VLSI Design, 2002 VOL. 15 (1), pp. 441–454

Taylor & Francis
Taylor & Francis Group

Basic Algorithms for the Asynchronous Reconfigurable Mesh

YOSI BEN-ASHER* and ESTI STEIN

Computer Science Department, Haifa University, Haifa 31905, Israel

(Received 31 January 2001; Revised 27 June 2001)

Many constant time algorithms for various problems have been developed for the reconfigurable mesh (RM) in the past decade. All these algorithms are designed to work with synchronous execution, with no regard for the fact that large size RMs will probably be asynchronous. A similar observation about the PRAM model motivated many researchers to develop algorithms and complexity measures for the asynchronous PRAM (APRAM). In this work, we show how to define the asynchronous reconfigurable mesh (ARM) and how to measure the complexity of asynchronous algorithms executed on it. We show that connecting all processors in a row of an $n \times n$ ARM (the analog of barrier synchronization in the APRAM model) can be solved with complexity $\Theta(n \log n)$. Intuitively, this is average work time for solving such a problem. Next, we describe general a technique for simulating T-step *synchronous* RM algorithms on the ARM with complexity of $\Theta(T \cdot n^2 \log n)$. Finally, we consider the simulation of the classical synchronous algorithm for counting the number of non-zero bits in an n bits vector using $(k < n) \times n$ RM. By carefully optimizing the synchronization to the specific synchronous algorithm being simulated, one can (at least in the case of counting) improve upon the general simulation.

Keywords: Reconfigurable mesh; Asynchronous; APRAM; Asynchronous reconfigurable algorithm

INTRODUCTION

One of the most interesting models of parallel computation is the reconfigurable mesh (RM). The RM consists of a mesh augmented by the addition of a dynamic bus system whose configuration changes in response to computational and communication needs. More precisely, a RM of size $n \times m$ consists of nm identical SIMD processors positioned on a rectangular array with n rows and m columns. It is assumed that every processor knows its own coordinates within the mesh: we let $p_{i,j}$ denote the processor in row i and column j. Each processor $p_{i,j}$ is connected to its four neighbors $p_{i-1,j}$, $p_{i+1,j}$, $p_{i,j-1}$ and $p_{i,j+1}$, provided they exist, and has four ports denoted by N, S, E and W. Local connections within the processor between these ports can be established, under program control, creating a powerful bus system that changes dynamically to accommodate various computational needs. This yields a variety of possible bus topologies for the mesh, where each connected component is viewed as a single bus.

The ability to broadcast in a single time unit on such a bus motivated many researchers to develop constant time (or sub logarithmic) algorithms which outperform their PRAM counterparts. Such algorithms include transitive closure [16], convex hull and other geometrical problems [3,14,17] string matching [4], and many of the constant time sorting algorithms [10]. However, all these algorithms have been developed for the synchronous RM where, in each step, all the processors of the mesh simultaneously reconfigure, broadcast and read incoming data. A similar model is the synchronous PRAM (PRAM). A PRAM consists of a collection of n sequential processors, each with its own private local memory, communicating with one another through a shared global memory. Each processor has its own local program, and all n processors execute synchronously. In practice, it is unlikely to synchronize large amount of processors. This fact motivated several attempts to define an asynchronous PRAM (APRAM) model and develop asynchronous algorithms. The processors of an APRAM run asynchronously, i.e. each processor executes its instructions independently of the timing of the other processors. There is no global clock.

We first describe how asynchronous execution is handled in the APRAM model. An important problem in studying APRAM, is to define a complexity measure for asynchronous algorithms. In general, asynchronous execution assumes that every possible execution order is allowed. This means that in every "step" an adversary

*Corresponding author. E-mail: yosi@cs.haifa.ac.il

ISSN 1065-514X print/ISSN 1563-5171 online © 2002 Taylor & Francis Ltd
DOI: 10.1080/106551402100002057

chooses a subset of processors that are allowed to proceed and execute the next instruction in the underlying algorithm. Intuitively, we can say that the difficulty in obtaining such a complexity measure is to overcome the adversary's ability to delay some processors for arbitrarily long periods that need not be counted by the complexity measure. What we would really like to measure is the ability of the adversary to force the algorithm to "waste" steps by making a certain choice with regard to the execution order. Several suggestions have been made as to how to analyze asynchronous execution, particularly for the PRAM model, which is the relevant model for this work. Gibbons [9] introduced a model in which the computation is divided into phases which are separated by barrier synchronization. The length of a phase is the length of the longest computation in it. The APRAM of Cole and Zajicek [5] does not require any synchronization, but the algorithm is restricted to working in "rounds" wherein each processor must be advanced by at least one step. The execution time is the sum of maximal steps per processor in every round. The power of the adversary to choose the worst execution order was later relaxed in [6,13]. This was achieved by assigning a probability distribution on the set of all possible execution orders and setting the execution time to be the average execution time of every execution order (according to the probability distribution). We refer to this model as Nishimura's model and use it to measure the complexity of our Asynchronous Reconfigurable Algorithms (ARA). We remark that PRAM (and not message passing) is the most relevant model for RMs because the two models follow the similar steps. For example, except for the reconfiguration, PRAM's read/write operations to a shared cell are equivalent to the read/broadcast operations of the RM.

The application of Nishimura's model to RMs is not straightforward. In order to develop correct algorithms on an asynchronous RM (ARAs), some of the unique features which are not present in the APRAM had to be contended with:

1. Waiting for a message that does not arriving because the processor that is about to send it is delayed by the adversary.
2. A desired bus configuration is delayed because some of the processors that should create the desired bus are delayed.
3. A desired bus configuration can be "destroyed" when the adversary allows some of the processors participating in that bus to advance before the reconfiguration of the whole bus is completed.
4. A broadcast is "run over" by another message before it had the chance to be read.
5. A processor may receive a "wrong" message either because the bus configuration is not ready yet or was changed before the "right" message had the chance to arrive.
6. The initial configuration of the mesh when the power is turned on is not known. Hence, an ARA must be able to synchronize the processor without assuming anything about the initial configuration.

For example, consider the problem of establishing a straight bus segment (called a "line") connecting all the processors in one row of an $n \times n$ RM. Assume that some processor was able to read a synchronizing message sent from one end of the row to the other end and back. Clearly, even after such a message has been read, this processor can never be sure that a straight bus segment has been generated. This is because such a message could travel through a "non-straight" bus using other rows of the mesh. It follows that developing correct ARAs might be more difficult than developing correct APRAM algorithms. This difficulty is due to the need to maintain a desired bus configuration until all the processors have read the messages broadcast on that bus. So, in order to develop correct ARAs, it seems right to start with a basic set of problems such as connecting all processors in a row and then continue to more complex problems such as counting or sorting. The inherent difficulties in developing ARAs might partially explain why this subject has yet to be addressed by the RM community.

The asynchronous RM is defined in second section. A set of ARAs for solving basic barrier-like problems is described in third section, along with their asynchronous complexity, using Nishimura's model. A general asynchronous simulation of synchronous reconfigurable algorithms is given in Theorem 3.5. Finally, the complexity of the classical counting algorithms is studied in fourth section.

THE ASYNCHRONOUS RECONFIGURABLE MESH: TERMS AND DEFINITIONS

There are now quite a few RM models. These include the RN model of [2], the LRN of [1] the PARBS [16], the RMBM model [15] and the Rmesh [11]. These models differ in terms of word size, various restrictions on the topology of the mesh, how conflicting broadcasts on the same bus are handled, and in particular, the set of allowed configurations a processor can use. However, none of these parameters are relevant to the problem of asynchronous execution. Consequently, the following definition of the asynchronous RM reflects the common notion of the model and is not concerned with the above parameters. On the other hand, some of the finer details characterizing ARA will be addressed, making the proposed definition significantly different from standard.

An asynchronous RM (ARM) is a mesh of $n \times n$ processing units $p_{i,j}$, where each $p_{i,j}$ is connected to its four neighbors in the mesh through four edges denoted by their endpoints: East $e_{i,j}$, North $n_{i,j}$, West $w_{i,j}$ and South $s_{i,j}$. Thus, the edge starting with $e_{i,j}$ connects $p_{i,j}$ to $p_{i,j-1 \bmod n}$ and $n_{i,j}$ connects $p_{i,j}$ to $p_{i+1 \bmod n,j}$. A processor $p_{i,j}$ can "reconfigure" by joining the endpoints of its edges. The joining of two endpoints x, y is denoted by

$\langle x - y, \ldots \rangle$. For example, $\langle e{-}w{-}n,\ s \rangle$ indicates that a message coming from $e_{i,j}$ will continue to both $w_{i,j}$ and $n_{i,j}$. A write operation can be executed only from an endpoint (connected or non-connected), and the message will be broadcasted to all the processors that are currently connected to this endpoint. We assume a broadcasting time of zero. This means that the write is immediately broadcast to all the relevant processors. A read operation is applied to an endpoint (either connected or disconnected) and should return the last value broadcasted through that endpoint.

Each processor executes a sequence of atomic steps according to a program or an algorithm whose syntax will be defined next. As in regular RM, each step contains four operations: (1) a read of incoming messages, (2) a constant time local computation, (3) reconfiguration, and (4) write operations on some (or possibly none) of the endpoints. A processor halts when the program being executed by it reaches the last step. The following definition of the execution of a given ARA is similar but not identical to that of Nishimura [13].

DEFINITION 2.1 In an asynchronous execution of an RM:

1. there is uniform probability that one of the potentially active processors will be selected and allowed to execute the next step in its program.
2. The number of times a processor is selected is measured by means of "balls" that accumulate in a "jar" attached to that processor.
3. The complexity of a given ARA is the average number of balls that have accumulated in all jars of the potentially active processors when all the processors halt.

This is in fact the well-known "coupon collector" [7] process. The rationale for taking the average total number of balls as opposed to the average maximal number of balls per jar is to conceal the relatively high probability that a few processors will accumulate a significant larger number of balls then the average. A "bad" ARA is an algorithm in which many balls are accumulated by a majority of the processors, and not by just a few.[2]

In addition to defining complexity, we also need to define correctness:

DEFINITION 2.2 An execution of a given ARA is "fair" if there is a finite number $f(n)$ such that every processor is activated at least once every consecutive $f(n)$ steps. A given ARA is correct if for every input and every possible fair execution:

- All the processors reach a halting state.
- When a processor halts, the final configuration of all the processors in the RM should satisfy a pre-designated condition. ("There is a bus segment connecting all the processors in row i which does not pass through another processor in the mesh," is an example of such a condition).
- When a processor halts, a pre-designated set of processors should have the correct output.

It is now possible to determine an upper bound to the execution time of a given ARA:

THEOREM 2.1 Let A be an ARA that terminates after T *synchronous* steps, i.e. an asynchronous execution (see Definition 2.1) where a processor cannot receive it's t'th ball before the rest of the processors have received t balls. We say that A is "run over" if there are two possible executions O_A, O'_A with the same first t steps, such that a message m broadcast on the t'th step satisfies that:

- m is read by processor $p_{i,j}$ for the first time at a later step $k > t$ in O_A.
- (The first message that $p_{i,j}$ reads in O'_A after step t is not m.

If there are no such two possible executions, we say that A is "run over free". If A is run-over free, then the asynchronous complexity of A is $O(T{\cdot}n^2 \log n)$.

Proof The proof is based on the well-known fact [7] that when balls are thrown at random to n jars, an average of $n \log n$ balls must be thrown in order that each jar will contain at least one ball. We can divide each possible execution O_A into sub-sequences of steps $O_A = E_i; E_2; \ldots$ such that E_i contains the next sequence of steps (after E_{i-1}) wherein each processor advances by at least one step (received one ball). Accordingly, the average size of each E_i over all possible O_A is $O(n^2 \log n)$. However, since A is run-over free, then in each E_i every processor reads the messages broadcast in $E_{j<i}$. This is because none of these messages can be run over in E_i, nor in the previous $E_{j<i}$s. Every asynchronous execution O_A has a matching synchronous execution O^s_A with the same set of messages. Assuming by induction that the state (local memory, next instruction and configuration) of every processor after $E_i \in O_A$ is equivalent to the state of equivalent processor after step t in O^s_A, then:

- All the messages broadcasted at step $t + 1$ will be broadcasted at E_{i+1}.
- Each message broadcasted in E_{i+1} is also broadcasted at step $t + 1$ in O^s_A.
- All the messages read in step $t + 1$ of O^s_A will be read either in E_{i+1} or in E_{i+2} of O_A. This follows from the fact that no message can be run over by another message in O^s_A and that every processor performs at least one step in E_{i+1}/E_{i+2}. Note that we need two "rounds" to read all the messages. This is because O_A could have scheduled the read operation of some processor before the matching write operation. Thus, two rounds (E_{i+1}/E_{i+2}) are required to guarantee that all the messages broadcast at step t have been read.

The base of the induction for $t = 1$ and $i = 1$ is immediate. Consequently, every O_A must terminate after

$E_1,\ldots,E_{2t}$ rounds, and the average number of balls must be $O(T{\cdot}n^2 \log n)$. ■

We remark that the opposite claim is not necessarily true. A lower bound of $\omega(T)$ on a synchronous execution of a run-over free ARA does not imply $\omega(T{\cdot}n^2 \log n)$ on the asynchronous complexity. For example, consider the problem of determining the ID of a processor that holds a non-zero input bit. Assume that there are at least $(3/4)n^2$ non-zero bits in every possible input and that initially all the endpoints are connected ($\langle e{-}w{-}s{-}e\rangle$). The ID is computed when there is at least one processor that knows the ID of another processor with non-zero input bit. The underlying ARA works by letting each processor with a non-zero input bit to broadcast its ID and continue reading until a non-zero ID (other than its own) has been detected. On the average, after a constant number of balls, a non-zero processor will broadcast its ID, and another processor will read it. Thus, the asynchronous complexity is $O(1)$ and not $\Omega n \log n$, as would be required if the opposite claim to Theorem 2.1 were true.

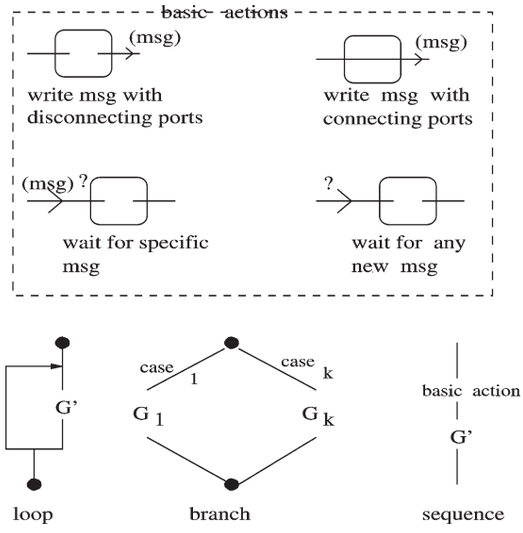

FIGURE 1 Rules for constructing the control flow graph of an ARA.

CONNECTING AND DISCONNECTING IN A ROW

Here we address the problem of connecting all processors in one row of an $n \times n$ ARM, that is, creating one horizontal bus on that row. The initial configuration is not known. Hence, messages broadcast on that row do not necessarily reach their destination via that row, but can reach it via the outer rows of the mesh. This is the first step in establishing ARAs, as creating a bus along one row of the mesh establishes a communication line and a known configuration on which future steps of the algorithm can rely. Creating such a bus is actually the reconfigurable analog of a barrier synchronization [8]. As such its importance in the development of ARAs is evident. Indeed, we show how to apply this ability to connect in a row to a general asynchronous simulation of regular synchronous RM algorithms on the ARM. We also consider the complementary problem of asynchronously disconnecting all the processors in a given row and show that this is a much more difficult task. Finally, the syntax of ARA is defined in Fig. 1, which shows the control flow graph of the program executed by every $p_{i,j}$, where each node represents the next step to be executed.

DEFINITION 3.1 An ARA solves the problem of connecting in a row if, regardless of the initial configuration of the ARM, the following hold:

- When a processor in a row halts, all the processors in that row are in state $\langle e{-}w, n, s\rangle$.
- All processors in the row will reach a halting state.
- None of the processors in the other rows of the mesh is used by the algorithm.

We describe three solutions to the problem of connecting in a row. These demonstrate:

- that optimal ARAs may require some careful planning.
- the use of Theorem 2.1 to compute asynchronous complexity.
- that Nishimura's model can distinguish between several ARAs according to their intuitively expected efficiency, since each algorithm improves upon its predecessor. This is not straightforward: since we measure the average work, one might suspect that efficient "sequential" solutions could turn out to be optimal ARAs.

THEOREM 3.1 Let $A^{c1}(r)$ be an algorithm where $p_{r,i}$ (for $i > 0$) first disconnects its edges $\langle e, w, s, n\rangle$, waits for a message from its left neighbor, transmits its ID to its right neighbor, connects its horizontal edges $\langle e, w, s, n\rangle$ and waits for a message from $P_{r,n-1}$ (as described in Fig. 2). Then $A^{c1}(r)$ solves the problem of connecting in a row,

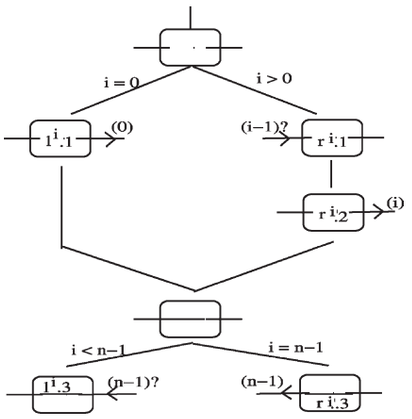

FIGURE 2 Linear connecting in a row.

and the asynchronous complexity of $A^{c1}(r)$ is $O(n^2 \log n)$, when an $n \times n$ ARM is used.

Proof The correctness of this algorithm is immediate. Synchronously, this algorithm takes $n+1$ steps, since processor i waits for processor $i-1$. Obviously no message is run over. The next message is broadcast after the last message has been received, and thus, by Theorem 2.1 the asynchronous complexity is $O(n^2 \log n)$ where the number of potentially active processors is only n. ■

THEOREM 3.2 Let $A^{c2}(r)$ be an algorithm (formally described in Fig. 3), where at the t'th iteration, every two consecutive horizontal buses at row r are joined into subbuses of length 2^{t+1}. Then $A^{c2}(r)$ solves the problem of connecting in a row, and the asynchronous complexity of $A^{c2}(r)$ is $O(n \log^2 n)$.

Proof The proof is similar to that of Theorem 3.1. The fact that no message is run over is due to the following claims:

- A message sent by action $l^i{\cdot}1$ (see Fig. 3) cannot run over another message sent by action $l^j{\cdot}1$ of the same type. This is because the message sent by $l^i{\cdot}1$ is transmitted to the right, and $p_{r,i}$ disconnects its edges before the transmission takes place.
- A message sent by $l^i{\cdot}1$ cannot run over a message sent by $r^k{\cdot}3$ since no message is sent by $r^k{\cdot}3$, before $p_{r,k}$ has received the message from $l^i{\cdot}1$.
- A message sent by $r^k{\cdot}3$ cannot run over a message of the same type sent by $r^j{\cdot}3$ $(j < k)$ since there must be a receiving processor $p_{r,i}$ between $p_{r,k}$ and $p_{r,j}$ which blocks $r^k{\cdot}3$.

The correctness is immediate because when two buses are joined ($r^i{\cdot}2$), both ends of the resulting bus remain disconnected. ■

The intuition that connecting in a row can be done in $O(n)$ in a constant number of asynchronous steps is thus somewhat of a surprise.

THEOREM 3.3 Let $A^{c3}(r)$ be an algorithm (formally described in Fig. 4), where $p_{r,i}$: disconnects, transmits its ID to the left, waits for its right neighbor's ID, transmits its ID to the right, waits for its left neighbor's ID, and then waits for two messages from both ends $*i = 0$ and $i = n-1$. $p_{r,n-1}$ continues sending a confirmation message at $r^{n-1}{\cdot}3$ until it receives an acknowledgment from $p_{r,0}$. Then $A^{c3}(r)$ solves the problem of connecting in a row, and the asynchronous complexity of $A^{c3}(r)$ is $O(n \log n)$.

Proof The fact that no message in $A^{c3}(r)$ is broadcast before the broadcasting processor is in a disconnecting state implies that no message can be run over. Synchronously, $A^{c3}(r)$ terminates after a constant number of steps. The main difficulty is to show the correctness of the algorithm, which is based on the following argument.

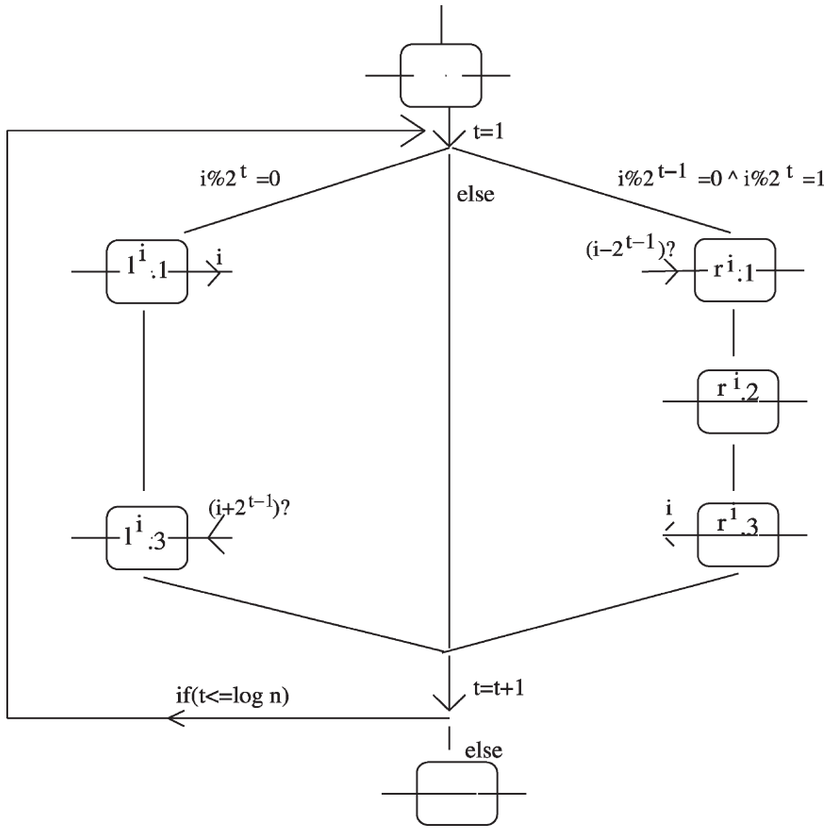

FIGURE 3 Tree like connecting in a row.

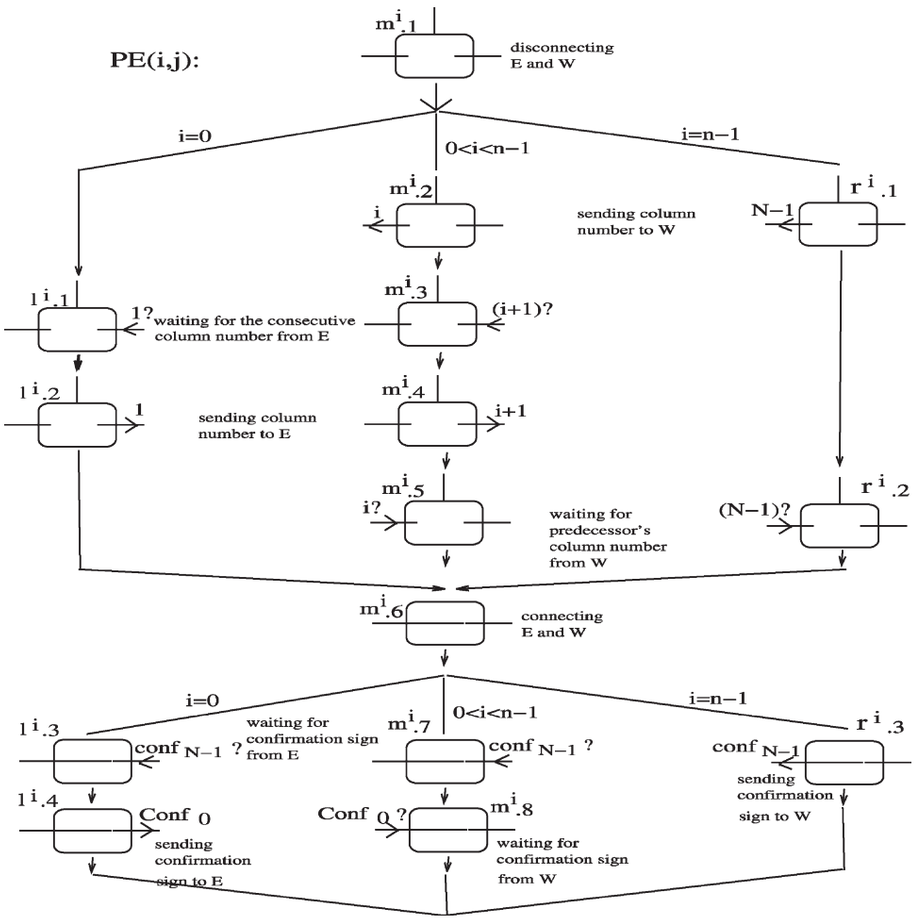

FIGURE 4 Constant time ARA for connecting in a row.

Assume (w.o.l.o.g) that $p_{r,i}$ does not execute $A^{c3}(r)$'s protocol, yet some neighbor $p_{r,i+1}$ starts to execute $A^{c3}(r)$. As a result, $p_{r,i+1}$ will not receive the message in action $m^{i+1}{\cdot}5$ and will remain in a disconnected state. Consequently, the broadcast of $r^{n-1}{\cdot}3$ cannot pass through $p_{r,i+1}$. A similar claim holds for the left neighbor of $p_{r,i}$ and for $i = 0; i = n-1$. Thus, if the message broadcast by $r^{n-1}{\cdot}3$ reachs $p_{r,0}$, then it must bypass $p_{r,i+1}$ (that is travel through others rows), using a "band" at some $p_{r,k>i+1}$ of the form $\langle e{-}n, \ldots\rangle$ or $\langle e{-}s, \ldots\rangle$. However, in this case, $p_{r,k}$ could not have started the algorithm and, by the same argument, $p_{r,k+1}$ must be in a disconnected state and $r^{n-1}{\cdot}3$ could not have bypassed $p_{r,i+1}$ by passing through $p_{r,k}$ instead. Consequently, the second phase of messages from the two ends cannot terminate while there are processors in a disconnected state before or after bypasses. Therefore, the only possibility for $r^{n-1}{\cdot}3$ and $l^0{\cdot}4$ to pass through some $p_{r,l}$ is if all the processors in the row are in a connected state. ■

We turn now to discuss the problem of disconnecting all the processors. We can use Definition 3.1, to define this problem, except that here all the processors in the row must be in a disconnected state when the algorithm terminates. Despite the similarity to the problem of connecting in a row, this problem cannot be solved efficiently.

THEOREM 3.4 The asynchronous complexity of disconnecting all the processors in a row r is $\theta(n^2 \log n)$.

Proof The upper bound follows, which is formally described in Fig. 5, is based on an algorithm similar to that of Theorem 3.1 Basically, each $p_{r,i}$ goes into a disconnected state, waits for a broadcast from $p_{r,i-1}$, broadcasts it ID to $p_{r,i+1}$, waits for a "confirm" broadcast from $p_{r,i+1}$, and sends a confirm broadcast to its right neighbor, $p_{r,i-1}$. Obviously, when the confirm broadcast is received by a processor, all other processors must be in a disconnected state. No message can be run over because a message is broadcast only after the preceding message has been read. The number of synchronous steps is $2n$. An asynchronous complexity of $O(n^2 \log n)$ is therefore implied.

To obtain the lower bound, we assume that all the processors in the row are initially connected $\langle e{-}w, n, s\rangle$. We concentrate on $p_{r,n-1}$, which must be informed by directly or indirectly that all other $n-1$ processors are in a disconnected state. The only way for some $p_{r,i<n-1}$ to inform $p_{r,i-1}$ that it and possibly all $p_{r,j<i}$ are in

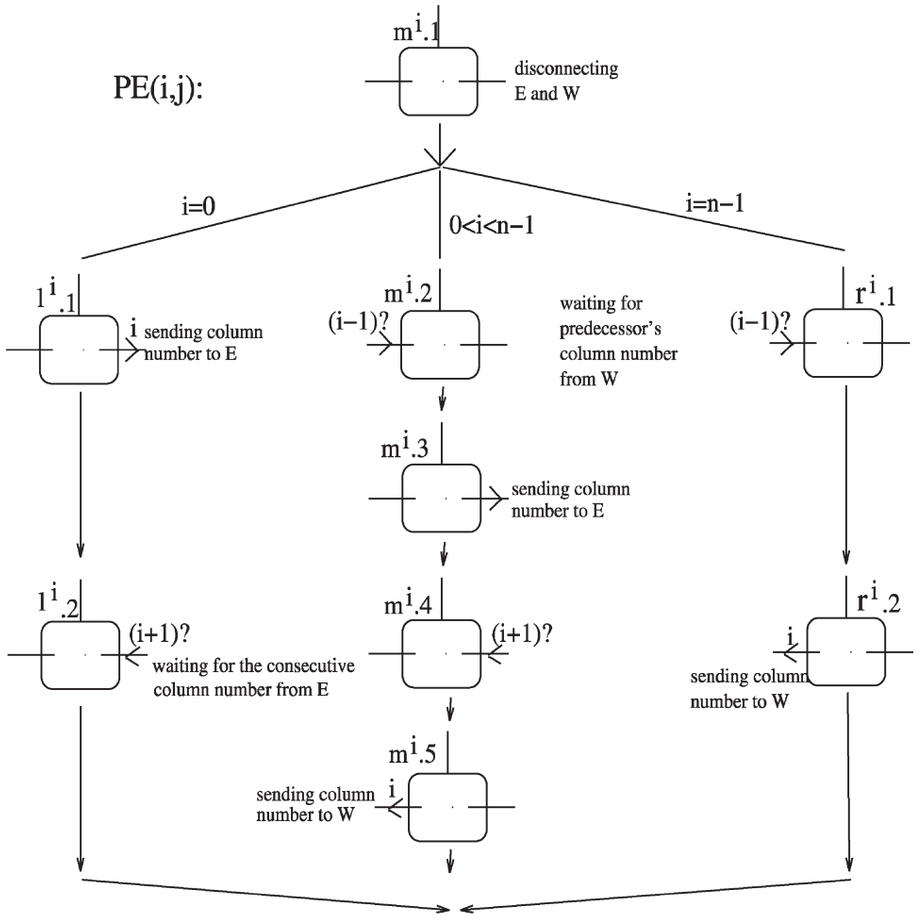

FIGURE 5 Direct disconnecting in a row.

a disconnected state is to use a bus from $p_{r,i}$ to $p_{r,n-1}$. The bus can be used because $p_{r,i}$ through $p_{r,n-1}$ are all connected. Moreover, $p_{r,0}$ through $p_{r,i}$ must be in a disconnected state: Otherwise, another bus connecting $p_{r,k<i}$ to $p_{r,n-1}$ must be created. This would, however, invalidate the information that $p_{r,i}$ is in a disconnected state. Therefore, the next processor to inform $p_{r,n-1}$ that it is in a disconnected state is $p_{r,i+1}$. This yields the desired lower bound. ■

The only way to disconnect all the processors in a row r faster than the lower bound is to use an extra row, dedicated to the algorithm, above the current row. First, each processor in row r disconnects and inform the upper neighbor in the auxiliary row to begin executing the algorithm for connecting in a row (Theorem 3.1–3.3). When all the processors in the auxiliary row are connected, then the processors in row r can be certain that they are all in a disconnected state, as described in Fig. 6.

We can now show how a regular synchronous RM algorithm can be executed on the ARM. In the proposed simulation, each processor of the original RM is simulated by a 3×3 ARM. This enables the processor to "signal" a neighboring 3×3 ARM without changing or affecting the state of the simulated mesh.

LEMMA 3.1 One step of a reconfigurable processor in a given RM can be simulated in $O(1)$ complexity by a 3×3 sub-ARM (called a "large processor") embedded in a larger ARM such that:

- the large processor (LP) simulates the new configuration of the simulated processor using four out of the 12 external edges connecting the LP to the rest of the processors in the larger ARM.
- No message is broadcast on the four edges of the LP used to simulate the original processor.
- At the end of the simulation, the LP uses one of the $12 - 4$ unused edges to signal a neighboring processor in the larger mesh that the simulation is over.

Proof We use the center processor $p_{1,1}$ (see Fig. 7) to simulate the configuration of the processor being simulated. The rest of the processors are called "external". Note that no matter what the configuration simulated by the sub mesh, the edges connecting an external processor to its external neighbors are always disconnected. Let a "round" denote a sequence of 8 asynchronous steps in which each external processor disconnects its edges

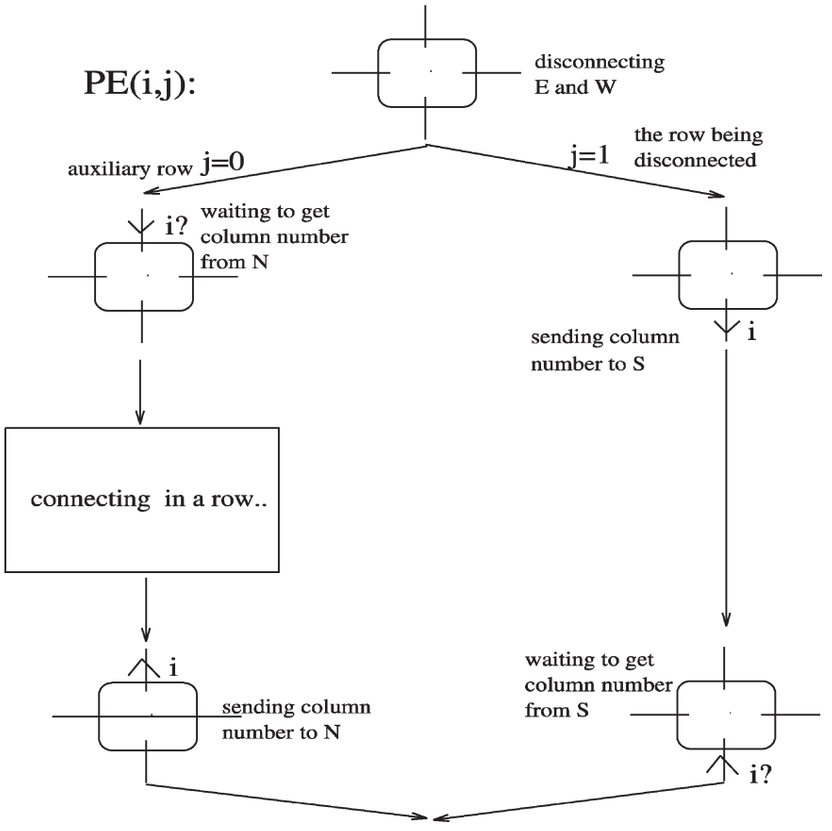

FIGURE 6 Using an auxiliary row to implement disconnecting in a row.

$\langle e, n, s, w \rangle$, reads the messages waiting on them, and signals its left neighbor to do the same (following the order given in Fig. 7). The round is over when $p^1_{2,0}$ gets a signal from $p^8_{2,1}$. Clearly, the messages sent during a round cannot affect the state of the large mesh. The simulation of the step is performed as follows:

1. A round is performed for disconnecting the LP from the large mesh.
2. A second round is performed wherein $p^4_{0,1}$ signals $p_{1,1}$ to execute the next step of the simulated processor. Let m be the message that $p_{1,1}$ is waiting to receive on it upper edge. $p^4_{0,1}$ signals $p_{1,1}$ by broadcasting m', a message containing the original m plus a signaling information. Note that since the LP is disconnected from the rest of the large mesh, m' cannot run over another message in the larger mesh.
3. The center processor $p_{1,1}$ waits for m'. It then changes its state (according to the simulated processor) and broadcasts a message m'' to all four directions. This message m'' contains the new broadcasts made by $p_{1,1}$ or the old information that $p_{0,0}$ read on its four edges.
4. A third round is performed where $p^4_{0,1}, p^6_{1,2}, p^8_{2,1}$ and $p^2_{1,0}$ wait to receive m'' before they continue the round. Upon receiving m'' and before continuing the round, each $p^4_{0,1}, p^6_{1,2}, p^8_{2,1}$ or $p^2_{1,0}$ restore from m'' the original message m and store it in its local memory.
5. A round is performed in which each $p^4_{0,1}, p^6_{1,2}, p^8_{2,1}$ connects $p_{1,1}$ to the rest of the larger mesh before it continues the round (restoring the bus configuration of the external processors as described in Fig. 7).
6. A last round is performed in which each external processor broadcasts the message that it stored in the third round on the suitable bus.
7. $p_{2,0}$ can now signal its lower neighbor in the large mesh that the simulation is over.

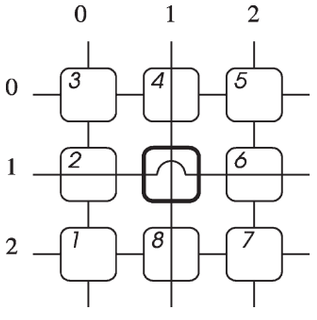

FIGURE 7 Asynchronous simulation of an original RM processor by a 3×3 ARM.

The proof of validity and complexity is immediate. ■

THEOREM 3.5 Let A be an algorithm which terminates after T steps on the synchronous $n \times n$ RM. Then A can be executed on $a3(n+1) \times 3(2{\cdot}n)$ ARM, with asynchronous complexity of $\Theta(T{\cdot}n^2 \log n)$.

Proof The simulation of A is carried out step by step. A barrier synchronization is performed between two steps. Thus, before a processor starts to execute step $t+1$ of A, it is guaranteed that all the other processors executing A have completed step t. Several technical details are omitted and we essentially describe only the major details needed for understanding and validating the algorithm. Each row of the original RM uses an auxiliary row to determine when all the processors in row r have completed the execution of the current step. This is done by executing the operation of connecting in a row on the auxiliary row. (This is similar to the use of its operation in the disconnecting operation illustrated in Fig. 6). A special dedicated column is used to determine (by applying the operation of connecting in a row to column 0 of the ARM) when all the auxiliary rows are connected. Thus, when column 0 is connected, it follows that all the processors of the original RM have completed one step of A. The formal description of the barrier synchronization step is given in Fig. 8 on the left, while the general example of the configuration of the simulating ARM is given on the right (the auxiliary rows and column are marked by thick lines).

The buses that are generated during the execution of A must pass through the processors of the auxiliary rows. This is why the state of all the processors in the auxiliary rows is $\langle e, w, n{-}s\rangle$ (see Fig. 8). Note that when connecting in a row is executed by the auxiliary rows, There is no need to change the $\langle n{-}s, \ldots\rangle$ connection. Thus, the buses of the simulated RM are not affected by the operation of connecting in a row and remain stable until it is guaranteed that all the processors of the original RM have completed executing the current step.

There is one problem with this method. How can a processor of the original mesh safely signal its upper neighbor in the auxiliary row without either running over unread messages or changing its configuration? This is solved using large processors as described in Lemma 3.1. Thus, the asynchronous complexity of the simulation is bounded by $O(T{\cdot}n^2 \log n)$.

Finally, it is easy to find a synchronous RM algorithm for which the above bound is also optimal. Consider the problem of computing the Boolean AND of an $n \times n$ RM where each processor holds one bit. Synchronously, this can be solved in two steps by connecting all the edges $\langle e{-}w{-}n{-}s\rangle$ and letting each processor with zero input broadcast 1 on the common bus. Next, each processor reads the value broadcast on the bus. If no such value has been broadcast, the Boolean AND is true. An average of $\Omega(n^2 \log n)$ balls is necessary to guarantee that every processor will execute at least one step [7,13]. This fact yields that the complexity of the general simulation is $\Theta(T{\cdot}n^2 \log n)$. ■

ASYNCHRONOUS COUNTING

In this section, we show how connecting in a row can be used to solve more complex problems whose synchronous solution over the RM requires a non-constant number of iterations. The goal is to optimize the synchronization such that the proposed solution will outperform the asynchronous simulation of Theorem 3.5. Let us consider the well known problem of counting the number ($S(n)$) of 1s in a binary vector of size n which is given as input to the lowest row of a $k \times n$ RM. The solution to this problem is a classical RM technique and is used as a basic block in many reconfigurable algorithms [12]. In order to simplify the description, we consider a variant of this problem wherein the maximal power of t such that $k^t \leq S$ is to be computed

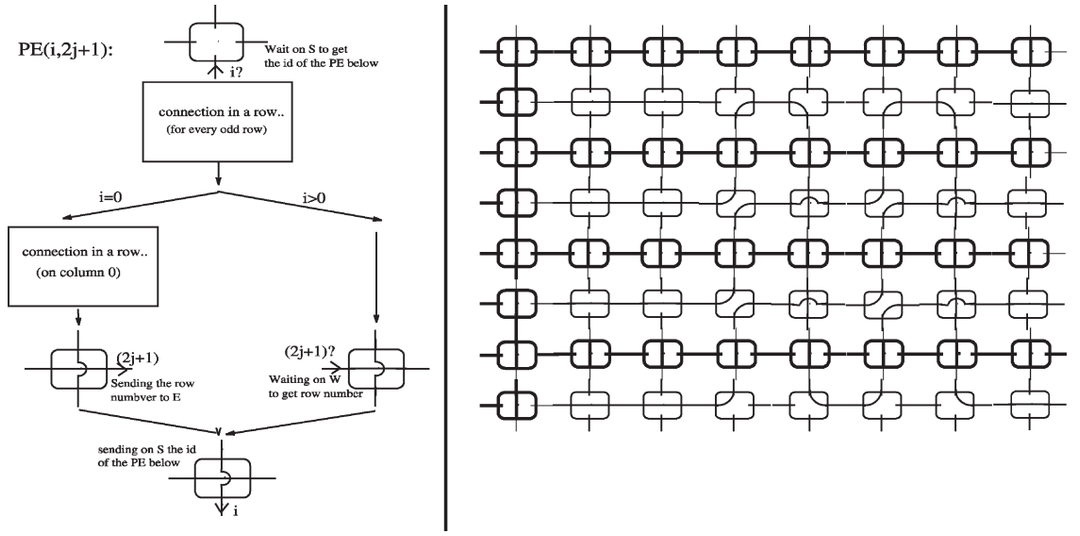

FIGURE 8 A barrier synchronization (left) and using auxiliary rows while maintaining original bus configuration (right).

(i.e. computing the value of $\lfloor \log_k S(n) \rfloor$). The synchronous solution uses repeated iterations such that at each iteration we filter every k consecutive 1s to a single representative. This representative is called the "leader". The filtering continues until the number of leaders become less than k.

In general, there are two types of columns (per input bit): "input columns" used for the actual counting operation, and the "modulo columns" used to redirect signals from the topmost row to the bottom row of the mesh. Initially, the input bits are broadcast along the input columns. A 0-bit input broadcast along an input column causes all the processors in the column to create horizontal connections $\langle e{-}w, n, s \rangle$ and then remain in this ("passive") state until the last iteration. A 1-bit input broadcast along an input column causes all the processors in the column to create "bands" $\langle e{-}n, w{-}s \rangle$. As a result, any broadcast on row i coming from the east will continue in row $i+1$ to the west. Initially, the set of leaders contains all the input columns with input value "1". There is an auxiliary row whose processors are disconnected if they correspond to a leader column, and connect if they do not correspond to such a column. Thus, in the auxiliary row, there is a horizontal bus between every two consecutive leaders. Following is our version of the Synchronous Counting Algorithm (SCA), also depicted in Fig. 9:

1. The counting signals $S_1 \ldots S_k$ of the current iteration are broadcast to the various rows.
2. Each counting signal will jump up one row every time it passes through an active/leader input column. Since we have only $k < n$ rows, each counting signal will be redirected to the bottom row every time it hits the upper row.
3. We can decide whether a processor $p_{\text{row,col}}$ that is in a leader column should become passive ($\langle e{-}w, n, s \rangle$) or remain a leader. His decision is based on the source row number of the incoming signal. In other words, if the signal was sent from the first row, and row $\neq k$, then $p_{\text{row,col}}$ should go into a passive state.
4. The lower processor in a leader column going into a passive state informs its representative in the auxiliary row to go into a connected state.
5. The lower processor in a remaining leader column informs its representative in the auxiliary row to broadcast its ID to the left.
6. The rightmost processor in the auxiliary row always broadcasts its ID to the left.
7. If the main processor (see Fig. 9) receives the ID of the last processor in the auxiliary row, then it is clear that there are no remaining leaders. Therefore, the algorithm terminates with the number of iterations as output. Otherwise, the main processor instructs its column's processors to start sending the counting signals of the next iteration.

Basically, the proposed ARA attempts to use the same steps as the synchronous Algorithm. The only difference is that the proposed ARA requires the synchronization of all the leaders at the end of every iteration. It would be possible to use Theorem 3.5 to transform the synchronous algorithm to an asynchronous one. However, since the number of leaders decreases in each iteration by a factor of $1/k$, we can design a special synchronization algorithm whose complexity depends only on the number of the leaders that remain at the beginning of every iteration. We will show that synchronizing only the remaining leaders is significantly better than synchronizing all the processors at each iteration.

The synchronization at the end of each iteration of the above SCA can be partitioned into two stages:

Vertical synchronization: verifying that all the processors in a leader column that are going into a passive state in fact moved to that state.

Horizontal synchronization: verifying that all the vertical synchronizations have been completed.

The solution to these problems is given in the following two lemmas.

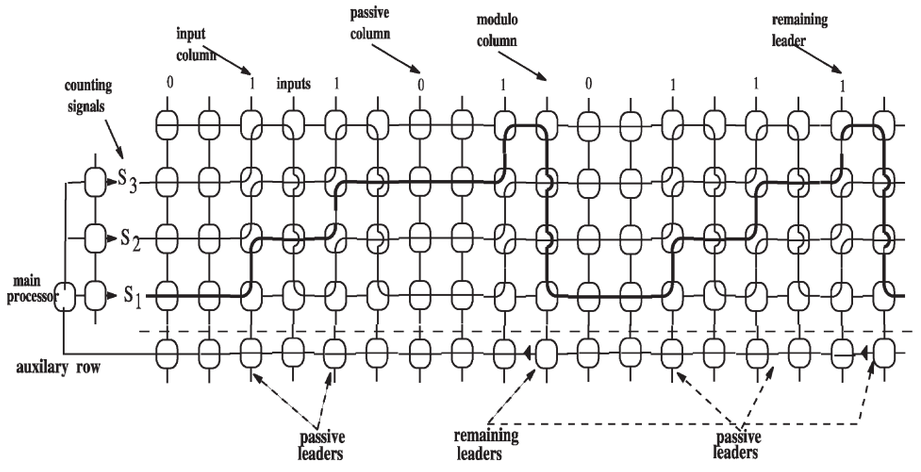

FIGURE 9 First iteration of the counting with $n = 9$ and $k = 3$.

LEMMA 4.1 Vertical synchronization in a leader column at the end of every iteration of the SCA can be done asynchronously with a complexity of $O(k \log k)$, using an auxiliary column of k processors.

Proof The main difficulty is to perform the synchronization after each processor changed its state to passive without running over the S_i's (counting signals) that may not have been read yet by other processors. The proposed solution is to rebroadcast the original counting signal S_i together with the messages needed for the new synchronization (denoted by S_i', S_i''). Thus, if S_i', runs over S_i, the original counting signal is not lost. Assume that we add an auxiliary column between every input column and its modulo column (see Fig. 10, initial state). The main stages of the algorithm are depicted in Fig. 10 as follows:

1. After receiving a counting signal (either S_i, S_i', S_i''), processor $p_{i,j}$ goes into a disconnected state. It then signals its right neighbor $p_{i,j+1}$ (in the auxiliary column) to start executing the instructions given in the next item by rebroadcasting S_i' to it. Finally, $p_{i,j}$ waits for an answer S_i'' from $p_{i,j+1}$.
2. After receiving S_i', $p_{i,j+1}$ connects its vertical ports $\langle e-w, n-s\rangle$ and performs connecting in a column. When the terminating signal ("go") of this operation is received, $p_{i,j+1}$ knows that all the processors in the input column are in a disconnected state and notifies $p_{i,j}$ of this by rebroadcasting S_i''.
3. When $p_{i,j}$ receives S_i'', it goes into the passive state $\langle e-w, n-s\rangle$ and performs connecting in a column (as described in Fig. 10). Once the final ("go") signal of this operation passes through, we know that all the processors in the leader column are now in a passive state.
4. Special care must be taken to synchronize the upper processor of the modulo column. This is necessary in order to guarantee that it has changed its state from $\langle w-s, e, n\rangle$ to $\langle e-w, n, s\rangle$. The synchronization can be carried out by means of additional messages of the form S_i'''. The technical details are straightforward and thus omitted.

The validity of the algorithm is due to the following claims:

- Since the algorithm is executed after S_i has been broadcast, there is no possibility that some S_i will reach a processor it is not supposed to reach (i.e. changes in state caused by the algorithm cannot direct S_i to undesirable locations).
- A processor that is supposed to read S_i in the synchronous algorithm will receive either S_i or S_i'/S_i''. Hence, no information is lost because of the algorithm.
- S_i', S_i'' are broadcast after $p_{i,j}$ disconnects its vertical bus. Thus, S_i', S_i'' cannot run over other S_k', S_k'' messages broadcast by $p_{i,l>j}$ processors.

The time complexity is dominated by the two operations of connecting in a column. Thus, the overall complexity of the algorithm is bounded by $O(k \log k)$. ■

The second synchronization problem is solved by the following algorithm:

LEMMA 4.2 Horizontal synchronization at the end of an iteration of the SCA can be performed with asynchronous complexity $O(m \log m)$ where m, is the number of leader columns in the current iteration.

Proof The algorithm uses three auxiliary rows (called T-row, C-row and B-row) to perform the horizontal synchronization. Initially we assume that all the leaders of the current iteration in the T-row are in a disconnected state $\langle e, w, n, s\rangle$, while all the other processors in the T-row are connected. The C-row is a copy of the T-row and

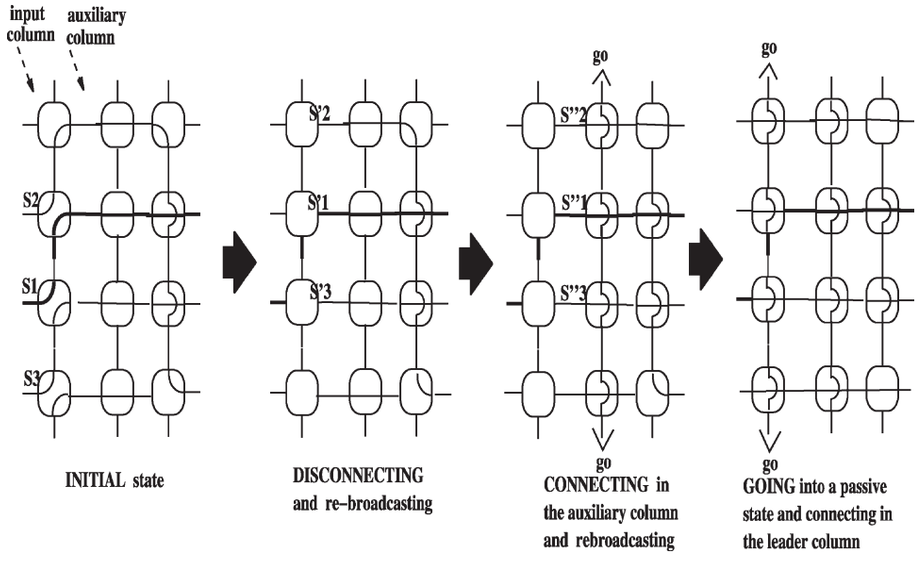

FIGURE 10 Verifying that all the processors in a leader column moved to a passive state.

the B-row is fully connected (Fig. 11). The main stages of this special variant of connecting in a row (depicted in Fig. 11) are as follows:

1. At the end of a vertical synchronization, a representative of the column in the T-row (called the passive leader) is activated. Upon activation, any passive leader in the T-row connects its buses $\langle e{-}w, n, s\rangle$, so that every group of k consecutive leaders form one connected segment in the T-row.
2. Next, each leader in the T-row performs the operation of connecting in a segment (i.e. connecting in a row restricted to the segment). Thus, every processor in the T-row knows that its new segment has been completed.
3. After the operation of connecting in a segment, each passive leader in the T-row signals its lower neighbor (going to state $\langle e{-}w, n{-}s\rangle$) in the C-row to connect its vertical buses.
4. Similarly, a remaining leader in the T-row signals its corresponding processor in the B-row to go into a disconnected state $\langle e, w, n{-}s\rangle$.
5. Each disconnected processor in the C-row (corresponding to a remaining leader) waits for a confirmation message indicating that the corresponding processor in the B-row is now disconnected.
6. The synchronization of all the leaders is realized by performing connection in a row at the C-row.
7. When the operation of connecting in a row completes in the C-row, the B-row has a copy of the remaining leaders of the T-row and the processors in the C-row are fully connected. Thus, for the next iteration we can use the same algorithm, with the B-row and C-row switching roles.

The validity of this algorithm is immediate as the algorithm performs connecting in a row of all the leaders/trepresentatives of columns that are going into a passive state. For every iteration of the SCA, this algorithm:

- may perform $\frac{m}{k}$ separate operations of connecting in a segment, where each segment may contain up to k representatives of leaders going into a passive state and m is the total number of representatives in the T-row. According to Nishimura's model, the complexity of these operations is bounded by $O(m \log m)$ balls.
- may copy the state of the representatives from the T-row to the B-row and the C-row, which are also bounded by $O(m \log m)$.
- may perform the final operation of connecting the remaining leaders in the B-row, which is also bounded by $O(m \log m)$.

Thus the complexity of horizontal synchronization is bounded by $O(m \log m)$. ■

It is now possible to state the main result:

THEOREM 4.1 The maximal power of "1" bits (out of n bits) can be computed in asynchronous complexity of $O[\max(n{\cdot}k \log k, n{\cdot}\log n)]$, using an $(k+3) \times 3{\cdot}n$ ARM.

Proof As explained previously, we use the SCA algorithm and synchronize at the end of every iteration.

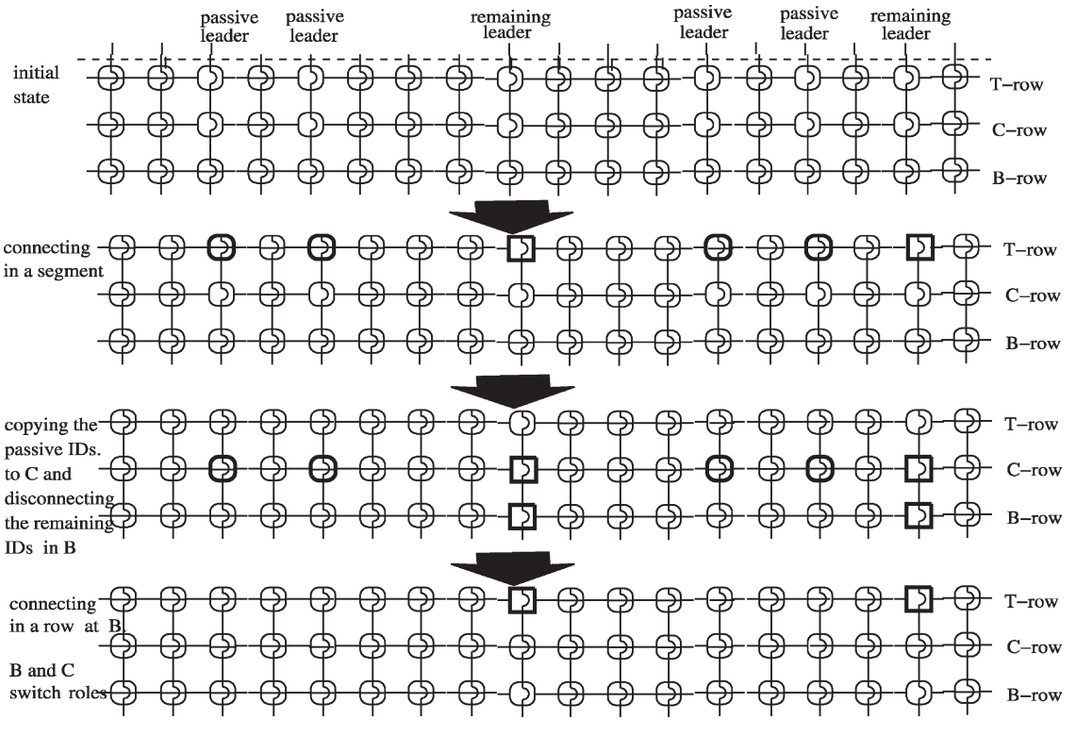

FIGURE 11 Efficient synchronization of leaders.

This is done by performing vertical synchronization (Lemma 4.1) in every active column that is going into a passive state, and horizontal synchronization (Lemma 4.2) to verify that all the columns that are supposed to go into a passive state have indeed done so. Thus, after the horizontal synchronization, the main processor (Fig. 9) can safely transmit the counting signals for the next iteration. If the main processor is activated before the synchronization is complete, the counting signals of the next iteration can override the counting signals of the last iteration before they are read. Moreover, a processor that should go into a passive state and did not do so can direct a counting signal of the current iteration to a wrong location.

At the end of a vertical synchronization, the lowest processor in the auxiliary column signals the corresponding processor in the T-row to begin executing the algorithm of Lemma 4.2. When the last connection in a row of Lemma 4.2 ends, the leftmost processor in the C-row can signal the main processor to start the next iteration. Thus, the validity of this algorithm follows from Lemma 4.2 and Lemma 4.1. There are at most $\log_k n$ iterations such that at the t'th Iteration, $t = 0, \ldots, \log_k n$, there are at most $m = n/k^t$ leaders. Thus, the complexity of synchronization at the t'th iteration is bounded by $O[n/k^t(k \log k + \log n/k^t)]$. The total complexity of the counting operation is bounded by $O[\max(n \cdot k \log k,\ n \cdot \log n)]$. ■

This yields a complexity of $O[n \cdot k \cdot \log_k n \cdot \log n]$, which is a significant improvement to the complexity yielded by Theorem 3.5.

CONCLUSIONS

In this work, we have shown how to define the asynchronous reconfigurable mesh (ARM) and how to measure the complexity of algorithms executed by it. We have studied the fundamental problem of barrier synchronization and given optimal solutions its two variants: connecting and disconnecting in a row. These operations were used to obtain a general optimal simulation for synchronous RM algorithms over the ARM. We used the problem of counting the number of non-zero bits in an input vector to show that this simulation can in some cases produce non-optimal complexities. We believe that even more conclusive examples of optimal algorithms for the ARA (outperforming the general simulation of the best known synchronous algorithms) can be obtained. Some of the techniques developed for this algorithm seem general enough to be useful as basic units for future asynchronous algorithms. These techniques include: copying/switching rows in the horizontal synchronization of Lemma 4.2) and using Large Processors in Lemma 3.1.

Some open problems remain:

- More conclusive examples of algorithms whose synchronization can be optimized beyond Theorem 3.5 should be obtained.
- An ARA which does not use any form of synchronization (like the "pure" asynchronous pointer-jumping PRAM algorithm described in [13]) would be nice to have.
- The inner structure of dependencies exposed by a given ARA is not handled well by Nishimura's model. Consider, for example, two ARAs $A1$ and $A2$, wherein each processor performs k steps over the $n \times n$ ARM. In $A1$ there are no dependencies or broadcasts. In $A2$, however, each processor depends on all the other processors to reach step t, after which it may execute its next step, $t + 1$. In spite of different types of dependencies, both models have the same complexity of $\Theta(k \cdot n^2 \log n)$. Can we find a better model and corresponding ARAs?

References

[1] Ben-Asher, Y., Gordon, D. and Schuster, A. "Efficient self simulation algorithms for reconfigurable arrays". In *1st European Symp. on Algorithms*, September 1993.

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

[3] Bokka, V., Gurla, H., Lin, R., Olariu, S., Schwing, J.L. and Zhang, J. "Constant time algorithms for point sets on reconfigurable meshes". In *Reconfigurable Architecture Work-shop. IEEE*, April 1994.

[4] Chen, G.-H. (1992) "An $o(1)$ time algorithm for string matching", *Intern. J. Computer Math.* **42**, 185–191.

[5] Cole, R. and Zajicek, O. "The APRAM: Incorporating asynchrony into the PRAM model". In *Symp. Parallel Algorithms and Architectures*, June 1989, pp. 169–178.

[6] Cole, R. and Zajicek, O. "The expected advantage of asynchrony". In *Symp. Parallel Algorithms and Architectures*, No. 2, July 1990, pp. 85–94.

[7] Feller, W. (1950) An Introduction to Probability Theory and Its Applications (Wiley, New York).

[8] Freudenthal, E. and Gottlieb, A. "Process coordination with fetch-and-increment". In *Intl. Conf. Architect. Support for Prog. Lang. and Operating Syst.*, No. 4, April 1991, pp. 260–268.

[9] Gibbons, P.B. "A more practical PRAM model". In *Symp. Parallel Algorithms and Architectures*, June 1989, pp. 158–168.

[10] Lin, R., Olariu, S., Schwing, J.L. and Zhang, J. "Sorting in $o(1)$ time on an $n \times n$ reconfigurable mesh". In *Proceedings of EWPC'92*, 1992.

[11] Miller, R., Prasanna, V.K., Reisis, D. and Stout, Q.F. "Meshes with reconfigurable buses". *Proc. of 15th MIT Conference on Advanced Research in VLSI*, March 1988, pp. 163–178.

[12] Nakano, K. "Efficient summing algorithm for a reconfigurable mesh". In *Reconfigurable Architecture Workshop. IEEE*, April 1994.

[13] Nishimura, N. "Asynchronous shared memory parallel computation". In *Symp. Parallel Algorithms and Architectures*, No. 2, July 1990, pp. 76–84.

[14] Olariu, S., Schwing, J.L. and Zhang, J. "Constant-time convex polygon algorithms on reconfigurable meshes". In *Proceedings of SPIE-Int. Soc. Opt. Eng.*, 1992, pp. 111–121.

[15] Thiruchelvan, R.K., Trahan, J.L. and Vaidyanathan, R. "On the power of segmenting and fusing buses". In *Proceedings of 7th International Parallel Processing Symposium, IEEE*, April 1993, pp. 79–83.

[16] Wang, B. and Chen, G. (1990) "Constant time algorithms for the transitive closure and some related graph problems on processor arrays with reconfigurable bus systems", *IEEE Trans. Parallel Distributed Systems* **1**(4), 500–507.

[17] Wang, B.F. and Chen, G.H. (1990) "Constant time algorithms for sorting and computing convex hulls", Proceedings of the International Computing Symposium (Tsing-Hua University, Taiwan).

Yosi Ben-Asher received his Ph.D. in computer science from Hebrew University in 1990. He is currently a lecturer at the Department of Computer Science, University of Haifa, Israel. His research interests include parallel systems, compilers, operating systems and reconfigurable networks. Current projects include: VLIW scheduling, extracting parallelism from sequential code using probabilistic evaluation of circuits (P2NC) and HParC a programming language for highly parallel servers.

Esti Stein is currently a Ph.D. student in computer science in Haifa University. She got her B.Sc in the Technion, Israel. Her M.A. in the Haifa University was "Automatic Transformation of Shared Memory Parallel Programs into Sequential Programs". Her Ph.D. thesis under the supervision of Dr Yosi Ben-Asher includes development of algorithms for asynchronous Reconfigrable meshes and extending compiler-scheduling techniques for VLIW processors using fast evaluation of algebraic circuits.