Asher, D. Pelg, R. Ramaswami, and A. Shuster, "The power of reconfiguration," *Journal of Parallel and Distributed Computing*, pp. 139-153, 1991.

[2] Z. Guo, "Optically interconnected processor arrays with switching capability," *Journal of Parallel and Distributed Computing*, pp. 314-329, 1994.

[3] 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.

[4] J. Jang and V. K. Prasanna, "An optimal sorting algorithm on a reconfigurable mesh," in *Proc. Int. Parallel Processing Symp.*, pp. 130-137, 1992.

[5] T. W. Kao, S. J. Horng, Y. L. Wang, and H. R. Tasi, "Designing efficient parallel algorithms on CRAP," *IEEE Trans. Parallel and Distributed Systems*, pp. 554-560, 1995.

[6] P. Lalwaney, 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.

[7] J. M. Maberg and E. Gafni, "Sorting in constant number of row and column phases on a mesh," *Algorithmica*, pp. 561-572, 1988.

[8] R. Melhem, D. Chiarulli, and S. Levitan, "Space multiplexing of waveguides in optical multiprocessor systems," *The Computer Journal*, 32 (4): 362-369, 1989.

[9] M. Nigam and S. Sahni, "Sorting n numbers on $n \times n$ reconfigurable meshes with buses," *Journal of Parallel and Distributed Computing*, pp. 37-48, 1994.

[10] S. Pavel and S. G. Akl, "On the power of arrays with reconfigurable optical buses," *Technical Report 95-374*, Department of CIS, Queen's University, 1995.

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

[12] 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*, pp. 187-192, 1990.