Optimal Simulations in Reconfigurable Arrays

Yosi Ben-Asher • Dan Gordon • Assaf Schuster

Abstract

There are several reconfiguring models of parallel computation that are considered in the published literature, depending on their switching capabilities. Three of the more popular ones are the Horizontal-Vertical Reconfigurable Mesh (HV-RN model) [LM89a, LM89b, ML88], the Linear Reconfigurable Mesh (LRN model) [MPRS87, BPRS91], and the General Reconfigurable Mesh (RN model) [Wan91, Sch91].

Can these reconfigurable models (in particular two dimensional arrays) be the basis for the design of massively parallel computers? Perhaps the most basic related question is: given an algorithm which is designed for a large reconfigurable mesh, can it be executed efficiently on a smaller reconfigurable mesh? We call it the simulation problem. Despite the fact that machines already exist and a large number of efficient algorithms are known for reconfigurable arrays, this question was left open for all reconfiguring models of computation.

In this work we give several positive answers to the simulation problem. We have the following results for the simulation of an \( n \times n \) reconfiguring mesh by the \( p \times p \) reconfiguring mesh.

1. Using standard techniques the simulation in the HV-RN model is carried optimally with slowdown \( \Theta((n/p)^2) \).

2. Although using the same method fails for the LRN model, a novel technique is developed, achieving asymptotically optimal simulation of LRN meshes, with slowdown \( \Theta((n/p)^2) \).

3. Efficient simulations in the RN model is achieved too, paying an extra factor which is polylogarithmic in the size of the simulated mesh. The slowdown achieved is \( O((n/p)^2 \log n \log(n/p)) \).

Although the simulation algorithms are complex and require extensive bookkeeping operations, the required space is asymptotically optimal.

In addition to the above we give a bunch of inter-model simulation results. These results give some additional insight into the gap in the computational power of the various reconfiguring models [Sch91, BPS91].

---

*Department of Mathematics and Computer Science, University of Haifa, Haifa 31905, Israel. E-mail: yosi@mathcs2.haifa.ac.il

†Department of Mathematics and Computer Science, University of Haifa, Haifa 31905, Israel. E-mail: gordon@mathcs2.haifa.ac.il

‡(CONTACT AUTHOR:) Department of Computer Science, Technion, Technion City, Haifa 32000, Israel. E-mail: assafs@cs.technion.ac.il
1 Introduction

Most currently available parallel computers are based on a collection of processors and switching nodes connected by a fast interconnection network. While traditional designs are based on a network of fixed topology, the concept of a reconfigurable network (RN) has been around for a while. However, only recently did this idea become the focus of much interest, due to technological developments that have made it more viable.

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 on the RN, each switch of the network fixes its local configuration by partitioning its collection of edges into some allowable combination of subsets. Two adjacent edges grouped by a switch into the same pair are viewed as (hardware) connected, and the processor may choose to listen to any incoming or passing message.

1.1 Previous and Related Work

There are several different reconfiguring models that are considered in the published literature, depending on their switching capabilities. In the major part of this work we consider arrays (actually, meshes) operating in three of the more popular models (a formal definition is given later).

Horizontal-Vertical Reconfigurable Mesh (HV-RN model): Also known in the literature as the Polymorphic Processor Array. It consists of a two-dimensional array of processors connected to a bus network. The configuration of the bus network can be dynamically changed so that subbuses of different lengths are formed along rows and columns. A VLSI chip called YUPPIE (Yorktown Ultra Parallel Polymorphic Image Engine) has been implemented to demonstrate the feasibility of the model [LM89a, LM89b, ML89].

Linear Reconfigurable Mesh (LRN model): consists of an $n \times n$ array of processors overlaid with a reconfigurable bus system [MPRS87]. Buses that are formed dynamically at each step consist of any connected paths of network edges. Many recent results present constant time algorithms on the linear reconfigurable mesh. Nakano, Masuzawa and Tokura present a constant time algorithm for the addition of $n$ binary bits on a $n \times \log^2 n$ mesh [NMT91]. In [BPRS91] it is shown how to sort $n$ items in constant time using (the lower-bound of) $n^2$ VLSI area and $n^{1+c}$ processors. The paper also gives a method for “automatic compilation” of reconfigurable mesh programs, and suggests a guided optic implementation for reconfigurable meshes. There are many other recent results, e.g. [BS92a, JS91, EW91, Lin91, BS92b, PJ91].

In a series of papers, originating back to 1974, Rothstein et. al. investigated the Bus Automata
long-distance communication. Many efficient algorithms, complexity bounds, and the connection of BA to other scientific areas are considered in Rothstein's works.

General Reconfigurable Mesh (RN model): is a reconfigurable two-dimensional array, where the buses are any partition of the network into edge-disjoint subgraphs. Chen and Wang et. al. present several constant time algorithms on the general reconfigurable mesh, summarized in [Wan91].

Another, much more powerful reconfiguring model allows for all kinds of gates at the processors, resulting in transformations that are performed on messages issued on the buses. Such networks and chips were actually built [GK89, Kea88, VK88].

In [Sch91, BPS91] the expanding volume of reconfigurable results and architectures was given a theoretical treatment. For example, it was shown that if the linear and the general models are equivalent in power then the connectivity problem is logspace computable. This partially explained the existence of a fairly simple connected components algorithm on a general reconfigurable cube [WC90], while no such equivalent is known at the linear reconfiguring side. The semi-formal introduction to these results is postponed to Section 7, where we also give additional insight into the gap in the computational power of the various reconfiguring models.

Results concerning the universality of the mesh were given in [Sch91, BPS91]. For example, paying a quadratic blow-up of the number of processors, any reconfiguring network may be simulated by a mesh. Other results concern an automatic "programming" of the mesh and the slowdown involved.

1.2 This work - Optimal Simulations

The question we are interested in is whether reconfigurable models (in particular two dimensional reconfigurable arrays) can be 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. The assignments of logical processors tasks to the available physical ones is automatically done by the compiler, and is "transparent" to the programmer. Thus, the ability of performing this mapping efficiently is an extremely important property of the model. In its absence, it is not likely that it will be chosen for a direct implementation on existing architectures. Note that the compiler actually writes a simulation program of a large machine having many processors by a smaller machine.

Despite the fact that real machines already exist and the large number of efficient algorithms that are known for reconfigurable arrays, none of the models was previously shown to support optimal simulations. In this work we give several positive answers for this problem. We present asymptotically optimal and almost optimal 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 (i.e. the polymorphic array
concept) exhibits optimal simulations (Section 3).

(2) Although using the same method fails for the linear reconfigurable mesh (Section 4) a novel technique may be used to achieve asymptotically optimal simulation there, too (Section 5).

(3) A third algorithm presents simulations of general reconfigurable meshes, paying with an extra slowdown which is polylogarithmic in the size of the simulated mesh (Section 6).

The simulation algorithms are very complex and require lots of bookkeeping operations. We show that the required space for these is asymptotically optimal, too. Yet, to avoid painful reading, we do not cope with constants minimization, hence the algorithms may seem wasteful and non-optimized at first glance. In addition, although given for the mesh, the simulation results may be applied to arbitrary rectangles as well.

The paper is organized as follows: Section 2 gives a formal definition of the reconfigurable networks model and three of its main variants. Section 3 gives the details of the simulation for a mesh in the HV-RN model. Section 4 gives an example for which the method of Section 3 fails for the linear reconfigurable mesh. Section 5 presents the new method for asymptotically optimal linear reconfigurable mesh simulation. In Section 6 the general reconfigurable mesh is shown to come close to optimal simulations. Finally, Section 7 gives a bunch of inter-model simulation results for general topology networks, as well as for meshes. These are motivated by both applicational and theoretical considerations.

2 Reconfiguring Models of Computation

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 a 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 processor individually, depending on the input and on the contents of messages that were previously read from approaching buses.

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 referred to as 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.

### 2.1 Switch Operations

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

**General RN (RN):** 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 configurations are any network partition of edge-disjoint connected subgraphs.

**Linear RN (LRN):** 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 configuration is a partition of the network into paths, or a set of edge-disjoint linear buses.

**Horizontal-Vertical RN (HV-RN):** This is a degenerate LRN model which applies to reconfigurable *arrays* only. For example, in two-dimensional arrays buses are formed along rows (horizontal) or columns (vertical) only.

### 2.2 Simulations and Slowdown

Let $\mathcal{R}$ and $\mathcal{R}'$ be two reconfigurable networks operating in any of the models defined in section 2.1. $\mathcal{R}$ and $\mathcal{R}'$ 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 $\mathcal{R}'$ *simulates* a single step of $\mathcal{R}$ with slowdown $C$ if for any single-step algorithm that is executed by $\mathcal{R}$ there is a $C$-steps algorithm that is executed by $\mathcal{R}'$ achieving the same computational task. We say that $\mathcal{R}'$ simulates $\mathcal{R}$ with slowdown $C$ if for any algorithm $A$ that is executed by $\mathcal{R}$ there is an algorithm $A'$ that is executed by $\mathcal{R}'$ achieving the same computational task, where each step of $A$ is simulated by $\mathcal{R}'$ 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 $\mathcal{R}$ to be executed by processors from $\mathcal{R}'$. We say that processor $x$ in $\mathcal{R}'$ simulates processors (say) $y, z, w$ in $\mathcal{R}$ 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 $M$ and a simulation of a single, arbitrary step which is consistent with $M$ is shown with slowdown $d$. The latter is thus a simulation of any algorithm with slowdown $d$, given that the input is consistent with $M$.

The above definition captures the intuitive notion of "simulation". However, since $\mathcal{R}$ and $\mathcal{R}'$ may have a different structure, a certain computational task may require different 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 presenting the simulation we will have to determine the memory requirements by the simulating processors, e.g. when a large network is simulated by a smaller one. However, we always assume that (except for the switching capabilities) 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 $\text{sizeof}(id)$ in the simulated network).

2.3 The RLA and the Mesh

The most simple connected topology is the Reconfigurable Linear Array (RLA). An RLA consists of $n$ processors labeled $\{0, \cdots, n-1\}$. 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 an 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.

The reconfigurable mesh is the underlying topology which is most popular in the literature, and which is also the topology considered in the main part of this work. The $n \times n$ reconfigurable mesh, called the $\kappa$-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 $i$th row and the $j$th column as $[i-1, j-1]$.

In the linear reconfiguring model (LRN) there are 10 local configurations that may be taken by a switch of the mesh. These configurations are depicted in Figure 1. Out of these, only four are supported by the switch of the HV-RN mesh; these are $\{L-R, U-D\}, \emptyset, \{U-D\}, \{L-R\}$. For

![Figure 1: Local configurations allowable for a switch of the mesh in the LRN model.](image)
convenience, we envision the mesh as embedded in the plane so that row 0 is at the bottom, and column 0 is to the left.

The following function will be used in several of the algorithms for mapping large meshes into smaller ones.

\[ \text{FOLD}(m) = \begin{cases} m \mod p & \text{if } m \div p \text{ is even} \\ p - 1 - (m \mod p) & \text{else} \end{cases} \]

We use

\[ [r, c] \rightarrow \{\text{FOLD}(r), \text{FOLD}(c)\} \]

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 non-formal 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 \(i\)th 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 Optimal Simulations in HV-RN meshes

In this section we show how simulation is performed with optimal asymptotic slowdown on meshes that operate in the HV-RN model. Although the algorithm that is presented in Section 5, showing optimal simulations for meshes in the LRN model, could also be applied to the HV-RN model, the algorithm which is presented here is much simpler and involves smaller constants.

The algorithm is based on the simulation of an \(n/p\)-submesh of the \(n\)-mesh by a single node of the \(p\)-mesh. For the sake of clarity of description, we assume that \(n/p\) is an integral integer.

**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 emulated 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) is completed by a single processor with slowdown \(2l\).

The simulation of a \(l\) processors RLA by a single processor \(x\) is carried in a “double pass” of \(x\) on the RLA. In the first pass, \(x\) simulates processors \(0, 1, \ldots, l - 1\) in that order. For each \(0 \leq i \leq l - 1\), \(x\) determines the local configuration of processor \(i\). If some processor \(j\), where \(j < i\), is a speaker and is connected to processor \(i\) (a fact which could have already been detected by \(x\)) and if \(i\) is a reader, then \(x\) writes \(j\)'th message in \(i\)'th buffer (recall that in the local memory of \(x\) there are buffers assigned to all the processors which are simulated by it). If more than a single speaker were already detected on
i'th bus (including the case that i is the second speaker) and j is a reader, then \( x \) sets the buffer of \( i \) as if an error was detected on the bus.

By the end of the first pass the buffer of the highest id'ed processor at each bus contains the message transmitted on that bus, or a detected bus error. Thus, a second pass of the RLA simulating the processors in a reversed order \( 1-1, 1-2, \ldots, 0 \) will put the correct message in each of the processors buffers. 

We can now state the main theorem of this section.

**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. Input and output data is contained in the buffers of the simulated processors from the \( n \)-mesh, as allocated in the memory of the simulating processors of the \( p \)-mesh. Each processor in the \( p \)-mesh is attached with an \( n/p \)-sub-mesh of the \( n \)-mesh. More precisely, processor \([i, j]\) of the \( p \)-mesh (where \( 0 \leq i, j \leq p - 1 \)) is attached with the sub-mesh of the \( n \)-mesh consisting of the processors \([k, l]\), for all \( in/p \leq k < (i + 1)n/p \) and \( jn/p \leq l < (j + 1)n/p \).

**Algorithm HV-RN Simulation:**

The algorithm consists of three phases. In the first phase, each processor of the \( p \)-mesh simulates the sub-mesh of the \( n \)-mesh that is attached to it. By Lemma 3.1, this phase takes \( 4(n/p)^2 \) steps. It is straightforward to enhance the algorithm from Lemma 3.1, so that it is written in the buffer allocated to the leftmost processor in each row whether it is connected by a bus to the "rightmost" processor in that row, and vice versa (similarly, for the highest and lowest processors of each column). Moreover, the buffer contains the information whether there was a speaking or a reading processor on the "internal part" of any bus reaching a leftmost or a rightmost processor.

The second phase includes \( O(n/p) \) steps. This corresponds to the \( n/p \) rows (columns) of the \( n \)-mesh that are "contracted" in every row (column) of the \( p \)-mesh. Let \( x \) be a processor of the \( p \)-mesh. At the \( i \)'th step of this phase, \( x \) collaborates with the rest of the processors in its row (column) to simulate the inter-sub-mesh bus connections of the \( i \)'th row (column) of the \( n \)-mesh which is associated with the row (column) of \( x \). We next give the details of this collaboration for the simulation of one of the rows. The details for the simulation of a column are similar.

Let \( x \) be the leftmost processor of the \( i \)'th row in the sub-mesh of the \( n \)-mesh which is simulated by \( x \). Let \( y \) be the corresponding rightmost processor. If we are to give explicit indices, then the relation is as follows:

\[
x = [x_1, x_2] \quad ; \quad z = [x_1n/p, x_2n/p + i] \quad ; \quad y = [(x_1 + 1)n/p - 1, x_2n/p + i]
\]

Basically, \( x \) simulates the connections of \( z \) and \( y \) to their neighbors (in the simulated row) which belong to neighboring sub-meshes. These connections denoted \( z_{\text{LEFT}} \) and \( y_{\text{RIGHT}} \), correspond to the (left and right) edges of \( x \), denoted \( z_{\text{LEFT}} \) and \( x_{\text{RIGHT}} \). We consider two cases.

1. If \( z_{\text{LEFT}} \) and \( y_{\text{RIGHT}} \) are taking part in different buses, then \( x \) disconnects \( z_{\text{LEFT}} \) from \( x_{\text{RIGHT}} \).

   Let \( B_{\text{LEFT}} \) and \( B_{\text{RIGHT}} \) denote the sub-buses in which \( z_{\text{LEFT}} \) and \( y_{\text{RIGHT}} \) take part, correspondingly. If the part of \( B_{\text{LEFT}} \) that is contained in the sub-mesh that is associated with \( x \)
contains a speaker, then \( z \) speaks on \( x_{LEFT} \) (while checking the state of the bus for an error). If, on the other hand, it contains only readers, then \( z \) reads from \( x_{LEFT} \). As explained above, this information is stored in the buffer allocated to \( z \). Similar decisions are made for reading and speaking on \( x_{RIGHT} \) depending on the data stored in the buffer of \( y \).

2. If \( x_{LEFT} \) and \( y_{RIGHT} \) take part in the same bus, then \( z \) connects \( x_{LEFT} \) to \( x_{RIGHT} \). \( z \) reads and speaks according to the operations that are taken by the processors of the \( i \)’th row of the sub-mesh to which it is associated. Again, this information is found in the buffers of both \( z \) and \( y \).

The third phase involves a single pass of each processor on the associated sub-mesh. Information that was read during the second phase is copied into the buffers of all the processors taking part in buses which cross sub-mesh border lines. This is completed in \( (n/p)^2 + O(n/p) \) steps.

### 4 A Lower bound for the simulation by contraction

The method used in Section 3 for the simulation of an \( n \)-mesh by a \( p \)-mesh, involves a contraction of a full, connected sub-mesh of the larger mesh, so that it is simulated by a single processor of the smaller mesh. In other words, this is a contraction mapping of simulated to simulating processors, and it remains fixed throughout the simulation. This is actually the general approach of simulating a processor array with a smaller one, e.g. in the fixed connection model of computation [Lei91, pp. 234].

In the reconfiguring context there are other important characteristics of the method which is presented in Section 3. The following modus operandi is also followed in algorithm HV-RN Simulation, and is a “natural” first trial to solve the simulation problem.

**All-or-nothing assumption:** any border-crossing bus is simulated in a single step by a bus, solely dedicated for that purpose.

For simplicity, we shall not give a formal definition of the all-or-nothing assumption, nor shall we formalize our claims in the form of theorems. Rather, we give an example how the straightforward approach of the contraction mapping and the all-or-nothing assumption fails to achieve efficient simulations.

Consider the situation in which a LRN \( n \)-mesh is to be simulated by a LRN \( n/2 \)-mesh. Using a contraction mapping, we decide that processor \([i,j]\) in the \( n/2 \)-mesh is to simulate processors \([2i,2j],[2i+1,2j],[2i,2j+1],[2i+1,2j+1]\) of the \( n \)-mesh. Consider a simulation of a step of the \( n \)-mesh in which the bus configuration is as depicted in figure 2. The figure shows the mapping of processors of the \( n \)-mesh, represented by circles, to the simulating processors from the \( n/2 \)-mesh, represented by squares. Thus each square contains four circles. Buses are drawn by thick lines. Let us describe the configuration assumed by the \( n \)-mesh, as is shown in the figure.

Bus 1 starts at processor \([n-1,0]\) (in the \( n \)-mesh) and goes straight, all the way down to processor \([0,0]\). Bus 2 starts also at processor \([n-1,0]\), goes right to processor \([n-1,1]\), down to \([n-3,1]\), right to \([n-3,2]\) and then all the way down to \([0,2]\). Note that bus 2 shares with bus 1 the Down exit of the simulating processor \([n-1,0]\). Hence, because of the all-or-nothing assumption we need two different steps to simulate bus 1 and bus 2. It may be viewed as if bus 2 has two stair-like bends.
Bus 3 has 3 such bends: start \([n - 2, 0]\), right to \([n - 2, 1]\), down to \([n - 4, 1]\) (parallel to bus 1), right to \([n - 4, 3]\), down to \([n - 6, 3]\) (parallel to bus 2), right to \([n - 6, 4]\) and down all the way to \([0, 4]\). Clearly, since bus 3 shares edges of the simulating mesh with both bus 1 and bus 2, and because of the all-or-nothing assumption, bus 3 is simulated in a step when both bus 1 and bus 2 are not simulated. We can proceed in the way described above in order to construct \(n/2 + 1\) such buses. Bus \(i\) shares a

Figure 2: Lower bound example: buses are “shuffled” so that bus \(i\) shares an exit of a simulating mesh processor with all the buses \(0, 1, \ldots, i - 1\).

\(n/2\)-mesh exit with all the buses \(j\) for \(1 \leq j < i\). We conclude that for the given configuration and under the given assumptions, the simulation takes at least \(\Omega(n)\) steps, regardless of the size of the simulating mesh.
A similar bus construction and the same arguments hold when using the contraction mapping and the all-or-nothing assumption in simulations of meshes in all other reconfiguring models, except for the HV-RN.

5 Optimal Simulations for LRN meshes

The main result of this section is the following theorem.

Theorem 5.1 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 components, which may have interest in its own right.

5.1 LCC - linear-connected components

Definition 5.1 A graph \( G = (V, E) \) is called linear if the degree of every vertex is \( \leq 2 \), and \( G \) is acyclic.

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

Lemma 5.1 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 the contraction mapping (as described in Section 4) of a 2-submesh of the 2n-mesh. We call the processors of the given n-mesh v-processors. A bus in the n-mesh can be represented by two paths in the 2n-mesh, see also Corollary 7.4 and Figure 9. These two buses may be viewed as if they were directed, which will be necessary later for choosing a label on a linear bus. 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 \([i, j]\) that holds a 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 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 \( \leq 2 \).

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 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 in the direction of that 1.
(d) similar to (c), but for the column.

Fig. 5.b shows the 1's of a \( 9 \times 9 \) adjacency matrix and the corresponding bus configuration.

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

\[
V_{LCC} = \{[i,j] | m_{i,j} = 1\} \quad ; \quad E_{LCC} = \left\{ \{[i,j],[k,l]\} \mid \begin{array}{l} \text{[i,j],[k,l] are connected by a} \\ \text{horizontal or a vertical bus segment.} \end{array} \right\}
\]

Denote by \( G' \) the dual graph of \( G \). Namely,

\[
G' = (E, \{\{e_1,e_2\} | e_1 \cap e_2 \in V})
\]

Claim 5.1 \( GLCC \) 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:

\[
e_1 \rightarrow [i,j] \quad ; \quad e_2 \rightarrow \left\{ \begin{array}{l} [i,k] \quad \text{if } e_2 = (i,k) \\ [k,j] \quad \text{if } e_2 = (j,k) \end{array} \right\}
\]

The maps of \( e_1,e_2 \) are connected in \( GLCC \), 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. \( \square \)

Back to the LCC algorithm: every connected component in \( GLCC \) 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 beginning of the proof). 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 \( GLCC \) to its left - see \( \leftarrow \) in fig. 5.b. 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. 5.b in paranthesis to its left.
This completes the proof of Lemma 5.1.

Remark: With a little effort we can also solve the LCC problem for a graph of \( n \) vertices on the \( n \)-mesh. This can be done by splitting the set of vertices and merging the solutions. As was promised earlier we avoid painful details.

5.2 The LRN Simulation Algorithm

If \( m \leq 4 \), we simulate the \( n \)-mesh with one processor. Else, let \( p = \lfloor m/4 \rfloor \). We shall use a \( p \)-mesh to traverse the \( n \)-mesh. According to Lemma 5.1, the \( m \)-mesh solves the LCC problem on \( 2p \) vertices in \( \mathcal{O}(1) \) time, given that the inputs reside in every alternate 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.

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):

The basic idea is to traverse the \( n \)-mesh with the \( p \)-mesh in snake-like order. At every window position, the following occurs: New bus segments are encountered, old bus segments enter the window, and some old bus segments join up with others (see fig. 3). 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.

Figure 3: buses \( a, b, c \) encountered and joined inside a window.

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 2.3. Thus, in moving from one window to the next in the snake-like order, the 2 rows or columns on either side of the boundary are simulated by the same row or column of the window. Information about broadcasters is also stored in this manner.

In general, every window position is bordered by up to 4 windows, of which some are “old” window positions, and some are “future” window positions. Furthermore, one old window is the immediate
predecessor of the current position, and one future window is the immediate successor (this will be
called the “next” window). Since the buses are linear, each bus segment may have 2 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 2 or more bus segments merge
in a window (due to the LCC operation), they become a single bus segment, with only 2 ends. The
processors “in charge” of these ends are informed about the id of the single bus segment. The precise
method by which all this is done is explained below.

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. 4):

**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 respect to the predecessor. Such
a segment is called an “awakening” segment.

**terminating segment**: The segment terminates in the window - it is called an “ending” or “termi­
nating” 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.

![Figure 4: Bus segments in a new window position.](image)
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). 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, in a manner similar to choosing a label in a connected component of an LCC graph (Section 5.1).

4. Now each processor in charge of an entering or awakening segment has information about zero, one, or two other processors-in-charge to which it is connected (these connections are inside or outside the window). 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. 5. Fig. 5a. shows a 5 x 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. 5b.

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 5.1 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. Fig. 5b. also shows the bus segments resulting in the 2p-mesh as a result of the LCC operation.

![Diagram showing bus segments and LCC labels](image)

Figure 5: Example for the LCC process applied to a bus configuration:

a. (left) bus segments in a 5 \( \times \) 5 window  
b. (right) the corresponding LCC graph

7. Consider now a single bus and the situation of its current endpoints. Each endpoint may be in one of 3 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's. It will be updated during the back-sweep of the window (subsection 5.5).

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 \( x \). But 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 subsection describes our solution.

### 5.3 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 doesn’t 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. There are four cases to consider:

- **a.** \( B \) is dormant. Let \( P2 \) be the processor through which the bus connected to \( B \) enters (or awakes in) the window (same as \( P1 \) with respect to \( A \)), for example see fig. 6. 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.

![Figure 6: Case a. both ends are dormant.](image)

- **b.** \( B \) terminates in the window. Let \( P2 \) be the processor through which the segment terminating in \( B \) entered (or awoke in) the window. 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).
c. **B** continues into the next window. Let **P**₂ be the processor through which the bus connecting to **B** entered (or awoke in) the window. Again, **P**₁ and **P**₂ exchange information during the LCC, but now **P**₁ does not initiate an update of **A**. The continuing end **B** will update (or initiate an update of) **A** in some future window.

d. The bus ending in **B** goes to sleep in the window. Let **P**₂ be the processor through which the bus connecting to **B** entered (or awoke in) the window. As above, **P**₁ and **P**₂ have each other's information about **A** and **B** after the LCC. **P**₁ checks if **A** will wake up before **B**. If so, **P**₁ 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 **P**₁. Note that in this case (d.), **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.

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 that window, 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 **P**₂ has been processed and so there will be no further updates (**B** will not update **A** because it wakes up after **A**).

### 5.4 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 \( 2n/p - 2 \), 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 \( 2n/p - 2 \) window positions away. Therefore, we can use a circular array of size \( 2n/p - 2 \) 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.

### 5.5 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.
2. For every window position, we also have the following information:
   
   2a. 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.
   
   2b. 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.

   2c. All processors that were in charge of entering or awakening bus segments hold the bus-id’s 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-going-to-sleep (they were awakening segments 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 2 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 “conflict” 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 5.1.

5.6 Other traversal orders

Proposition 5.1 The algorithm in the proof of Theorem 5.1 can be modified to handle any order of traversal of the n-mesh by the m-mesh.

Proof: In any order of traversal, a current window is bordered by windows of the following type:

Old window (there may be 0, 1, 2, 3, or 4 such bordering window); Immediate predecessor (0 or 1) Immediate successor (0 or 1) Future window (0, 1, 2, 3, or 4).

Thus, our categorization of bus segments inside a window, namely, entering, awakening, going-to-sleep and continuing, are exhaustive, and are also relevant for other traversals. So basically, we have to handle everything in a similar manner. The differences are due to the fact that the number of bordering window of any given type may be different than in the snake-like order. These differences are handled as follows:
1. There may be up to 4 bordering old windows, and thus there may be up to 4 times as many awakening segments in a window. This can be handled by specifying that the column-stack mechanism is set to update not only the top row of processors (as in the snake-like order), but also the bottom row and the two sides.

2. Handling dormant ends: Again, we adopt the lazy approach described earlier. In case there is no active end continuing into a next window, we also use the column-stack to update the end that will wake up before the other (it is assumed that the traversal order is known to all the processors).

3. Using the LCC: In the snake-like order, bus ends entered a window either from the predecessor or from some (single) bordering old window. Thus, there were at most 2p processors-in-charge of bus segments that participated in the LCC algorithm. Now, there may be up to 4p such processors, so in order to use the LCC, we may have to take an even smaller window (p = m/8 instead of p = m/4).

4. Using the column-stack: Our note that a dormant endpoint will awake after at most \(2(n/p) - 2\) window positions is no longer valid. Therefore, every processor’s local array may have to be \(4(n/p)^2\). The factor of 4 is required because a processor’s column stack may need to update up to 4 boundary processors in the worst case.

6 Almost Optimal RN Simulations

For the RN model we can show the following result which may not be asymptotically optimal. 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. Finally, 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.

**Theorem 6.1** 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 5. We keep using windows for the simulation of p-submeshes of the n-mesh. Similarly, we keep the method of assigning id’s (representatives, labels) to the bus segments that are discovered, from the set of id’s of window boundary processors. The major 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 Subsection 6.1. The i'th iteration (\(i \geq 0\)) collects information for 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, relying on the result of Section 5. Thus, for consistency, the folding mapping is also used here for RN simulations of n-meshes by a p-mesh.

6.1 Connected Components Results

We use the following lemma by Miller et. al.
Lemma 6.1 [MPRS87, 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.

The proof consists of a simple adaptation of the $O(\log n)$ steps connected components algorithm given by Shiloach and Vishkin [SV82] for the CRCW PRAM model. In the RN simulation algorithm to be presented in this section we need a slightly different result.

Lemma 6.2 Let $G = (V,E)$ be an undirected graph having $|V| = n$ nodes and $|E| \leq n/2$ 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 at each processor, the connected components of $G$ can be determined in $O(\log n)$ steps.

We remark that this is not a trivial corollary of Lemma 6.1, since moving the edges to their appropriate places according to the adjacency matrix may require $\Omega(n^{1/2})$ steps, e.g. when there is a need to set a $n^{1/2}$-submesh of the adjacency matrix. Also, the number of input graph edges may be higher than $n/2$, as long that it is $O(n)$ and that the edges are evenly distributed among the processors of the leftmost column.

The results from Section 5, Lemma 6.1 and Lemma 6.2 imply

Corollary 6.1 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 $[\text{FOLD}(i), \text{FOLD}(j)]$ 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 $|E| \leq n/2$ 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 6.1 will be sufficient for our purposes, we remark that the connected components results are actually 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 the following.

Let $G = (V,E)$ be an undirected graph having $|V| = n$ nodes.

Corollary 6.2 1. Given the adjacency matrix of $G$ distributed so that element $[i,j]$ of the matrix is stored in processor $[i,j]$ of the HV-RN reconfigurable $n$-mesh, the connected components of $G$ can be determined in $O(\log n)$ steps.

2. Using also Theorem 3.1: given the adjacency matrix of $G$ distributed so that element $[i,j]$ of the matrix is stored in processor $[i \text{ div } p, j \text{ div } p]$ of the HV-RN reconfigurable $p$-mesh, the connected components of $G$ can be determined in $O((n/p)^2 \log n)$ steps.
3. Given that $|E| \leq n^{1/2}/2$, so that the edges of $G$ are arbitrarily distributed at the processors of the leftmost column of the HV-RN reconfigurable $n$-mesh, such that there is at most one edge at each processor, the connected components of the graph can be determined in $O(\log n)$ steps.

4. Using also Theorem 3.1: Given that $|E| \leq n^{1/2}/2$, so that the edges of $G$ are arbitrarily distributed at the processors of the leftmost column of the HV-RN 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.

**Proof:** (of Lemma 6.2) We use a variant of the connected components algorithm given in [MPRS87] which, in turn, is an adaptation of [SV82]. We assume that the reader is familiar with these algorithms, and proceed to describe the necessary changes. As a strengthening of the result, we allow for an edge to appear as an input on several of the leftmost column processors. The total number of processors holding input edges is bounded by $n/2$.

We first add an initializing phase at the beginning of the algorithm.

**Sort:** The edges are sorted according to their left coordinate as a primary key and their right coordinate as a secondary key. Empty processors on the leftmost column are assumed to contain an input edge $(\infty, \infty)$. The LRN sorting algorithm which was presented in [BPRS91] may be used. A straightforward variant of this algorithm sorts $n$ keys given on the leftmost column of a reconfigurable mesh in $O(1)$ steps, and outputs the keys back on the leftmost column, smallest on bottom-left corner processor.

**Multiplicity avoidance:** Reading the edge in the processor above it (in the leftmost column) and the one below it, each processor holding an edge $(i, j)$ decides whether it is the highest of all processors holding $(i, j)$ edges. All other processors eliminate the edges they are holding.

**Packing:** Using the packing algorithm from (say) [BS92b], all remaining edges are packed into the lower part of the leftmost column. The resulting column of edges remains sorted by both left and right coordinates as primary and secondary keys.

**Init:** Let $(i, j)$ be the $k$'th highest edge, so that $(i, j)$ is stored in processor $[k-1, 0]$ of the $n$-mesh. This processor defines a variable PARENT and initializes it to $j$. Recall that the basic algorithm is based on $O(\log n)$ iterations, each of which assigns a parent to each node in the input graph. PARENT will be updated at the end of each iteration to contain the id of the parent of $j$. Next the $k$'th row is notified on the identity of $(i, j)$, so that processor $[k-1, j]$ defines itself as a communicator. Communicators will be used for the parent-update operation.

We now describe the necessary change at each iteration of the algorithm. The only difference is at step 2.(a) of the [MPRS87] algorithm, where each node $i$ with a current parent $j$ examines his edges $(i, k)$. The goal is to find a $k$ for which $parent(k) < j$. If found, $parent(k)$ is notified to the $i$'th row. This is part of the hooking operation, as described in [SV82].

The new algorithm uses bus-splitting (see e.g. [MPRS87]) and the part of the leftmost column occupied by edges $(i, \star)$, so that the highest processor holding an $(i, \star)$ edge, $p_i$, is notified by the lower
processors on some parent(k), if any exists, p, broadcasts this value on its row, then the i'th processor of this row broadcasts this value on its column, then processor [i - 1, i - 1] broadcasts it to the i'th row.

For a processor x holding an input edge (i, j) it remains to show that the variable PARENT is updated at the beginning of each iteration, i.e. it contains parent(j). This is true for the first iteration, by the initialization part. For each successive iteration, when the new parent(j) is determined by the j'th row, it is transmitted by [j - 1, j - 1] on the j'th column, where it is read by all communicators which, in turn, transmit it on their rows where it is read by the processors of the leftmost column holding edges (*, j). Finally, these processors update their PARENT variables with the new value.

6.2 The Algorithm

Back to the proof of Theorem 6.1, the algorithm consists of three phases. We first sketch these phases informally, then proceed to give the details.

Notation: 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".

Algorithm RN Simulation (sketch):

1. This phase consists of log(n/p) iterations, gathering configuration information and constructing a spanning forest for the set of buses. During iteration 0, each p-window is simulated by the given p-mesh and a representative for each bus segment is elected. During iteration \(i > 0\) all windows of size \(2^ip\) are considered, deducing information for each of them out of the information of its four composing sub-windows. This is achieved by computing the connected components of a graph representing the bus configuration.

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

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 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 \(+\text{sizeof}(id)\) 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 asymptotically optimal.
6.3 Phase (1): Iterations (Recursion on window size)

We will switch terminology, viewing this phase as either iterations or recursion on window size. The recursion consists of \( \log(n/p) \) levels. In level \( i \) we process information for windows of size \( 2^ip \).

6.3.1 Recursion base:

For \( i = 0 \) which is the recursion base, a window of size \( p \) is simulated by the \( p \)-mesh. 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 picked 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 iterations of the algorithm. 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 window boundary processors only. Moreover, we pick them from those processors where the bus actually exits to a processor outside the window. There may be up to \( 4(p-1) \) such processors. Corner processors may represent two different buses, but this will not slow the algorithm down. Picking a representative for all buses of any window may be done in \( O(\log \log p) \) steps, using "binary search" of the boundary processors addresses.

6.3.2 Recursion step:

Moving through levels of the recursion, we find new representatives for the buses which cross the boundaries of the composing sub-windows. Let us first sketch the general idea, and then give the details of the data movements.

Suppose we are given the bus representatives in four sub-windows \( A, B, C \) and \( D \), each of size \( 2^{i-1}p \). \( A, B, C \) and \( D \) compose a window \( W \) of size \( 2^ip \), as depicted in figure 7. 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 \( G_W \), whose nodes are all the bus representatives in the sub-windows. Let \( a \) be an id of a node of \( G_W \). So

\[
a = [r_12^{i-1}p + r_2, c_12^{i-1}p + c_2]
\]

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

\[
r_2 = 0 \quad ; \quad c_2 = 0 \quad ; \quad r_2 = 2^{i-1}p - 1 \quad ; \quad c_2 = 2^{i-1}p - 1
\]

An edge \((a, b)\) of \( G_W \) represents two buses represented by \( a \) and \( b \) in neighboring sub-windows, which are found to be connected at their joint boundary. Note that any bus leaving a sub-window in the direction of one of the other sub-windows 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. A closer investigation of the algorithm reveals that the constants of this estimate may be reduced considerably. As promised before, we shall not get into this.\(^1\)

\(^1\)There may, however, exist an asymptotically faster connected components algorithm, due to the small number of
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 of the recursion. If data is stored appropriately, this can be done by using Corollary 6.1 in $O(2^{2i}(i + \log p))$ steps.

Let $T(n)$ denote the recursion time to compute the tree of representatives, as described above, for the global $n$-mesh. If we assume that the interface of the output of the $i$'th and the input of the $i + 1$'th recursion levels is correct, consistent and takes $O(2^{2i})$ steps (in addition to the connected component edges in the input graph. However, such a result for the CRCW PRAM would solve open theoretical problems, e.g. it implies that Logspace has circuits of depth $o(\log^2 n)$. Hence looking for such a result it would probably not be a good approach to resort to PRAM algorithms, as is commonly done.
complexity), then

\[
\begin{align*}
T(n) &= 4T(n/2) + O((n/p)^2 \log n) + O((n/p)^2) \\
T(p) &= O(\log \log p)
\end{align*}
\]

hence

\[
T(n) = O((n/p)^2 \log n \log (n/p))
\]

as claimed. We now proceed to show that the assumptions on the data movements hold.

### 6.3.3 Mapping:

The (LRN) mesh on which the connected components algorithm is to be executed 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 applicable, we define a function $\Upsilon_i()$. This function maps the facets of a window of size $2^i p$ and the facets of its sub-windows, i.e. the $16 \cdot 2^{i-1} p$ nodes of the input graph to the connected components algorithm, one-to-one into the rows of the virtual mesh, i.e. into $\{0, \ldots, 16 \cdot 2^{i-1} p - 1\}$. Although we do not explicitly specify $\Upsilon_i()$, it should fulfill two requirements, which are referred to as its properties. The role of these will be made clear later.

1. It maps all id's of the same facet of a sub-window in a successive order.
2. A row facet is mapped in order that increases with its column indices, similarly for a column facet and its row indices. As an exception, $\Upsilon_0$ is defined as follows.

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

Clearly, such a mapping exists for all $i$ and is easily computed in $O(1)$ steps and $O(1)$ space by each processor.

We cannot, however, complete the description of the algorithm by referring to the virtual mesh only, since virtual meshes change in different recursion levels. We thus map the virtual mesh into the $p$-mesh, using $FOLD()$.

\[
\Upsilon = FOLD \circ \Upsilon_i
\]

Observe that the mapping $\Upsilon$ does not depend on $i$. This relies on the properties of the functions $\Upsilon_i$, the operation of $FOLD$, and the fact that facets are of size which is a multiplicity of $2p$. This important feature of the mapping implies that mapping of the same processor-id of the $n$-mesh, as a candidate in the connected components algorithm, does not change in different levels of the recursion.

### 6.3.4 Representatives Election

As before, we assume the connected components algorithm to compute bus representatives for bus segments in a window $W$ of size $2^i p$. The algorithm uses the id of the nodes of the input graph $G_W$ as indices for computing processors. Thus, we cannot allow the nodes 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, that is, the processors which compose their facets.
Recall that the representative of a bus is, in fact, an id of a processor sitting on the bus. 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 in respect to a certain window to be an id of a processor sitting on the boundary of that window. Clearly, this is the case for windows of size $p$. For larger windows, however, we have to revise the connected components algorithm inductively.

First, we have to make sure that the id of a processor which is the intersection of the bus and the window borderline is in the pool of the candidates for being chosen as the representative. Consider again the choice of representatives of buses in a $p$-window $W_p$. $W_p$ is a sub-window of larger windows, let $W_{2p}$ denote the window containing $W_p$ which is of size $2p$. 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 the boundaries, 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 representatives 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 such a 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

$$\begin{cases} n & \text{if } c = 0 \text{ or } r = 0 \\ 2^k p & \text{else} \end{cases}$$

Computing the above for any of the processors of any other facet 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 the beginning of the recursion.

Next, we have to make a similar revision of the connected components algorithm. Again, the choice of representatives gives precedence to id's of processors sitting on the boundary of larger windows. Recall that the connected components algorithm computes representatives for a $2p$-window $W$ out of the representatives of its sub-windows. Let $x$ be a node in the input graph, where (by induction hypothesis) $x$ is a boundary processor of one of the sub-windows, say $A$, and $x$ is in the only facet of $W$ taking part in the boundary of a window which is much larger than $W$. So according to our policy, $x$ is given priority in becoming a representative. This is done as follows. All processors in row $T_i(x)$ of the virtual mesh know $LABEL(x)$ which was chosen as the representative of the connected component of $x$. Processor $[T_i(x), T_i(LABEL(x))]$ uses bus-splitting in column $T_i(LABEL(x))$ in order to inform processor $[T_i(LABEL(x)), T_i(LABEL(x))]$ of the status of $x$. $[T_i(LABEL(x)), T_i(LABEL(x))]$ chooses one of the applicants having the same status, say $T_i(x)$, to replace $T_i(LABEL(x))$ as the new representative of the bus. It then informs the processors in his column of his decision, which is distributed to all rows $T_i(y)$ where $y$ is in his component, so that the component is re-labeled to $x$. The full process takes constant number of steps (on the virtual mesh).

Finally, if $x$ is a boundary processor of a larger window in which two facets of $W$ take part, then (if necessary) a similar constant time re-labeling process may be performed.
6.3.5 Message collection and Error detection

Yet another revision of the connected components algorithm has to do with the message transmission on the bus and the detection of bus error states.

Consider iteration $i$, 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 $s_x$. 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 6.3.1). Moreover, row $T(x)$ is storing this message, if exists. Column $T(x)$ is used to move the message to processor $[T(LABEL(x)), T(x)]$. Now all messages transmitted on sub-segments reside in processors of the row of the mesh to which the new representative is mapped. Next, all processors in row $T(LABEL(x))$ transmit the messages, so that all processors of the row can either store that message or detect an error state of the bus, representing an error state of the corresponding simulated bus segment $s$.

Similarly, if $i > 1$ then we assume inductively that the information about transmission in sub-segments is stored at the rows of the virtual mesh. The information is moved to the new representative of each segment, which also decides on the detection of a bus error.

6.3.6 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 at a previous recursion level, the representative of $x$ is elected during the connected component algorithm and is stored at row $T_i(x)$ of the virtual mesh as $LABEL(x)$. $LABEL(x)$ is stored at row $T(x)$ 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 previously in Subsection 6.3.2 for a window $W$. These are formed at the end of recursion level $i - 1$ and are the input to the connected components executions of level $i$. This is also a temporary data for internal use of Phase (1).

Pointers: For each pair, $x$ and its chosen representative $LABEL(x)$ we define a pointer $(x \rightarrow LABEL(x))$. This pointer is duplicated and stored at all the processors of row $T(x)$. The total collection of all pointers constructs 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 $(x \rightarrow LABEL(x))$ for each representative $x$. Recall that there are initially at most $2n^2/p$ representatives which are evenly distributed by $T$ 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. It is stored, together with the message that was transmitted on that bus (or with an error indication) by all the processors at row $T(x)$. An ancient parent is easily discovered during the connected components algorithm when it is elected as a representative of some component, but is not at the boundary of the window, or the bus does not leave the window.
6.3.7 Data Movement: the way it works

We now describe the substeps taken by iteration $i$ 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 sub-windows of $W$, row $\Upsilon(x)$ of the $p$-mesh knows $\text{LABEL}(x)$. This is obvious at level 1, after the bus algorithm for leader election that is executed at the bottom of the recursion. Suppose this holds at the beginning of iteration $i$. We shall see that it holds at the beginning of the next iteration, too.

Knowing $\text{LABEL}(x)$, row $\Upsilon(x)$ creates a pointer $(x \rightarrow \text{LABEL}(x))$ 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.

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 $\text{NBR}(x)$ as the id of the processor which is a neighbor of $x$ in the neighboring facet (see Figure 7).

The property of $\Upsilon$ (see Subsection 6.3.3) implies that for every $x$ which was elected as a representative at iteration $i - 1$ and which is not on the boundary of a $2^i p$-window

$$\Upsilon(\text{NBR}(x)) = \Upsilon(x)$$

Hence the same row of the $p$-mesh, namely $\Upsilon(x)$, knows both $\text{LABEL}(x)$ and $\text{LABEL}(\text{NBR}(x))$. Row $\Upsilon(x)$ composes a new edge $(\text{LABEL}(x), \text{LABEL}(\text{NBR}(x)))$ and stores it at its leftmost processor. Also, for the needs of the connected components algorithm as described in Subsection 6.1, the "sister edge" $(\text{LABEL}(x), \text{LABEL}(\text{NBR}(x)))$ is produced and stored at the same place.

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

Conceptually, here is where the virtual mesh of size $2^i p$ and the mapping $T_i$ into its rows, are replaced by the virtual mesh of size $2^{i+1} p$ and the mapping $T_{i+1}$ into its rows. Again, since the mapping $\Upsilon$ remains the same, there is no need to move edges around. There is, however, some work carried by the leftmost processor of each row of the $p$-mesh to assign a single edge to each of the rows of the new virtual mesh.

We are now at the position of activating the (improved) connected components. The algorithm, taking $O(2^{2i} \log(2^i p))$ steps, stores the representatives at the correct places: for each $y$ at the boundary of $W$ row $\Upsilon(y)$ knows $\text{LABEL}(y)$, as was required at the beginning of this subsection.

6.4 Phase (2): simple tree contraction

As already explained, the iterations of Phase (1) construct a tree of maximal height $\log(n/p)$ where each node points to his parent. The tree information is saved in terms of pointers (tree edges) $(x \rightarrow \text{LABEL}(x))$ where $x$ is a node in the tree and $\text{LABEL}(x)$ is his 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 emulated $n$-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. those having column coordinate equal \( l \div n/p \) will be evaluated by the \( l \)th column of the \( p \)-mesh. Each column is in charge of evaluating one of these sets.

The column is connected as a single bus and works independently and sequentially on the evaluation of his set. This process is a trivial exercise, assuming the processors store the pointers in a way which gives them a fast, \( O(1) \) steps access to the table of \( O((n/p)^2) \) pointers that is stored by each of them. Note that a \( p \)-mesh processor \( y = [c, r] \) stores a (single) pointer \( (x \rightarrow LABEL(x)) \) for each of the \( O((n/p)^2) \) \( x \)'s 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 index. As mentioned earlier, we assume a single step access to such tables.

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) \( \text{msg}(z) \). This information is stored by all column processors as an \textit{info-pair} \((z, \text{msg}(z))\) in a special \( O((n/p^2))\)-sized table. This table is used at Phase (3).

### 6.5 Phase (3): backsweep

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, is 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 \((z, \text{msg}(z))\) 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 each boundary processor \( z \), if \( z \) was elected as the representative of its bus segment at this window then \([FOLD(r), FOLD(c)]\) transmits \( \text{msg}(z) \).

This completes the proof of Theorem 6.1.

### 7 Inter-Model Simulations Results

In this section we give several simulation results between different variations of the basic models considered so far.

#### 7.1 More Switches

We consider several variants of the basic models that were defined in Section 2.1.

**Directed RN (DRN):** This model is similar to the RN model, except that edges are directed, so messages travel in one direction only. Consequently, each connected subset of edges is split into
in-edges and out-edges. A message entering the switch for the first time via either one of the in-edges, proceeds via all the out-edges connected to it.

Non-Monotone RN (NMRN): This model is the same as the Directed RN model, but a switch has an additional "inversion" capability. When this operation is activated by the switch, a signal going via the switch is inverted. That is, a "0" ("no signal") turns into a "1" ("signal on") and vice versa.

Directed LRN (DLRN): This model is similar to the DRN model, except that, connecting is done as in the LRN model: a switch may partition its collection of (directed) edges into any combination of connected pairs and singletons. Hence buses are of the form of a directed path (or a directed cycle).

No-Cycles LRN (NC-LRN): This model is similar to the LRN model, except if in some time-step a cycle is formed and a processor tries to speak on it, then an error state occurs on the bus.²

tree-RN: This model is similar to the RN model, except that the formed buses are trees (as subgraphs of the underlying topology).

Note that the notion of a bus for DRNs and NMRNs is somewhat different than that of LRNs and RNs. For DRNs and NMRNs suppose some processor \( z \) transmits at time step \( t \), and let \( H_t \) denote the global configuration that was chosen by the network during step \( t \). Then the message issued by \( z \) on some connected set of out-edges reaches the subgraph of \( H_t \) consisting of all nodes that may be reached from \( z \) by a directed path starting at those out-edges. The notion of bus error state for DRNs and NMRNs changes, too: a node \( y \) detects an error during step \( t \) if, in the configuration \( H_t \), \( y \) is reachable from two different speakers transmitting different messages. It may happen that a message issued by some speaker \( z \) will be correctly received by a reader, while other readers that are reachable from \( z \) detect an error since they are reachable from other speakers too.

There are two remarks related to the definition of the tree-RN model. First, the motivation for its definition comes from the "conditioned" theoretical gap between the power of RN's and LRN's, as was shown in [BPS91, Sch91]. Indeed Tree-RN's is a model having switch capabilities that are "between" those of the processors of the LRN model and those of the RN model. Thus, simulations results concerning this model improve our understanding of where this gap comes from. Second, the definition of the tree-RN model is very special, in that it uses the global notion of a tree to describe the buses, rather than describe all possible local switch configurations, as is with the other models.

### 7.2 Reconfigurable Complexity Classes

In the context of the complexity classes that were defined in [BPS91], the simulation results of this section show relations that are instructive from a theoretical point of view. For a formal and full definition of the complexity classes, We refer the interested reader to [BPS91]. Here, however, we have chosen to avoid this painful reading, by giving the intuitive notion of the classes.

²The motivation for the definition of the NC-LRN model comes from guided optic implementations where stability problems may occur on "cycles".
Informally, the class $RN$ is the set of all the problems that are solvable in constant time by a reconfigurable network which

- operates in the RN model,
- consists of a polynomial number of processors,
- completes the computation in constant time.

The classes $LRN$, $NC-LRN$, $HV-RN$, tree-$RN$, $DRN$ and $NMRN$ are defined analogously. The results from [Sch91, BPS91] for the relations between reconfigurable complexity classes and parallel and traditional complexity classes are summarized in Figure 8. Established connections are drawn by arrows. Downward vertical arrows hold trivially and are omitted.

### 7.3 Undirected models

Let us first consider the relative power of the LRN and the NC-LRN models. NC-LRN may be easier to implement than LRN. That is since in certain technologies when a signal is transmitted on a cycle of connected links it may not stabilize with the correct value. We thus say that an error was detected on the bus. On the other hand, it may be easier to design algorithms in the LRN model, as one does
not have to avoid cycles in the resulting buses. In any case, the following implies that, up to a factor of \( 2 \) in the algorithm runtime, it does not matter which of the models is actually implemented.

Claim 7.1 Let \( \mathcal{R} \) be a reconfiguring network, and let \( \mathcal{R}_{LRN} \) and \( \mathcal{R}_{NC-LRN} \) denote the network \( \mathcal{R} \) operating in the LRN model and the NC-LRN models, correspondingly. Then \( \mathcal{R}_{NC-LRN} \) simulates \( \mathcal{R}_{LRN} \) and \( \mathcal{R}_{LRN} \) simulates \( \mathcal{R}_{NC-LRN} \) with slowdown \( 2 \).

In the context of complexity classes, Claim 7.1 implies:

Corollary 7.1 \( LRN = NC-LRN \).  

Proof: Let us show how \( \mathcal{R}_{NC-LRN} \) simulates a step of \( \mathcal{R}_{LRN} \), in two steps. The location of inputs and outputs is the same for the simulating and the simulated networks, hence the simulation may be extended to algorithms of any runtime with slowdown of \( 2 \).

During both steps of the simulation of (the only step of) the original algorithm \( A \), a processor \( x \) which is on a bus \( B \) but is not a speaker of \( B \), keeps the corresponding ports connected as in \( A \). If it is a reader in \( A \), then it reads the message that is transmitted on \( B \) during the second simulation step. We will make sure that this is the same message that is read by \( x \) on \( B \) in \( A \). Suppose now that \( x \) is a speaker of \( B \). Without loss of generality, we assume that \( x \) has two ports \( p_1 \) and \( p_2 \) connecting edges which participate in \( B \).

During the first simulation step, \( x \) disconnects \( p_1 \) from \( p_2 \) but all other connections are kept as in \( A \). \( x \) then transmits his \( id_x \) on \( p_1 \) while listening on \( p_2 \). There are three possible results:

1) If \( x \) reads \( id_x \) on port \( p_2 \), then \( B \) is a cycle with \( x \) as its only speaker. Thus, in this case, at the next step \( x \) keeps the same local configuration, and transmits the original message, making sure every reader on \( B \) gets it.

2) If \( x \) reads \( id_y \) on port \( p_2 \) then clearly there are at least two speakers on \( B \). This indicates that in \( A \) there is an error detected on \( B \). Thus at the next step \( x \) connects \( p_1 \) and \( p_2 \) and transmits the original message on the resulted bus. This would cause an error state on \( B \) during the 2nd simulation step, since all speakers of \( B \) will be doing it whether \( B \) is a cycle or not.

3) If \( x \) does not detect anything on port \( p_2 \) during the first simulation step, then there may be of two possible situations.

   (i) \( B \) is a cycle having at least one more speaker.

   (ii) \( B \) is not a cycle.

In any case, \( x \) connects \( p_1 \) to \( p_2 \) and transmits the original message on \( B \) (like it did in case 2). It is easy to verify that what is heard by the readers is consistent with \( A \).

Two steps simulation by \( \mathcal{R}_{LRN} \) of a step of \( \mathcal{R}_{NC-LRN} \) is analogous to the above. The only difference is that we need a special message that is transmitted by \( x \) in case 1 (\( x \) is the only speaker of a bus \( B \) which is a cycle), telling all the readers of \( B \) that in the original step \( A \) there was an error detected on the bus.  

As is depicted in Figure 8, it is our belief that there is a gap in the power of \( LRN \) and \( RN \). One example is Transitive Closure which is known to be in \( RN \) [Wan91] whereas \( s-t \) connectivity is not known to be computed in logspace \((L)\). Here we would like to strengthen our understanding of what is the exact property of the RN model which makes it stronger. Indeed the processors of the tree-RN model posses the same capability as the processors of the RN model in connecting any subset of links, rather than pairs only as is with the LRN model. Yet, it is not possible in the tree-RN model to connect cycles.

The following claim shows that, as opposed to the equivalence in the power of LRN and NC-LRN, the main source of the extra power of the RN model is the ability to connect cycles. In other words, the proof shows that when a RN algorithm uses buses having no cycles, or when cycles can be easily avoided, then there is an efficient LRN algorithm for the same task.

**Claim 7.2** Let \( \mathcal{R} \) be a network computing in the tree-RN model. Let \( \mathcal{R}' \) be a network computing in the LRN model, where \( \mathcal{R}' \) has the same topology of \( \mathcal{R} \), except that each edge (bus component) \( e \) in \( \mathcal{R} \) is replaced by two edges \( e^1, e^2 \) in \( \mathcal{R}' \). Thus, each node of \( \mathcal{R}' \) has twice the number of ports as the corresponding node in \( \mathcal{R} \) and the network \( \mathcal{R}' \) has twice the number of edges of \( \mathcal{R} \). In this setting \( \mathcal{R}' \) can simulate \( \mathcal{R} \) with no slowdown.

In the context of complexity classes, Claim 7.2 implies:

**Corollary 7.2** \( LRN = \text{tree} - RN \).  

**Proof:** Let \( x \) be a processor of \( \mathcal{R} \) and \( x' \) the corresponding processor in \( \mathcal{R}' \). Suppose at some step \( x \) assumes a local configuration, connecting a set of \( J \) of \( x' \)'s edges, say \( \{e_0,e_1,\ldots ,e_{J-1}\} \), taking part in the same bus. \( x' \) simulates \( x \) at that step by connecting the matching \( \{e_1^1,e_{i+1 \mod J}^2\} \) for \( 0 \leq i \leq J-1 \). Note that for an edge \( e \) which is not connected by \( x \) to any other edge, \( x' \) connects \( \{e^1,e^2\} \). When \( x \) either reads or speaks on some port \( e \), \( x' \) simulates this operation using the port \( e^1 \).

Any connected bus \( B \) which is formed in \( \mathcal{R} \) results in a connected bus \( B' \) which is formed in \( \mathcal{R}' \), where \( B \) and \( B' \) connect the same network nodes.  

It is instructive to look at the same problem on the mesh topology.

**Claim 7.3** An \( n \)-mesh computing in the tree-RN model is simulated with no slowdown by a \( 2n \)-mesh computing in the LRN model.

**Proof:** Basically, each processor \( x \) in the \( n \)-mesh is simulated by four processors (a \( 2 \times 2 \) sub-mesh) in the \( 2n \)-mesh. These essentially follow the same procedure carried by \( x' \) in the proof of Claim 7.2.

Putting in the results from Section 5 we thus conclude the following.

**Corollary 7.3** The simulation of the tree-RN \( n \)-mesh by the LRN \( p \)-mesh \((p \geq 2)\) is completed with slowdown \( \Theta((n/p)^2) \).
7.4 Directed models

We now turn to consider simulations of directed models by undirected models and vice versa.

Claim 7.4 A p-mesh computing in the RN model is simulated with no slowdown by a 2p-mesh computing in the DRN model.

Proof: The direction of the rows and columns of the 2p-mesh is changing, so that odd rows are directed leftward and even rows are directed rightward, odd columns are directed downward and even columns are directed upward. Again, each processor \( z \) in the \( p \)-mesh is simulated by a 2-sub-mesh in the \( 2p \)-mesh.

![RN configurations](image)

![DRN configurations](image)

Figure 9: Simulating RN mesh-configurations by DRN mesh-configurations, four DRN nodes per original RN node. RN nodes drawn with dashed lines, DRN nodes with solid lines. Direction of edges is given by arrows.

Several simulating configurations are depicted in figure 9. A port \( p \) of the RN mesh is replaced by an in-port \( p_{in} \) and an out-port \( p_{out} \) of the DRN mesh. When \( p \) is connected to other ports of the switch then both \( p_{in} \) and \( p_{out} \) are connected to the corresponding in and out ports. It is easy to verify that all configurations can be consistently constructed this way.

Corollary 7.4 A p-mesh computing in the LRN model is simulated with no slowdown by a 2p-mesh computing in the DLRN model.

Proof: Use a subset of the local configurations as in the proof of Claim 7.4.
Claim 7.5 A p-mesh computing in the DLRN model is simulated with no slowdown by a p-mesh computing in the LRN model.

Proof: Use the same bus configurations, except that speaking processors disconnect the bus in the direction which is not supported by the directed model.

Acknowledgements

David Peleg made the initial suggestion of using the window method rather than contraction.

Several of the observations that are made throughout this article were also made independently by Koji Nakano [Nak].

References


