JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **30**, 1–22 (1995)

Efficient Self-Simulation Algorithms for Reconfigurable Arrays[1]

YOSI BEN-ASHER,*[,2] DAN GORDON,*[,3] AND ASSAF SCHUSTER†[,4]

*Department of Mathematics and Computer Science, University of Haifa, Haifa 31905, Israel; and †Department of Computer Science, Technion, Haifa 32000, Israel

There are several reconfiguring-network models of parallel computation that are considered in the published literature, depending on their switching capabilities. Can these reconfigurable models be the basis for the design of massively parallel computers? Perhaps the most fundamental related issue is *virtual parallelism*, or *the self-simulation problem*: Given an algorithm which is designed for a large reconfigurable mesh, can it be executed efficiently on a smaller reconfigurable mesh? In this work, we give several positive answers to the self-simulation problem. We show that the simulation of a reconfiguring mesh by a smaller one can be carried optimally and using standard methods on the model in which buses are established along rows or along columns. A novel technique is shown to achieve asymptotically optimal self simulation on models which allow buses to switch column and row edges, provided that a bus is a "linear" path of connected edges. Finally, for models in which a bus is any subgraph of the underlying mesh, efficient simulations are presented, paying by an extra factor which is polylogarithmic in the size of the simulated mesh. Although the self-simulation algorithms are complex and require extensive bookkeeping operations, the required space is asymptotically optimal. © 1995 Academic Press, Inc.

1. INTRODUCTION

The basic idea of a reconfigurable network is to enable flexible connection patterns, by allowing nodes to connect and disconnect their adjacent edges in various patterns. This yields a variety of possible topologies for the network and enables the program to exploit this topological variety in order to speed up the computation.

Informally, a reconfigurable network operates as follows. Essentially, the edges of the network are viewed as building blocks for larger *bus* components. The network dynamically reconfigures itself at each time step, where an allowable configuration is a partition of the network into a set of edge-disjoint buses. A crucial point is that the reconfiguration process is carried out *locally* at each processor (or *switch*) of the network. That is, at the beginning of each step during the execution of a program, each switch in the network fixes its *local configuration* by partitioning its collection of edges into some allowable combination of subsets. Two adjacent edges that are grouped by a switch into the same partition are viewed as if they were (hardware) connected.

There are several reconfiguring models that are considered in the published literature, depending on their switch capabilities. In this work, we focus on two-dimensional arrays (or, meshes) operating in three of the more popular models:

Horizontal–Vertical Reconfigurable Mesh (*HV-RN Model*). The switches may change the configuration of the network so that buses of different lengths are formed horizontally along rows and vertically along columns. Thus, a single bus cannot "change directions" by using both horizontal and vertical bus components (mesh edges) [15, 12, 23]. A VLSI chip called YUPPIE (Yorktown Ultra Parallel Polymorphic Image Engine) has been implemented to demonstrate the feasibility of this reconfiguration style [14, 16].

Linear Reconfigurable Mesh (*LRN Model*). A bus may consist of any connected path of edges, not only vertical or only horizontal. In this model, however, only "linear" buses are composed, so that a bus component is attached to at most one other bus component at each end. Many results present efficient algorithms on the linear reconfigurable mesh. Some of these algorithms achieve constant running time (even when this is not possible using the popular PRAM model), and some match known *Area* $\times$ *Time*^{2} lower bounds. These results include arithmetic operations [19, 8, 21], sorting and selection [2, 10, 9, 20, 6], image processing applications [11, 7], and others [18, 2, 4].

General Reconfigurable Mesh (*RN Model*). A configuration of buses is any partition of the network into edge-disjoint subgraphs, so buses are not necessarily linear. Efficient algorithms were presented on the general reconfigurable mesh, including for example a constant time transitive closure algorithm [26, 27]. A version of this model, namely, the CAAPP (Content Addressable Array Parallel Processor), consisting of a 2-D array of 512×512 bit-serial processors, was implemented [28, 29].

[1] A short version of this work was presented at the 1st European Symposium on Algorithms, Sept. 1993.
[2] E-mail: yosi@mathcs2.haifa.ac.il.
[3] E-mail: gordon@mathcs2.haifa.ac.il.
[4] E-mail: assaf@cs.technion.ac.il.

In [3, 24], the expanding volume of reconfigurable results and architectures was given a theoretical treatment. For example, it was shown that the set of problems computable in constant time on a polynomial size mesh in the linear model is exactly the set of problems computable by a log-space Turing machine. The corresponding set in the general model contains exactly all the problems that are computable by a logspace Turing machine having a symmetric logspace oracle. Thus the general model is expected to be more powerful than the linear one. In particular, these results may explain the existence of a fairly simple constant time connected components algorithm on the general reconfigurable mesh [26], while no such equivalent is known on the linear reconfiguring mesh.

Some work was carried in the direction of simulating general networks using a *larger* two-dimensional mesh. It was shown that any constant degree reconfiguring network may be simulated with no slow-down by a reconfiguring two-dimensional mesh, paying by a quadratic blow-up of the number of processors [3, 24]. This result was improved for the case of d-dimensional meshes: An n^d-nodes d-dimensional mesh can be simulated by a two-dimensional mesh with $O(n^{2d-2})$ processors [22].

1.1. This Work: Optimal Simulations

The question we are interested in this work is whether reconfigurable models (in particular, two-dimensional reconfigurable arrays) can form the basis for the design of massively parallel computers. Perhaps the most basic aspect of this question is the efficiency and ease of algorithms design. Usually, for a certain problem, the solution is given by an algorithm which is suitable for input of size n, where the number of computing processors may be a function of n. It is assumed by the algorithm designer that as many processors as required by his algorithm are simultaneously available for his program. This assumption frees him from the need to know the exact size of the machine he is working on, and thus considerably eases the programming task. Furthermore, independence of machine size is also a desired feature for reasons of software portability.

By the above discussion, it is desirable that the assignment of logical processors to the available physical ones be automatically determined by the compiler. To this end, the compiler should write an efficient *self-simulation program* of a large machine having many processors by a smaller machine with less processors. Hence the ability of efficiently achieving the logical to physical mapping is an extremely important property of any model for parallel computation. In its absence, it is not likely that the model will be chosen for a direct implementation on existing architectures.

Despite the large number of efficient algorithms that are known for reconfigurable arrays, none of the models was previously shown to support optimal self simulations. In this work, we give several positive answers to this problem. We present asymptotically optimal and almost optimal self-simulation results of large reconfigurable-mesh machines by smaller ones. We have the following (informally stated) results:

(1) Using standard simulation techniques the mesh in the HV-RN model exhibits optimal self simulations (Section 3.1).

(2) Although using the same method fails in the LRN model (Section 3.2), a technique is developed to achieve asymptotically optimal self simulation for that model, too (Section 4).

(3) A third algorithm presents self simulations in the RN model, paying by an extra slowdown which is polylogarithmic in the size of the simulated mesh (Section 5).

The self-simulation algorithms are very complex and require lots of bookkeeping operations. We show that in all of our algorithms the required space for the bookkeeping is asymptotically optimal. Yet, to avoid painful reading, we do not cope with constants minimization. In addition, although given for the mesh, the simulation results may be applied to arbitrary rectangles as well.

2. RECONFIGURING MODELS OF COMPUTATION—PRELIMINARIES

A reconfigurable network is a network of processors operating synchronously. The processors residing at the nodes of the network perform the same program, taking local decisions and calculations according to the input and locally stored data. Input and output locations are specified by the problem to be solved, so that initially, each input item is available at a single node of the network, and eventually, each output item is stored by one.

A single node of the network may consist of a computing unit, a memory unit and a switch with reconnection capability. In the sequel, we use the notions of *switch*, *processor*, and *network node* in an interchangeable manner.

A single time step of a reconfigurable network computation is composed of the following substeps:

Substep 1. The network selects a *configuration H* of the buses, and reconfigures itself to H. This is done by local decisions taken at each switch.

Substep 2. One or more of the processors connected by a bus transmit a message on the bus. These processors are called the *speakers* of the bus.

Substep 3. Several of the processors connected by the bus attempt to read the message transmitted on the bus by the speaker(s). These processors are called the *readers* of the bus.

Substep 4. Some local computation is taken by every processor.

At each time step, a bus may take one of the following three states. *Idle*, no processor transmits; *Speak*, there is one or more speakers, all sending the same message; *Error*, there is more than one speaker, and two or more messages are different. An *Error* state is detectable by all processors connected by the corresponding bus, but the messages are assumed to be destroyed. Thus, the bus accessing capability is similar to that of the popular Collision CRCW PRAM.

2.1. Switch Operations

The general reconfigurable network model, as presented above, does not specify the exact operation of the switches. In this paper, we consider three basic variants:

General RN (*RN Model*). The switch may partition its collection of edges into any combination of subsets, where all edges in a subset are connected as building blocks for the same bus. Thus the possible network configurations are any partition into edge-disjoint connected subgraphs.

Linear RN (*LRN Model*). The switch may partition its collection of edges into any combination of connected pairs and singletons. Hence buses are of the form of a path (or a cycle) and the global configurations is a partition of the network into paths, or a set of edge-disjoint linear buses.

Horizontal–Vertical RN (*HV-RN Model*). Buses are formed either along rows (horizontally) or along columns (vertically), but may not contain building blocks from both dimensions.

Observe that a network operating in the HV-RN model has a subset of the set of possible configurations of the same network operating in the LRN model. The same applies to the LRN and the more general RN models.

We omit the description of other switching variants that are considered in the literature. Nevertheless, our methods may be applied to some of these models as well.

2.2. Simulations and Slowdown

Let $\mathfrak{R}$ and $\mathfrak{R}'$ be two reconfigurable networks operating in any of the models defined in Section 2.1. $\mathfrak{R}$ and $\mathfrak{R}'$ may have the same underlying topology or they may differ in their structure. They may be operating in the same model, or they may have different types of switches. We say that $\mathfrak{R}'$ *simulates* a single step of $\mathfrak{R}$ with *slowdown C* if for any single-step algorithm that is executed by $\mathfrak{R}$ there is a C-step algorithm that is executed by $\mathfrak{R}'$ achieving the same computational task. We say that $\mathfrak{R}'$ *simulates* $\mathfrak{R}$ with slowdown C if for any algorithm A that is executed by $\mathfrak{R}$ there is a step-by-step simulation algorithm A' that is executed by $\mathfrak{R}'$ achieving the same computational task, and in which each step of A is simulated with slowdown C. When $C = 1$ we say that the simulation is carried with no slowdown.

Typically, we assign the tasks carried by processors from $\mathfrak{R}$ to be executed by processors from $\mathfrak{R}'$. We say that processor x in $\mathfrak{R}'$ simulates processors (say) y, z, w in $\mathfrak{R}$ if x carries the tasks of y, z, and w during the simulation. Mapping of simulated to simulating processes may change at different steps of the simulated algorithm. More commonly, however, a fixed mapping $\mathcal{M}$ and a simulation of a single, arbitrary step which is consistent with $\mathcal{M}$ is shown with slowdown d. The latter is thus a simulation of any algorithm with slowdown d, given that the input is consistent with $\mathcal{M}$.

The above definitions formalize the intuitive notion of "simulation." However, since $\mathfrak{R}$ and $\mathfrak{R}'$ may have a different structure, a certain computational task may require a specific placement and timing of the input and output items. For the sake of simplicity, we will not define these requirements in a formal fashion. Rather, we assume that they are fulfilled in some "satisfactory" way.

Another issue is the resource requirements by the simulating network. In particular, when presenting a simulation we have to determine the memory requirements by the simulating processors. For example, each processor in a large network which is simulated by a processor in a smaller network is allocated a special buffer in the memory of the simulating processor, in which information about the simulated processor is stored: input and output, local configuration, information about crossing buses, readers and writers on these buses, etc. However, we always assume that the computing power of the processors is equivalent for the simulating and the simulated networks. Moreover, the bus bandwidth, namely, the maximal number of bits in a message that is transmitted on the bus in a single step, is the same for both networks (however, not less than *sizeof*(*id*) in the simulated network).

2.3. The RLA and the Bus-Splitting Method

The most simple connected topology is the Reconfigurable Linear Array (RLA). An RLA consists of n processors labeled $\{0, ..., n-1\}$. Processor i is connected to processors $i-1$ and $i+1$ if $1 \leq i \leq n-2$, to processor 1 if $i = 0$ and to processor $n-2$ if $i = n-1$. Thus, for a processor of a RLA there are only two local configurations: either connect or disconnect the edges. The set of allowable global configurations of an RLA is the set of all 2^{n-2} splittings of the $n-1$ edges into smaller linear arrays. We may view the RLA as a path going from "left" to "right."

One of the most basic techniques in computing with reconfiguring arrays is called *bus splitting* [18]. Consider the RLA and suppose some arbitrary subset of its processors store input values, a single value at each processor. Then, in a single step, the rightmost processor in the RLA can have one of these values. This is done as follows: The processors which do not have input values connect their edges, while the others disconnect and transmit their values to the right. Clearly, the rightmost input value (if exists) will be read by the rightmost processor of the RLA.[5]

[5] Although the bus splitting method is very simple, it gives an indication of the power of the reconfiguring model. In particular, if the input values are 1's and the processors with no input are assumed to store 0's, then the OR of the input is computed in a single step. In comparison, for the same operation in the CREW model, one needs time which is logarithmic in the number of processors [5].

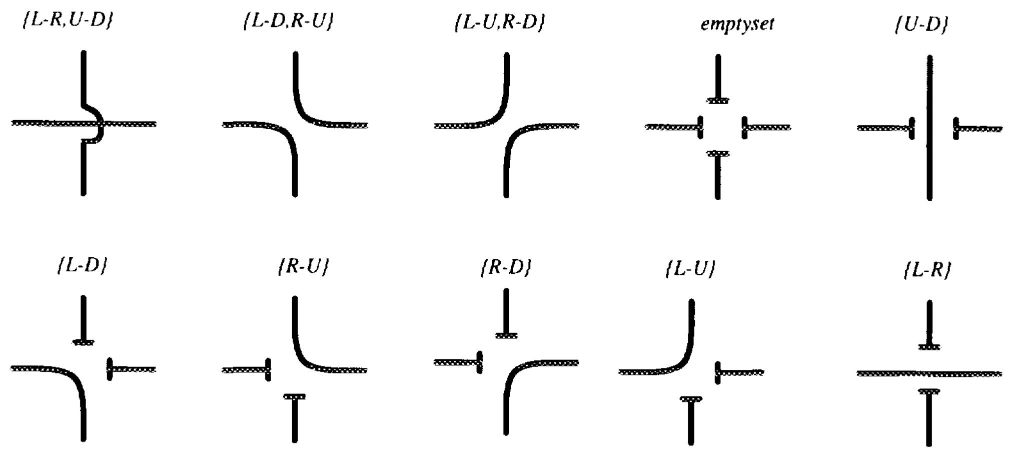

FIG. 1. Local configurations supported by the switch of the mesh in the LRN model.

2.4. The Mesh and the Folding Mapping

The *reconfigurable mesh* is the underlying topology which is the most popular in the literature and which is also the topology considered in this work. One of the reasons for the popularity of the mesh is its *universality*. For example, by paying a quadratic blow-up in the number of processors, any network may be simulated by a two-dimensional mesh with no slowdown [24, 3].

The $n \times n$ reconfigurable mesh, called the n-mesh, or the mesh of size n, is composed of an array of n columns and n rows of processors, with edges connecting each processor to its four neighbors (or fewer, for borderline processors). We refer to the processor at the ith row and the jth column as $[i - 1, j - 1]$.

Only four local configurations are supported by the switch of the HV-RN mesh. These are $\{L - R, U - D\}$, $\emptyset, \{U - D\}, \{L - R\}$. The linear reconfiguring mesh (LRN) supports six additional local configurations. All ten LRN local configurations are depicted in Fig. 1. The RN mesh further supports five additional configurations that are not supported by the LRN mesh. These are depicted in Fig. 2.

For convenience, we envision the mesh as embedded in the plane so that row 0 is at the top and column 0 is to the left.

The following function is sometimes used for mapping large meshes into smaller ones:

$$FOLD(m) = \begin{cases} m \bmod p & \text{if } m \operatorname{div} p \text{ is even} \\ p - 1 - (m \bmod p) & \text{otherwise} \end{cases}.$$

We use

$$[r, c] \rightarrow [FOLD(r), FOLD(c)]$$

for mapping a processor $[r, c]$ of a large mesh into the p-mesh. This has the same effect as that of folding a large page of paper several times into a square of size $p \times p$. A point on the p-sized square "simulates" all points of the folded page that are stabbed when pushing a pin at this point.

In order to simplify the presentation of our algorithms, we describe the global configurations taken by the network and the actions performed by the processors in a nonformal way. For example, by saying that "each processor of the bottom row transmits (some message) on its column" we mean the following. First, set the global configuration so that all columns are connected, forming vertical buses, and processors at different columns are disconnected. This is done by using the $\{U - D\}$ local configuration at all processors. Then, let $[0, i - 1]$, the ith processor of the bottom row, be the speaker of the bus created on column $i - 1$. The readers are all the rest of the processors of that column.

3. THE CONTRACTION METHOD

In this section, we show that the straightforward approach of simulating each submesh with a single processor gives a self-simulation algorithm with optimal asymptotic

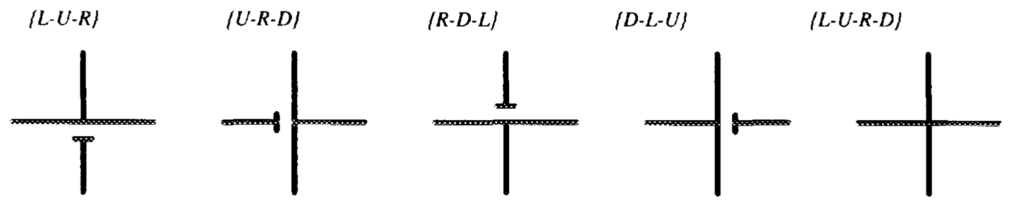

FIG. 2. Local configurations that are supported by a switch of the mesh in the RN model and are not supported by a switch of the mesh in the LRN model.

slowdown on meshes that operate in the HV-RN model.[6] We call this technique *the contraction method*, as submeshes may be viewed as if they are contracted into a single processor. We then proceed to show that the contraction method fails to achieve efficient self simulation for the LRN and the RN models.

3.1. Optimal Self Simulations in HV-RN Meshes

The HV-RN algorithm is based on the simulation of an n/p-submesh of the n-mesh by a single processor of the p-mesh (We assume that n/p is an integer), as shown in the following lemma.

LEMMA 3.1. *In the HV-RN model, a single processor can simulate the l-mesh with slowdown* $4l^2$.

Proof. In the HV-RN model, buses are formed either along rows or along columns. This implies that movement of a single message on a bus can be determined by considering one-dimensional arrays of processors of the l-mesh. In other words, we need only to consider the simulation of a row or a column of the l-mesh. The proof will be completed by showing that the simulation of a single row (similarly, a single column) can be done by a single processor with slowdown $2l$.

The simulation of an l-processor RLA by a single processor is carried in a double pass of on the RLA: left-to-right and then right-to-left. In the first pass, the simulating processor simulates RLA processors $0, 1, \dots l - 1$ in that order. The operations and decisions of each processor are determined, and the simulating processor collects data about transmitted messages or error states. When a simulating processor disconnects, so that one bus is terminating and one is starting, the data concerning the terminating bus is stored. In the reverse pass, the data stored at the rightmost processor of each bus is dispersed to the rest of the processors of this bus which are readers. ■

Using Lemma 3.1, we get the following theorem.

THEOREM 3.1. *In the HV-RN model, the simulation of the n-mesh by a p-mesh can be completed with slowdown* $5(n/p)^2 + O(n/p)$.

Proof. We will show the simulation of a single step. Each processor in the p-mesh simulates an n/p-submesh of the n-mesh. More precisely, processor $[i, j]$ of the p-mesh (where $0 \le i, j \le p - 1$) simulates the submesh of the n-mesh consisting of the processors $[k, l]$, for all $in/p \le k < (i + 1)n/p$ and $jn/p \le l < (j + 1)n/p$. The processors $[k, l]$ for which k or l get extreme values in these ranges compose the *boundary* of the submesh. A row in a n/p-submesh is called a *row-segment* (of the corresponding row of the n-mesh). The *rightmost* and *leftmost* processors of a row-segment belong to the boundary. Similar terminology apply to column and *column-segments*.

[6] A result similar to the one obtained in Section 3 was shown independently by Maresca and Li [17].

ALGORITHM HV-RN Simulation. The algorithm consists of three phases. In the first phase, each processor of the p-mesh simulates the corresponding submesh. By Lemma 3.1, this phase takes $4(n/p)^2$ steps. We make a minor change in the algorithm from Lemma 3.1 as follows. The algorithm notifies a boundary processor in each segment whether it is connected by a bus to the other boundary processor in that segment. Moreover, the information includes whether there was a speaking or a reading processor in the "internal part" of any bus reaching a boundary processor.

The second phase consists of n/p steps. These correspond to the n/p rows (columns) of the n-mesh that are simulated by every row (column) of the p-mesh. Let x be a processor of the p-mesh and let M_x the submesh which is simulated by x. During the second phase x collaborates with the rest of the processors in its row (column) to simulate the intersubmesh bus connections. In the ith step of this phase, x simulates intersubmesh connections for the ith row-segment (column-segment) of M_x. Let us consider simulation of row only, the details for the column are similar.

Let z be the leftmost processor of the ith row-segment in M_x and let y be the corresponding rightmost processor. During the second phase x simulates the connections of z and y to their neighbors which belong to neighboring submeshes. The connections, denoted z_{LEFT} and y_{RIGHT}, correspond to the left and right edges of x, which are denoted x_{LEFT} and x_{RIGHT}, respectively. We consider two cases:

1. If z_{LEFT} and y_{RIGHT} belong to different buses, then x disconnects x_{LEFT} from x_{RIGHT}. Let B_{LEFT} and B_{RIGHT} denote the subbuses in which z_{LEFT} and y_{RIGHT} take part, respectively. If the part of B_{LEFT} that is contained in M_x contains a speaker, then x speaks on x_{LEFT} (while checking the state of the bus for an error). If, on the other hand, it contains only readers, then x reads from x_{LEFT}. As explained above, this information is stored in x during the first phase. Similar decisions are made for reading and speaking on x_{RIGHT}.
2. If z_{LEFT} and y_{RIGHT} take part in the same bus, then x connects x_{LEFT} to x_{RIGHT}. x reads and speaks according to the operations that are taken by the processors of the ith row-segment of M_x.

The third phase involves a single pass of each processor on the corresponding simulated submesh. Information that was read during the second phase is informed to all simulated processors which take part in buses which cross submesh border lines. This is completed in $(n/p)^2 + O(n/p)$ steps. ■

3.2. The Contraction Method Fails in Stronger Models

The contraction method used above for the simulation of an n-mesh by a p-mesh, involves splitting the larger mesh into submeshes, each of which is simulated by a

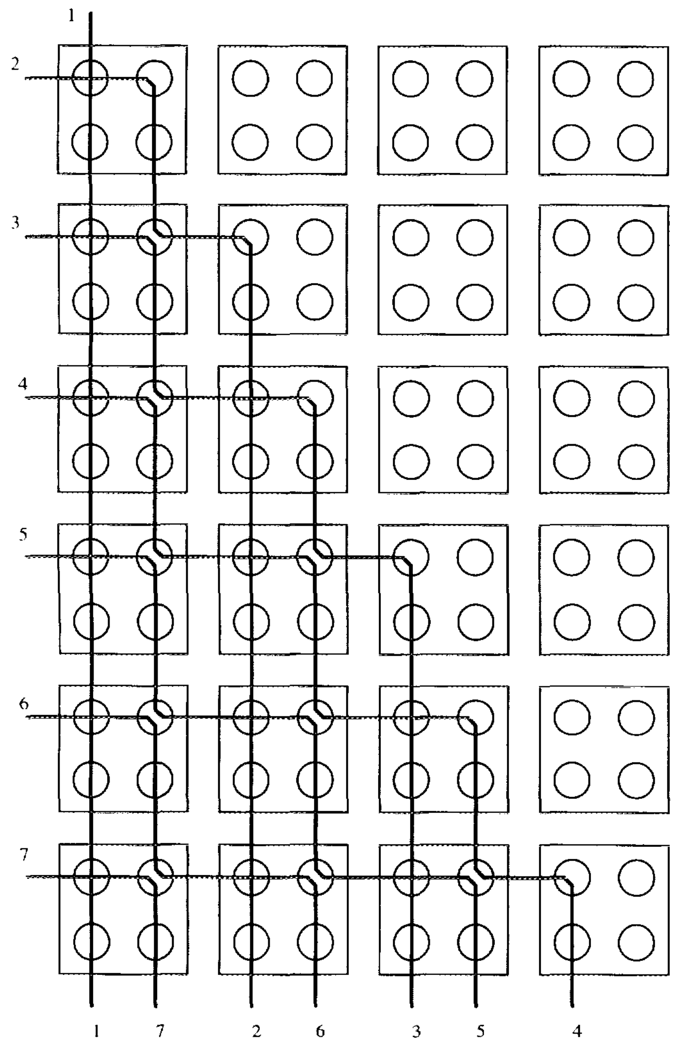

FIG. 3. Lower bound example: buses are shuffled so that bus i shares an exit of a simulating mesh processor with all the buses 1, 2, ..., $i - 1$.

processor of the smaller mesh (see first and third phases of the algorithm).[7] Another characteristic of the above simulation algorithm is that a bus which crosses submesh boundaries is simulated in a single step by a bus, solely dedicated for that purpose (second phase of the algorithm). We now give an example how this straightforward approach fails to achieve efficient simulations in the LRN or the RN models.

Suppose an LRN n-mesh is to be simulated by an LRN $n/2$-mesh. We let processor $[i, j]$ in the $n/2$-mesh simulate processors $[2i, 2j]$, $[2i+1, 2j]$, $[2i, 2j+1]$, and $[2i+1, 2j+1]$ of the n-mesh. Consider a simulation of a step of the n-mesh in which the bus configuration is as shown in Fig. 3. The figure shows the mapping of processors of the n-mesh, represented by circles, to the simulating processors of the $n/2$-mesh, represented by squares. Buses are drawn by thick black lines. Let us describe the configuration of the n-mesh, as shown in Fig. 3:

Bus 1 starts at processor [0, 0] (in the n-mesh) and goes straight, all the way down to processor $[n - 1, 0]$. Bus 2 starts also at processor [0, 0], goes right to processor [0, 1], down to [2, 1], right to [2, 2], and then all the way down to $[n - 1, 2]$. Note that bus 2 shares with bus 1 the Down exit of the simulating processor [0, 0]. Hence, according to our policy of simulating a full (boundary crossing) bus in a single step, we need two different steps to simulate bus 1 and bus 2. Bus 2 has two stair-like bends. Bus 3 has three such bends: Start [2, 0], right to [2, 1], down to [4, 1] (parallel to bus 1), right to [4, 3], down to [6, 3] (parallel to bus 2), right to [6, 4], and down all the way to $[n - 1, 4]$. Again, since bus 3 shares edges of the simulating mesh with both bus 1 and bus 2, it is simulated in a step when both bus 1 and bus 2 are not simulated.

We may proceed in the way described above in order to construct $n/2 + 1$ such buses. Bus i shares exits of the simulating mesh with all the buses j for $1 \leq j < i$. We conclude that for the given configuration and under the given assumptions on the operation of the simulating algorithm, the simulation takes at least $\Omega(n)$ steps regardless of the size of the simulating mesh.

4. OPTIMAL SIMULATIONS FOR LRN MESHES

Although it was demonstrated in Section 3.2 that the straightforward approach fails to achieve efficient self simulations of LRN meshes, we show in this section a more involved technique which obtains optimal simulation results. The main result of this section is the following theorem.

THEOREM 4.1. *The simulation of the n-mesh by the m-mesh in the LRN model is completed with slowdown* $\Theta((n/m)^2)$. *The simulation algorithm uses* $\Theta((n/m)^2)$ *extra space at each processor of the m-mesh.*

The algorithm uses a variant of a connected components algorithm for graphs having only linear and noncyclic components, which may be of interest in its own right.

4.1. Preliminaries

We first note that a processor writing on a linear bus may do so in two different modes: (a) the bus is connected inside the processor, and the processor simply writes on the bus; (b) the bus is not connected inside the processor, and the processor writes a message on each end of the bus (in theory, these messages may be different).

Our notion of a linear bus allows cycles. Whenever we have a configuration of buses and speakers on the buses, we say that *condition* NSC (No Speaker Cycle) *holds* if there is no speaker on a cycle. Note that a speaker in mode (b) above cannot be on a cycle, because the bus is not connected inside the processor. Our next observation is that any configuration of linear buses and speakers can

[7] Indeed, this is the common approach of simulating a processor array with a smaller one, e.g., in the fixed connection model of computation [13, p. 234].

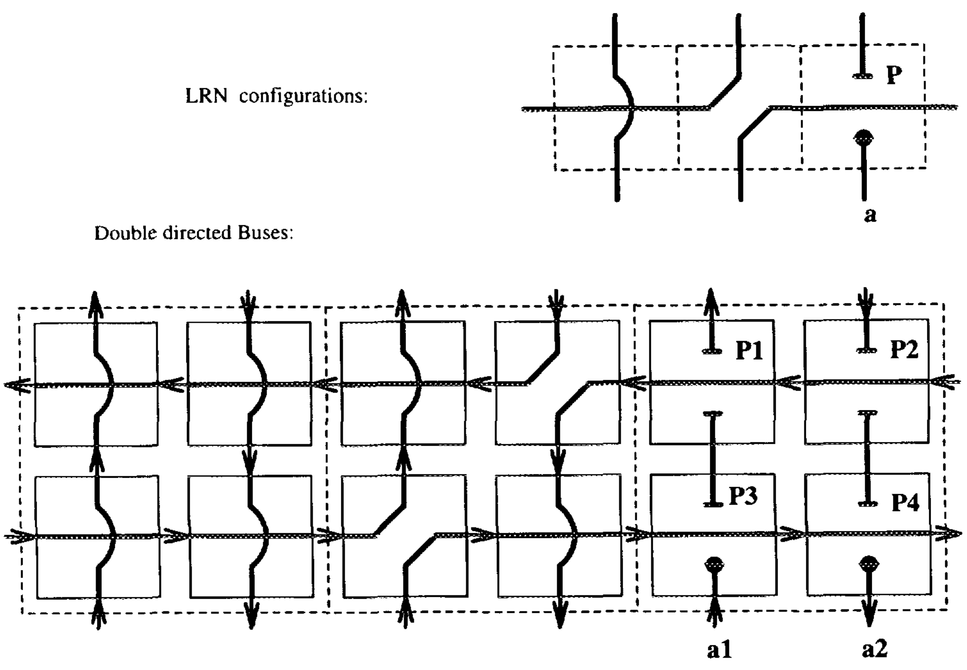

FIG. 4. LRN bus configurations in an n-mesh, and the corresponding double directed buses in a $2n$-mesh.

be simulated in two steps in a straightforward way by a configuration satisfying NSC as follows:

Step 1. A processor that broadcasts in mode (a) does not connect the buses inside, and simply broadcasts its message in mode (b) on the two bus ends; it also listens to the two ends.

Step 2. If the above processor detected an error at one or both bus ends, it now broadcasts a special error message on both bus ends. All processors receiving such an error message will know that the simulated bus (which may be a cycle) had inconsistent messages written on it. If the processor did not detect an error, it now connects the bus ends and listens (without broadcasting), in case some processor broadcasts an error message.

The *leader election problem* for a bus is the problem of all the processors agreeing on one of them being a leader. When the bus is linear and not a cycle, this problem can be solved in $O(1)$ time by simulating an n-mesh with a $2n$-mesh as follows: Every processor of the n-mesh is simulated by a 2×2 square of processors on the $2n$-mesh, and every bus is simulated by a double bus on the $2n$-mesh. This double bus can be viewed as having 2 directions, as illustrated in Fig. 4. In the $2n$-mesh, bus directions alternate on the rows and also on the columns. A speaker on the n-mesh is simulated by some of the simulating four processors broadcasting only in outgoing directions. The two sets of buses are used to prevent concurrent write by the end processors.

To explain this, note from our previous comments that it is sufficient to simulate processors writing in mode (b). Consider the case of processor P (in Fig. 4) writing on bus a. P is simulated by processors $P1$ to $P4$, and bus a is simulated by buses $a1$ and $a2$. Only processor $P4$ writes on the outgoing bus $a2$. $P3$ only listens to its bus $a1$. Naturally, each step of the n-mesh must be simulated by three steps of the $2n$-mesh:

- An internal step for each 2×2 square assigning the tasks to the four processors.
- An external step, where each processor interacts with the processors outside the squares.
- An internal step, where the four processors exchange the information from step 2.

The leader is elected by having each endpoint of the bus transmit its id towards the other end. All processors on the bus listen, and the processor with the smallest id is chosen by all as the leader. Note that on each bus, only two processors will transmit.

4.2. LCC: Linear-Connected Components

DEFINITION 4.1. A graph $G = (V, E)$ is called linear if the degree of every vertex is ≤ 2, and G is acyclic.

The LCC problem is defined in the following lemma, stating our main result for 4.2.

LEMMA 4.1. *Let $G = (V, E)$ be a linear graph, $|V| = n$. Assume that the adjacency matrix $M = (m_{i,j})$ of G is stored on the $2n$-mesh, where $m_{i,j}$ is stored on processor $[2i, 2j]$. Then the connected components of G can be found in constant time, and the output is stored such that for every $0 \leq i < n$, $[2i, 0]$ holds some j such that vertices i and j are connected, and two connected vertices hold the same value.*

Proof. We consider an n-mesh in which every processor is the image of a 2-submesh of the $2n$-mesh. We call the processors of the given n-mesh v-processors (virtual processors). A bus in the n-mesh can be represented by two paths in the $2n$-mesh, so that a leader may be elected as described above. The rest of the proof is described in terms of the n-mesh, so that $[i, j]$ denotes the v-processor in row i, column j.

We now describe the algorithm solving the LCC problem:

1. Every processor $[i, j]$ that has $m_{i,j} = 1$ determines which of the following cases hold:
   (a) it is a unique 1 in its row;
   (b) it is a unique 1 in its column;
   (c) if it is not unique in its row, the direction (left or right) of the other 1 in the same row;
   (d) if it is not unique in its column, the direction (up or down) of the other 1 in the same column.

Note that there are at most two 1's in a row or column, because the degree of every vertex is ≤ 2.

2. If $m_{i,j} = 0$, $[i, j]$ connects the buses $\{L - R, U - D\}$. Else:
   (a) if $m_{i,j}$ is the unique 1 in its row, $[i, j]$ does not connect itself to anything in its row;
   (b) similar to (a), but for the column;
   (c) if there is another 1 in the same row, $[i, j]$ connects itself to the bus in the direction of that 1.
   (d) similar to (c), but for the column.

Note that conditions (a)–(d) can be tested using bus-splitting. Figure 8b below shows the 1's of a 10×10 adjacency matrix and the corresponding bus configuration. Observe that when $[i, j]$ connects itself to a 1 in its column and to a 1 in its row, the corresponding bus segments are connected together inside $[i, j]$.

Define G_{LCC} to be the graph formed according to step 2 (above), i.e., $G_{LCC} = (V_{LCC}, E_{LCC})$, where

$$V_{LCC} = \{[i,j] \mid m_{i,j} = 1\};$$

$$E_{LCC} = \left\{ \{[i,j], [k,l]\} \;\middle|\; \begin{array}{l} [i,j], [k,l] \text{ are connected together by} \\ \text{a horizontal or a vertical bus segment} \end{array} \right\}.$$

Denote by G' the *dual* graph of G. Namely,

$$G' = (E, \{\{e_1, e_2\} \mid e_1 \cap e_2 \in V\}).$$

CLAIM 4.1. *G_{LCC} contains exactly two isomorphic copies of G'.*

Proof. Consider a connected component of G', which is a path of maximal length $e_1, e_2, \ldots, e_k$, and $e_i \cap e_{i+1} \in V$. For some $v_i, v_j \in V$, $e_1 = (v_i, v_j)$. Assume w.l.o.g. that $i < j$. The isomorphic copy of $e_1, \ldots, e_k$ is obtained as follows:

$$e_1 \to [i,j]; \quad e_2 \to \begin{cases} [i,k] & \text{if } e_2 = (i,k) \\ [k,j] & \text{if } e_2 = (j,k) \end{cases}.$$

The maps of e_1, e_2 are connected in G_{LCC}, either on row i or column j. This construction continues for $e_3, \ldots, e_k$. The second copy of the connected component $e_1, \ldots, e_k$ is obtained by mapping e_1 to $[j, i]$ (when $i < j$), and proceeding in the same manner as above. Note that the two copies of the connected components are *reflections* of each other about the main diagonal of the mesh. ■

Back to the LCC algorithm: every connected component in G_{LCC} now chooses a unique label as follows: A v-processor $[i, j]$ knows that it is the end of a linear component when it holds a unique 1 in its row or column. Each end $[i, j]$ transmits $\min(i, j)$ to the other end (using the double path mentioned in the preliminaries). Now all v-processors on the linear component choose the minimum number that was transmitted as the component's label. Note that the reflection component about the diagonal will have the same label.

To transmit the information to column 0, we do the following: Every v-processor holding a 1 transmits i (the component label) to the left, provided there is no edge of G_{LCC} to its left; see $\leftarrow$ in Fig. 8b. This transition is done by letting every v-processor holding a 0 connect $\{L - R\}$, while v-processors which do have a 1 disconnect their edges ($\emptyset$) and transmit i to the left. Note that between them, the two isomorphic copies of a connected component transmit all necessary information to column 0. The label transmitted to every processor in column 0 is shown in Fig. 8b in parentheses to its left.

This completes the proof of Lemma 4.1. ■

The LCC′ Problem. Consider now a graph G such that every vertex degree is ≤ 2, but we now allow cycles. The LCC′ problem is defined the same as the LCC problem, except that all processors that are on *any* cycle are considered as belonging to the same component. The component label for such processors will be some special value CYCLE. A simple modification of the LCC will solve the LCC′ problem: At the stage where each end $[i, j]$ transmits $\min(i, j)$ to the other end, v-processors that are on a cycle will not detect anything on the bus; from this they will conclude that they are on a cycle, and assign themselves the component value CYCLE. For convenience, we henceforth use LCC to refer to LCC′.

4.3. The LRN Simulation Algorithm: Introduction and Sketch

Recall that n denotes the size of the simulated mesh and m is the size of the simulating mesh. If $m \leq 4$, we simulate the n-mesh with one processor. Else, let $p = m/4$. We shall use a p-mesh to traverse the n-mesh. As we have seen, the m-mesh solves the LCC problem on $2p$ vertices in $O(1)$ time, given that the inputs reside in every alternative processor.

We define a mapping of the p-mesh into the m-mesh so that its image can simulate the p-mesh with no slowdown.

Mapping. Processor $[i, j]$ of the p-mesh is mapped to processor $[4i, 4j]$ of the m-mesh.

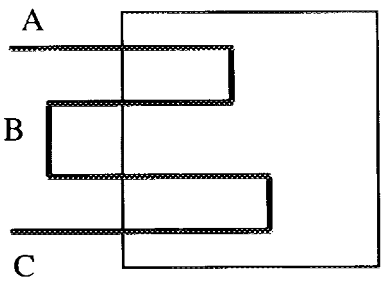

FIG. 5. Buses A, B, and C encountered and joined inside a window.

Simulation. Processors of the m-mesh which are not the image of the p-mesh fix their configuration as $\{L - R, U - D\}$. It is straightforward to see that in this way the m-mesh can simulate the p-mesh and can also solve the LCC problem on $2p$ vertices. In the rest of the algorithm description, we use the term p-mesh to refer to the image of the p-mesh in the given m-mesh.

ALGORITHM LRN Simulation (Sketch). We assume that n is divisible by p. We divide the n-mesh into $(n/p)^2$ submeshes of size $p \times p$; each such submesh is called a *window*. The basic idea is to traverse the n-mesh with the p-mesh in snake-like order. The p-mesh moves from one window to the next one, keeping track of all necessary bus information. At every window position of the p-mesh, the following occurs: New bus segments are encountered, old bus segments enter the window from a previously traversed window, and some old bus segments join up with others (see Fig. 5). In addition, some processors may write on a bus. Every new bus segment that is encountered is given some unique *id*, and when bus segments join, the combined segment is given a single *id*; this is where the LCC is used. At the end of the forward traversal of the window, we have all the separate buses, each identified by a unique *id*, and we also have all the necessary broadcast information for each bus. The window is then moved over the n-mesh in the opposite order. At every position, the bus segments in the window are set up, and the broadcast information for each segment is broadcast from one of its endpoints. Note that every bus which is contained entirely inside some window can be handled in a simple manner, so we assume from now on that we are only dealing with buses that cross window boundaries.

According to our previous comments, it suffices to consider the case when condition NSC holds. Note that there may still be cycles, but there will be no speakers on a cycle. Also, any bus cycle eventually results in a cycle in the LCC graph, and the processors on such a cycle will become aware of it by getting the value CYCLE, as explained in the description of the LCC′ problem (end of Section 4.2). During the backsweep, such bus cycles will all have the *id* CYCLE, but the broadcast message will be null.

To begin with, we assume that all the bus connections in the n-mesh are stored in the processors of the p-mesh in the folding mapping, as described in Section 2.4. Thus, in moving from one window to the next in the snake-like order, the two rows or columns on either side of the boundary are simulated by the same row or column of the window. Information about speakers is also stored in this manner.

In general, every window position is bordered by up to four windows, of which at most two are "old" window positions, and at most two are "future" window positions. Furthermore, one old window is the immediate predecessor of the current position, and one future window is the immediate successor (called the "next" window). Since the buses are linear, each bus segment may have two ends leaving a window to a future window position. The border processor at which a bus end leaves a window retains all relevant information about the bus, including the status of the other end of the bus and the identity of the processor "in charge" of the other end. Whenever two or more bus segments merge in a window (due to the LCC operation), they become a single bus segment, with only two ends. The processors "in charge" of these ends are informed about the *id* of the single bus segment.

As long as one end of a bus continues from the current window to the next, it retains all relevant information about the bus (including the status of the other end). However, when a bus end does not continue into the next window (e.g., if the bus segment terminates in the window), that same bus may still be encountered in a future window. In this case, a special mechanism (called the *column stack*) is used to convey all necessary information to that bus in the future window position.

The rest of this section is organized as follows. Section 4.4 introduces some necessary concepts and explains in detail everything that happens inside a window. Section 4.5 gives the details of how dormant endpoints are updated. Section 4.6 describes the column stack mechanism for updating dormant endpoints. Section 4.7 gives a detailed description of the back-sweep.

4.4. The LRN Simulation Algorithm: Technical Details

We need some terminology to explain the method. A bus segment that intersects a window is called "live" (or active) in the window. If a segment was live in some window $W1$ and is not live in some successive window $W2$ (and did not terminate prior to W2), it is called "dormant" with respect to $W2$. For each bus segment active in a window, the following is true (see Fig. 6):

New Segment. The segment starts in the window; such a segment is called a "new" bus.

Entering Segment. The segment enters the window from the immediate predecessor window; we call such a segment an "entering" segment.

Awakening Segment. The segment enters the window from an old window which is not the immediate predecessor, which means that the segment was dormant with re-

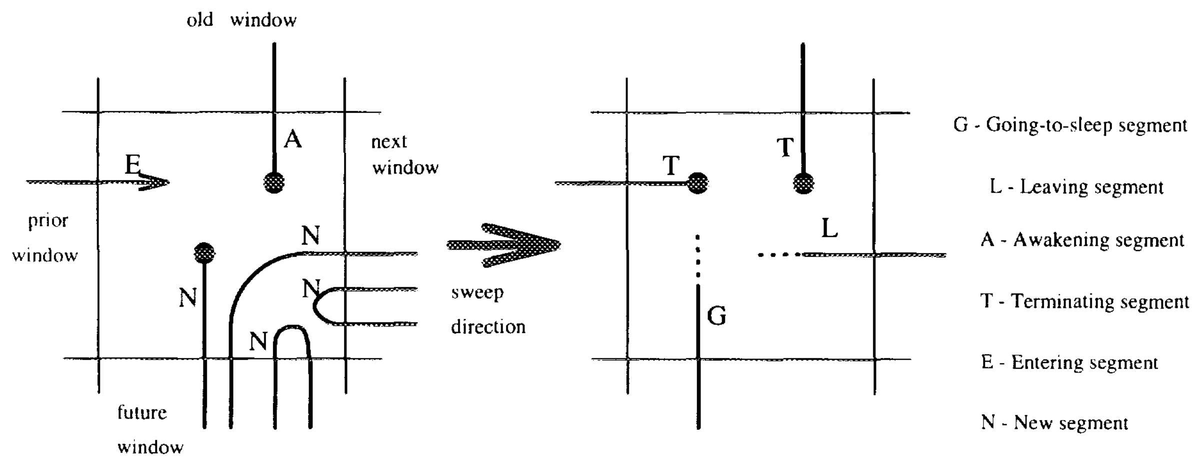

FIG. 6. Bus segments in a new window position.

spect to the predecessor; such a segment is called an "awakening" segment.

Terminating Segment. The entering or awakening segment terminates in the window; it is called an "ending" or "terminating" segment.

Leaving Segment. The segment enters the next window; it is then called a "leaving" segment.

Going-to-Sleep Segment. The segment enters a future window which is not the next window, so it is dormant with respect to the next window; we call such a segment a "going-to-sleep" segment.

We now describe everything that happens inside a window. At first, the window configures all the buses in it according to the information stored in the window's processors. New buses are given *id*'s by the processors through which they leave the window (recall that we are only handling buses which cross window boundaries). In case a new bus leaves through two processors, one of them is elected by the bus segment to assign the new bus an *id*. The simulation algorithm performs the following steps:

1. If an entering segment has an awakening partner (the other end of a bus segment with the same *id*), it informs the awakening end of new developments, which may include a new bus *id* and a broadcast message. Recall that a live *id* is always aware of the status of its dormant end and of the processor that retains this information.
2. Some awakening segments may not have an active "other end" entering the window. The processor in charge of the awakening segment is updated by a so-called "column stack" which will be explained in detail later. This mechanism ensures that the processor is updated in constant time. The information in the column stack is placed there by the other end of the bus at some previous window in which the other end did not continue.
3. Every processor in charge of an entering or awakening segment now has the current id of the bus, and the situation of the other current end of the bus, which may also be entering or awakening. We now configure the buses inside the window, according to the basic information about the configuration of the buses. Every local bus segment (i.e., an intersection of a bus with the window) is used so that each processor at the end of the local segment has the information about the other end. This is done by having both ends transmit the information along the local segment, using the "double path" existing in the underlying $2p$-mesh, as mentioned in the preliminaries.
4. Consider a processor P, in charge of an entering or awakening segment; see Fig. 7. Consider the linear bus through P: one end leaves the window at the point where the bus entered or woke up in the window. This end of the bus may eventually reenter the same window, and if it does so after passing only through windows that have been processed, it will enter the window (or wake up) through some processor-in-charge Q. The bus may reenter the window more than once; we assume Q is the *first* such processor along the bus from the direction of P. Our method of retaining and conveying information will guarantee that P will have all the necessary information about Q. The other end of the linear bus through Q may continue inside the window and exit through another processor-in-charge R. So P has information about zero, one, or two

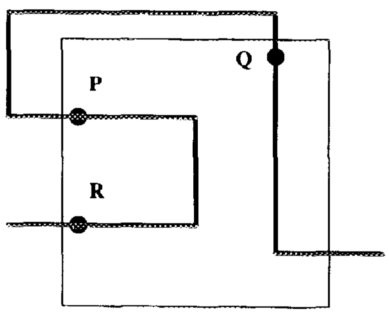

FIG. 7. A processor in charge P connected to processors in charge Q and R.

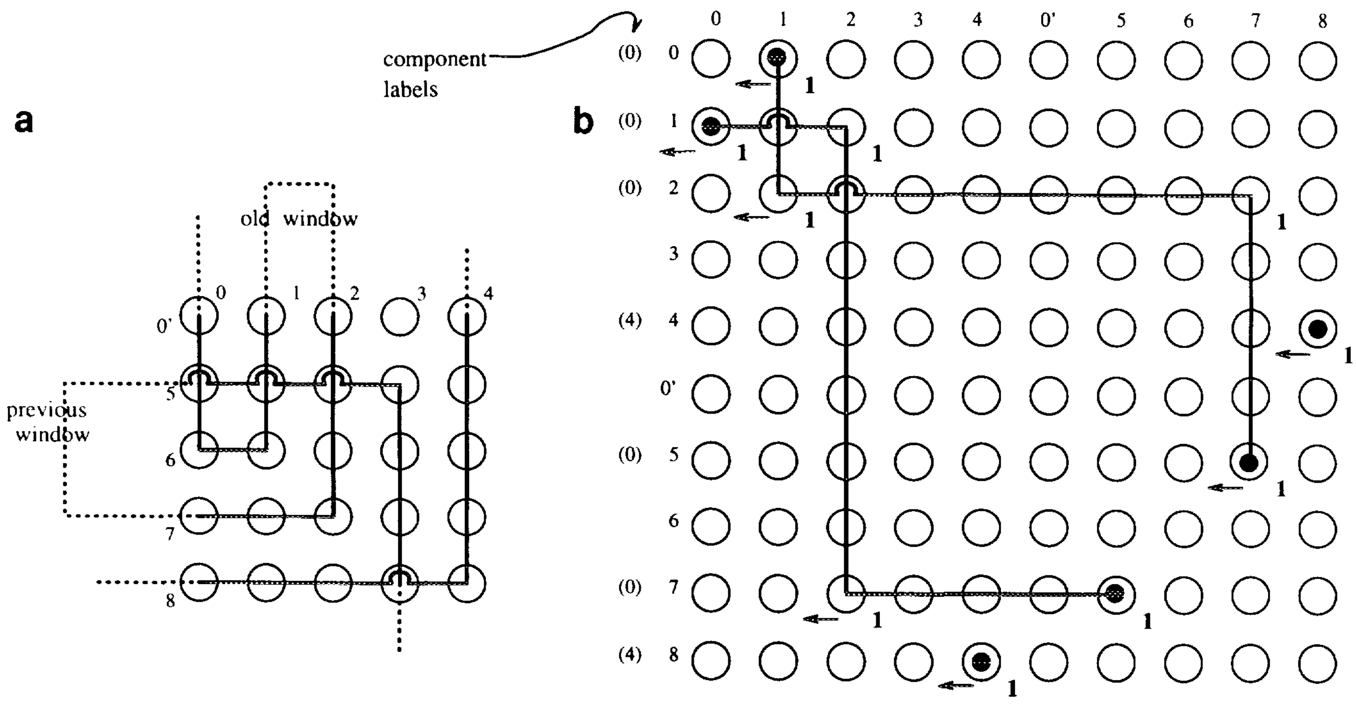

FIG. 8. Example of the LCC process applied to a bus configuration: (a) bus segments in a 5 × 5 window; (b) the corresponding LCC graph.

other processors-in-charge to which it is connected (at most one such connection is inside the window and at most one is outside the window). The fact that P is connected this way to at most two processors is crucial, because it ensures that the graph representing these connections is linear, thus enabling the use of the LCC algorithm to find the connected components and assign a unique id to each component. Note that [0, 0] is a special case, since it may be in charge of two segments, one entering and one awakening. It is therefore regarded in the following as two processors, numbered 0 (potentially in charge of an awakening segment) and 0′ (potentially in charge of an entering segment. We label the processors in the top row 0, 1, 2, ..., $p - 1$, and the processors in the left column 0′, p, $p + 1$, ..., $2p - 2$. All these processors are now mapped in a straightforward manner to the top row and the left column of the $2p$-mesh, in the order 0, 1, ..., $p - 1$, 0′, p, $p + 1$, ..., $2p - 2$.

5. At this stage, the adjacency matrix corresponding to the connections found above is set up in the $2p$-mesh, as follows: horizontal buses are configured, and the processors in column 0 (of the $2p$-mesh) transmit the processor labels (0, 0′, 1, 2, ..., $2p - 2$) to which they are connected, at most two labels per bus. All processors listen to the broadcast, and if a processor in column k reads the label k, it knows that it should hold the value 1 in the adjacency matrix. Next, vertical buses are configured, and the above step is repeated with the columns of the $2p$-mesh.

EXAMPLE. An example is given in Fig. 8. Figure 8a shows a 5 × 5 window, moving from left to the right, and the buses that are configured in it. Bus connections outside the window are shown as dotted lines. Processors 5, 7, and 8 are in charge of entering bus segments, and processors 0, 1, 2, and 4 are in charge of awakening bus segments. Each processor-in-charge knows whether it is connected outside the window to another processor-in-charge; awakening bus segments are informed of this by the column stack mechanism. After the local segments are configured, all processors-in-charge know to which other processors-in-charge they are directly connected (inside or outside the window). The $2p$-mesh holding the adjacency matrix is shown in Fig. 8b. Note that if one of the buses is a cycle, it will eventually result in a cycle in the corresponding LCC graph (a unique 1 in both its row and column is considered a cycle).

6. An LCC is performed on the adjacency matrix on the $2p$-mesh. Since the actual mesh is of size $m \times m = 4p \times 4p$, by Lemma 4.1 the LCC can be done in $O(1)$ time. The LCC operation chooses one unique bus *id* for all the segments that were found to belong to one bus. Figure 8b also shows the bus segments resulting in the $2p$-mesh as a result of the LCC operation. In the case of a cycle, it follows from our NSC assumption that there is no speaker on this bus, so the LCC will assign the value CYCLE to its processors.

7. Using the connections of the LCC, any processor carrying a message from a speaker now transmits its message. At this stage, it is possible that there will be two or more different messages, in which case the processors will retain a special value ERROR as the bus message.

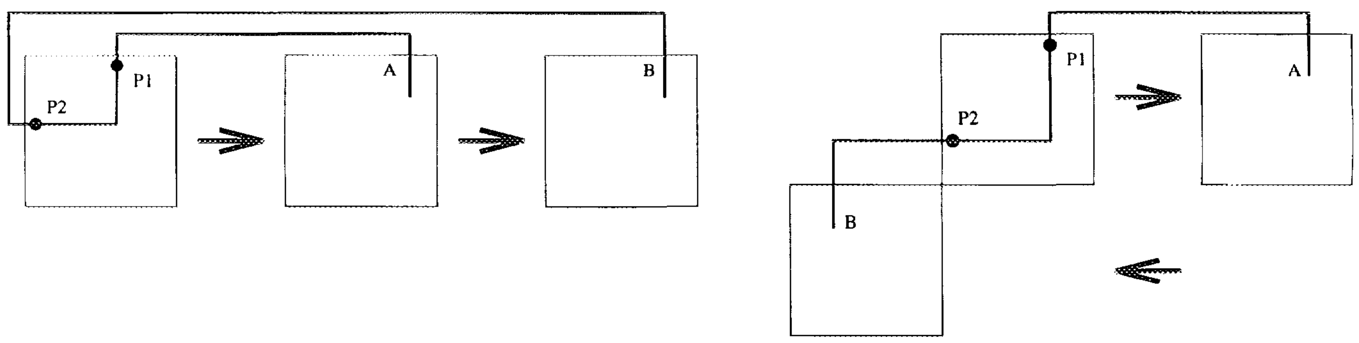

FIG. 9. Case (a): two situations in which both ends are dormant.

8. Consider now a single bus and the situation of its current endpoints. Each endpoint may be in one of three states: inside the window, outside the window and dormant, or outside the window but not dormant (this happens if the window has already passed over an actual endpoint of the bus). The current endpoints need to be updated of the new bus *id* (and perhaps of the broadcast information), and this is done as follows:

(a) If a current endpoint is in the window, no updating is necessary, since it has all the necessary information after the LCC.

(b) If a current endpoint is outside the window and not dormant, we do not update it at this stage, because it will not participate in any future LCC operations. It will be updated during the back-sweep of the window (Section 4.7).

(c) Consider now a current endpoint that is dormant. We have to ensure that when it awakens, it will hold the latest *id* (and any other relevant information) that has been assigned to the bus. A naive approach is to try to update it every time a new *id* is found, but this is inefficient because a dormant end may be updated many times before it wakes up, so some method has to be found that will choose the latest of these updates. The problem is that in some window, the same physical processor, say x, may be in charge of several different dormant ends (which will wake up in different future windows), so updating it in constant time in the current window is impossible. One approach may be to store all these updates in the column of x. However, this, in turn, will require the extraction of the latest update from the column (when the dormant endpoint awakes), which cannot be done in constant time. The following section describes our solution.

4.5. Updating a Dormant Endpoint

Our basic solution to updating a dormant end is to adopt the lazy approach: As long as one end of the bus is active and continues into the next window, it will take care of the dormant end in the future. Only when it does not continue, does it initiate an update operation for the dormant end.

Consider now the portion(s) of the bus inside the window. Our assumption is that one current end—call it A—is dormant, and let $P1$ denote the processor through which A enters the current window, so $P1$ either borders the previous window or some past window. Let B be the other current end of the bus, and let $P2$ be the processor through which the bus connected to B enters (or wakes in) the window. There are four cases to consider, illustrated in Figs. 9–12. In all these figures, the window with processors $P1$ and $P2$ is the current window:

(a) B is dormant; see Fig. 9. It can be seen that during the LCC, $P1$ and $P2$ correspond to the ends of a connected component. We specify now that during the LCC the endpoints of such a component exchange all their information about their current bus ends. So now both $P1$ and $P2$ have all the information about A and B. The update action is the following: $P1$ checks which of A or B is due to wake earlier. If A is due to wake before B (or in the same window as B), then $P1$ initiates an update of A using the "column-stack mechanism" described later. Note that because B is also dormant, then, by symmetry, $P2$ will do the same for bus-end B. This guarantees that the dormant end which wakes earlier will be able in the future to update the other dormant end.

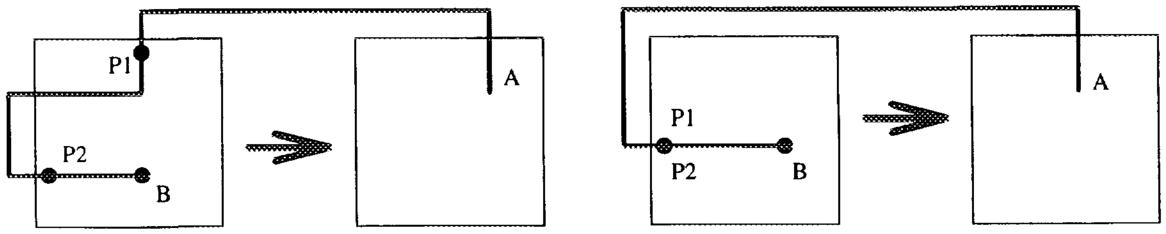

FIG. 10. Case (b): B terminates in the current window.

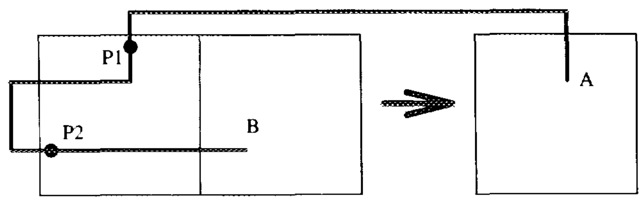

FIG. 11. Case (c): G continues into the next window.

(b) B terminates in the current window (Fig. 10). As in case (a) above, $P1$ and $P2$ exchange information during the LCC, and now $P1$ initiates a future update of A (using the column-stack). Note that $P1$ and $P2$ may coincide.

(c) B continues from the current window into the next window (Fig. 11). Again, $P1$ and $P2$ exchange information during the LCC, but now $P1$ does not initiate an update of A. The continuing end B will update (or initiate an update of) A in some future window. Again, $P1$ and $P2$ may coincide.

(d) The bus ending in B goes to sleep in the current window (Fig. 12). As above, $P1$ and $P2$ have each other's information about A and B after the LCC. $P1$ checks if A will wake up before B. If so, $P1$ initiates an update of A. Otherwise (i.e., if B wakes up before A), A will be updated in the future by B, so no update of A is initiated by $P1$. Note that in this case, A and B cannot wake up together in the same window. In the case that B wakes up before A, B requires no updating since the entire bus segment connecting A to B has already been processed by the window. Here, $P1$ and $P2$ can also coincide.

It remains to verify that the dormant end A is updated once and only once by the above technique. It is easy to see that it is updated at least once, depending on what happens to B. To see that only one update is done, note first that as long as B remains active, no update is done, and this continues until B enters the window in which A wakes up. So now consider all the other cases, and consider the first time that A is updated; we shall see that this is the only update to A. This update falls into categories (a), (b), or (d) above, as follows.

If the first update is done according to case (a), then in the window where the update was initiated, both A and B are dormant and were found to belong to the same bus. Hence, the entire bus section connecting A to B has already been processed by the window, so no further updates to A are possible (B will wake up after A or together with A if an update was initiated).

If the first update was done according to case (b), then there will be no further updates, because the entire bus segment connecting A to B has been processed by the window.

If the first update was done according to case (d), then the entire bus segment between A and $P2$ has been processed and so there will be no further updates (B will not update A because it wakes up after A).

4.6. The Column Stack

The problem can be specified as follows: In some window position, $[i, 0]$ wishes $[0, k]$ to receive message m in a future window w. We are given that for every $0 \leq i < p$, $[i, 0]$ has just one such message to send. This is done as follows:

1. Horizontal buses are configured.
2. All processors in column 0 that have a message to send broadcast the triple (k, w, m) on the horizontal bus.
3. Every $[i, j]$ reads the triple (k, w, m) from the bus. If $j = k$, $[i, j]$ stores m in an internal array $M[1..W]$, where W denotes the total number of window positions, i.e., $M[w] = m$.

The retrieval of information from the column stack is done at the beginning of every new window position w as follows:

1. Vertical buses are configured. Only processors in the first row listen to the bus.
2. Every $[i, j]$ checks $M[w]$. If it is not empty, $[i, j]$ broadcasts $M[w]$ on the bus.
3. Every processor in the first row that had a message will now receive it.

As noted earlier, every awakening segment will receive at most one such message, so in every column, at most one processor will have a message to broadcast.

The size of the internal array $M[1..W]$, as described above, has to be $(n/p)^2$. However, by using relativized window numbers, we can modify it so that we only need $n/p - 1$ memory locations, as follows:

Note that when a message has to be sent to a dormant endpoint, that endpoint will awake in a window that is at most $2n/p - 1$ window positions away. Also, when the current window position advances horizontally one position, the window in which bus segments go to sleep gets closer by *two* window positions (and the last window in a horizontal run has no such "going to sleep" window). Therefore, we can use a circular array of size $n/p - 1$ to store and retrieve the messages. As the window advances, we advance along the circular array, and always retrieve messages from the current position on the array.

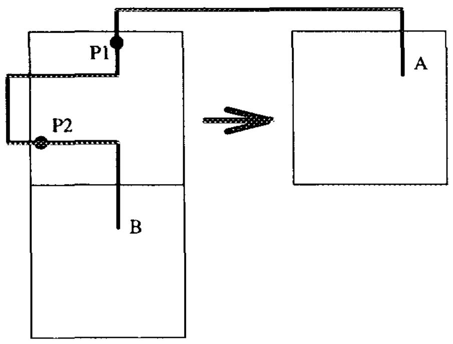

FIG. 12. Case (d): B goes to sleep in the current window.

4.7. The Back-Sweep

At the end of the forward sweep, the simulating window has the following information:

1. For every bus: the final *id* of the bus, the last window which the bus intersected, and the broadcast value (if any) on the bus. Recall that the broadcast value may be ERROR in the case when there were two or more different messages.
2. For every window position, we also have the following information:
   (a) For every bus segment in the window (recall that we are only considering segments that cross window boundaries), both local endpoints of the bus segment know the *id* of the bus that was valid after the LCC in that window.
   (b) Every processor knows whether it had a 0 or 1 in the adjacency matrix of the LCC in that window, and the configuration of the LCC graph.
   (c) All processors that were in charge of entering or awakening bus segments hold the bus-*ids* that were valid before and after the LCC.

The window is now swept in the exact reverse direction. At every window position, the situation is somewhat analogous to the forward sweep, but the actions—mainly the transfer of information—are different. In order to avoid confusion with the forward sweep, we shall add the prefix "b-" to all words which may have ambiguous meaning. Thus, b-first means the first time (in the back-sweep), which is also the last time in the forward-sweep.

At every window position in the back-sweep, some or all of the following may occur:

• A b-first bus is encountered. This means that in the forward-sweep, this was the last window in which that particular bus intersected the window.

• A b-new bus segment is encountered. Such a segment may be a b-first bus, but it may also be a segment of a non-b-first bus, e.g., in case the bus was encountered before (in the back-sweep) and both its ends became b-dormant.

• Some bus segments are b-entering (they were leaving in the forward sweep).

• Some bus segments are b-continuing (they were entering segments in the forward sweep).

• Some bus segments are b-awakening (they were going-to-sleep in the forward sweep).

• Some bus segments are b-going-to-sleep (they were awakening segments in the forward sweep).

Denote by *fid* the final *id* that was given to a bus. For convenience, we assume that the *fid* also codes the broadcast value (if any) on the bus. In case of two or more conflicting broadcast values, this is easily detected by the time we reach the last window (in the forward sweep) which intersected the bus, and we assume that the *fid* also codes a special ERROR value.

The *fid* of a bus is transmitted to the various bus segments in the various window positions in the following manner:

At the b-first time that a bus is encountered, we have the *fid* of the bus. One or both endpoints of such a segment is b-continuing or b-going-to-sleep, so each such endpoint will hold the *fid* of the bus.

At every window position, the following is done:

1. Every b-entering or b-awakening segment holds the *fid* of its bus.
2. The column-stack is activated so that b-new (but not b-first) segments now also hold the *fid* of their bus. Actually, only b-new segments that initiated a column-stack message in the forward-sweep retrieve their *fid* in this manner; the others will be updated in step 3 below.
3. The LCC is configured in the same way as in the forward-sweep. In every (linear) component of the LCC, at least one processor is in charge of either a b-first bus, a b-new bus with an *fid* from the column-stack, a b-entering segment, or a b-awakening segment. All such processors hold the *fid* of the bus and they broadcast it along the linear component of the LCC graph. Now, row and column buses are configured so that processors in the first row and column (in charge of local bus segments) receive the *fid* from their row or column, in a manner similar to the transfer of a component label in the LCC algorithm.

Invariant: *Every processor in charge of a local (to the window) bus segment holds the fid of its bus.*

4. The buses are now configured in the window, and every processor in charge of a local bus segment broadcasts the *fid* along the local segment to all processors in the window which are on that segment.
5. We also take action to guarantee that b-new (but not b-first) segments in b-future windows will receive their *fid*. Recall that segments that were awakening (in the forward-sweep) received information from segments that were not continuing with their window. Although not mentioned earlier for the sake of clarity, we now specify that among the information that was passed was the *id* of the processor in charge of the segment that did not continue. So now, every processor that received information from the column stack in the forward sweep uses the column-stack mechanism to pass the *fid* to the same processor in charge of the b-new segment in the b-future window.

In order to prove the correctness of the algorithm, we need to show that the invariant is true at every window position. In the b-first window position (the very first position in the back-sweep), any local bus segments are b-first segments, so, as explained above, they have the *fid* of the bus as found in the forward-sweep.

Assume that the invariant is true in all window positions up to, and including, window position w; we shall show that it remains true in the b-next position, which we denote by w'. Consider all the local bus segments of w'; they fall into the following categories: b-entering, b-awakening, b-first, b-new (but not b-first).

A b-entering segment has the *fid* because it had it, by the invariant, in the b-previous window position w. A b-awakening segment has its *fid*, because, according to the invariant, it had it in the b-old window position in which it b-went-to-sleep.

Any b-first bus segments have their correct *fid*, as has already been explained. A b-new (but not b-first) segment receives its *fid* either from the column-stack, as explained above, or (if it did not initiate a column-stack message in the forward sweep) from the LCC configuration. Note that the correct *fid* is retrieved from the column-stack because, by the invariant, all bus segments had their *fid*'s in the window in which the information was placed on the column-stack.

The backsweep completes the proof of Theorem 4.1.

4.8. Other Traversal Orders

It should be noted that the above algorithm can be modified to handle other traversal orders, at the expense of a larger memory at each processor. The details, which may be found in [1], are straightforward, and we omit them for brevity. The larger memory is needed because the small circular array used in the column-stack may no longer be sufficient.

5. SELF SIMULATIONS OF MESHES—RN MODEL

For the RN model, we can show the following result which may be suboptimal. In order to simplify the presentation of the main algorithm of this section, we assume that $n = 2^k p$ for some positive integer k. Also, throughout the presentation we will not care for constants, but rather will try to make the asymptotic results as clear as possible. For a table of size $O((n/p)^2)$ which is stored at the local memory of a processor, it is assumed to be accessed in $O(1)$ steps by that processor. Note that this implies a $\Theta(\log n)$-bits word model for the simulating machine. This is not unreasonable, as the simulated and simulating processors are assumed to be of the same size.

THEOREM 5.1. *The simulation of the n-mesh by the p-mesh in the RN model is completed with slowdown $O((n/p)^2 \log n \log(n/p))$. The simulation algorithm uses $O((n/p)^2)$ extra space at each processor of the p-mesh.*

The general method is different from the one used in the LRN model simulation from Section 4. We keep using *windows* for the simulation of p-submeshes of the n-mesh. Similarly, we keep the method of assigning *ids* (representatives, labels) to the bus segments that are discovered, from the set of *ids* of *window boundary* processors. The main difference comes from the way that the bookkeeping is handled. The algorithm here is based on iterations, where the basic iteration step is a connected components algorithm, presented in Section 5.1. The ith iteration ($0 \le i \le \log(n/p)$) collects information on bus segments that are contained in *windows* of size $2^i p \times 2^i p$ of the n-mesh. The algorithm makes use of LRN simulations as subroutines, applying the result from Section 4. Thus, for consistency, the folding mapping is also used here for RN simulations of n-meshes by a p-mesh.

5.1. Connected Components

The RN self simulation makes heavy use of an algorithm for finding the connected components of a sparse graph. The result is stated in the following lemma, and may be of interest in its own right.

LEMMA 5.1. *Let $G = (V, E)$ be an undirected graph having $|V| = n$ nodes and $|E| \le n$ edges. Given the edges of G arbitrarily distributed at the processors of the leftmost column of the LRN n-mesh, such that there is at most one edge stored at each processor, the connected components of G can be determined in $O(\log n)$ steps.*

The restriction on the number of edges may be alleviated: For any $c \ge 1$ (even when c is a function of n), if $|E| \le cn$ then the connected components may be found in $O(c \log n)$ steps. This may be obtained as a corollary of the algorithm, by repeating each step $O(c)$ times.

The connected components algorithm is a variant of the algorithm by Miller *et al.* proving the following result.

LEMMA 5.2 [18, Thm. 4.4]. *Given the adjacency matrix of an undirected graph with n vertices distributed so that element (i, j) of the matrix is stored in processor $[i, j]$ of the LRN n-mesh, the connected components of the graph can be determined in $O(\log n)$ steps.*

However, Lemma 5.1 is not a trivial corollary of Lemma 5.2, since moving the edges to their appropriate places according to the adjacency matrix may require $\Omega(n^{1/2})$ steps, e.g., when a full $n^{1/2}$-submesh of the adjacency matrix is set.

The results from Section 4, Lemma 5.2, and Lemma 5.1 imply the following corollary.

COROLLARY 5.1. 1. *Given the adjacency matrix of an undirected graph with n vertices distributed so that element $[i, j]$ of the matrix is stored in processor $[FOLD(i), FOLD(j)]$ of the LRN reconfigurable p-mesh, the connected components of the graph can be determined in $O((n/p)^2 \log n)$ steps.*

2. *Let $G = (V, E)$ be an undirected graph having $|V| = n$ nodes and $|E| \le n$ edges. Given the edges of G arbitrarily distributed at the processors of the leftmost column of the LRN reconfigurable p-mesh, such that there are at most n/p edges at each processor, the connected components of the graph can be determined in $O((n/p)^2 \log n)$ steps.*

Although Corollary 5.1 will be sufficient for our purpose, we remark that the connected components results are in fact stronger than what is stated. A closer inspection of the algorithms involved reveals that they use buses which are configured along columns and rows only. We thus conclude that the results of Lemmas 5.2 and 5.1 hold also for the HV-RN model. Using also Theorem 3.1, we have that Corollary 5.1 holds also for the HV-RN model.

Proof of *Lemma* 5.1. We use a variant of the n-vertices graph connected components algorithm given in Miller *et al.* [18] which, in turn, is an adaptation of the $O(\log n)$ steps CRCW PRAM algorithm by Shiloach and Vishkin [25]. We assume that the reader is familiar with the Shiloach–Vishkin algorithm, and proceed to describe its implementation on the $n \times n$ reconfigurable mesh. Components are labeled by vertex numbers, where all the processors of the ith row of the mesh "know" the label of the ith vertex and store it in a variable called $LABEL$.

The following initialization procedure is carried.

- Simultaneously for all rows i move the input edge from $[i, 0]$ to $[i, i]$.
- Simultaneously for all columns i, suppose the input edge at $[i, i]$ is (j, k) then it is moved to $[j, i]$. In this step column broadcast is used, so that the information about the edge (j, k) is kept at all column processors.
- At all rows i all processors define a variable $LABEL$ which is initialized to i.

The crucial point at the end of the initialization (and which is different from the Miller *et al.* algorithm) is that for column i there is at most one k so that there is a row j containing an input edge (j, k) at its intersection with the column, namely at $[j, i]$.

The Shiloach–Vishkin algorithm consists of $O(\log n)$ iterations, during which the PRAM algorithm exploits two fundamental operations to update component labels. At each point during the algorithm, the *current* label of vertex i, namely the value of $LABEL$ in row i, may be viewed as a pointer from i to its current label, so we call $parent(i)$ the value of $LABEL$ in row i (e.g., after the init procedure above $parent(i) = i$). In what follows we briefly describe the update operations, along with reconfigurable mesh implementations.

The first operation, called *shortcutting*, consists of every vertex "connecting" itself to its grandparent. This is implemented as follows.

- Use column broadcasts from every diagonal processor $[i, i]$ so that every processor $[k, m]$ knows the current parent of m.
- Simultaneously for all rows i, the value $parent(parent(i))$ is broadcast in row i from processor $[i, parent(i)]$ so that all processors $[i, j]$ know the grandparent of vertex i.

The second operation is called *hooking*, which consists of every vertex k that points to a root (i.e., $parent(k) = parent(parent(k))$) trying to hook the root $parent(k)$ to a vertex i in the same component, such that $parent(i) < parent(k)$. This is accomplished as follows:

- Simultaneously for all processors $[i, j]$: Suppose one of the processors in column j, say $[k, j]$, holds an input edge (k, i). $[i, j]$ uses column broadcast to notify $[k, j]$ the value $parent(i)$. After receiving, processor $[k, j]$ knows both $parent(k)$ and $parent(i)$, and consequently it computes which of them is greater. If $parent(i) < parent(k)$ then $[k, j]$ takes part in the following step.
- By using bus splitting simultaneously in all rows k, $[k, 0]$ receives $parent(i)$ from some processor in row k which holds the input edge (k, i) and for which $parent(k) > parent(i)$. If no value is received by $[k, 0]$ or if $parent(k)$ is not a root then $[k, 0]$ does not transmit in the following step.
- $[k, 0]$ uses row broadcast to notify that $parent(i)$ is the new label of vertex k, so $parent(i)$ (if received) is stored as the new value of $LABEL$ in all the processors of row k.

Obviously the implementations of the shortcutting, hooking, and initializing operations take $O(1)$ steps. Therefore the whole algorithm takes $O(\log n)$ steps. ■

5.2. The Algorithm

Back to the proof of Theorem 5.1, the algorithm consists of three main phases. We first sketch these phases informally, then proceed to give the details.

One of the notions we use extensively is that of a *window*. A submesh of the n-mesh is called a *window of size m*, or an m-window, when it is one of the $(n/m)^2$ m-submeshes in a partitioned n-mesh. Thus a window of size m is always "aligned to a boundary of size m." The *boundary* of a window consists of all the processors from which edges exit the window (or from which there is a direction with no edge at all). The boundary of a window is composed out of four *facets* in the intuitive way (each corner processor belongs to two facets).

The algorithm consists of three basic phases, as described below.

ALGORITHM RN Simulation (Basic Phases).

Phase (1) This phase consists of $\log(n/p)$ iterations, called *levels*, which gather configuration information and construct a spanning forest over the set of buses. During iteration 0 (level zero), each p-window is simulated by the given p-mesh and a *representative* for each bus segment is elected. During iteration $i > 0$ (level i) all windows of size $2^i p$ are considered, deducing information for each of them out of the information of its four composing subwindows. This is achieved by computing the connected components of a graph which represents the bus configuration.

Phase (2) In this phase the data gathered in Phase (1) is used to associate a message (or an error indication) with each processor which is a bus representative in some p-window.

Phase (3) The p-mesh is moved through all p-windows. Each p-window is simulated for a single step, in which the representative of each of its buses transmits the appropriate message (similar to the "back-sweep" of the LRN simulation algorithm).

We now turn to the detailed description of the phases and the data management. Our goal is to show that Phase (1) terminates in $O((n/p)^2 \log n \log(n/p))$ steps, phase (2) terminates in $O((n/p)^2 \log(n/p))$ steps, and phase (3) takes only $O((n/p)^2)$ steps. We will also consider the space requirements: if we define a single unit of space by the number of bits (*+sizeof(id)*) required by the algorithm at each processor of the original mesh, then the simulation requires $O((n/p)^2)$ space units at each processor, which is optimal up to a constant factor.

5.3. Phase (1): Gathering Information

We view this phase as $\log(n/p) + 1$ iterations with a growing window size. Each iteration is called a *level*, where level i of the algorithm, $0 \le i \le \log(n/p)$, processes information for windows of size $2^i p$. Level $i + 1$ uses the information gathered in level i to process larger windows of size $2^{i+1}p$, each consisting of four windows of size $2^i p$, etc.

5.3.1. Level Zero

For $i = 0$ (level zero), windows of size p are simulated one by one by the p-mesh in some arbitrary order. Since buses which do not cross the window boundaries are simulated in a single step, we assume w.l.o.g. that all buses do cross window boundaries.

Similar to the algorithm for the LRN simulation, some *id* is chosen for each bus. This *id* is referred to as the *representative* processor of that bus in the simulated window. It is going "to represent" this window in subsequent levels. Thus the representative stores any message which is heard on the bus and which originated from a speaker of the bus in its window. It also stores the state of the bus.

We pick representatives from *ids* of window boundary processors only. For each bus we pick one of the boundary processors when the bus actually exits the window. There may be up to $4(p - 1)$ such processors. Picking a representative for all buses of any window may be done in $O(\log p)$ steps by applying a "binary search" on the address space of the boundary processors.

There is one complication, however. Corner processors may represent two different buses (since two edges leave the window in the corner). We thus refer to a corner processor as two different logical processors. For this purpose, we add a least significant bit to the *id* of corner processors, thus having two different (but successive) new *id*'s. Suppose the corner is an upper-left one. Then one of the *id*'s belonging to that corner will be a candidate for representing the bus which leaves the corner going up (if exists) and the other one will be a candidate for representing the bus which leaves the corner going left (if exists). If the buses are the same one, i.e., they join inside the window, then one of the candidates will be elected according to the above mentioned algorithm.

In view of the above discussion, in the rest of this section we refer to each corner processor as two different processors. In the mappings to be described later, mapping a corner processor means mapping separately its two different *id*'s. Moreover, as each of the *id*s refers to an edge exiting the window, we consider each of the *id*'s as belonging to a different window facet according to the direction of the corresponding edge. Finally, we note that using this terminology, the boundary of a $2^i p$-window consists of $4 \cdot 2^i p$ processors.

5.3.2. Higher Levels

Moving to a higher level, we find new representatives for the buses which cross the boundaries of the composing subwindows. Let us first sketch the general idea, and then give the details of the data movement.

Suppose we are given the bus representatives in four subwindows A, B, C, and D, each of size $2^{i-1}p$. A, B, C and D compose a window W of size $2^i p$, as depicted in Fig. 13. The computation of the representatives of W consists of the fact that there are at most $4 \cdot 2^{i-1}p$ different buses at each of the sub-windows. We define a new graph G_W, called the *input graph* whose nodes are all the bus representatives in the subwindows. Let a be an *id* of a node of G_W. So

$$a = [r_1 2^{i-1}p + r_2, c_1 2^{i-1}p + c_2],$$

where $0 \le r_2, c_2 < 2^{i-1}p$ and at least one of the following equalities holds:

$$r_2 = 0; \quad c_2 = 0; \quad r_2 = 2^{i-1}p - 1; \quad c_2 = 2^{i-1}p - 1.$$

An edge (a, b) of G_W represents two bus segments represented by a and b in neighboring subwindows, and which are found to be connected at their joint boundary. Note that any bus leaving a subwindow in the direction of one of the other subwindows indicates such an edge. In summary, G_W has up to $16 \cdot 2^{i-1}p$ nodes and $4 \cdot 2^{i-1}p$ edges.[8]

The crucial point in the validity of the algorithm is that the connected components of G_W represent connected buses in W, just like the nodes of G_W represent bus segments in A, B, C and D. Thus, choosing representatives for connected components in W is the next step in moving to a higher level. If data is stored appropriately, this can be done by using Corollary 5.1 in $O(2^{2i}(i + \log p))$ steps by the p-mesh. In the following sections, we show that the output of the connected components algorithm of the ith level is readily available as input to the $(i + 1)$th level (i.e., it is stored in the appropriate processors), after some initializing operations which take only $O(2^{2i})$ steps.

Let $T(n)$ denote the time to compute the representatives, as described above, for the global n-mesh, i.e. to complete Phase (1). Level i involves connected compo-

[8] A closer inspection of the algorithm reveals that the constants of this estimate may be reduced considerably. We shall not consider the details here.

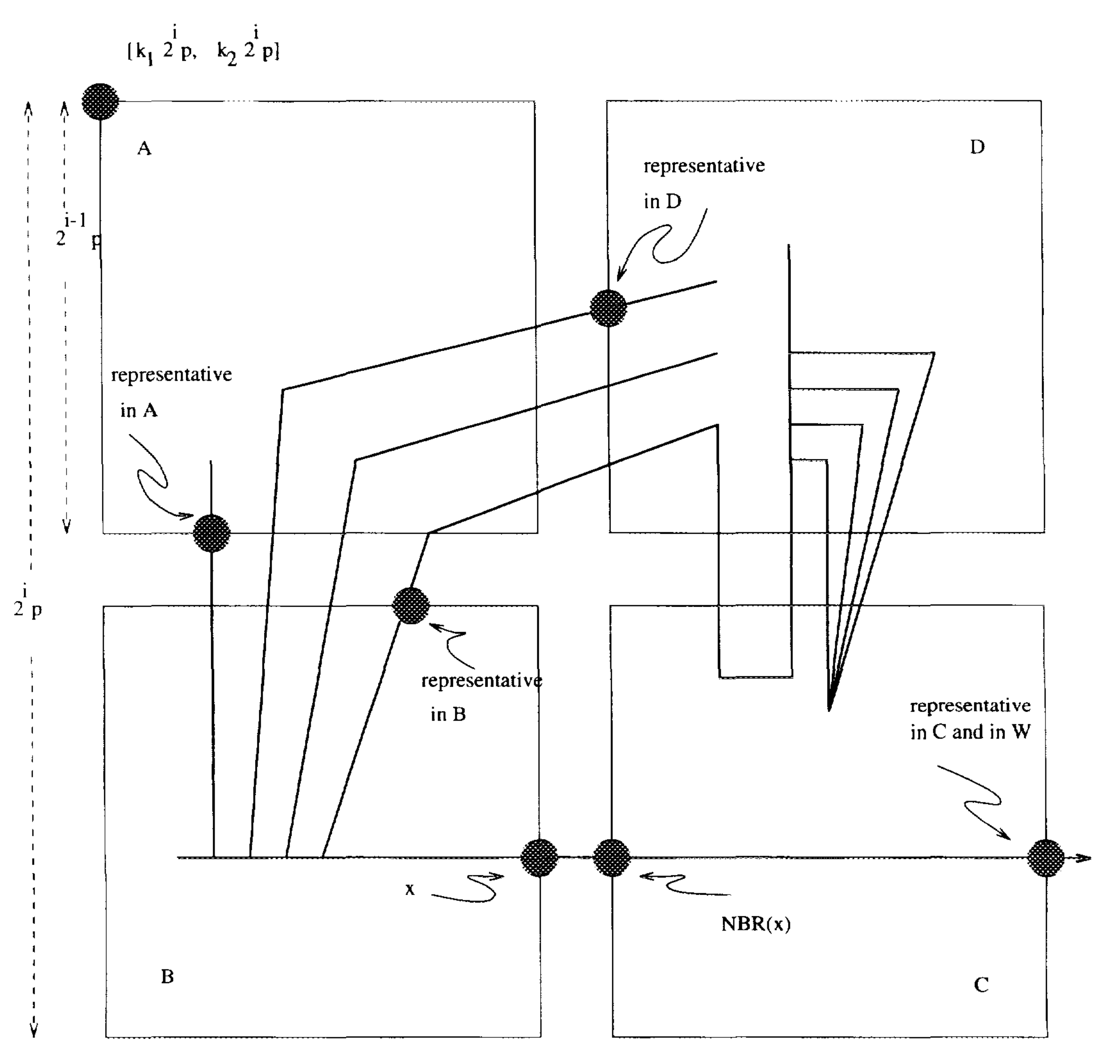

FIG. 13. Bus segments in four subwindows A, B, C, and D compose a larger segment in W.

nents and initializing operations for $(n/2^ip)^2$ windows of size 2^ip. Thus

$$\begin{aligned}T(n) &= \sum_{i=0}^{\log(n/p)} \left(\frac{n}{2^ip}\right)^2 \cdot (2^{2i} + 2^{2i}(i + \log p)) \\ &= O((n/p)^2 \log n \log(n/p)).\end{aligned}$$

We now proceed to show the consistency of locations for the output of level i and the input for level $i + 1$.

5.3.3. Mapping Representatives into the p-Mesh

The mesh on which the connected components algorithm is performed in level i is of size $16 \cdot 2^{i-1}p$. We refer to it as the *virtual mesh*. In order to make the (LRN) simulation of the virtual mesh by the given p-mesh applicable, we define a function $\Upsilon_i(\,)$. This function maps the facets of a window of size 2^ip and the facets of his subwindows, i.e., the $16 \cdot 2^{i-1}p$ nodes of the input graph, one-to-one into the rows of the virtual mesh, i.e., into $\{0, \cdots, 16 \cdot 2^{i-1}p - 1\}$.

As an exception, Υ_0 maps the facets of a window of size p (which has no subwindows), as follows:

$$\Upsilon_0([r, c]) = \begin{cases} FOLD(r) & \text{for a column facet} \\ FOLD(c) & \text{for a row facet} \end{cases}.$$

For $i > 0$, there are eight "generalized" facets to be mapped, called *g-facets*, as is depicted in Fig. 14. Although we do not explicitly specify $\Upsilon_i(\,)$, we require that it maps two adjacent processors of the same g-facet successively. Clearly, such a mapping exists for every i and is easily computed in $O(1)$ steps. Note that the requirement implies that each row g-facet is mapped into a consecutive subrange of $\{0, \cdots, 16 \cdot 2^{i-1}p - 1\}$ according to the processors column indices, where the first number in this range is a multiple of $2p$. A similar observation holds for column g-facets and their processors row indices.

Virtual meshes change in different levels. In each level, the corresponding virtual mesh is simulated by the given p-mesh. We thus map the virtual mesh (for $i > 0$) into the

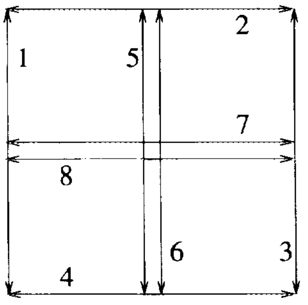

FIG. 14. Eight "generalized" facets (gfacets) of a window and its four composing subwindows.

p-mesh, using $FOLD(\ )$. In particular, this maps the rows of the virtual mesh into the rows of the p-mesh. The joint mapping $\Upsilon = FOLD \circ \Upsilon_i$ maps the facets of 2^ip-windows and their four sub-windows into the rows of the p-mesh. Observe, however, that the mapping Υ does not depend on the level i. In other words, a processor in the facet of some window and which is also in a facet of a larger window will be mapped by Υ to the same row of the p-mesh. This is due to the property of the functions Υ_i, the operation of $FOLD$, and the fact that facets are of size which is a multiplicity of $2p$. This feature will become important later when the output of the connected component algorithm in one level is to become the input of this algorithm at the next level.

5.3.4. Representative Election

As before, we assume the connected components algorithm to compute bus representatives for bus segments in a window W of size 2^ip. The algorithm uses the *id*'s of the nodes of the input graph G_W as indices to the rows of the virtual mesh, by using the mapping described in Section 5.3.3. Thus, we cannot allow the nodes of G_W to have arbitrary *id*'s. We solve this by restricting the set of possible *id*s to the set of boundary processors of W and of its four composing sub-windows, i.e., the processors which compose their facets. This is done as follows.

Recall that the representative of a bus is, in fact, an *id* of a processor through which the bus passes. Also, a bus has a representative corresponding to a certain window only if it exits this window. We conclude that we can always choose the representative of a bus with respect to a certain window to be an *id* of a processor on the boundary of that window. Clearly, this is the case for windows of size p. For larger windows, however, we have to change the connected components algorithm inductively. Basically, while choosing a representative out of a set of candidates, we prefer a candidate which resides on the boundary of the largest window. Note that if we do not include this change at a certain level, then we might run into trouble at some higher level, with the following situation: suppose r is the elected representative of a bus segment s_r in a 2^ip-window W_{2^ip}, and suppose W_{2^ip} is contained in a 2^jp-window $(j > i)$ W_{2^jp}, and suppose s_r exits W_{2^jp} via a single node $\hat{r}$ in the boundary of W_{2^jp} which is also in the boundary of W_{2^ip}, and suppose $r \neq \hat{r}$. Since we choose r and not $\hat{r}$ at level i, we arrive at level j with $\hat{r}$ not in the pool of representative candidates for the bus containing s_r in W_{2^jp}. Hence we are forced to choose a representative for this bus (say r) which is not on the boundary of W_{2^jp}, contradicting our previous discussion.

Consider again the choice of representatives of buses in a p-window W_p. W_p is a subwindow of larger windows, let W_{2p} denote the window of size $2p$ containing W_p. Two of the facets of W_p f_1, f_2 take part in the boundary of W_{2p}. For larger sized windows, either both f_1, f_2 take part in border lines, or just one of them (say, f_1 in this case), or none. In choosing representatives for buses in W_p we revise the algorithm so that *id*s from f_1 are chosen with priority over others, *id*s from f_2 are next, others chosen only for buses which do not intersect f_1 and f_2. This adds only two steps to the representative choice in W_p, given that each borderline processor knows the size of the largest window containing W_p for which it is still a borderline processor.

Let $x = [r, c]$ be a borderline processor. Suppose

$$r = r'2^{k_1}p; \quad c = c'2^{k_2}p; \quad r + 1 = r''2^{k_3}p; \quad c + 1 = c''2^{k_4}p;$$

where r', c', r'' and c'' are odd positive integers. If $k = \max\{k_1, k_2, k_3, k_4\}$ then the largest window W_x containing x in its boundary is of size

$$\begin{cases} n & \text{if } c = 0 \text{ or } r = 0 \\ 2^kp & \text{otherwise} \end{cases}.$$

Computing the above for three other processors, one from each of the other three facets of W_x, x may decide in $O(1)$ steps his status as a representative candidate against other borderline processors of W_x. This computation is taken only once at level zero.

Next, we have to make a similar revision of the connected components algorithm, so that the choice of representatives "prefers" *id*s of processors on the boundary of larger windows. Recall that the connected components algorithm computes representatives for a 2^ip-window W out of the representatives of its subwindows. Let x be a node in the input graph, where (by induction hypothesis) x is a boundary processor of one of the subwindows, say A, and x is in a facet of W taking part in the boundary of a window which is much larger than W. So according to our policy, x is a preferred representative of its bus segment in W. This is done as follows. By the end of the ith level connected components algorithm all processors in row $\Upsilon_i(x)$ of the virtual mesh know $LABEL(x)$ which was chosen as the representative of the component of x. Suppose $LABEL(x) \neq x$. Processor $[\Upsilon_i(x), \Upsilon_i(LABEL(x))]$ uses bus splitting in column $\Gamma_i(LABEL(x))$ in order to inform processor $[\Upsilon_i(LABEL(x)), \Upsilon_i(LABEL(x))]$ about the higher priority of x. Note that the bus-splitting method guarantees that only one candidate having higher priority than $LABEL(x)$ succeeds, and let us assume it is x. $[\Upsilon_i(LABEL(x)), \Upsilon_i(LABEL(x))]$ chooses x to replace $LABEL(x)$ as the new representative of the bus. It then informs the processors in its column of its decision, which, in turn, is distributed to all rows $\Upsilon_i(y)$ for which $LABEL(x)$ was the previous elected representative, so that the component is relabeled to x. The whole process takes constant number of steps (on the virtual mesh).

5.3.5. *Message Collection and Error Detection*

We now turn to the message transmission on the bus and the detection of bus error states.

Consider level i, in which windows of size $2^i p$ are considered. Let $LABEL(x)$ be chosen as a representative of a bus segment s, where x is a representative of a partial segment s_x in one of the subwindows. If $i = 1$, so that s_x is a segment contained in a single p-window, then x "knows" whether there was a message transmission during the simulated step by one of the processors in s_x (see Section 5.3.1). This information is stored in a row of the level-zero virtual mesh, namely, $\Upsilon_0(x)$. By the property of Υ, we know that this row and $\Upsilon_1(x)$ are mapped to the same row of the p-mesh. Thus, this information is readily available to the virtual processors of the row $\Upsilon_1(x)$ in the virtual mesh of level 1. Column $\Upsilon_1(x)$ is used to move the message to processor $[\Upsilon_1(LABEL(x)), \Upsilon_1(x)]$. Now all messages transmitted on subsegments reside in processors of the row of the virtual mesh to which the new representative is mapped. Next, row $\Upsilon_1(LABEL(x))$ is connected, and all its processors transmit the stored messages simultaneously, so that all processors of the row can either store a message or detect an error state of the bus (indicating an error state of the corresponding simulated bus segment s).

Similarly, if $i > 1$ then we assume inductively that the information about transmission in subsegments is stored at the rows of the virtual mesh of level $i - 1$. As before, these rows and the corresponding rows of the virtual mesh of level i are mapped by Υ to the same rows of the p-mesh, hence the same process may be carried.

5.3.6. *Data Types and Storage*

There are four types of data items that are produced, stored and fetched during Phase (1):

Representatives. For each x which was elected as a representative in a previous level, the representative of x is elected during the connected components algorithm and is stored at row $\Upsilon_i(x)$ of the virtual mesh as $LABEL(x)$. Thus $LABEL(x)$ is stored at row $\Upsilon(x)$ of the p-mesh. The representatives are temporary data existing during Phase (1) only.

Edges. This is an edge (x, y) of the graph G_W as defined in Section 5.3.2 for a window W. These are formed at the end of level $i - 1$ and are the input to the connected components algorithm of level i. This is a temporary data for "internal use" of Phase (1).

Pointers. For each pair, x and its chosen representative $LABEL(x)$, we define a pointer $\langle x \rightarrow LABEL(x)\rangle$. This pointer is duplicated and stored by all the processors of row $\Upsilon(x)$. The collection of all pointers is a spanning forest of all representatives, in which there is a single tree for each bus. This forest is the output of Phase (1) and the input to Phase (2).

Note that there is at most one pointer $\langle x \rightarrow LABEL(x)\rangle$ for each representative x. Recall that there are initially at most $2n^2/p$ representatives which are evenly distributed by Υ among the rows of the p-mesh. Thus the total number of pointers that are stored at each row is $O((n/p)^2)$.

Ancient Parent. Finally, a chosen representative which is global to its bus (rather than to some partial segment) is called an ancient parent (because it is a root of a tree in the pointer forest). Let x be an ancient parent. x is stored, together with the message that was transmitted on that bus (or with an error indication) by all the processors at row $\Upsilon(x)$. An ancient parent is easily identified during the connected components algorithm when it is elected as a representative of some component in a certain window: either that it is not on the boundary of the window, or it is on the boundary but the corresponding bus does not leave the window.

5.3.7. *Data Movement: The Way It Works*

We now describe the substeps taken by level i of Phase (1) for a single window, say W (of size $2^i p$), together with the bookkeeping that is performed.

We assume that at the beginning of the phase and for each x which is a boundary processor of the subwindows of W, row $\Upsilon(x)$ of the p-mesh knows $LABEL(x)$. This is obvious at level 1, after the bus algorithm for representation election that is executed during level zero. Suppose this holds at the beginning of level i. We shall see that it holds at the beginning of the next level, too.

Knowing $LABEL(x)$, row $\Gamma(x)$ creates a pointer $\langle x \rightarrow LABEL(x)\rangle$ which is stored at all its processors. Since the boundary processors are evenly distributed among the rows, there are at most $O(2^i)$ such pointers that are created at each row of the p-mesh.

Let x be a boundary processor of a sub-window of W, and suppose that x is in a facet facing a facet of another sub-window of W. We define $NBR(x)$ as the *id* of the processor which is a neighbor of x in the neighboring facet (see Fig. 13).

The property of Υ (see Section 5.3.3) implies that for every x, $\Upsilon(NBR(x)) = \Upsilon(x)$. Hence the processors of the same row of the p-mesh, namely $\Upsilon(x)$, know both $LABEL(x)$ and $LABEL(NBR(x))$. Row $\Upsilon(x)$ composes a new edge $(LABEL(x), LABEL(NBR(x)))$ and stores it at its leftmost processor.

This process takes $O(1)$ (row) steps per edge, and there are $O(2^i)$ edges stored at each (leftmost processor of each) row of the p-mesh.

Conceptually, here is where the virtual mesh of size $16 \cdot 2^{i-1}p$ and the mapping Υ_{i-1} into its rows, are replaced by the virtual mesh of size $16 \cdot 2^{i}p$ and the mapping Υ_i into its row. Again, since the mapping Υ remains the same, so edges of the input graph to the level i connected components algorithm do not have to switch locations in the p-mesh.

After the neighbors connecting edges are produced and stored, the connected component algorithm of level i can be executed. The algorithm (including the change as described in Section 5.3.4), takes $O(2^{2i} \log(2^i p))$ steps. It stores the representatives at the correct places: For each y in the boundary of W, row $\Upsilon(y)$ knows $LABEL(y)$, as was required at the beginning of this subsection.

5.4. Phase (2): Simple Tree Contraction

Phase (1) constructs trees of maximal height $\log(n/p)$. The tree information is saved by pointers (tree edges) $\langle x \rightarrow LABEL(x)\rangle$ where x is a node in the tree and $LABEL(x)$ is its parent. The total of all pointers is replicated and saved at each column of the p-mesh.

All nodes of the trees are processors of the simulated n-mesh which are boundary processors of p-windows. We want to evaluate for each such processor its ancient parent in the pointers forest. It is easy to split the total number of n^2/p such boundary processors into p sets; e.g., let the lth set, $0 \le l \le p - 1$, consist of all processors with column coordinate equal to l div n/p. The lth column is in charge of evaluating the ancient father for each of the processors in the lth set. The whole column is connected as a single bus and works independently and sequentially on the evaluation of his set. This process is straightforward, assuming the processors store the pointers in a way which gives them constant time access to the table of $O((n/p)^2)$ pointers that is stored by each of them. Note that a p-mesh processor $y = [r, c]$ stores a (single) pointer $\langle x \rightarrow LABEL(x)\rangle$ for each of the $O((n/p)^2)$ xs where $r = \Upsilon(x)$. Hence the pointers table at each processor, as a table of child-parent edges, may be accessed using the *id*s of the children as the entry indices.

Finally, after $O(\log(n/p))$ steps for each p-window boundary processor z, the processor storing the pointer to the ancient parent of z in the pointers forest transmits the *id* of the ancient parent together with the attached message (or an error indication) $msg(z)$. This information is stored by all column processors as an *info-pair* $\langle z, msg(z)\rangle$ in a $(O((n/p)^2)$-sized) table. This table is used during Phase (3).

5.5. Phase (3): Back-Sweep

At this final phase, the information that was collected and arranged at Phases (1) and (2) is dispersed to the bus segments at the p-windows. Note that at the end of Phase (2) all representative information, namely the info-pairs, are replicated and stored at each row of the p-mesh. During Phase (3) the p-mesh simulates the p-windows in some arbitrary order, as follows. For each p-window, for each row y, there are exactly four boundary processors z for which $y = \Upsilon(z)$. The row is connected to collectively find the information about them: for each $z = [r, c]$ the info-pair $\langle z, msg(z)\rangle$ is fetched by row $\Upsilon(z)$ and is moved to $[FOLD(r), FOLD(c)]$. All this takes $O(1)$ steps.

In one additional step, the p-mesh is connected to form the exact configuration of the simulated p-window. For a boundary processor z, if z was elected as the representative of its bus segment in this window then $[FOLD(r), FOLD(c)]$ transmits $msg(z)$.

This completes the proof of Theorem 5.1.

6. CONCLUDING REMARKS

In this work, we gave the first results for the simulation of large reconfiguring meshes by smaller ones. We believe that efficient self simulation results are essential for any parallel model of computation. Our algorithms may also be applied to other variants of the reconfigurable mesh model that are discussed in the literature. For example the optimal LRN self simulation implies optimal self simulation of a model called *tree-RN*. The tree-RN model is similar to the general RN model, except that no cycles are allowed in the configuration. Notice that this is a global restriction on the set of configurations, whereas the restrictions which differentiate the HV-RN, LRN, and the RN models are applied locally at every switch. In contrast to the tree-RN model, it seems that in the general RN model an additional polylogarithmic factor is inherent in the complexity of the self simulation algorithm, since otherwise much faster connected components algorithms must be found. Yet, there may exist a faster algorithm than the one given for the RN model. This and the related lower bound problems are the subjects for futher research.

REFERENCES

1. Y. Ben-Asher, D. Gordon, and A. Schuster, Optimal simulations in reconfigurable arrays. *Proc. 1st European Symposium on Algorithms*, Sep. 1993. (Also TR #716, CS, Technion.)
2. Y. Ben-Asher, D. Peleg, R. Ramaswami, and A. Schuster, The power of reconfiguration. *J. Parallel Distrib. Comput.* **13**(2), Oct. 1991. [Special Issue on Massively Parallel Computation]
3. Y. Ben-Asher, D. Peleg, and A. Schuster, The complexity of reconfiguring networks models. *Proc. of the Israel Symposium on the Theory of Computing and Systems*, May 1992; *Inform. and Comput.*, to appear.
4. Y. Ben-Asher and A. Schuster, The bus-usage method for the analysis of reconfiguring networks algorithms. *Proc. of the Intl. Parallel Processing Symp.*, Beverly Hills, Mar. 1992, pp. 146–149; *J. Algorithms*, to appear.
5. S. Cook, C. Dwork, and R. Reischuk, Upper and lower bounds for parallel random access machines without simultaneous writes. *SIAM J. Comput.* 15(1) (1986).
6. E. Hao, P. D. MacKenzie, and Q. F. Stout, Selection in $O(\log n)$ time on the reconfigurable mesh. *4th Symp. on Frontiers of Massively Parallel Computing*, 1992.

7. J. Jang, H. Park, and V. K. Prasanna, A fast algorithm for computing histogram on reconfigurable mesh. *Proc. Frontiers of Massively Parallel Computation,* 1992, pp. 244–251.
8. J. Jang, H. Park, and V. K. Prasanna, An optimal multiplication algorithm on reconfigurable mesh. *Proc. Symp. on Parallel and Distributed Processing,* 1992, pp. 381–391.
9. J. Jang and V. K. Prasanna, A fast sorting algorithm on higher dimensional reconfigurable mesh. *26th Conf. on Information Sciences and Systems,* 1992.
10. J. Jang and V. K. Prasanna, An optimal sorting algorithm on reconfigurable mesh. *Proc. 6th Inter. Parallel Processing Symp.*, Mar. 1992, pp. 130–137.
11. J. F. Jenq and S. Sahni, Reconfigurable mesh algorithms for the hough transform. *Proc. 20th Intl. Conference on Parallel Processing,* Chicago, Aug. 1991.
12. M. Kaufmann, J. F. Sibeyn, and R. Raman, Randomized routing on meshes with buses. *Proc. 1st European Symp. on Algorithms,* Sep. 1993.
13. F. T. Leighton, *Introduction to Parallel Algorithms and Architectures.* Morgan Kaufmann, San Mateo, CA, 1991.
14. H. Li and M. Maresca, Polymorphic-torus architecture for computer vision. *IEEE Trans. Pattern Anal. Mach. Intell.* **11**(3), 233–243 (1989).
15. H. Li and M. Maresca, Polymorphic-torus network. *IEEE Trans. Comput.* **38**(9), 1345—1351 (1989).
16. M. Maresca and H. Li, Connection autonomy in SIMD computers: A VLSI implementation. *J. Parallel Distrib. Comput.* **7**(2), 302–320 (1989).
17. M. Maresca and H. Li, Virtual parallelism support in reconfigurable processor arrays. Unpublished manuscript, 1993.
18. R. Miller, V. K. Prasanna-Kumar, D. I. Reisis, and Q. F. Stout, Parallel computations on reconfigurable meshes. *IEEE Trans. Comput.* **42**(6), 678–692 (June 1993).
19. K. Nakano, T. Masuzawa, and N. Tokura, A sub-logarithmic time sorting algorithm on a reconfigurable array. *IEICE Trans. E* **74**(11), 3894–3901 (Nov. 1991).
20. M. Nigam and S. Sahni, Sorting n numbers on $n \times n$ reconfigurable meshes with buses. *Proc. of Intl. Parallel Processing Symposium,* Apr. 1993, pp. 174–181.
21. H. Park, V. K. Prasanna, and J. W. Jang, Fast arithmetic on reconfigurable meshes. In *Proc. Intl. Parallel Processing Symp.*, Aug. 1993, pp. III-236–243.
22. R. Vaidyanathan and J. L. Trahan, Optimal simulation of multidimensional reconfigurable meshes by two-dimensional reconfigurable meshes. *Inform. Process. Lett.* **47,** 267–273 (Oct. 1993).
23. S. Rajasekaran, Mesh connected computers with fixed and reconfigurable buses: Packet routing, sorting and selection. *Proc. 1st European Symp. on Algorithms,* Sep. 1993.
24. A. Schuster, Dynamic Reconfiguring Networks for Parallel Computers: Algorithms and Complexity Bounds. Ph.D. thesis, Hebrew University, Jerusalem, Israel, Aug. 1991.
25. Y. Shiloach and U. Vishkin, An $O(\log N)$ parallel connectivity algorithm. *J. Algorithms* **3,** 57–67 (1982).
26. B. Wang and G. Chen, Constant time algorithms for the transitive closure and some related graph problems on processor arrays with reconfigurable bus systems. *IEEE Trans. Parallel Distrib. Systems* **1**(4), 500–507 (1990).
27. B. F. Wang, Configurational computation: A new algorithm design strategy on processor arrays with reconfigurable bus systems. Ph.D. thesis, National Taiwan University, June 1991.
28. C. C. Weems, S. P. Levitan, A. R. Hanson, and E. M. Riseman, The image understanding architecture. *Internat. J. Comput. Vision* **2,** 251–282 (1989).
29. C. C. Weems, R. Deepak, D. B. Shu, and G. Nash, Reconfiguration in the low and intermediate levels of the image understanding architecture. Technical Report COINS TR 90-10, Univ. of Massachusetts at Amherst, Feb. 1990.

Received September 10, 1993; revised July 5, 1994; accepted January 16, 1995

YOSI BEN-ASHER received his Ph.D. degree in computer science from the Hebrew University in 1989. Currently he is a lecturer in the Department of Mathematics and CS, University of Haifa. His research interests include parallel systems, parallel languages, and reconfigurable networks. Dr. Ben-Asher is currently developing the PCoLAN project, a system implementing a mixed model of shared memory and message passing on local area networks.

DAN GORDON received the B.Sc. and M.Sc. degrees in mathematics from the Hebrew University, and the D.Sc. degree in mathematics from the Technion, Israel Institute of Technology, in 1976. He is currently a senior lecturer of computer science at the University of Haifa, and has also taught at various other universities. His research interests include reconfigurable processor arrays, data structures and algorithms, and computer graphics. Dr. Gordon is a member of the ACM and EATCS (European Association for Theoretical Computer Science).

ASSAF SCHUSTER received the B.A., M.A., and Ph.D. degrees in computer science from the Hebrew University of Jerusalem (graduated 1991). He is currently a lecturer at the Technion, Israel Institute of Technology. His main interests include parallel and distributed computation, networks and routing algorithms, optical computation and communication, dynamically reconfigurable architectures, distributed shared memory, and parallel computing on clusters of workstations.