1999

Scaling Simulations of Reconfigurable Meshes.

Jose Alberto Fernandez zepeda

Louisiana State University and Agricultural & Mechanical College

SCALING SIMULATIONS OF RECONFIGURABLE MESHES

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Electrical and Computer Engineering

by
José Alberto Fernández Zepeda
B.S., Universidad Nacional Autonóma de México, 1991
M.S., Universidad Nacional Autonóma de México, 1994
December 1999

Acknowledgments

I sincerely thank my advisors Dr. Jerry L. Trahan and Dr. Ramachandran Vaidyanathan for their supervision, guidance, patient, and for all the time that they dedicated to me during my studies at Louisiana State University.

I am grateful for the financial support provided by Consejo Nacional de Ciencia y Tecnología (CONACYT), the program Fulbright-IEE, the National Science Foundation, and the Department of Electrical and Computer Engineering at LSU.

I dedicate this dissertation to my mother Ana María Zepeda for her love and encouragement throughout my life.

Finally, I would like to thank all my friends for a memorable time at LSU, especially to Anu Bourgeois for her sincere friendship.

Table of Contents

List of Figures

Fernández Zepeda, José Alberto, B.S., Universidad Nacional Autonóma de México, 1991
M.S., Universidad Nacional Autonóma de México, 1994
Doctor of Philosophy, Fall Commencement, 1999
Major: Electrical Engineering; Minor: Mathematics
Scaling Simulations of Reconfigurable Meshes
Thesis directed by Associate Professor Jerry L. Trahan and Associate Professor Ramachandran Vaidyanathan
Pages in thesis, 128. Words in abstract, 312.

ABSTRACT

This dissertation deals with reconfigurable bus-based models, a new type of parallel machine that uses dynamically alterable connections between processors to allow efficient communication and to perform fast computations. We focus this work on the Reconfigurable Mesh (R-Mesh), one of the most widely studied reconfigurable models.

We study the ability of the R-Mesh to adapt an algorithm instance of an arbitrary size to run on a given smaller model size without significant loss of efficiency. A *scaling simulation* achieves this adaptation, and the *simulation overhead* expresses the efficiency of the simulation. We construct a scaling simulation for the Fusing-Restricted Reconfigurable Mesh (FR-Mesh), an important restriction of the R-Mesh. The overhead of this simulation depends only on the simulating machine size and not on the simulated machine size. The results of this scaling simulation extend to a variety of concurrent write rules and also translate to an improved scaling simulation of the R-Mesh itself.

We present a *bus linearization* procedure that transforms an arbitrary non-linear bus configuration of an R-Mesh into an equivalent acyclic linear bus configuration implementable on an Linear Reconfigurable Mesh (LR-Mesh), a weaker version of the R-Mesh. This procedure gives the algorithm designer the liberty of using buses of arbitrary shape, while automatically translating the algorithm to run on a simpler platform. We illustrate our bus linearization method through two important applications. The first leads to a faster scaling simulation of the R-Mesh. The second application adapts algorithms designed for

R-Meshes to run on models with pipelined optical buses.

We also present a simulation of a Directional Reconfigurable Mesh (DR-Mesh) on an LR-Mesh. This simulation has a much better efficiency compared to previous work. In addition to the LR-Mesh, this simulation also runs on models that use pipelined optical buses.

Chapter 1

Introduction

In recent years, reconfigurable models have drawn considerable interest and numerous fast algorithms have been proposed for them. These models use dynamically alterable connections between processors not only to allow efficient communication, but also to perform computation faster than on conventional "non-reconfigurable" models. Researchers have proposed a number of reconfigurable models including the Reconfigurable Mesh (R-Mesh) [20, 26, 33], Reconfigurable Network (RN) [6], Polymorphic Processor Array (PPA) [30], Processor Array with Reconfigurable Bus System (PARBS) [54], Reconfigurable Multiple Bus Machine (RMBM) [50], Reconfigurable Buses with Shift Switching (REBSIS) [28], and Distributed Memory Bus Computer (DMBC) [42]. Nakano [37] presented a bibliography of published research on reconfigurable models.

This dissertation deals with the ability of a reconfigurable model to adapt an algorithm instance of an arbitrary size to run on a given smaller model size without significant loss of efficiency. A *scaling simulation* achieves this adaptation, and the *simulation overhead* expresses the efficiency of the simulation. For most conventional models, such a scaling simulation is trivial. For reconfigurable models, however, the problem presents several challenges as explained later.

In this dissertation, we focus our attention on the R-Mesh (one of the most widely studied reconfigurable models) and some of its variants. We present new and faster scaling simulations for these models using various write rules. We also demonstrate how some of these models can simulate each other, and describe some important applications of these results.

We organize the remainder of this chapter as follows. Section 1.1 illustrates the main features of reconfigurable models through some examples. Section 1.2 defines the concept of scaling simulations for models of parallel computation and describes previous work on this aspect of reconfigurable models. Section 1.3 describes the scope of the dissertation and Section 1.4 details the main contributions of this work. Finally, Section 1.5 outlines the organization of the dissertation.

1.1 Reconfiguration

A reconfigurable (bus-based) model operates by creating elaborate patterns of "buses" between processors. Some of the most important features of such a model are the following.

1. It uses an internal port connection mechanism to segment or fuse buses.

2. Each processor can independently change its internal port connections at each step.

3. It assumes a constant propagation delay on buses.

4. It uses its buses as a computational resource.

We now present an example that illustrates these features and their use in constructing a very fast algorithm. Figure 1.1 shows an eight-processor reconfigurable linear array (RLA). Each processor connects directly to its neighboring processors through two ports (left and right). Each processor is permitted to internally connect or disconnect its ports.

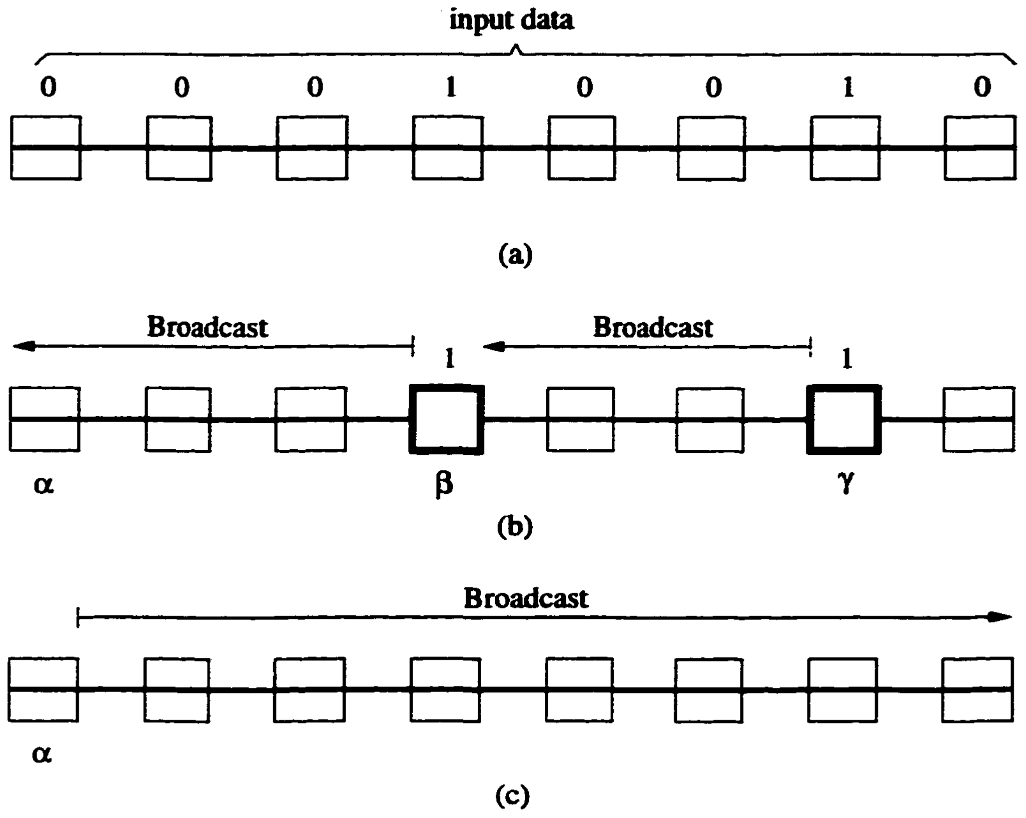


Figure 1.1: A reconfigurable linear array computing the OR function: a) initial configuration and input data for each processor; b) processors holding '1' split the bus and write '1' to their left bus; c) processor α broadcasts the result to all the processors.

In this example, the RLA calculates the OR function of eight bits. Each processor holds an input bit and assumes the port connection depicted in Figure 1.1(a). The algorithm proceeds as follows.

Each processor that holds a '1' splits the bus by disconnecting its ports; otherwise, it keeps the bus intact (see Figure 1.1(b)). Each processor that holds a '1' writes to the bus through its left port and the leftmost processor, α, reads the result of the OR function from its left port. If all processors hold '0's, then there is no write to the bus and processor α reads a "null value" indicating that the result is '0'. If at least one processor holds a '1', then processor α reads a '1' from the bus. This value is written by the processor nearest to α that holds a '1' (processor β in Figure 1.1(b)). Processors holding '0's just provide an unbroken bus, so the value written by processor β reaches processor α. Finally, processors connect their ports (as in Figure 1.1(c)) and α broadcasts the result to all processors.

This example illustrates most of the basic features of a reconfigurable model. The method of this example readily generalizes to a constant-time OR algorithm for N bits on an N-processor RLA. Notice how each processor can disconnect or connect its ports to split the bus or fuse bus segments. Also notice how each processor can change its port connections at each step. This is a local decision (based only on input data or data read from the bus) and is independent of the decision taken by neighboring processors. In general, reconfigurable models are Single-Instruction, Multiple-Data (SIMD) machines, where all processors execute the same program, the program for computing the OR function in this case. These models operate synchronously, so all processors change port connections, write to buses, read from buses, and perform computations at predefined cycles of a master clock. Notice that in this example we assume that the propagation delay is constant, so processor α broadcasts the result to all the processors connected to the bus in a single step. (This assumption is a good approximation for medium sized machines [26, 30, 33, 42].) In contrast, in an N-processor (non-reconfigurable) linear array, a broadcast takes $O(N)$ time. Notice how the data paths play an important role in determining the answer to the problem in question. The Parallel Random Access Machine (PRAM), a very popular model of parallel computation, requires $\Omega(\log N)$ time to compute the OR function on N bits, if only exclusive writes are allowed. In the above example, the RLA solves the problem in constant time using only exclusive writes; notice how writing processors (such as β and γ in Figure 1.1(b)) write to different buses.

The one-dimensional RLA readily extends to a two-dimensional Reconfigurable Mesh (R-Mesh). An R-Mesh processor has four ports connecting it to its neighbors to the North, South, East, and West. (We formally define the R-Mesh in Chapter 2.) In our next illustration, we use an R-Mesh to compute the sum of 8 bits (Figure 1.2); for simplicity, we show only the bottom four rows of the R-Mesh.

Each processor in the bottom row of the R-Mesh holds one input bit. The algorithm proceeds as follows. The processor at the bottom of each column broadcasts its bit to all the processors in its column. Each processor reading a '1' from its vertical bus, connects its West port to its North port, and its East port to its South port; otherwise, it just connects its West and East ports (see Figure 1.2).

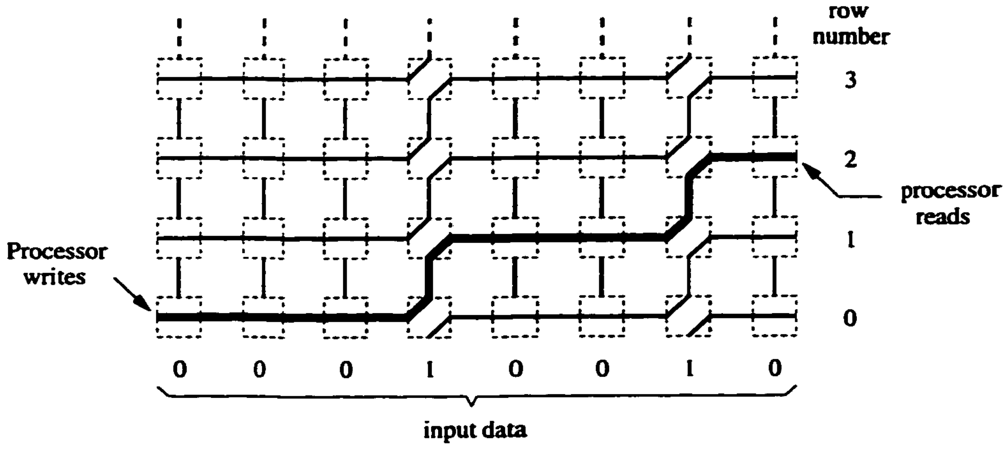
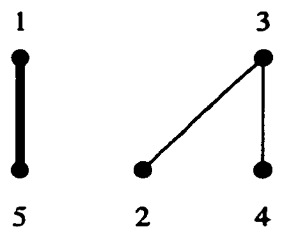


Figure 1.2: Summation of eight binary bits on an R-Mesh.

This internal port connection combined with the external links between processors, creates a staircase-like bus structure. In fact, the bus originating at the bottom left processor (shown in bold in Figure 1.2) steps up by one row for each '1' in the input. The processor on the bottom left corner writes a signal on its West port and each processor of the rightmost column reads from its East port. The sum of the input bits equals x ($x = 2$ in our example) if and only if the signal arrives at the East port of the rightmost processor of row x. In general, this method sums N bits in constant time on an $(N+1) \times N$ R-Mesh.

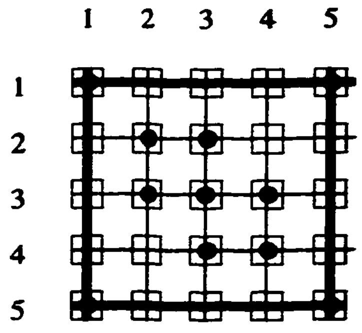
The main idea of this algorithm is to configure buses so that a signal sent at a fixed point of the R-Mesh "arrives at the answer." Similar techniques can be used to solve a variety of fundamental problems, such as prefix sums, multiplication, Boolean matrix multiplication, and sorting [16, 21, 22, 29, 35, 36, 38].



(a)

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 1 & 0 & 0 & 0 & 1 \\ 2 & 0 & 1 & 1 & 0 & 0 \\ 3 & 0 & 1 & 1 & 1 & 0 \\ 4 & 0 & 0 & 1 & 1 & 0 \\ 5 & 1 & 0 & 0 & 0 & 1 \end{array}$$

(b)



(c)

Figure 1.3: Example of a graph algorithm: a) Graph with 2 components; b) Adjacency matrix; c) Embedding on the R-Mesh. The "bold" buses in (c) correspond to the component shown in bold in (a).

Another class of algorithms where reconfiguration is advantageous is graph algorithms. We illustrate one such algorithm in Figure 1.3, where the $N \times N$ adjacency matrix of an N-node graph is directly embedded into an $N \times N$ R-Mesh. This embedding has the property that nodes i and j have a path between them in the graph if and only if diagonal R-Mesh processors (i, i) and (j, j) are incident on the same bus. This property allows the R-Mesh to solve problems like s-t connectivity, connected components, and transitive closure in constant time [54]. Similar techniques have been used to construct constant time algorithms for other graph problems, such as spanning trees, biconnected components, Euler tour, and tree traversal [2, 9, 25, 31, 49].

The planar topology of the R-Mesh also makes it suitable for problems in image processing (where each processor represents a pixel of the image) and planar computational geometry. These include image labeling, template matching, histogramming, convex hull, dominance counting, Voronoi diagrams, and other proximity problems [1, 10, 11, 18, 20, 23].

1.2 Scaling Simulations and Previous Work

Let $\mathcal{M}(N)$ denote an N-processor instance of a model, $\mathcal{M}$, of parallel computation. A *scaling simulation* for $\mathcal{M}$ is an algorithm that simulates an arbitrary step of $\mathcal{M}(N)$ on a smaller instance $\mathcal{M}(P)$, for any $P < N$. In general, this simulation runs in $\Theta\left(\frac{N}{P}f(N,P)\right)$ steps. Clearly, the work of N processors on P processors takes $\Omega\left(\frac{N}{P}\right)$ steps, therefore, $f(N,P)$, a non-decreasing function, is the *simulation overhead.* (The scaling simulation serves to establish that any algorithm designed to run in T steps on $\mathcal{M}(N)$ can run in $O\left(\frac{N}{P}f(N,P)\cdot T\right)$ steps on $\mathcal{M}(P)$.)

Definition 1 For any $P < N$, let $\mathcal{M}(P)$ simulate a step of $\mathcal{M}(N)$ in $O\left(\frac{N}{P}f(N,P)\right)$ time.

(i) Model $\mathcal{M}$ has an *optimal scaling simulation* iff $f(N,P) = O(1)$.

(ii) Model $\mathcal{M}$ has a *strong scaling simulation* iff $f(N,P)$ is independent of N and $f(N,P) = o(P)$.

(iii) Model $\mathcal{M}$ has a *weak scaling simulation* iff it does not have an optimal or strong scaling simulation. ■

If a model possesses an optimal scaling simulation, then a programmer need not be concerned with the actual size of the machine on which a program is to run. In this case, the best algorithm serves well on all model and problem sizes. On a model with a strong scaling simulation, a single algorithm will serve all problem sizes as the simulation overhead is independent of N, the simulated model size. A compiler, that is in any case local to the model or machine instance, can map logical processors (defined by the algorithm and problem instance) to physical processors. On a model with a weak scaling simulation, however, the fastest algorithm for given problem and model sizes may not be the fastest (after scaling) for other problem and model sizes; it may have to be fine-tuned or possibly even replaced by another algorithm for different problem sizes. (This approach is taken in practice in some parallel machines, for example [8, 16].) With a weak scaling simulation, however, different problem sizes would call for different algorithms, and so different programs, to run in the best possible time on the same available machine. For a model with an optimal or strong scaling simulation, algorithmic results have significance whether or not problem size matches machine size. Algorithm development itself may be easier on these models (for example, using the work-time framework [19] for PRAM algorithms).

In traditional "non-reconfigurable" models, a large model instance $\mathcal{M}(N)$ can be simulated optimally by a small model instance $\mathcal{M}(P)$, simply by letting a processor of $\mathcal{M}(P)$ simulate $\frac{N}{P}$ processors of $\mathcal{M}(N)$. Thus these models have optimal scaling simulations. The difficulty in scaling algorithms for reconfigurable models stems from the fundamentally different way in which they perform computation. A sequence of steps typical to many algorithms is as follows: (a) processors configure themselves locally to establish a global pattern of buses interconnecting the processors; (b) a designated processor issues a special signal at a fixed position in the bus structure;

(c) the processors deduce an answer depending on where the signal arrives. (The summing algorithm of Figure 1.2 is an example of such an algorithm.) In this setting, consider a problem that has N possible answers. If a model instance is not large enough to accommodate N distinct answers (positions for signal arrival), then the above method will not work.

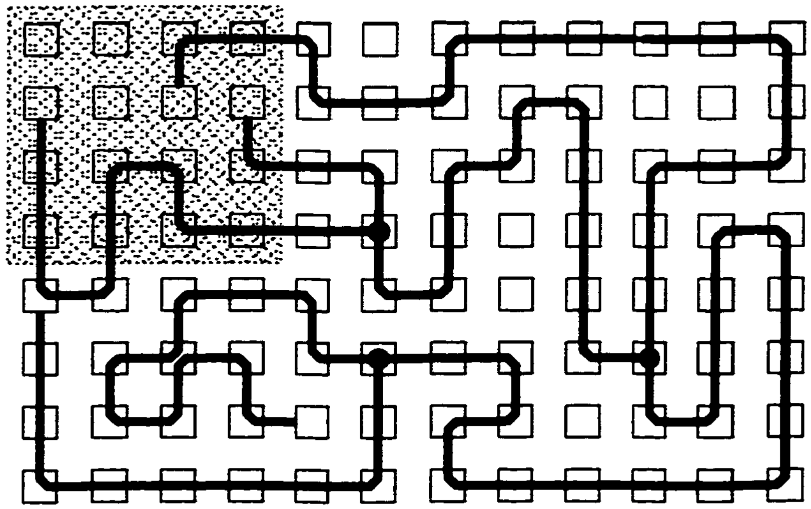


Figure 1.4: Simulating an R-Mesh using an smaller R-Mesh (shadow square).

The great variety of bus shapes in an R-Mesh, especially branches and cycles, make it difficult to design an efficient scaling simulation. To illustrate this problem, consider Figure 1.4, which shows a 4×4 R-Mesh (shown as a shaded square) simulating a "window" of an 8×12 R-Mesh. In the window, the simulating R-Mesh detects four buses that are separated within the window, but are, in fact, part of a single bus. The simulating machine has to label each bus, keep track of them through the entire simulation, and update their labels when they fuse to other buses. This task is not trivial because of the large number of possible bus configurations and is the main cause of the simulation overhead.

This notion of scaling simulation can be generalized to that of scaling simulations between different models. Specifically, we will consider scenarios where a single step of $\mathcal{M}_1(N)$, a model of size N is simulated by $\mathcal{M}_2(P)$ a smaller instance of a different model in $O\left(\frac{N}{P}f(N,P)\right)$ time.

Reconfigurable models possess a large body of fast algorithms, yet only a handful of results exist for scaling algorithms on these models. Previously, Maresca [30] established that the Polymorphic Processor Array (PPA) possesses an optimal scaling simulation. The PPA restricts the pattern of buses that can be created, severely curtailing the power of the model [50]. Ben-Asher *et al.* [4] proved that the Linear Reconfigurable Mesh (LR-Mesh), a restriction of the R-Mesh, has an optimal scaling simulation. The LR-Mesh admits only certain patterns of buses, making it unsuitable for some fundamental problems such as graph connectivity. Murshed and Brent [34] defined certain global restrictions on bus configurations and designed simpler optimal scaling simulations for LR-Meshes under these restrictions. Ben-Asher *et al.* [4] developed a (weak) scaling simulation for an $N \times N$ (unrestricted) R-Mesh on a $P \times P$ R-Mesh that has a simulation overhead of $\log N \log \frac{N}{P}$. Matias and Schuster [32] proposed a randomized scaling simulation for the unrestricted R-Mesh on the LR-Mesh; their method has a constant (with high probability) simulation overhead, only when $P \leq \frac{N}{\log N \log\log N}$, and uses the "ARBITRARY" concurrent-write rule, a rule not easily implementable on a bus. On other reconfigurable models, Trahan *et al.* [48] developed an $O(\log N)$ simulation of an $N \times N$ Directional R-Mesh on an $O(N^4 \times N^4)$ LR-Mesh. Trahan and Vaidyanathan [51] have shown that, for certain restrictions of local connections, the Reconfigurable Multiple Bus Machine (RMBM) has a strong scaling simulation. Trahan *et al.* [47] developed a number of algorithms that scale with optimal overhead on the Linear Array with Reconfigurable Pipelined

Bus System (LARPBS), a reconfigurable model that uses pipelined optical buses for communication.

1.3 Scope of the Dissertation

All prior approaches to scaling reconfigurable models either severely restrict the simulated model and/or grant extra capabilities to the simulating model (in order to achieve constant overhead), or incur a high simulation overhead. We consider a restriction of the R-Mesh, called the Fusing Restricted R-Mesh (FR-Mesh), for which we construct a strong scaling simulation. The FR-Mesh is as "powerful" as the unrestricted R-Mesh [45], though it allows only two of the fifteen internal connections possible on the R-Mesh (see Section 2.2). Further, the FR-Mesh admits constant time algorithms for fundamental problems (such as *s*-*t* connectivity, connected components, transitive closure, and cycle detection [2, 25, 54]) that are unlikely to be solvable in constant time on the LR-Mesh; many such problems are fundamental to algorithm development in general [53].

In Chapter 3, we construct a strong scaling simulation of the FR-Mesh in which the simulation overhead is logarithmic in the simulating machine size, and entirely independent of the simulated machine size. More precisely, we establish that for any $P < N$, a step of an $N \times N$ FR-Mesh (that has N^2 processors) can be simulated by a $P \times P$ FR-Mesh in $O\left(\frac{N^2}{P^2} \log P\right)$ time.

Additionally, we identify the bottleneck producing the simulation overhead of the FR-Mesh scaling simulation as "leader election." Thus, any improvement in techniques for leader election will immediately translate to a further reduction of the overheads for scaling simulations of both the FR-Mesh and the R-Mesh. Indeed, an

FR-Mesh that can resolve concurrent writes by the "PRIORITY" rule has an optimal scaling simulation (Section 3.6).

Although most of the dissertation uses the "COMMON" concurrent write rule, we also consider other rules such as "COLLISION" and "COLLISION^{+}" (Section 3.6) that are well known in the context of PRAM algorithms [19, 24]. For these rules, the FR-Mesh still has a strong scaling simulation with a logarithmic simulation overhead.

The strong scaling simulation of the FR-Mesh also leads to an improved (weak) scaling simulation of the R-Mesh (Section 3.7). The simulation overhead for this simulation is $\log P \log \frac{N}{P}$; the previous fastest scaling simulation for the R-Mesh [4] had a simulation overhead of $\log N \log \frac{N}{P}$ (see Table 1.1).

The R-Mesh can create bus structures of many different shapes (see Figure 2.1). On the one hand, flexibility in shaping buses facilitates algorithm design and can reduce running time, but on the other hand, complex bus shapes complicate implementation of these models. In Chapter 4, we present a procedure called *bus linearization* that transforms a bus of any shape allowed by the R-Mesh into one with an equivalent linear (non-branching) structure.

Specifically, we prove that an $N \times N$ LR-Mesh (an R-Mesh restriction that permits only linear buses) can simulate an arbitrary step of an $N \times N$ R-Mesh in $O(\log N)$ time. We illustrate the use of bus linearization through two important applications. The first constructs the best known deterministic scaling simulation for the R-Mesh. This approach has $\log N$ simulation overhead, which improves on the simulation overhead of $\log P \log \frac{N}{P}$ (Chapter 3). The second application adapts algorithms designed for the R-Mesh to run on reconfigurable models that use optical buses [40, 44, 45].

Bus linearization also improves the FR-Mesh scaling simulation of Chapter 3 to a weaker simulating model (see Table 1.1); the simulation in Chapter 3 requires a

"CRCW" FR-Mesh (with the ability to perform concurrent writes), while the simulation of Chapter 4 needs only a "CREW" LR-Mesh (without the need to perform concurrent writes).

In Chapter 5, we present the simulation of a Directed Reconfigurable Mesh (DR-Mesh) on an LR-Mesh. The DR-Mesh has directed buses with the ability to restrict data propagation to only one direction. We simulate an $N \times N$ DR-Mesh in $O(\log^2 N)$ time on an $O(N \times N \times \frac{N}{\log N})$ (three-dimensional) R-Mesh and on an $O(\frac{N^2}{\log N} \times \frac{N^2}{\log N})$ (two-dimensional) R-Mesh. This result is a substantial improvement on the only previous work on this problem [48], where an $O(N^4 \times N^4)$ R-Mesh achieves $O(\log N)$ time. Furthermore, we showed that the simulating machine can be a Pipelined Reconfigurable Mesh (PR-Mesh) or an equivalent pipelined optical model, thereby extending the scope of the simulation.

1.4 Contributions of This Work

On the whole, this dissertation provides a better understanding about scalability of reconfigurable models in general, and the R-Mesh and its variants, in particular. It presents new approaches to problems in scalability, many of which could be useful for scaling simulations on other reconfigurable models as well. We present several simulations among reconfigurable models. Tables 1.1 and 1.2 summarize the contribution of these simulations in the context of existing results.

Some algorithms run more efficiently on LR-Meshes than on FR-Meshes and vice-versa. The class of *separable R-Mesh algorithms* comprises algorithms in which each step runs on either an LR-Mesh or an FR-Mesh. Separable algorithms provide an extremely rich array of fast and efficient algorithmic building blocks. The main result in Chapter 3 is a strong scaling simulation of the FR-Mesh. This result, coupled with the optimal scaling simulation for the LR-Mesh [4] allows separable algorithms to scale with an overhead that depends only on the simulating machine size.

Table 1.1: Summary of scaling simulations

Simulated model	Simulating model	Simulation overhead	Reference
CRCW LR-Mesh	CRCW LR-Mesh	$O(1)$	[4]
CRCW FR-Mesh	CRCW FR-Mesh	$O(\log P)$	This work
	CREW LR-Mesh	$O(\log P)$	This work
	CREW PR-Mesh	$O(\log P)$	This work
CRCW R-Mesh	CRCW R-Mesh	$O(\log N \log \frac{N}{P})$	[4]
	CRCW R-Mesh	$O(\log P \log \frac{N}{P})$	This work
	COLLISION CRCW LR-Mesh†	$O(\log P)$ w.h.p.	[32]
	ARBITRARY CRCW LR-Mesh†	$O(1)$ w.h.p.	[32]
	CREW LR-Mesh	$O(\log N)$	This work
	CREW PR-Mesh	$O(\log N)$	This work

The sizes of the simulated and simulating models are $N \times N$ and $P \times P$, respectively.
†The simulating model is randomized and its simulation overhead is with high probability.

The main result of Chapter 4 is the construction of a bus linearization algorithm. This algorithm transforms any R-Mesh algorithm to run on the optimally scalable LR-Mesh. Bus linearization gives an algorithm designer the liberty of using buses of arbitrary shape, while automatically translating the algorithm to run on a more implementable platform. Bus linearization runs on an LR-Mesh with only exclusive writes, whereas the simulating machine of all prior scaling simulations required concurrent writes (see Table 1.1).

Furthermore, bus linearization also transforms FR-Mesh algorithms to run on an LR-Mesh. This feature automatically allows any separable algorithm to run on an LR-Mesh maintaining a simulation overhead of $\log P$. Bus linearization also facilitates the simulation of the R-Mesh and FR-Mesh on reconfigurable pipelined optical models, which require linear acyclic buses. Thus, the importance of bus linearization lies in its use in translating the vast body of R-Mesh and FR-Mesh algorithms to run on the LR-Mesh and reconfigurable pipelined optical models.

Table 1.2: Summary of DR-Mesh simulations

Simulated model	Simulating model size	Simulating model	Time	Reference
	$O(N^4 \times N^4)$	CRCW R-Mesh	$O(\log N)$	[48]
CRCW DR-Mesh	$O\left(N \times N \times \frac{N}{\log N}\right)$	CREW LR-Mesh	$O(\log^2 N)$	This work
	$O\left(N^2 \times \frac{N}{\log N}\right)$	CREW PR-Mesh	$O(\log^2 N)$	This work

The size of the simulated model is $N \times N$.

The main result of Chapter 5 is a simulation of a CRCW DR-Mesh on a CREW LR-Mesh and can be extended to reconfigurable pipelined optical models. Ben-Asher *et al.* [5] established that a constant time simulation of an $N \times N$ DR-Mesh on an R-Mesh bounded with a polynomial number of processors is not likely, while Trahan *et al.* [48] designed a simulation of each DR-Mesh step on an $O(N^4 \times N^4)$ R-Mesh in $O(\log N)$ time. The target of our simulation is to reduce the number of processors. Its contribution is a useful technique that dramatically reduces the size of the simulating model by a factor of $O(N^4 \log^2 N)$. Table 1.2 shows these results.

In addition to these broad results, this work has also generated several tools and techniques that may be of independent interest, such as prefix assimilation (Section 3.4.1), double bus structure (Section 4.1.2), and a terse representation for connectivity within R-Mesh "tiles" (Section 5.4).

1.5 Organization of the Dissertation

Chapter 2 defines the R-Mesh, some of its variants, and basic concepts such as concurrent write rules and leader election. Chapter 3 describes the FR-Mesh simulation and its use in obtaining an improved scaling simulation for the R-Mesh. Chapter 4 presents bus linearization and its applications. Chapter 5 deals with the simulation of the directed R-Mesh on an LR-Mesh. Finally, Chapter 6 summarizes this work and presents some directions for future work in the area.

Chapter 2

Definitions and Terminology

This chapter describes the reconfigurable models we use in this work. Also discussed are the definitions of several concurrent write rules and some mapping techniques used in scaling simulations.

2.1 The R-Mesh

An $R \times C$ Reconfigurable Mesh (R-Mesh) is a two-dimensional array of processors connected in an $R \times C$ grid. Each processor in the R-Mesh has direct connections to adjacent processors through its North, South, West, and East input/output ports. A processor can internally partition its set of four ports so that all ports in the same block of a partition are fused. This allows the R-Mesh to construct various bus patterns and to change them dynamically according to the requirements of the problem at hand. Figure 2.1 shows a 3×5 R-Mesh, depicting the fifteen possible port partitions of a processor. These partitions, along with external connections between processors, define a global bus structure consisting of a set of buses that weave through the ports. A *component* is a set of buses and ports that have a common connection. The R-Mesh and all its restricted versions assume a constant propagation delay on buses [26, 33, 42].

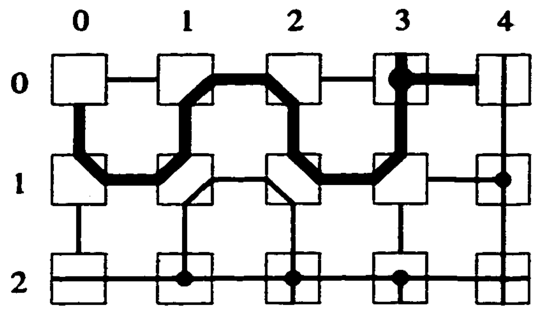


Figure 2.1: Internal connections of a 3 × 5 R-Mesh.

2.2 The FR-Mesh

The Fusing Reconfigurable Mesh (FR-Mesh) is a restricted version of the R-Mesh. Trahan *et al.* [45] proved that the class of languages accepted by an FR-Mesh is equivalent to the class SL (the same as the R-Mesh) of languages accepted in symmetric logarithmic space, on a Turing machine. That is, the FR-Mesh is as "powerful" as the R-Mesh though it allows only two of the fifteen internal connections possible on the R-Mesh, a *fusing* and a *cross-over* connection. A fusing connection joins all four ports (processor (0,2) in Figure 2.2), and a cross-over connection joins the North port with the South port and the West port with the East port (processor (0,0) in Figure 2.2). Because of the FR-Mesh connections, assume without loss of generality that each processor in an FR-Mesh has only two ports, the *vertical port* (North and South ports) and the *horizontal port* (East and West ports). The connections in an FR-Mesh allow a processor to directly connect to any other processor in its row (or column) via a *horizontal* (or *vertical*) *bus*. Figure 2.2 shows a 3 × 5 FR-Mesh with one component shown in bold. This component consists of buses in row 1 and column 3 and all the ports connected to these buses.

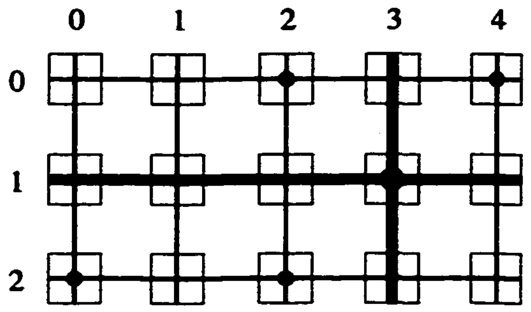


Figure 2.2: 3 × 5 FR-Mesh.

2.3 The LR-Mesh

The Linear Reconfigurable Mesh (LR-Mesh) is a restricted version of the R-Mesh. Each port in the LR-Mesh can connect to at most one other port in the same processor, so the LR-Mesh allows ten of the fifteen connections of the R-Mesh (all of the configurations of Figure 2.1 except those of processors (0,3), (1,4), (2,1), (2,2), and (2,3)). Ben-Asher *et al.* [4] constructed an optimal scaling simulation for the LR-Mesh. Ben-Asher *et al.* [5] also proved that the class of languages accepted by an LR-Mesh (resp., R-Mesh) is equivalent to the class L (resp., SL) of languages accepted in logarithmic space (resp., symmetric logarithmic space [39]) on a Turing machine. Although it has not been proved, the class L is conjectured to be a proper subset of the class SL, so the LR-Mesh is likely to be a weaker model than the R-Mesh. Figure 2.3 shows a 3×5 LR-Mesh with some typical bus configurations. The figure also shows a component in bold.

2.4 Concurrent Writes

Most of this dissertation deals with concurrent read, concurrent write (CRCW) reconfigurable models, in which several processors may simultaneously read from or

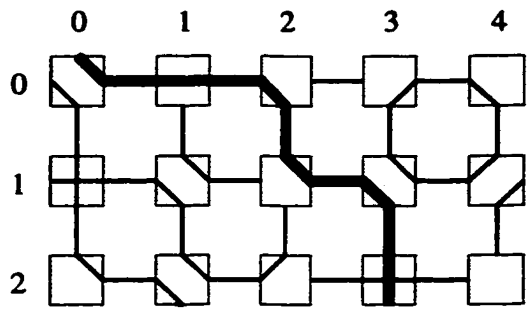


Figure 2.3: 3 × 5 LR-Mesh.

write to the same bus. Concurrent writes are resolved by the COMMON, COLLISION, COLLISION$^+$, PRIORITY, or ARBITRARY rules. These rules are well known in the context of PRAM algorithms [19, 24]. The COMMON rule allows concurrent writes only if all the values written to a component are equal. Under the COLLISION rule, if more than one processor attempts to write to a component, then a collision symbol is written. The COLLISION$^+$ rule behaves like the COMMON rule when all processors attempt to write the same value to a component, and like COLLISION otherwise. In the PRIORITY rule, the processor with highest priority among those attempting to write (usually the lowest indexed processor) wins the write conflict and writes its value. In the ARBITRARY rule, an arbitrary processor wins the write conflict and writes its value.

On a distributed shared resource such as a bus, only certain write rules such as COMMON, COLLISION, and COLLISION$^+$ [4] are feasible. Other rules exist, such as ARBITRARY and PRIORITY, whose physical implementations on a bus are not feasible, on the one hand, but on the other hand, they are a very useful algorithmic abstraction that simplifies algorithm design. In Sections 3.6 and 4.2.3, we present procedures to simulate models with different write rules and in Section 4.2.5, we present a procedure to remove the need of concurrent writes for the LR-Mesh.

In the following discussion, we refer to the process of choosing an element from a candidate pool by the PRIORITY (resp., ARBITRARY) rule as priority resolution (resp., arbitrary selection).

Leader election: This is a procedure that, for each component, selects a leader among a set of marked processors. (Marked processors may, for example, be the processors attempting to write.) The following procedure performs leader election by *priority resolution.* Let each processor have a unique $O(\log P)$-bit key (the key may be its index). By examining the keys bit-by-bit, the procedure reduces the set of potential candidates for the leader so that the elected leader is one with a highest (or lowest) key. Consequently, a COMMON CRCW R-Mesh can perform leader election among P processors in $O(\log P)$ time. Section 3.6.3 describes procedures to perform leader election by priority resolution using the COMMON, COLLISION, and COLLISION$^+$ rules.

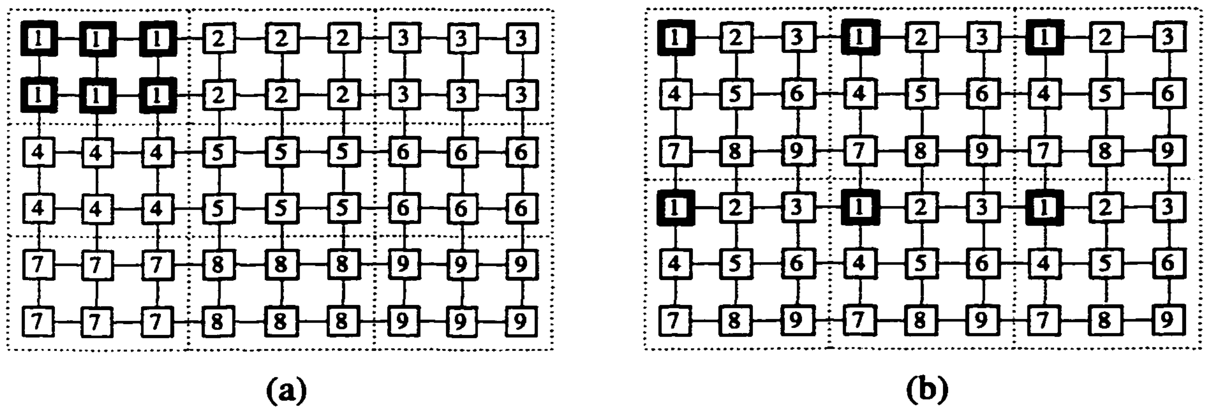


Figure 2.4: Mappings of 6×9 R-Mesh to 3×3 R-Mesh, where numbers indicate processors of the 3×3 R-Mesh: a) Contraction mapping; b) Windows mapping.

2.5 Contraction and Windows Mappings

Let $\mathcal{Q}$ be an $N \times N$ R-Mesh and $\mathcal{R}$ a $P \times P$ R-Mesh, where $P \leq N$. A scaling simulation for an R-Mesh involves the simulation of a step of $\mathcal{Q}$ on $\mathcal{R}$. For this, processors of $\mathcal{Q}$ must map to processors of $\mathcal{R}$.

The most obvious mapping is to let each processor of $\mathcal{R}$ simulate an $\frac{N}{P} \times \frac{N}{P}$ "sub-R-Mesh" of $\mathcal{Q}$. Ben-Asher *et al.* [4] called this the *contraction mapping* (see Figure 2.4(a), where the bold processors map to processor 1 of $\mathcal{R}$).

The *windows mapping* [4] divides $\mathcal{Q}$ into $\frac{N^2}{P^2}$ "windows", each a $P \times P$ sub-R-Mesh. In the simulation, each processor of $\mathcal{R}$ simulates the same processor from each window (see Figure 2.4(b), where the bold processors map to processor 1 of $\mathcal{R}$).

Ben-Asher *et al.* [4] proved that the contraction mapping will not allow an optimal scaling simulation for the LR-Mesh. Using a similar argument, an FR-Mesh scaling simulation that uses the contraction mapping has an overhead of $\Omega\left(\sqrt{N}\right)$ (proved in Section 3.2). Hence, we use the windows mapping to perform the scaling simulation of the FR-Mesh (see Chapter 3).

Chapter 3

FR-Mesh Scaling Simulation

This chapter presents a strong scaling simulation for the FR-Mesh [13, 14]. The following theorem summarizes the main result of this chapter.

Theorem 3.1 *For any $P < N$, any step of an $N \times N$* COMMON *CRCW FR-Mesh can be simulated on a $P \times P$* COMMON *CRCW FR-Mesh in $O\left(\frac{N^2}{P^2} \log P\right)$ time.* ■

The main objective is to simulate an arbitrary step of an $N \times N$ FR-Mesh, $\mathcal{Q}$, on a $P \times P$ FR-Mesh, $\mathcal{R}$. In the simulation, each processor of $\mathcal{R}$ simulates $\frac{N^2}{P^2}$ processors of $\mathcal{Q}$. Therefore, $\mathcal{R}$ can simulate local actions of processors of $\mathcal{Q}$ in $O\left(\frac{N^2}{P^2}\right)$ time. The global structure of $\mathcal{Q}$ is more difficult to simulate.

Chapter 3 is organized as follows. Section 3.2 establishes a lower bound for the FR-Mesh scaling simulation using the contraction mapping. Section 3.3 gives a general description of the FR-Mesh scaling simulation. Sections 3.4 and 3.5 describe the two main parts into which we have divided the FR-Mesh scaling simulation and that constitute the proof of Theorem 3.1. Section 3.6 explains how to modify the FR-Mesh scaling simulation to accommodate different write rules. Finally, Section 3.7 describes a new R-Mesh scaling simulation using the FR-Mesh.

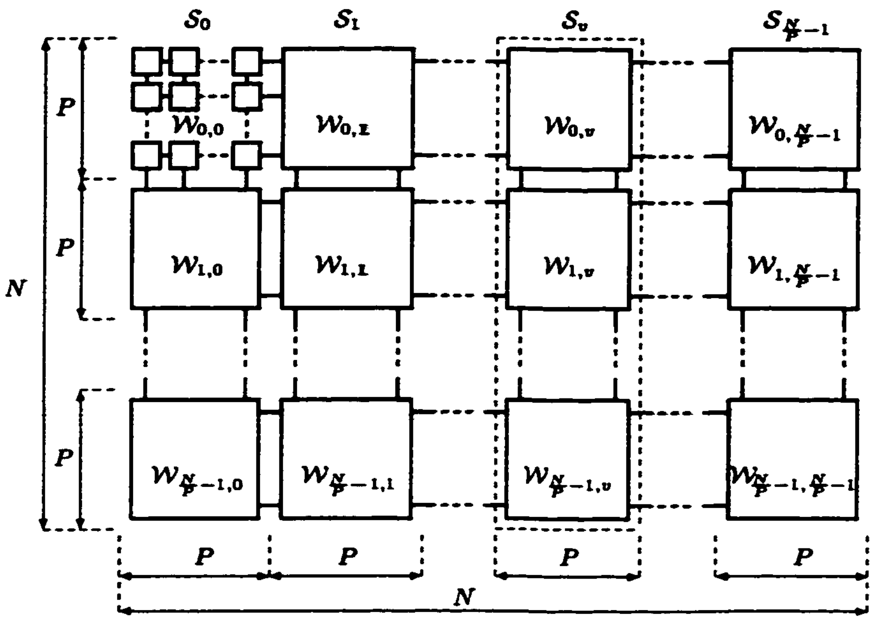

$\mathcal{S}_0$
$\mathcal{S}_1$
$\mathcal{S}_v$
$\mathcal{S}_{\frac{N}{P}-1}$
$\mathcal{W}_{0,0}$
$\mathcal{W}_{0,1}$
$\mathcal{W}_{0,v}$
$\mathcal{W}_{0,\frac{N}{P}-1}$
$\mathcal{W}_{1,0}$
$\mathcal{W}_{1,1}$
$\mathcal{W}_{1,v}$
$\mathcal{W}_{1,\frac{N}{P}-1}$
$\mathcal{W}_{\frac{N}{P}-1,0}$
$\mathcal{W}_{\frac{N}{P}-1,1}$
$\mathcal{W}_{\frac{N}{P}-1,v}$
$\mathcal{W}_{\frac{N}{P}-1,\frac{N}{P}-1}$
P
N

Figure 3.1: Slices and windows of the simulated $N \times N$ FR-Mesh.

3.1 Scaling Simulation Terminology

Let $\mathcal{Q}$ be the simulated machine, an $N \times N$ CRCW FR-Mesh. Let $\mathcal{R}$ be the simulating machine, a $P \times P$ CRCW FR-Mesh, where $P \leq N$. Without loss of generality, assume that $\frac{N}{P}$ is an integer. We now define the terminology used in presenting a scaling simulation.

Slice: The simulated FR-Mesh, $\mathcal{Q}$, contains $\frac{N}{P}$ slices, each an $N \times P$ sub-FR-Mesh (see Figure 3.1). Denote slice v by $\mathcal{S}_v$, where $0 \leq v < \frac{N}{P}$.

Window: Each slice, $\mathcal{S}_v$, contains $\frac{N}{P}$ windows, each a $P \times P$ sub-FR-Mesh (see Figure 3.1). For $0 \leq u, v < \frac{N}{P}$, denote window u of $\mathcal{S}_v$ by $\mathcal{W}_{u,v}$.

Bus index: This is an identifier assigned to each horizontal and vertical bus in $\mathcal{Q}$ according to its position. The horizontal bus in row i has bus index i, while the vertical bus in column j has bus index $j + N$, where $0 \le i, j < N$. For $0 \le b < 2N$, let $bus(b)$ denote the bus with index b.

Bus data: This is the value available on a bus at the end of a write cycle. If there is no write on a bus at the current cycle, then the bus data on the bus is said to be *null*; otherwise, it is *non-null*. Since the FR-Mesh permits resolution of concurrent writes by rules such as COLLISION, the data a port reads from the bus may not be the same as the value the same port wrote to the bus.

Component number: This is an identifier assigned to a component. In the simulation presented in this chapter, the component number is equal to the largest bus index among all buses in the component. Initially, when the simulation is unaware of any connections between buses of $\mathcal{Q}$, it assigns to each bus its bus index as its component number.

3.2 Mapping for FR-Mesh

Ben-Asher *et al.* [4] established that simulating the LR-Mesh using the contraction mapping (see Section 2.5) requires $\Omega(N)$ overhead, which does not allow an optimal or even a strong scaling simulation for the LR-Mesh.

By a similar argument, the required overhead to simulate an FR-Mesh using the contraction mapping is $\Omega\left(\sqrt{N}\right)$, and so we cannot use the contraction mapping to design a strong scaling simulation for the FR-Mesh.

Lemma 3.2 *For any $P < N$, simulating any step of an $N \times N$ FR-Mesh on a $P \times P$ FR-Mesh using the contraction mapping requires $\Omega(\sqrt{N})$ time.*

Proof: We construct an example of a bus configuration in an $N \times N$ FR-Mesh, $\mathcal{Q}$, that requires $\Omega(\sqrt{N})$ steps to simulate using a $P \times P$ FR-Mesh, $\mathcal{R}$. First, we describe this bus configuration.

Assign each vertical and horizontal bus in $\mathcal{Q}$ a label between 1 and $\sqrt{N}$; the labels for vertical buses, starting at the left, are as follows:

$1, 2, 1, 3, 1, 4, \ldots, 1, \sqrt{N}, 2, 3, 2, 4, \ldots, 2, \sqrt{N}, 3, 4, 3, 5, \ldots, 3, \sqrt{N}, \ldots, \sqrt{N}-1, \sqrt{N}$.

The labels for horizontal buses, starting at the top, are as follows:

$1, 2, 3, \ldots, \sqrt{N}, 1, 2, 3, \ldots, \sqrt{N}$, and so on.

Each processor at the intersection of a vertical bus and a horizontal bus that have the same label sets a fusing connection; all other processors set cross-over connections. Figure 3.2 shows this bus configuration.

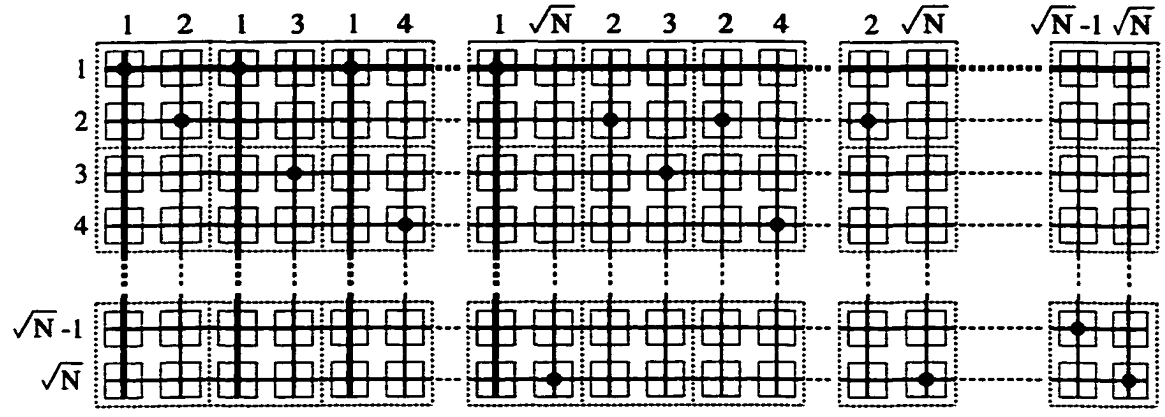


Figure 3.2: Contraction mapping for an FR-Mesh prevents a strong scaling simulation.

Assume that $P = \frac{N}{2}$. By using the contraction mapping, a single processor of the simulating machine, $\mathcal{R}$, performs the work of four processors of $\mathcal{Q}$ (dotted lines in Figure 3.2). Vertical buses with label 1 share processors with vertical buses with labels $2, 3, 4, \ldots, \sqrt{N}$. Each column of processors of $\mathcal{R}$ only can simulate one vertical bus of $\mathcal{Q}$ at a time. When $\mathcal{R}$ simulates the component with label 1 (bold bus in Figure 3.2), it cannot simulate components with labels $2, 3, 4, \ldots, \sqrt{N}$ at the same time. Similarly, vertical buses with label 2 share processors with vertical buses with labels $1, 3, 4, \ldots, \sqrt{N}$. When $\mathcal{R}$ simulates the component with label 2, it cannot simulate components with labels $1, 3, 4, \ldots, \sqrt{N}$ at the same time, and so on.

In general, given the above configuration, $\mathcal{R}$ can simulate at most one component of $\mathcal{Q}$ at a time. Consequently, the simulation of $\mathcal{Q}$ by $\mathcal{R}$ using the contraction mapping takes $\Omega\left(\sqrt{N}\right)$ time. ■

We will therefore use the *windows mapping* [4] (see Figure 2.4(b)).

3.3 General Description of the Simulation

Ben-Asher *et al.* [4] developed an optimal scaling simulation for the LR-Mesh that uses the windows mapping. This algorithm uses the fact that each bus has only two end-points (as each bus in the LR-Mesh is linear); consequently, it is possible to track buses across windows by considering at most two ports per bus. On the other hand, a bus in the FR-Mesh can have $\Theta(P)$ end-points in a window (because of fusing connections) and necessitates an entirely different approach.

The simulation of a step of $\mathcal{Q}$, an $N \times N$ FR-Mesh, on $\mathcal{R}$, a $P \times P$ FR-Mesh, progresses in two phases. During the first phase, *component determination*, the simulating machine $\mathcal{R}$ labels buses of $\mathcal{Q}$ with their component numbers. This allows $\mathcal{R}$ to treat ports with the same component number as if they were connected by a common bus, without the need to physically configure its buses exactly as in $\mathcal{Q}$. $\mathcal{R}$ ascertains the connection pattern of $\mathcal{Q}$ gradually by performing a window-by-window vertical sweep down each slice of $\mathcal{Q}$, and performing a slice-by-slice horizontal sweep across $\mathcal{Q}$. The biggest obstacle is that fusing connections outside a window can affect the component numbers of buses that are not connected within the window. Our main approach to overcoming this problem is assimilating into the current window the effect of connections in previously examined windows.

After assigning a component number to each bus, $\mathcal{R}$ proceeds to the second phase, *data delivery*. This phase conveys data written to one or more ports of a component to all ports of the component (this data depends on the concurrent write rule). During this phase, $\mathcal{R}$ sweeps over each window of $\mathcal{Q}$ detecting and recording processors' attempts to write to each component. During a second sweep, $\mathcal{R}$ delivers the data that appears on each component to all processors that simulate processors of $\mathcal{Q}$ reading that component. Section 3.4 describes component determination and Section 3.5 describes data delivery.

3.4 Component Determination

During the simulation, $\mathcal{R}$ treats $\mathcal{Q}$ as a series of $\frac{N}{P}$ slices. Component determination includes two horizontal sweeps across these slices to label the components. In the first sweep, $\mathcal{R}$ applies the following procedures to each slice of $\mathcal{Q}$ in turn. *Horizontal prefix assimilation* embeds in the current slice the effect of bus fusings of all preceding slices by selectively adding fusing connections to the current slice. After horizontal prefix assimilation, $\mathcal{R}$ can consider the current slice in isolation. To simulate the current slice, $\mathcal{R}$ now moves downwards, applying a sequence of *vertical prefix assimilation* and *component numbering* procedures to each window in the slice. Like horizontal prefix assimilation, vertical prefix assimilation embeds in the current window the effect of bus fusings of all previous windows in the slice. After applying vertical prefix assimilation to a window, component numbering uses the leader election procedure (Section 2.4) to update the component numbers for that window. The update includes the effect of fusing connections in all previously visited windows and slices. The sequence of calls to the vertical prefix assimilation and component numbering procedures concludes in the bottom window of the slice. Component numbers of vertical buses in the bottom window include the effects of the entire slice, so $\mathcal{R}$ broadcasts these to vertical buses in upper windows, and from here to any horizontal buses connected to them.

At this point in the simulation, for each bus in the slice, $\mathcal{R}$ holds the component number incorporating the effects of that slice and all slices to its left in $\mathcal{Q}$, but none of the slices to the right. After completing the first horizontal sweep across all slices in $\mathcal{Q}$, $\mathcal{R}$ holds the final component number of each horizontal bus in $\mathcal{Q}$. In the second horizontal sweep, $\mathcal{R}$ broadcasts these values to each vertical bus connected to a horizontal bus. The pseudo-code in Figure 3.3 describes the organization of the component determination phase.

Sections 3.4.1 to 3.4.5 explain horizontal prefix assimilation, vertical prefix assimilation, component numbering, the second vertical component sweep, and the second horizontal component sweep.

3.4.1 Horizontal Prefix Assimilation

Horizontal prefix assimilation is key to the scaling simulation. It embeds in $\mathcal{S}_v$ the effect of fusing connections in slices $\mathcal{S}_0, \ldots, \mathcal{S}_{v-1}$. $\mathcal{R}$ accomplishes this task by strategically adding fusing connections to the slice. Let C be a component such that at least one bus in C is fused within the slice. The essence of horizontal prefix assimilation is to select for each such component C a unique vertical bus b that can be used to connect horizontal buses in component C. That is, $\mathcal{R}$ fuses all horizontal buses

```
for $v \leftarrow 0$ to $\frac{N}{P} - 1$ do for slice $\mathcal{S}_v$ in $\mathcal{Q}$
    horizontal prefix assimilation
    for $u \leftarrow 0$ to $\frac{N}{P} - 1$ do for window $\mathcal{W}_{u,v}$ in $\mathcal{S}_v$
        vertical prefix assimilation
        component numbering
    end
    second vertical component sweep
end
second horizontal component sweep
```

Figure 3.3: Pseudo-code for component determination.

known to be in component C (because of fusing to the left of the current slice) to vertical bus b within the current slice.

Example: Figure 3.4(a) shows the effect of fusings in previous slices on the current slice. Figure 3.4(b) shows how horizontal prefix assimilation embeds these fusings in the current slice by adding some fusing connections (shown circled). $\mathcal{R}$ chooses vertical bus B to fuse the horizontal buses of component $\{2, 6\}$ within the slice. Similarly, $\mathcal{R}$ chooses vertical bus A for component $\{3, 5\}$. On the other hand, components $\{4\}$ and $\{1, 7\}$ have no fusings within the slice, so they need no extra fusings because nothing in the slice changes their component numbers. ■

The following is a description of variables and registers used during horizontal prefix assimilation.

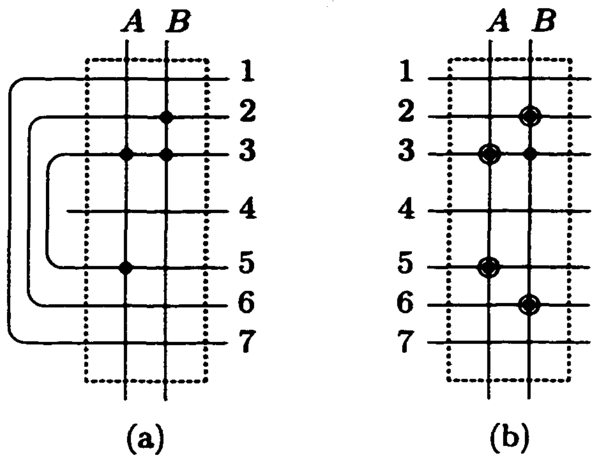


Figure 3.4: An illustration of horizontal prefix assimilation: a) Components and fusing connections in a slice before horizontal prefix assimilation; b) Slice after horizontal prefix assimilation, where added fusing connections are circled.

Let the ***horizontal fusing index*** of horizontal bus k be the index of the rightmost vertical bus (if any) in $\mathcal{W}_{u,v}$ that is fused to bus k. Denote this index as $hfi(k)$. Each processor connected to horizontal bus k holds this value (if any) in register hfi. Denote as $hcomp(k)$ ($vcomp(\ell)$, resp.) the component number of horizontal bus k (vertical bus ℓ, resp.). Each processor connected to horizontal bus k holds this value in register $hcomp$. Each processor in $\mathcal{R}$ possesses a set of $\frac{2N}{P}$ registers reg_m, where $0 \leq m < \frac{2N}{P}$. Register reg_m, in each processor of column ℓ, where $0 \leq \ell < P$, holds the horizontal fusing index for component $\frac{2N\ell}{P} + m$.

Assume that each processor in $\mathcal{R}$ holds the configuration of the processor it simulates in each of the $\left(\frac{N}{P}\right)^2$ windows of $\mathcal{Q}$. Also, assume that each processor in $\mathcal{R}$ holds the component numbers of its horizontal and vertical buses.

Horizontal prefix assimilation comprises four stages; each of these stages involves $\frac{N}{P}$ iterations. As we explain them, we will illustrate key ideas through the example of Figure 3.4.

Stage 1. In this stage, $\mathcal{R}$ determines by leader election (by priority resolution) the fusing index, $hfi(k)$, for each horizontal bus k in $\mathcal{S}_v$ and stores it in an appropriate processor and register. Stage 1 proceeds as follows.

for $u \leftarrow 0$ to $\frac{N}{P} - 1$ perform the following steps in window $\mathcal{W}_{u,v}$:

1. Each processor of $\mathcal{R}$ configures its ports as cross-over.

2. In each row k of $\mathcal{R}$, each processor, writing to its horizontal bus, uses leader election (by priority resolution) to find its $hfi(k)$ (if any).

3. In each row k of $\mathcal{R}$, the processor in column ℓ that satisfies the relation $\frac{2N\ell}{P} + m = comp(uP + k)$ for some m, where $0 \leq m < \frac{2N}{P}$, stores $hfi(k)$ (if any) in its register reg_m, overwriting any previously stored value.

$\mathcal{R}$ executes the leader election (by priority resolution) of Step 2 in $O(\log P)$ time, and Steps 1 and 3 in $O(1)$ time. $\mathcal{R}$ performs the steps above in $\frac{N}{P}$ windows, so it executes Stage 1 in $O(\frac{N}{P} \log P)$ time.

Example: In Figure 3.4(a), buses 2, 3, and 5 have fusing connections within the slice. The fusing indices for these buses are B, B, and A, respectively. Registers responsible for storing the fusing indices of component $\{2, 6\}$ hold only index B. On the other hand, registers for component $\{3, 5\}$ hold indices A and B.

Stage 2. Stage 1 places the set of possible fusing indices for a component in processors of the same column. Stage 2 uses leader election (by priority resolution) to select one fusing index for each of the $\frac{2N}{P}$ components per column of $\mathcal{R}$. Stage 2 proceeds as follows.

for $m \leftarrow 0$ to $\frac{2N}{P} - 1$ perform the following steps:

1. Each processor of $\mathcal{R}$ configures its ports as cross-over.

2. For each vertical bus of $\mathcal{R}$, use leader election (by priority resolution) to select a processor (if any) that has a non-null fusing index in register reg_m.

3. The selected processor broadcasts such a fusing index (if any) to all the processors in its column. Each processor in the column stores this index (if any) in register reg_m, overwriting any previously stored value.

$\mathcal{R}$ performs $\frac{2N}{P}$ leader elections (by priority resolution) in each column. Therefore, $\mathcal{R}$ executes Stage 2 in $O(\frac{N}{P}\log P)$ time.

Example: In Figure 3.4, this step selects index B (the only one available) for component $\{2,6\}$. Although it is not clear in Figure 3.4, assume that this step chooses index A (from the set $\{A,B\}$) for component $\{3,5\}$.

Stage 3 This stage completes horizontal prefix assimilation by adding a fusing connection between a horizontal bus and the vertical bus given by the fusing index selected in Stage 2 for its component. Stage 3 proceeds as follows.

for $u \leftarrow 0$ to $\frac{N}{P} - 1$ perform the following steps in window $\mathcal{W}_{u,v}$:

1. Each processor of $\mathcal{R}$ configures its ports as cross-over.

2. In each row k of $\mathcal{R}$, the processor in column ℓ that satisfies the relation $\frac{2N\ell}{P} + m = comp(uP+k)$ for some m, where $0 \leq m < \frac{2N}{P}$, broadcasts the contents (if any) of register reg_m to all processors in its row.

3. Each processor reads the horizontal bus. If the column index of some processor matches the value on the bus, then that processor replaces the original port configuration of the processor it simulates in $\mathcal{Q}$ by a fusing connection.

$\mathcal{R}$ performs these operations in each window in constant time, so $\mathcal{R}$ executes Stage 3 in $O(\frac{N}{P})$ time.

Example: In Figure 3.4, notice the circled fusing connections on vertical bus B (to embed component $\{2, 6\}$) and on vertical bus A (to embed component $\{3, 5\}$).

Altogether, $\mathcal{R}$ performs horizontal prefix assimilation in $O\big(\frac{N}{P}\log P\big)$ time.

Because horizontal prefix assimilation exploits the continuity of horizontal and vertical buses, this method does not extend to a scaling simulation of the (unrestricted) R-Mesh. For example, in Figure 3.4, if bus B was broken between buses 2 and 6, then placing a fusing connection at the intersection of buses 6 and B no longer accurately embeds the effects of buses 2 and 6 being in the same component. Since both buses are in different components of $\mathcal{R}$, one of them will be assigned a wrong component number, and a new problem that arises is that the segment of bus B below the break should be in a different component than bus 6. Furthermore, the number of possible components would be $O(N^2)$ instead of $2N$.

3.4.2 Vertical Prefix Assimilation

Vertical prefix assimilation embeds in $\mathcal{W}_{u,v}$ the effects of bus fusings in upper windows, $\mathcal{W}_{0,v}, \ldots, \mathcal{W}_{u-1,v}$. The procedure is a special case of horizontal prefix assimilation that uses $P \times P$ windows rather than $N \times P$ slices.

Vertical prefix assimilation includes an initial stage, described below, that provisionally replaces the component number of each vertical bus in window $\mathcal{W}_{u,v}$ by another that is $\log P$-bits long. This transformation maintains the size relation among the transformed component numbers. This stage proceeds as follows.

Small Component Numbers This stage assigns a component number of length $\log P$ bits to each of the P vertical buses in the window (the original component number is $\log N + 1$ bits long). This stage consists of the following steps.

1. Each processor of $\mathcal{R}$ configures its ports as cross-over.

2. The topmost processor in each column of $\mathcal{R}$ broadcasts its component number to all processors in its column.

3. Each main diagonal processor of $\mathcal{R}$ broadcasts the number it read to all processors in its row.

4. Each processor of $\mathcal{R}$ compares these two numbers. If both are equal, then the processor sets a provisional fusing connection in its ports.

5. The first row processors of $\mathcal{R}$, writing on their vertical buses, use leader election (by priority resolution) to find the largest column index for each component. Call the resulting index as the *small component number.*

$\mathcal{R}$ executes Steps 1-4 in $O(1)$ time and Step 5 in $O(\log P)$ time (since the length of a column index ℓ is $\log P$ bits). Therefore, $\mathcal{R}$ executes this stage in $O(\log P)$ time.

From this point, $\mathcal{R}$ applies a variation of horizontal prefix assimilation to window $\mathcal{W}_{u,v}$. We will describe vertical prefix assimilation by pointing out the differences with respect to horizontal prefix assimilation.

Notice that the prefix assimilation proceeds in a vertical fashion rather than horizontal. The number of possible small component numbers is at most P, so each processor in $\mathcal{R}$ has only one register (reg_1) (rather than the $\frac{2N}{P}$ registers reg_m) for storing the fusing index that corresponds to the small component number of its vertical bus.

Stage 1 In this stage, $\mathcal{R}$ determines by leader election (by priority resolution) the fusing index for each vertical bus ℓ in $\mathcal{W}_{u,v}$. Assume that vertical bus ℓ has small component number α, where $0 \leq \alpha < P$. Then, $\mathcal{R}$ stores the fusing index of column ℓ (if any) in register reg_1 of the processor located in row α and column ℓ.

Stage 2. The processors in each row α, writing on their horizontal bus, use leader election (by priority resolution) to select one fusing index for all vertical buses with small component α. Each processor in row α stores this index in its register reg_1.

Stage 3 This stage completes vertical prefix assimilation by adding the respective fusing connections as in horizontal prefix assimilation.

Component numbering also uses the small component number of each vertical bus. $\mathcal{R}$ performs vertical prefix assimilation in $O(\log P)$ time.

3.4.3 Component Numbering

The component numbering procedure assigns component numbers to buses (ports) in the current window. It incorporates the embedded effects of previous windows (gathered by the horizontal and vertical prefix assimilation procedures). The component numbering procedure works as follows.

Stage 1 Each processor of $\mathcal{R}$ configures its ports according to the configuration of the processor it simulates in $\mathcal{Q}$ (as altered by the prefix assimilation phases).

Stage 2 The processors at the top of each column, writing on the vertical buses, use leader election (by priority resolution) to find the vertical bus (column) with largest index in each component.

Stage 3 The processor at the top of a column with largest index in its component writes its original $\log N + 1$ bits long component number on its vertical bus. Each processor reads its vertical (horizontal, resp.) bus and stores the value in its register *vcomp* (*hcomp*, resp.).

The values in registers *vcomp* and *hcomp* are the new component numbers for vertical and horizontal buses, respectively, at this point in the simulation.

$\mathcal{R}$ performs the leader election (by priority resolution) of Stage 2 in $O(\log P)$ time and the remaining stages in $O(1)$ time. Therefore, $\mathcal{R}$ executes component numbering in $O(\log P)$ time.

Remark: If $\mathcal{R}$ used leader election (by priority resolution) to directly identify component numbers instead of Stages 2 and 3, then this would take $O(\log N)$ time, as the component numbers in $\mathcal{Q}$ could be $O(\log N)$ bits long.

Example: Figure 3.5 illustrates component numbering. For simplicity, it is designed to not require vertical prefix assimilation. Figure 3.5(a) shows the configuration of $\mathcal{S}_v$ after horizontal prefix assimilation. After applying component numbering to window $\mathcal{W}_{0,v}$ (Figure 3.5(b)), all buses in the same component receive the same component number, buses 0 and 2 are numbered B and A, respectively. The same effect occurs in windows $\mathcal{W}_{1,v}$ and $\mathcal{W}_{2,v}$ (Figures 3.5(c),(d)). Note that as the first vertical sweep advances, component numbers of vertical buses change (from A, B, C in $\mathcal{W}_{0,v}$, to B, B, C in $\mathcal{W}_{1,v}$, and finally to C, C, C in $\mathcal{W}_{2,v}$). These changes do not manifest themselves in upper windows until the second vertical component sweep.

3.4.4 Second Vertical Component Sweep

The second vertical component sweep propagates component numbers to each window in slice $\mathcal{S}_v$. After the first vertical sweep, all buses in window $\mathcal{W}_{\frac{N}{P}-1,v}$ include the

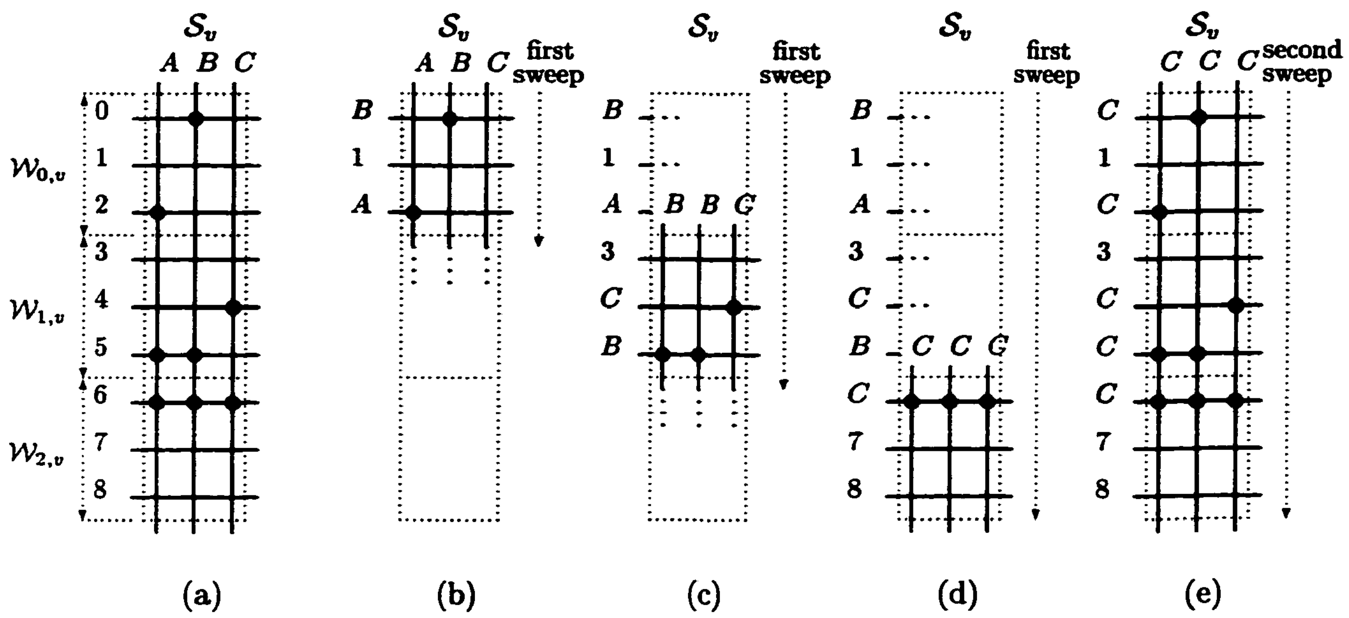


Figure 3.5: Example of vertical component sweep: a) Original slice connections and component numbers; b–d) First vertical component sweep; e) Second vertical component sweep.

effect of all fusing connections in the slice $\mathcal{S}_v$ and all previous slices (see window $\mathcal{W}_{2,v}$ in Figure 3.5(d)). During the second vertical component sweep, vertical buses convey these updated component numbers to horizontal buses in each window (see Figure 3.5(e)). For simplicity, this sweep follows the same direction as the first vertical sweep, starting in window $\mathcal{W}_{0,v}$ and moving down till window $\mathcal{W}_{\frac{N}{P}-1,v}$. Because vertical buses have the same component number in each window of the slice, the sweep over the windows can follow any order. $\mathcal{R}$ performs the following steps.

for $u \leftarrow 0$ to $\frac{N}{P} - 1$ perform the following steps in window $\mathcal{W}_{u,v}$.

1. Each processor of $\mathcal{R}$ configures its ports according to the configuration of the simulated window in $\mathcal{Q}$, plus the additional fusing connections obtained during the prefix assimilation sub-phases.

2. The topmost processor of each column writes its component number (the value of its register *vcomp*) on its vertical bus.

3. Each processor reads this value (if any) from its horizontal bus and stores it in its register *hcomp*, overwriting any previous value.

If some processor does not read any value in Step 3, then its horizontal bus is not fused to any vertical bus. In this case, each processor in that row retains the component number of its horizontal bus (in register *hcomp*) it had at the beginning of the simulation of the present slice.

$\mathcal{R}$ executes the second vertical component sweep in $O\left(\frac{N}{P}\right)$ time.

3.4.5 Second Horizontal Component Sweep

This phase propagates final component numbers to vertical buses in all slices in $\mathcal{Q}$. After the first horizontal sweep, all vertical and horizontal buses in the rightmost slice, $\mathcal{S}_{\frac{N}{P}-1}$, have the correct component number while vertical buses in other slices may not. Since each horizontal bus passes unbroken through all slices, $\mathcal{R}$ simply copies the final component number of a horizontal bus from the rightmost slice.

For each slice $\mathcal{S}_v$ of $\mathcal{Q}$ (the order of this sweep does not matter, but for simplicity start the sweep in $\mathcal{S}_0$ and finish it in $\mathcal{S}_{\frac{N}{P}-1}$) follow the next procedure:

for $u \leftarrow 0$ to $\frac{N}{P} - 1$ perform the following steps in window $\mathcal{W}_{u,v}$.

1. Each processor of $\mathcal{R}$ configures its ports according to the configuration of the simulated window $\mathcal{W}_{u,v}$, plus the additional fusing connections obtained during the prefix assimilation sub-phases.

2. The leftmost processor in each window writes its component number on its horizontal bus, then each processor in the window reads its vertical bus and stores this value (if any) in its register *vcomp*, overwriting any previous value.

The second horizontal component sweep runs in $O\left(\frac{N^2}{P^2}\right)$ time. The time to execute the component determination phase is $O\left(\frac{N^2}{P^2}\log P\right)$ time. Now, the conditions are set to apply the data delivery phase.

3.5 Data Delivery

So far, $\mathcal{R}$ holds the final component number of each bus in $\mathcal{Q}$. Using this information, ***data delivery*** ensures that each processor port receives its appropriate data. A component of $\mathcal{Q}$ must have the same data on all of its buses. Call the actions of $\mathcal{R}$ to ensure this property in its simulation of $\mathcal{Q}$ as ***data homogenization***. (Each processor in an FR-Mesh holds the bus data of its horizontal and vertical bus in registers *hdata* and *vdata*, respectively.) Data delivery employs two procedures, *window homogenization* and *slice homogenization*. In addition, data delivery uses the *second horizontal/vertical data sweeps* that parallel their counterparts in component determination.

The pseudo-code in Figure 3.6 describes the organization of data delivery. $\mathcal{R}$ applies window homogenization to all windows of the slice. After finishing with the bottom window, $\mathcal{R}$ applies the second vertical data sweep (similar to the second vertical component sweep) to broadcast any possible writes on lower windows to ports in upper windows. Because window homogenization acts locally at a window level, it cannot detect the relation between two buses with the same component number lying in different windows if they do not have an explicit connection between them in the slice. Therefore, after applying the second vertical data sweep, $\mathcal{R}$ performs slice homogenization to correct possible data inconsistencies that window homogenization

```
for $v \leftarrow 0$ to $\frac{N}{P} - 1$ do for slice $\mathcal{S}_v$ in $\mathcal{Q}$
    for $u \leftarrow 0$ to $\frac{N}{P} - 1$ do for window $\mathcal{W}_{u,v}$ in $\mathcal{S}_v$
        window homogenization
    end
    second vertical data sweep
    slice homogenization
end
second horizontal data sweep
```

Figure 3.6: Pseudo-code for data delivery.

cannot solve. After applying slice homogenization to all slices, $\mathcal{R}$ performs the second horizontal data sweep (similar to second horizontal component sweep) across all slices to ensure that writes in a slice reach slices to its left. We now describe the procedures in data delivery.

3.5.1 Window Homogenization

Window homogenization performs data homogenization within a window by integrating data entering a window through its borders with data written within the window, according to the concurrent write rule (COMMON, in this case). The main step of window homogenization is to physically connect within the window all buses in the same component. Once components are physically connected within $\mathcal{R}$, a write by the processors allows data homogenization through the window.

Window homogenization uses three different connection patterns to achieve data homogenization in every connected component of the window. The first connection pattern, the *HV* configuration, handles connected components that have both horizontal and vertical buses in the window (see buses *a* and *d* in Figure 3.7(b)). Similarly, the *VV* (*HH*, resp.) configuration handles connected components that have only vertical (horizontal, resp.) buses in the window. Figures 3.7(c) and 3.7(d) show examples of VV and HH configurations. Each component in $\mathcal{R}$ can easily determine its component type by checking whether or not it possesses fusing connections. The following routine sets configuration HV.

1. Each processor in $\mathcal{R}$ sets its port partition as cross-over.

2. The topmost processor in each column and the leftmost processor in each row broadcasts its component number to its column and row processors, respectively.

3. Each processor in the window compares the two numbers it receives. If they are equal, then it sets a fusing connection in its ports.

The following routine sets configuration VV (HH, resp.).

1. Each processor in $\mathcal{R}$ sets its port partition as cross-over.

2. The topmost (leftmost, resp.) processor in each column (row, resp.) broadcasts its component number to its column (row, resp.) processors. Then each main diagonal processor rebroadcasts the received number to its row (column, resp.) processors.

3. Each processor (including those in the main diagonal) in the window compares the two numbers it receives. If they are equal, then it sets a fusing connection in its ports.

Each processor in $\mathcal{R}$ that simulates a writer processor of $\mathcal{Q}$ holds data to be written on the bus. Additionally, each first row and first column processor holds data generated in adjacent windows. Window homogenization proceeds as follows.

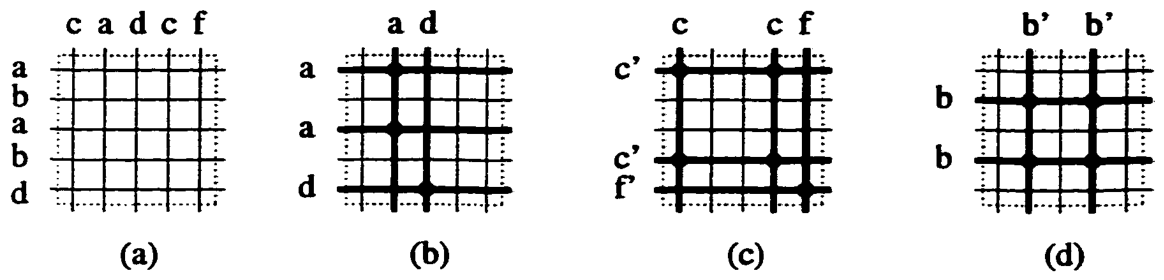


Figure 3.7: Configurations for window homogenization: a) Window after component determination; b–d) Configurations HV, VV, and HH for the same window.

1. $\mathcal{R}$ sets configuration HV (VV, HH, resp.). Only processors having ports attached to an HV (VV, HH, resp.) connected component take part in the following step.

2. Each processor that simulates a writing processor of $\mathcal{Q}$ writes its data on its respective bus or buses. Simultaneously, each border processor holding non-null bus data writes this value to the bus. Each processor reads the bus and stores the bus data.

The resulting data on each bus at the end of this step may be null or non-null. $\mathcal{R}$ performs window homogenization in constant time.

Example: Figure 3.7 illustrates window homogenization. Figure 3.7(a) shows the window after component determination, where the letters represent component numbers. Figure 3.7(b) shows the HV configuration. Notice how horizontal and vertical buses a form a connected component, as well as buses d. After applying a common write to each component, each processor port attached to the same component reads the same data. Figure 3.7(c) shows a VV configuration. Only buses with c and f take part in this operation. Observe that horizontal buses c' connect vertical buses c. The function of f' is similar but unnecessary in this case, since there is only one vertical bus f. Figure 3.7(c) shows the HH configuration with buses b' working as auxiliary buses to connect vertical buses b.

3.5.2 Second Vertical Data Sweep

The second vertical data sweep propagates bus data to each horizontal bus in slice $\mathcal{S}_v$. After applying window homogenization and component numbering to each window in the slice, all vertical buses include the effect of data writings in the slice. The bus data of horizontal buses in upper windows, however, may be altered by writings in lower windows. The second vertical data sweep solves these inconsistencies for those horizontal buses that have a fusing connection with some vertical bus in the slice; otherwise, we use slice homogenization. The second vertical data sweep works in the same way as the second vertical component sweep; the only difference is that each processor writes the contents of ***vdata*** (rather than *vcomp*) to its vertical bus, and each processor reading its horizontal bus stores the value in its register ***hdata*** (rather than *hcomp*)

$\mathcal{R}$ performs the second vertical data sweep in $O(\frac{N}{P})$ time. At this point, some data discrepancies may appear in horizontal buses of components without fusing connections in the slice. $\mathcal{R}$ applies slice homogenization to correct them.

3.5.3 Slice Homogenization

After applying window homogenization to all the windows in the slice, it is possible that some buses with the same component number have different bus data. Figure 3.8 illustrates this case. Since the two horizontal buses fuse the vertical bus in slice $\mathcal{S}_{v-1}$, after executing connected component determination, the three buses have the same component number (a in this case). The two horizontal buses do not fuse to any other bus in the remaining slices (from $\mathcal{S}_v$ to $\mathcal{S}_{\frac{N}{P}-1}$). Let some processor in slice $\mathcal{S}_v$ write α on the upper bus, while no processors write on the lower bus. After applying window homogenization to the entire slice, both buses will still have different bus data

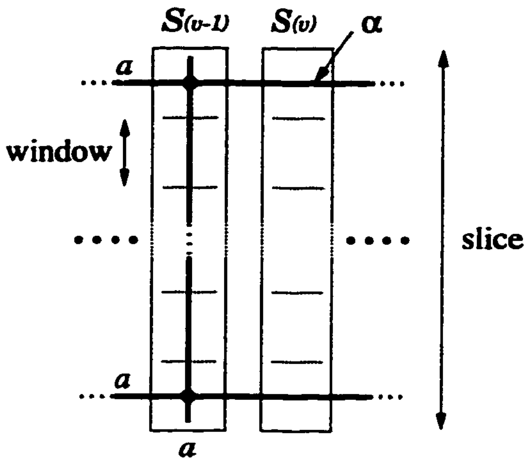


Figure 3.8: Example showing need for slice homogenization.

because they lie in different windows and S_v contains no explicit connection between them.

Slice homogenization acts over the entire slice, broadcasting data to buses placed in different windows (even when there is no direct connection between them in the slice) and correcting any remaining data discrepancy according to the concurrent write rule. Slice homogenization chooses one non-null datum (if any) for each component and broadcasts it to all the buses in its corresponding component (recall that the COMMON rule is assumed). This parallels the horizontal prefix assimilation algorithm that selects one fusing index (if any). Thus, the algorithm of Section 3.4.1 with minor modifications serves for slice homogenization. Slice homogenization performs the following steps.

Each row in $\mathcal{R}$ holds $2N$ registers ($\frac{2N}{P}$ registers per processor), as in horizontal prefix assimilation. Register reg_m, in each processor of column ℓ, where $0 \leq \ell < P$, holds the bus data for component $\frac{2N\ell}{P} + m$.

Stage 1. This stage groups the bus data of all horizontal buses according to their component numbers, and stores data for the same component in processors in the same column of $\mathcal{R}$. Stage 1 consists of the following steps.

for $u \leftarrow 0$ to $\frac{N}{P} - 1$ perform the following steps in window $\mathcal{W}_{u,v}$.

1. Each processor of $\mathcal{R}$ configures its port connections as cross-over.

2. Each processor of $\mathcal{R}$ writes the contents (if any) of register *hdata* to its horizontal bus.

3. In each row k of $\mathcal{R}$, the processor in column ℓ that satisfies the relation $\frac{2N\ell}{P} + m = comp(uP + k)$ for some m, where $0 \leq m < \frac{2N}{P}$, stores the data on the bus (if any) in its register reg_m, overwriting any previously stored value.

Remark: Since only common concurrent writes are allowed, we have only two possible classes of values, null and non-null.

$\mathcal{R}$ executes Stage 1 in $O(\frac{N}{P})$ time.

Stage 2. Since the bus data of buses having the same component number are in the same column, a common write among these values gives the final bus data for these buses.

for $m \leftarrow 0$ to $\frac{2N}{P} - 1$ perform the following steps in window $\mathcal{W}_{u,v}$.

1. Each processor of $\mathcal{R}$ configures its port connections as cross-over.

2. In each column ℓ, each processor writes the contents (if any) of register reg_m to its vertical bus.

3. In each column ℓ, each processor reads its vertical bus and stores that value in its register reg_m.

$\mathcal{R}$ executes Stage 2 in $O(\frac{N}{P})$ time.

Stage 3. **This stage returns the final bus data to the each processor in the slice.**

for $u \leftarrow 0$ to $\frac{N}{P} - 1$ perform the following steps in window $\mathcal{W}_{u,v}$.

1. Each processor of $\mathcal{R}$ configures its ports as cross-over.

2. In each row k of $\mathcal{R}$, the processor in column ℓ that satisfies the relation $\frac{2N\ell}{P} + m = comp(uP + k)$ for some m, where $0 \leq m < \frac{2N}{P}$, broadcasts the contents (if any) of register reg_m to all processors in its row.

3. Each processor reads the horizontal bus, and stores that value in register *data*.

$\mathcal{R}$ executes Stage 3 in $O(\frac{N}{P})$ time.

After applying window homogenization and slice homogenization to each window and slice in $\mathcal{Q}$, only the rightmost slice has its set of final bus data. $\mathcal{R}$ performs a reverse sweep, in which it broadcasts the data gathered during the first sweep, in order that each processor of $\mathcal{R}$ that simulates a reading processor of $\mathcal{Q}$ can read the final value from its vertical or horizontal bus or from both. This sweep starts in slice $\mathcal{S}_{\frac{N}{P}-1}$ and ends in slice $\mathcal{S}_0$.

Remark: We note that the actions of slice homogenization for all slices of $\mathcal{Q}$ can be deferred to a single slice homogenization on the last slice before the second horizontal data sweep. Our presentation of slice homogenization as a part of the data delivery phase allows each slice to be completely processed before the simulation moves on to the next slice.

3.5.4 Second Horizontal Data Sweep

The second horizontal data sweep propagates bus data to each vertical bus that is connected to a horizontal bus in $\mathcal{Q}$. After sweeping the $\frac{N}{P}$ slices of $\mathcal{Q}$ with slice homogenization, each port connected to a horizontal bus can read its final bus data while ports connected to vertical buses may not. Since each horizontal bus passes unbroken through all slices, $\mathcal{R}$ simply copies the final bus data of each horizontal bus from the rightmost slice. The second horizontal data sweep works similarly to the second horizontal component sweep; the only difference is that, in each slice, each processor writes the contents of *hdata* (rather than *hcomp*) to its horizontal bus, and each processor reading its vertical bus stores the value in its register *vdata* (rather than *vcomp*). So each processor in $\mathcal{Q}$ receives its correct final bus data.

$\mathcal{R}$ performs the second horizontal data sweep in $O\left(\frac{N^2}{P^2}\right)$ time. $\mathcal{R}$ also performs data delivery in $O\left(\frac{N^2}{P^2}\right)$ time.

This completes the simulation of an arbitrary step of an $N \times N$ COMMON CRCW FR-Mesh, $\mathcal{Q}$, on a $P \times P$ COMMON CRCW FR-Mesh, $\mathcal{R}$. On the whole, this simulation takes $O\left(\frac{N^2}{P^2}\log P\right)$ time; that is, it has a $\log P$ simulation overhead.

Remark: The FR-Mesh scaling simulation uses only $O\left(\frac{N^2}{P^2}\right)$ memory per processor, which is optimal. Also, the simulation applies to any $N_1 \times N_2$ FR-Mesh on a $P \times P$ FR-Mesh, with $P \leq N_1, N_2$.

3.6 Other Write Rules

The simulation explained in previous sections assumes the COMMON rule in $\mathcal{Q}$ and $\mathcal{R}$. In this section, we simulate $\mathcal{Q}$ on $\mathcal{R}$ and allow both of them to have any of the following write rules [12]: COMMON, COLLISION, COLLISION$^+$, ARBITRARY, or PRIORITY. We obtain the following results.

Theorem 3.3 *For any $P < N$, any step of an $N \times N$* COMMON, COLLISION, COLLISION$^+$, ARBITRARY, *or* PRIORITY *CRCW FR-Mesh can be simulated on a*

$P \times P$ COMMON, COLLISION, or COLLISION$^+$ CRCW FR-Mesh in $O\left(\frac{N^2}{P^2}\log P\right)$ time.

■

Theorem 3.4 *For any $P < N$, any step of an $N \times N$ COMMON, COLLISION, COLLISION$^+$, or ARBITRARY CRCW FR-Mesh can be simulated on a $P \times P$ ARBITRARY CRCW FR-Mesh in $O\left(\frac{N^2}{P^2}\right)$ time.* ■

Theorem 3.5 *For any $P < N$, any step of an $N \times N$ COMMON, COLLISION, COLLISION$^+$, ARBITRARY, or PRIORITY CRCW FR-Mesh can be simulated on a $P \times P$ PRIORITY CRCW FR-Mesh in $O\left(\frac{N^2}{P^2}\right)$ time.* ■

Though we view ARBITRARY and PRIORITY as too powerful for a bus-based model, the simulations indicate that the FR-Mesh can scale an ARBITRARY or PRIORITY algorithm with the same cost as scaling a COMMON algorithm (see Theorems 3.3, 3.4, and 3.5). Hence, algorithm development can be more flexible and efficient if the ARBITRARY or PRIORITY rules are permitted for algorithms that will be scaled and run on a model with a feasible concurrent write rule.

The cases where $\mathcal{R}$ uses the ARBITRARY or PRIORITY rules have a constant simulation overhead (see Theorem 3.4 and 3.5). This pinpoints leader election as the bottleneck in this scaling simulation. Thus any improvement in the technique for leader election immediately translates to a lower simulation overhead.

We prove Theorems 3.3 to 3.5 by performing the following simulations:

1. $N \times N$ COMMON, COLLISION, or COLLISION$^+$ CRCW FR-Mesh on an $N \times N$ ARBITRARY or PRIORITY CRCW FR-Mesh.

2. $N \times N$ PRIORITY CRCW FR-Mesh on a $P \times P$ PRIORITY CRCW FR-Mesh.

3. $N \times N$ ARBITRARY CRCW FR-Mesh on a $P \times P$ ARBITRARY CRCW FR-Mesh.

4. $P \times P$ PRIORITY CRCW FR-Mesh on a $P \times P$ COMMON, COLLISION, or COLLISION$^+$ CRCW FR-Mesh.

Simulations 1, 2, and 4 will establish Theorem 3.3, Simulations 1 and 3 will establish Theorem 3.4, and Simulations 1 and 2 will establish Theorem 3.5.

3.6.1 Simulation 1

An FR-Mesh using ARBITRARY or PRIORITY writes can simulate in constant time each step of an FR-Mesh of the same size using COMMON, COLLISION, or COLLISION$^+$. (The proof of this assertion mirrors the corresponding proof for PRAMs [19, 24].) Consequently, a simulation of an ARBITRARY or PRIORITY CRCW FR-Mesh with simulation overhead X implies simulations of FR-Meshes with any of the COMMON, COLLISION, or COLLISION$^+$ rules with the same simulation overhead X.

Lemma 3.6 *Any step of an $N \times N$* COMMON, COLLISION, *or* COLLISION$^+$ *CRCW FR-Mesh can be simulated on an $N \times N$* ARBITRARY *or* PRIORITY *CRCW FR-Mesh in $O(1)$ time.*

Proof: To prove Lemma 3.6 it is enough to show how to implement the COMMON, COLLISION, and COLLISION$^+$ rules on a CRCW bus that uses the ARBITRARY rule, since a CRCW bus that uses the PRIORITY rule simulates the ARBITRARY rule in constant time.

Common on Arbitrary: Under the COMMON rule, all the values written on the bus are equal. So, the bus using the ARBITRARY rule chooses anyone of the instances of the same value.

Collision on Arbitrary: In this simulation, accompany any writing with the index of the writing processor. Then, each writing processor compares its own index against the index it reads from the bus; if they differ, then the processor writes a collison symbol to the bus. If no processor writes a collision symbol, then the data on the bus is the final data.

Collision$^+$ on Arbitrary: Each writing processor writes its data to the bus. Then, each writing processor compares the data it wrote to the bus against the data it reads from the bus; if they differ, then the processor writes a collision symbol. If no processor writes a collision symbol, then the data on the bus is the final data. ∎

3.6.2 Simulation 2

Simulation 2 scales down the size of an FR-Mesh where both $\mathcal{Q}$ and $\mathcal{R}$ use the PRIORITY rule. Lemma 3.7 summarizes the result of Simulation 2.

Lemma 3.7 *Any step of an $N \times N$ PRIORITY CRCW FR-Mesh can be simulated on a $P \times P$ PRIORITY CRCW FR-Mesh in $O\left(\frac{N^2}{P^2}\right)$ time, that is, with a constant simulation overhead.*

Proof: We identify portions of the COMMON rule simulation that must change to accommodate the PRIORITY rule.

Priority on Priority: In the component determination phase (Section 3.4), $\mathcal{R}$ performs concurrent writes only during the following parts of the simulation:

1. Horizontal prefix assimilation (Section 3.4.1) in Stages 1 and 2,
2. Vertical prefix assimilation (Section 3.4.2) in Stages 1, 2, and to generate the small component numbers, and
3. Component numbering (Section 3.4.3) in Stage 2.

In these parts of the simulation, $\mathcal{R}$ uses concurrent writes to find a leader among a set of processors. In all the cases, the leader processor was always the one with highest index (we use the highest index for convenience, but the simulation can be easily modified to accept the lowest index). Using the COMMON rule, $\mathcal{R}$ performs leader election (by priority resolution) in $O(\log P)$ time. If $\mathcal{R}$ uses the PRIORITY rule, then it performs leader election in constant time. So, using the PRIORITY rule, $\mathcal{R}$ performs component determination in $O\left(\frac{N^2}{P^2}\right)$ time.

During the data delivery phase, $\mathcal{R}$ selects bus data for each bus. Under the PRIORITY rule, the data on the bus at the end of a writing cycle is the data written by the processor with highest priority connected to that bus. This simulation requires each bus datum written to have attached a *tag*. This tag is the index of the writing processor. Now, we identify the portions in the data delivery phase that change.

Window homogenization: By assigning the indices of processors in $\mathcal{R}$ in an analogous way to those of $\mathcal{Q}$, we ensure that the priority of writing processors in $\mathcal{R}$ reflects the priority of the corresponding writing processors in $\mathcal{Q}$. So using a concurrent write, $\mathcal{R}$ finds, for each component, the written data with highest priority within the window in constant time.

Next, $\mathcal{R}$ compares the data generated within the window against the data arriving from the top and left windows. Each border holds a unique bus datum per component. Processors of $\mathcal{R}$ perform three writings (data generated within the window, data from left border, and data from top border) in sequence for each component. Finally, each processor in $\mathcal{R}$ compares the tags to select the final bus data on its buses.

Second vertical data sweep: In this phase, $\mathcal{R}$ does not have to make any comparison, it only has to broadcast data. The broadcast data is the same for each component in the slice, so processors in $\mathcal{R}$ write their data and let the bus resolve the concurrent writes.

Slice homogenization: Remember that each horizontal bus in the slice has a component number and a bus datum with a tag. Also, in each row of processors of $\mathcal{R}$, there is a register where $\mathcal{R}$ stores a potential final bus datum for each component. We modify the slice homogenization algorithm of Section 3.5.3 in the following way. First, $\mathcal{R}$ stores the bus datum (and its tag) of each horizontal bus in its respective register (for each component that is present in the first window). Then, for each component, $\mathcal{R}$ compares the bus data tag in the first window against the one in the second window. $\mathcal{R}$ keeps in each register the bus datum with higher tag and repeats this procedure to cover all the windows in the slice.

Since this procedure takes $O\left(\frac{N}{P}\right)$ time per window, $\mathcal{R}$ executes slice homogenization in $O\left(\frac{N^2}{P^2}\right)$ time. For this reason, we defer slice homogenization to execute only once before the second horizontal data sweep, as mentioned in the remark at the end of Section 3.5. This phase proceeds as follows.

Stage 1. This stage is a combination of Stages 1 and 2 of slice homogenization (see Section 3.5.3). The first loop includes Steps 2 to 5; the second loop includes only Steps 4 and 5.

1. Each processor of $\mathcal{R}$ configures its port connections as cross-over.

for $u \leftarrow 0$ to $\frac{N}{P} - 1$ perform Steps 2 to 5 in window $\mathcal{W}_{u,v}$.

2. The leftmost processor in each row of $\mathcal{R}$ writes the contents (if any) of register *hdata* (and its tag) to its horizontal bus.

3. In each row k of $\mathcal{R}$, the processor in column ℓ that satisfies the relation $\frac{2N\ell}{P} + m =$

$comp(uP+k)$ for some m, where $0 \leq m < \frac{2N}{P}$, compares the tag of its bus data (if any) against the tag of the data stored in register reg_m. The processor keeps in register reg_m the data with higher priority tag.

for $m \leftarrow 0$ to $\frac{2N}{P} - 1$ perform Steps 4 and 5 in window $\mathcal{W}_{u,v}$.

4. In each column ℓ, each processor writes the contents (if any) of register reg_m to its vertical bus only if that register changed its contents in Step 3.

5. In each column ℓ, each processor reads its vertical bus and stores that value in its register reg_m.

Stage 2. This stage distributes the bus data to each bus in the slice. It is the same as Stage 3 of slice homogenization (Section 3.5.3).

Stage 1 determines the execution time of slice homogenization. $\mathcal{R}$ executes Stage 1 in $O\left(\frac{N}{P}\right)$ time per window, so $\mathcal{R}$ performs slice homogenization in $O\left(\frac{N^2}{P^2}\right)$ time.

Second horizontal data sweep: In this phase, $\mathcal{R}$ proceeds in the same way as in the second vertical data sweep. $\mathcal{R}$ broadcasts data and lets the bus resolve concurrent writes.

$\mathcal{R}$ performs data delivery in $O\left(\frac{N^2}{P^2}\right)$ time. On the whole, this simulation takes $O\left(\frac{N^2}{P^2}\right)$ time; that is, it has a constant simulation overhead.

3.6.3 Simulation 3

In the FR-Mesh simulation, we use leader election based on priority resolution. Priority resolution is a convenient and easy to implement method for choosing a leader. But in fact, any leader election method in the FR-Mesh simulation will work, not only priority resolution. So, by using a procedure similar to the one in Simulation 2, we obtain the following result.

Corollary 3.8 *Any step of an $N \times N$* ARBITRARY *CRCW FR-Mesh can be simulated on a $P \times P$* ARBITRARY *CRCW FR-Mesh in $O\left(\frac{N^2}{P^2}\right)$ time, that is, with a constant simulation overhead.*

3.6.4 Simulation 4

Now we prove that if $\mathcal{R}$ uses COMMON, COLLISION, or COLLISION$^+$, then $\mathcal{R}$ simulates a step of $\mathcal{Q}$ with a simulation overhead of log P. Lemma 3.9 summarizes the result of Simulation 4.

Lemma 3.9 *Any step of a $P \times P$* PRIORITY *CRCW FR-Mesh can be simulated on a $P \times P$* COMMON, COLLISION, *or* COLLISION$^+$ *CRCW FR-Mesh in $O(\log P)$ time.*

Proof: To prove Lemma 3.9, it is enough to show the simulation of a CRCW PRIORITY bus writing cycle on each of the COMMON, COLLISION, and COLLISION$^+$ simulating models. These procedures find the writing processor with highest index from the set of writing processors. Assume that each processor has a unique $O(\log P)$-bit key (the key may be its index). (To find the processor with lowest index, perform the procedures below, but each processor uses the 1's complement of its index rather than its real index.)

Priority on Common: In the first step, each processor writes a '1' on its bus if the most significant bit of its index is '1'.

If a processor with a '0' in this bit reads:

a) a '1', then it remains idle for the rest of the procedure, or

b) a null value, then it remains active in the following iteration.

In the second iteration, the remaining processors repeat the process but now using the second most significant bit, and so on. As the algorithm progresses, fewer processors remain active in the contest. Finally, after $2 \log P$ iterations, only the processor with highest index remains active in the contest. This winner processor writes its data to the bus.

Priority on Collision: In the first step, each processor writes a '1' on its bus if the most significant bit of its index is '1'.

If a processor with a '0' in this bit reads:

a) a '1', then there is only one writer, which wins the contest,

b) a collision symbol, then the processor remains idle for the rest of the procedure, or

c) a null value, then the processor remains active in the following iteration.

If a processor with a '1' in this bit reads:

a) a '1', then this processor wins the contest, and writes its data to the bus, or

b) a collision symbol, then that processor remains active in the following iteration.

In the second iteration, the remaining processors repeat the process but now using the second most significant bit, and so on. This process continues for at most $2 \log P$ iterations before finding winner. The winner processor writes its data to the bus.

Priority on Collision$^+$: The COMMON rule is a restricted case of the COLLISION$^+$ rule. Since the COMMON rule simulates the PRIORITY rule in $2 \log P$ steps, so does the COLLISION$^+$ rule.

By combining Lemmas 3.6, 3.7, and 3.9, we obtain Theorem 3.3. By combining Lemmas 3.6 and Corollary 3.8, we obtain Theorem 3.4. Similarly, by combining Lemma 3.6 and Lemma 3.7, we obtain Theorem 3.5.

3.7 Improved Scaling Simulation of the R-Mesh

This section describes an algorithm to simulate an arbitrary step of an $N \times N$ R-Mesh, $\mathcal{Q}$, on a $P \times P$ R-Mesh, $\mathcal{R}$, in $O\left(\frac{N^2}{P^2}\log P \log \frac{N}{P}\right)$ steps; this establishes a $\log P \log \frac{N}{P}$ simulation overhead. Our method applies our previous FR-Mesh scaling result (Theorem 3.3) to the R-Mesh scaling simulation of Ben-Asher *et al.* [4].

Ben-Asher *et al.* [4] proposed a scaling simulation for the unrestricted R-Mesh that simulates a step of an $N \times N$ R-Mesh on a $P \times P$ R-Mesh in $O\left(\frac{N^2}{P^2}\log N \log \frac{N}{P}\right)$ time. The $\log N$ factor in the simulation overhead is the contribution of a connected components algorithm used in the scaling simulation. This algorithm obtains the connected components of an N-node graph in $O(\log N)$ time using an $N \times N$ LR-Mesh, and in $O\left(\frac{N^2}{P^2}\log N\right)$ time after scaling it down.

We improve this simulation by replacing the connected components algorithm by one (based on the incidence matrix of the graph) that runs in $O(1)$ time on an $N \times N$ ARBITRARY (or PRIORITY) FR-Mesh, and in $O\left(\frac{N^2}{P^2}\log P\right)$ time after scaling it down to a $P \times P$ COMMON FR-Mesh. Thus, the improved scaling simulation has a simulation overhead of $\log P \log \frac{N}{P}$. The following theorem summarizes this result.

Theorem 3.10 *For any $P < N$, any step of an $N \times N$ COMMON, COLLISION, or COLLISION$^+$ CRCW R-Mesh can be simulated on a $P \times P$ COMMON, COLLISION, or COLLISION$^+$ CRCW R-Mesh in $O\left(\frac{N^2}{P^2}\log P \log \frac{N}{P}\right)$ time.* ■

Remark: If leader election can be done in $T = o(\log P)$ time, then the R-Mesh simulation overhead reduces to $T \log \frac{N}{P}$. In particular, if the simulating R-Mesh is allowed to use the ARBITRARY (or PRIORITY) rule, then the overhead is only $\log \frac{N}{P}$.

We now briefly describe the existing R-Mesh scalability simulation [4] that we will subsequently modify.

3.7.1 Existing R-Mesh Scalability Simulation

The method of Ben-Asher *et al.* [4] simulates a step of an $N \times N$ R-Mesh, $\mathcal{Q}$, on a $P \times P$ R-Mesh, $\mathcal{R}$. The most time-consuming part of this algorithm is the recursive procedure of Figure 3.9. This procedure identifies the components of $\mathcal{Q}$ and spends $\Theta\left(\frac{N^2}{P^2} \log N \log \frac{N}{P}\right)$ time. The remainder of the simulation runs in $O\left(\frac{N^2}{P^2} \log \frac{N}{P}\right)$ time.

```
Procedure $leaders(\mathcal{S}, X, P)$
/* Chooses a leader for each bus of an $X \times X$ R-Mesh, $\mathcal{S}$, using */
/*    a $P \times P$ R-Mesh. The output is the set, $\mathcal{L}$, of leaders.    */
    If $X > P$ then
        Divide $\mathcal{S}$ into four $\frac{X}{2} \times \frac{X}{2}$ sub-R-Meshes, $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{S}_3$, and $\mathcal{S}_4$
        for $j \longleftarrow 1$ to 4 do
            $\mathcal{L}_j \longleftarrow leaders(\mathcal{S}_j, \frac{X}{2}, P)$
    $components(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \mathcal{L}_4)$
end
```

Figure 3.9: Ben-Asher *et al.* procedure to calculate connected components.

To find a leader processor for each bus of an $X \times X$ R-Mesh $\mathcal{S}$, $\mathcal{R}$ calculates the connected components of an $8X$-node graph with at most $2X$ edges using the procedure $components(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \mathcal{L}_4)$. $\mathcal{R}$ obtains this graph from the connected components information generated on the four sub-R-Meshes of $\mathcal{S}$, each of size $\frac{X}{2} \times \frac{X}{2}$. That is, each sub-R-Mesh contributes $4\left(\frac{X}{2}\right)$ nodes (border ports), and there are at most $4\left(\frac{X}{2}\right)$ edges between the border ports of these sub-R-Meshes. When $X \leq P$, this problem reduces to finding a representative among $O(P)$ border ports for each bus, which $\mathcal{R}$ can accomplish in $O(\log P)$ steps.

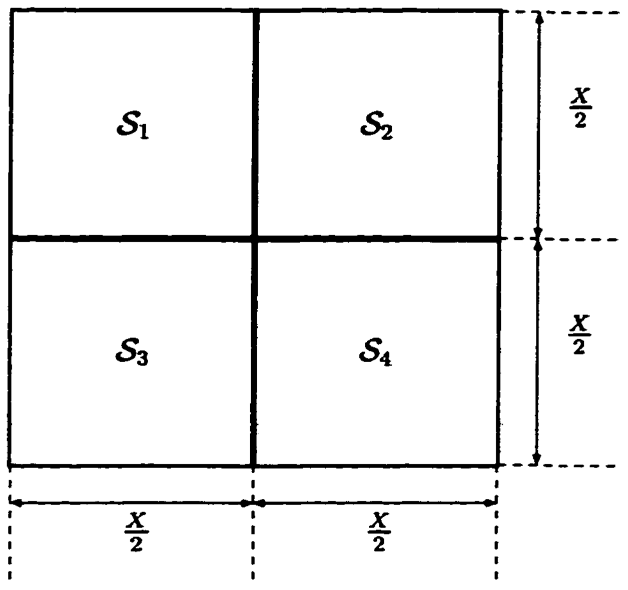


Figure 3.10: Decomposition of S into S_1, S_2, S_3, and S_4.

For $m \leq n$, let $t_c(n,m)$ denote the time required for an $m \times m$ R-Mesh to find the connected components of a $8n$-node graph with at most $2n$ edges. If $T(N,P)$ denotes the time to simulate an $N \times N$ R-Mesh on a $P \times P$ R-Mesh, then from the above discussion:

$$\begin{aligned} T(P,P) &= O(\log P) && \text{and} \\ T(N,P) &= 4T(\tfrac{N}{2},P) + t_c(N,P), && \text{for } P < N. \end{aligned}$$

This gives $T(N,P) = O\left(\left(\frac{N^2}{P^2}\right)\left(\log P + \sum_{i=1}^{\log \frac{N}{P}} 4^{-i} \cdot t_c\left(2^i P, P\right)\right)\right)$.

Ben-Asher *et al.* [4] showed that an $N \times N$ LR-Mesh can find the connected components of an $8N$-node, $2N$-edge graph in $O(\log N)$ time. Since an LR-Mesh is completely scalable, a $P \times P$ LR-Mesh can find *components*$(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \mathcal{L}_4)$ in $t_c(N,P) = O\left(\frac{N^2}{P^2} \log N\right)$ steps.

With this value of $t_c(N,P)$ in the equation for $T(N,P)$, we have

$$T(N,P) = O\left(\frac{N^2}{P^2}\log N \log\frac{N}{P}\right).$$

That is, the scalability factor of the simulation is $\log N \log \frac{N}{P}$.

3.7.2 The New Simulation

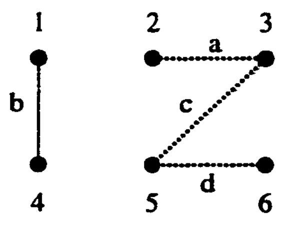
Our idea is to replace the LR-Mesh connected components algorithm used by *components*$(\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \mathcal{L}_4)$ with a faster, scalable FR-Mesh algorithm.

Lemma 3.11 *For any $P \leq N$, a $P \times P$ FR-Mesh (using the* COMMON *or* COLLISION *rule) can find the connected components of a graph with $O(N)$ nodes and $O(N)$ edges in $O\left(\frac{N^2}{P^2}\log P\right)$ time.*

Proof: Since the FR-Mesh scales with overhead $O(\log P)$ (Theorem 3.3), it suffices to prove that by embedding the incidence matrix of the c_1N-node, c_2N-edge graph (c_1 and c_2 are constants) into a $c_1N \times c_2N$ ARBITRARY (or PRIORITY) FR-Mesh, it can find the connected components of the graph in constant time. Let the vertical bus in column i of the FR-Mesh represent node i of the graph, and let the horizontal bus in row j of the FR-Mesh represent edge j of the graph. Now, embed the incidence matrix of the graph on the FR-Mesh as follows. Processor $p_{i,j}$ of the FR-Mesh sets a fusing connection if edge j is incident on node i; otherwise, it sets the cross-over connection.

Let (u, x_1), (x_1, x_2), ..., (x_k, v) be a set of edges that connects node u to node v in the graph. In the FR-Mesh, the vertical buses in columns u and x_1 are connected through the horizontal bus that represents edge (u, x_1). Similarly, the vertical buses in columns x_1 and x_2 are connected through the horizontal bus that represents edge

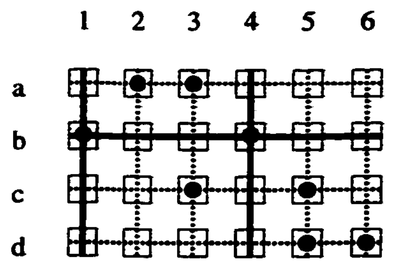
(x_1, x_2), and so on. Consequently, all nodes in the same component of the graph are also in the same component of the FR-Mesh. To choose a single label in each component, each processor on the first row now writes its column index (which is the identity of the node that column represents) on its vertical bus, then it reads the resulting component number from the same bus. ■



(a)

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline a & 0 & 1 & 1 & 0 & 0 & 0 \\ b & 1 & 0 & 0 & 1 & 0 & 0 \\ c & 0 & 0 & 1 & 0 & 1 & 0 \\ d & 0 & 0 & 0 & 0 & 1 & 1 \end{array}$$

(b)



(c)

Figure 3.11: Embedding the incidence matrix in an FR-Mesh: a) Two-component graph; b) Incidence matrix; c) The embedding.

Example: Figure 3.11(a) shows a two-component graph with six nodes and four edges, Figure 3.11(b) shows the incidence matrix of this graph, and Figure 3.11(c) shows the incidence matrix embedded in an FR-Mesh. Figure 3.11(c) shows the existence of a path between vertical buses 1 and 4 through horizontal bus *b* (this component is shown as solid bold lines). Also notice the path between vertical buses 2 and 6 through buses *a*, 3, *c*, 5, *d* (this component is indicated with dotted lines). When processors on the first row resolve concurrent writes, the top processors in columns 1 and 4 read '1', and the top processors in columns 2, 3, 5, and 6 read '2'.

Notice the importance of assuming the ARBITRARY (or PRIORITY) rule in the simulating machine to choose the component label. By using either of these write rules, the algorithm runs in $O(1)$ time. If we scale it down by using the FR-Mesh self-simulation (Theorem 3.3), then we get an overhead of $O(\log P)$, running on a $P \times P$ FR-Mesh with the COMMON or COLLISION rule. Otherwise, by initially assuming the COMMON or COLLISION rules, the connected components algorithm would run in $O(\log N)$ time, then scaling it down results in an overhead of $O(\log N \log P)$.

Lemma 3.11 establishes that $t_c(N, P) = O\left(\frac{N^2}{P^2} \log P\right)$. With this value for $t_c(N, P)$ in the equation for $T(N, P)$, we obtain

$$T(N,P) = O\left(\frac{N^2}{P^2} \log P \log \frac{N}{P}\right).$$

That is, the scalability factor is $\log P \log \frac{N}{P}$.

As noted before, the FR-Mesh scalability simulation of Theorem 3.3 has an $O(\log P)$ overhead due to leader election. Any improvement in leader election implies corresponding improvements in Lemma 3.11 and Theorem 3.10.

Chapter 4

Bus Linearization

The unrestricted R-Mesh can create bus structures of many different shapes (see Figure 4.2). On the one hand, flexibility in shaping buses facilitates algorithm design and can reduce running time, but on the other hand, certain bus shapes such as branches and cycles require more complicated hardware to implement these models. In this chapter, we present a procedure called *bus linearization* [15] that transforms a bus of any shape allowed by the R-Mesh into one with an equivalent linear structure. This procedure gives an algorithm designer the liberty of using buses of arbitrary shape, while automatically translating the algorithm to run on a more implementable platform.

We illustrate the use of bus linearization through two important applications. The first constructs a faster "scaling simulation" for the R-Mesh. The second application adapts algorithms designed for the R-Mesh to run on models that use optical buses [40, 44, 45].

The objective of bus linearization is to transform any "non-linear bus" (see Figure 4.1(a)) allowed by the R-Mesh into an "acyclic linear bus" (see Figure 4.1(b)). To this end, the LR-Mesh [4] can realize the resulting bus structure. Specifically, we prove that an $N \times N$ LR-Mesh (with only acyclic buses) can simulate an arbitrary step of an $N \times N$ R-Mesh in $O(\log N)$ time.

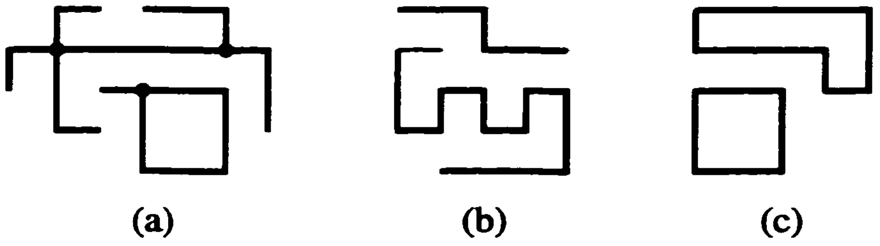


Figure 4.1: Type of buses: a) Non-linear; b) Acyclic linear; c) Cyclic linear.

This procedure iteratively grows a spanning tree for each bus in the R-Mesh. Simultaneously, it creates a "pseudo-Euler" tour for each tree; this is an equivalent acyclic linear bus representation of the spanning tree.

Matias and Schuster [32] also designed an algorithm that simulates an R-Mesh using an LR-Mesh. Although their algorithm targets a scaling simulation, one can use it for bus linearization. Their approach differs fundamentally from ours, however. They used a randomized simulation with (among others) the COLLISION rule for concurrently writing on buses; the overhead of their method (with both machines of size $N \times N$) is $\log N \log\log N$. Our method does not require concurrent writes, is deterministic, and has an smaller overhead of $\log N$ for the same problem. Section 4.3.1 gives more details contrasting their algorithm with ours.

Scaling Simulations In this chapter, we use bus linearization to construct a new deterministic scaling simulation for the unrestricted R-Mesh. This approach has a $\log N$ simulation overhead, which further improves the best previous deterministic simulation overhead (Section 3.7.2) of $\log P \log \frac{N}{P}$. Furthermore, the simulating

LR-Mesh uses only exclusive writes, whereas all prior scaling simulations required concurrent writes (see Table 1.1).

We also use bus linearization to allow the FR-Mesh scaling simulation of Section 3 to run on a weaker simulating model (see Table 1.1); the previous simulation requires a CRCW FR-Mesh, while the new simulation needs only a CREW LR-Mesh.

Optical Models Reconfigurable models with pipelined optical buses have attracted research interest because of their ability to handle communication-intensive algorithms efficiently. The Pipelined Reconfigurable Mesh (PR-Mesh) [45] is a variation of the R-Mesh that uses pipelined optical buses. Other optical models include the Array with Reconfigurable Optical Buses (AROB) [41] and the Array of Processors with Pipelined Buses using Switches (APPBS) [7, 17]. One can view the PR-Mesh as a version of the acyclic LR-Mesh with pipelined optical buses, or as a two-dimensional extension of the Linear Array with Reconfigurable Pipelined Bus System (LARPBS) [27, 40]. Due to the structure of its transmitting and receiving connections, the PR-Mesh (like other optical reconfigurable models) allows only acyclic linear connections.

We use bus linearization to translate the vast body of R-Mesh algorithms to run on the above optical models. Because of its similarity to the LR-Mesh, the PR-Mesh is the best suited to use bus linearization; in fact, bus linearization can run on a restricted version of the PR-Mesh. Table 1.1 shows our results for the PR-Mesh. Using results from Bourgeois and Trahan [7], we extend the PR-Mesh results to also include the AROB and APPBS.

The organization for this chapter is as follows. Section 4.1 gives some basic definitions. Section 4.2 details bus linearization and Section 4.3 uses bus linearization to construct new scaling simulations for the R-Mesh and FR-Mesh. Finally, Section 4.4 presents scaling simulations of the R-Mesh and FR-Mesh on reconfigurable pipelined optical models.

4.1 Definitions

This section introduces some basic concepts, such as the graph of an R-Mesh, the processor mapping used by the simulation, and two procedures to perform leader election on linear buses.

Figure 4.2(a) shows two buses in bold. Each bus induces a "component" in the R-Mesh. A component is the set of ports connected by the bus. It is possible for different buses to induce the same component. Two buses that induce the same component are said to be *equivalent*. The corresponding bold buses in Figure 4.2(a) and 4.2(b) are equivalent, though different in shape.

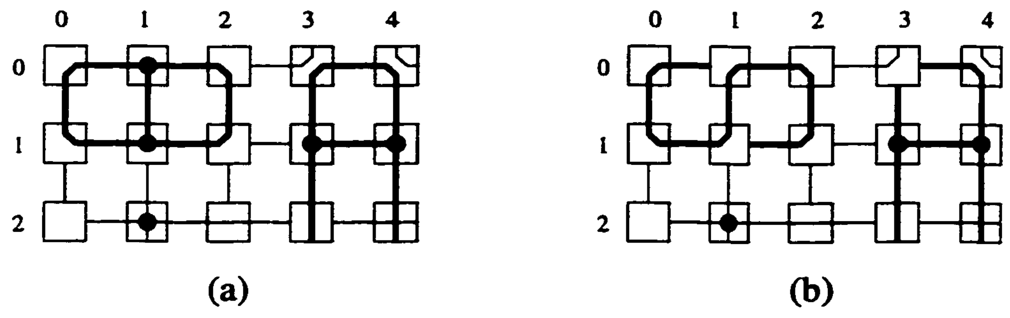


Figure 4.2: Port partitions of an R-Mesh.

A bus is *linear* iff it connects its ports only as allowed by the LR-Mesh; otherwise, the bus is *non-linear*. A linear bus could be cyclic or acyclic as shown in Figures 4.1(b) and 4.1c.

4.1.1 Graph of an R-Mesh

The graph, $\mathcal{G}$, of an R-Mesh configuration is a graphical representation of the connections between its ports. Each block in the port partition of a processor generates a node of $\mathcal{G}$. Thus, one can view each node of $\mathcal{G}$ as a set of ports internally connected within a processor of the R-Mesh. Let v_1 and v_2 be nodes of $\mathcal{G}$. An edge exists in $\mathcal{G}$ between v_1 and v_2 iff an edge exists in the R-Mesh between ports p_1 and p_2, where ports p_1 and p_2 are elements of the partition blocks that generate nodes v_1 and v_2, respectively. Figure 4.3(a) shows an R-Mesh configuration with its graph $\mathcal{G}$ in Figure 4.3(b); the dotted squares represent the corresponding processors for each partition of ports, but they are not part of the graph. Clearly each node has degree 0, 1, 2, 3, or 4. Let the term ***terminal node*** refer to a degree-1 node; ***linear node***, to a degree-2 node; and ***non-linear node***, to a degree-3 or degree-4 node.

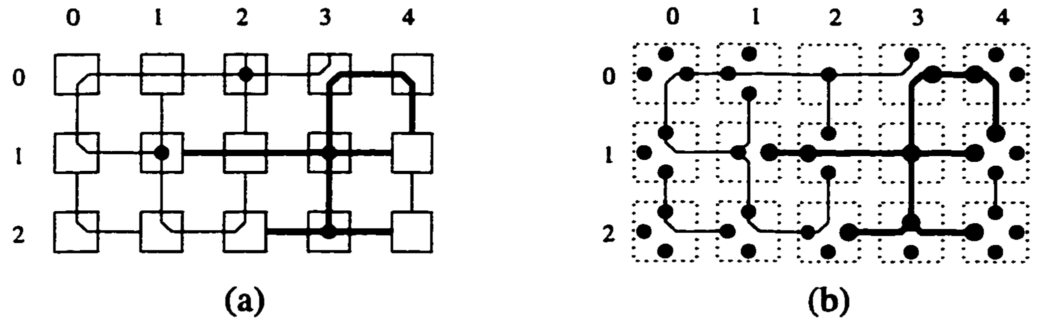


Figure 4.3: Graph of the R-Mesh: a) Configuration of an R-Mesh; b) Graph $\mathcal{G}$ of the R-Mesh.

4.1.2 Mapping R-Mesh Processors to LR-Mesh Processors

During the simulation of the R-Mesh by the LR-Mesh (Section 4.2), a ***group*** of four processors of the LR-Mesh simulates a single processor of the R-Mesh. For each port partition of the R-Mesh, there is an ***equivalent group configuration*** assumed by a group of processors of the LR-Mesh. Figure 4.4 shows representative configurations of an R-Mesh processor and its equivalent group configuration.

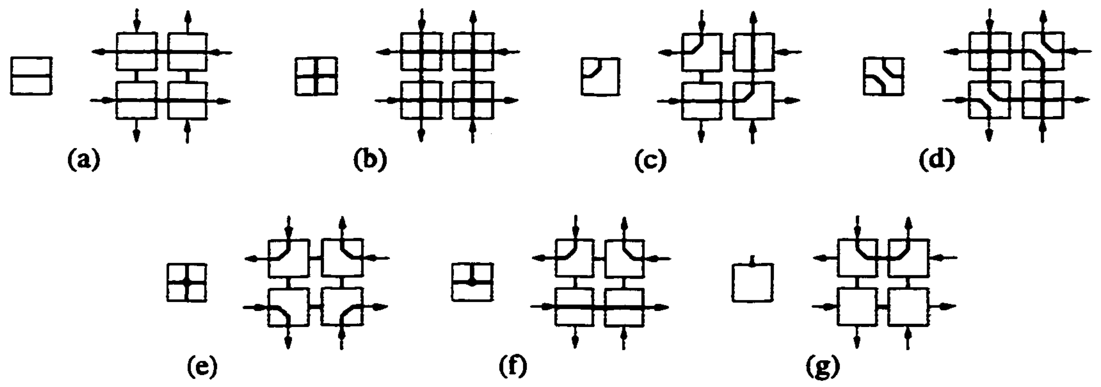


Figure 4.4: Equivalent group configurations for R-Mesh processors: a-d) Linear processors; e-f) Non-linear processors; g) Terminal processor.

Each group of four processors has eight ports through which it connects to neighboring groups. Assign a direction (incoming or outgoing) to each of these ports as shown in Figure 4.4. This assignment is only for ease of explanation and does not require the more powerful ***directional model*** [5] that can restrict information flow to only one direction on a bus because ports in the LR-Mesh processors will read only from incoming ports and segment and write only to outgoing ports. These oppositely "directed" buses, combined with the equivalent group configurations of the R-Mesh processors, create a ***double bus structure*** that is fundamental for some of the procedures presented in this chapter.

As is clear from Figure 4.4, each pair of incoming/outgoing ports of a group of processors in the LR-Mesh corresponds to a port of an R-Mesh processor. Since each node of graph $\mathcal{G}$ of the R-Mesh configuration corresponds to a set of ports, we will say that a node reads or writes to refer to reads and writes at the corresponding ports of the group.

4.1.3 Leader Election

Let S be a set of processors in the R-Mesh connected by a linear bus. Let $C \subseteq S$ be a set of *candidates* for leader. *Leader election* is the problem of selecting any one element from C. Leader election is a fundamental part of the R-Mesh simulation in Section 4.2. Furthermore, the ability of the LR-Mesh to perform leader election among processors connected by acyclic linear buses in constant time allows a fast R-Mesh simulation without the use of concurrent writes. We present two solutions to leader election corresponding to cyclic and acyclic linear buses. In these solutions we use a group of four processors to simulate a single processor of S (each square in Figure 4.5 is a group of four processors). We also use the equivalent group configurations of linear nodes (Figure 4.4(a) to 4.4(d)) to embed the connections among ports and to create the double bus structure described in Section 4.1.2. Assume that each group of processors has identified whether the linear bus to which it connects is cyclic or acyclic. (the first step of the simulation of Section 4.2 explains how to make this determination).

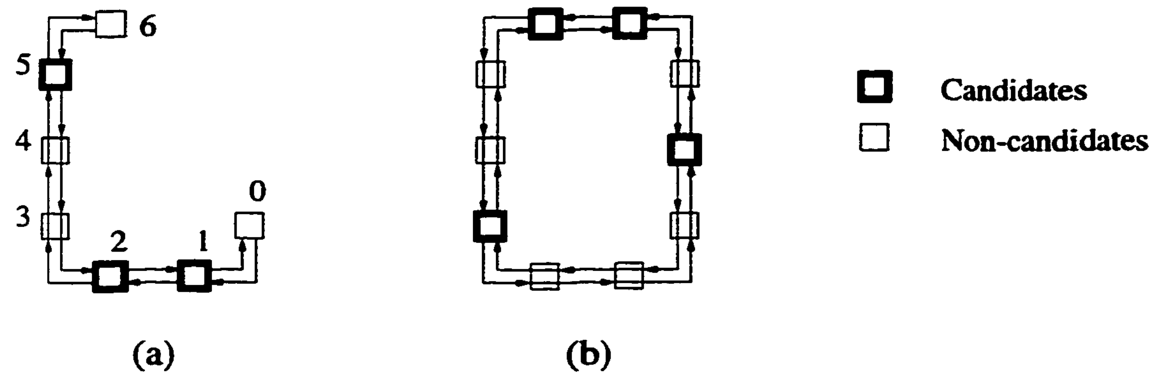


Figure 4.5: Leader election examples: a) Acyclic bus; b) Cyclic bus.

Leader Election on Acyclic Linear Buses: This leader election method consists of two steps: (1) select one of the terminal groups of the acyclic linear bus as a reference group; (2) select the candidate closest to the reference group as the leader.

To select the reference group, the two terminal groups exchange their indices (using the double bus structure described in Section 4.1.2) and decide upon the lower indexed group as the reference. To elect the leader, we use a procedure called neighbor localization that works as follows. If the reference group is a candidate, then it is elected as the leader; otherwise, each candidate segments its portion of linear bus, while non-candidate groups leave the bus unsegmented. A write by the reference group reaches the candidate closest to the reference. This candidate is elected leader. Notice that this procedure runs without concurrent writes and takes constant time.

Example: In Figure 4.5(a), the two terminal groups exchange their indices, 0 and 6. They select group 0 as the reference group. Then, group 0 writes on the bus, and group 1 (the closest candidate to the reference) reads the bus and declares itself the leader.

Lemma 4.1 *A CREW LR-Mesh can perform leader election on an acyclic linear bus (using a double bus structure) in constant time.*

Leader Election on Cyclic Buses: Finding a leader on a cyclic bus is more involved than on an acyclic bus and compels a different approach. The procedure we use selects the least indexed element of the set C as the leader. Non-candidate groups simply provide buses between adjacent candidate groups, while candidate groups split their bus (see Figure 4.5(a)). The idea is to shrink C until it contains only the leader. Reduce set C as follows.

Each candidate exchanges its index with its neighboring candidates (using the double bus structure described in Section 4.1.2). Since two outgoing ports are never adjacent, the exchange of indices is without conflict. Any candidate with an index larger than either of its neighbors cannot be the leader, so it excludes itself from $\mathcal{C}$ and connects its internal ports as a non-candidate group. Since the procedure removes at least one of each pair of neighboring candidates from $\mathcal{C}$, the number of candidates at least halves in each iteration. The procedure repeats on the reduced candidate set and stops when only one candidate remains. To determine this condition, test if a candidate is its own neighbor. Notice that this procedure does not require concurrent writes and takes $O(\log|\mathcal{C}|)$ time. Therefore, we have the following result.

Lemma 4.2 *A CREW LR-Mesh can perform leader election on a cyclic bus (using a double bus structure) with X candidates in $O(\log X)$ time.*

4.2 Bus Linearization

Bus linearization is a procedure for transforming non-linear buses of an R-Mesh into acyclic linear buses. This section describes a method for bus linearization that simulates an R-Mesh (that uses non-linear buses) on an LR-Mesh (that uses only linear buses). While linear buses are important because of their simple structure, a further restriction to acyclic buses has the advantage that they admit constant time leader election, an important procedure for eliminating concurrent writes. Pipelining on optical buses also assumes acyclic buses.

The idea of bus linearization is to generate a spanning tree (equivalent acyclic bus) of the graph $\mathcal{G}$ of the R-Mesh and find a pseudo-Euler tour (equivalent acyclic linear bus) of the spanning tree. The pseudo-Euler tour enables the LR-Mesh to handle the spanning tree, hence, R-Mesh bus structure, as a linear bus. We construct the spanning tree and pseudo-Euler tour iteratively, growing the spanning tree from an initial linear subgraph. An iteration starts with a collection of partial spanning trees and their pseudo-Euler tours. The iteration merges partial spanning trees and their pseudo-Euler tours for the next iteration. This method captures connectedness of a graph in a manner similar to the connected components algorithm of Shiloach and Vishkin [43].

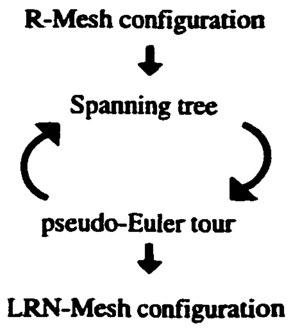


Figure 4.6: Linearization procedure.

This section establishes the following result.

Theorem 4.3 Bus linearization *Any step of an* $N \times N$ Common, Collision, Collision$^+$, Arbitrary, *or* Priority *CRCW R-Mesh can be simulated on an* $N \times N$ *CREW LR-Mesh in* $O(\log N)$ *time.*

The first step in the proof of Theorem 4.3 is an $O(\log N)$ time simulation of an $N \times N$ Common CRCW R-Mesh, $\mathcal{Q}$, on a $2N \times 2N$ Common CRCW LR-Mesh, $\mathcal{Z}$. We will later reduce the size of $\mathcal{Z}$ and describe the modifications required for introducing other write rules in $\mathcal{Q}$, and then eliminate the need for concurrent writes in $\mathcal{Z}$.

4.2.1 Simulation of R-Mesh by LRN-Mesh

We now describe the simulation with the running example of Figure 4.7. For clarity the example shows only one component. Each group of four processors of $\mathcal{Z}$ holds the processor index, port configuration, data to be written on buses, and the computation to be performed by the processor of $\mathcal{Q}$ it simulates.

Step 1 - Identifying bus types: This step classifies each bus of the R-Mesh as non-linear, cyclic linear, or acyclic linear. First, embed the graph $\mathcal{G}$ of the R-Mesh $\mathcal{Q}$ in $\mathcal{Z}$ using the configurations of Figure 4.4(a)-(d) for linear nodes and without any internal connection in the groups for terminal and non-linear nodes (see Figure 4.7(b)).

Each non-linear node now broadcasts a signal to all the nodes connected to it. If a node receives the signal, then its ports are on a non-linear bus; otherwise, its ports are on a linear bus. The next phase determines whether a linear bus is cyclic or acyclic, so only nodes on linear buses participate. Each terminal node writes a signal on its port. If a node receives this signal, then its ports are on an acyclic linear bus; otherwise, they are on a cyclic linear bus.

Step 2 - Eliminating cyclic linear buses: By Lemma 4.2, elect a leader in each cycle and cut the bus at the leader. At this point, all linear buses are acyclic. This step generates graph $\mathcal{G}'$, which is equivalent to graph $\mathcal{G}$ and is also embedded in $\mathcal{Z}$.

Next, $\mathcal{Z}$ performs the writing cycle on linear acyclic buses of graph $\mathcal{G}'$. In the writing cycle, writing nodes write their data to buses and reading nodes read and store the data. The remaining steps deal with the non-linear buses of graph $\mathcal{G}'$.

Step 3 - Constructing the raked graph: This step partially contracts the graph $\mathcal{G}'$ in a way analogous to raking (removing) the leaves of a tree. Specifically,

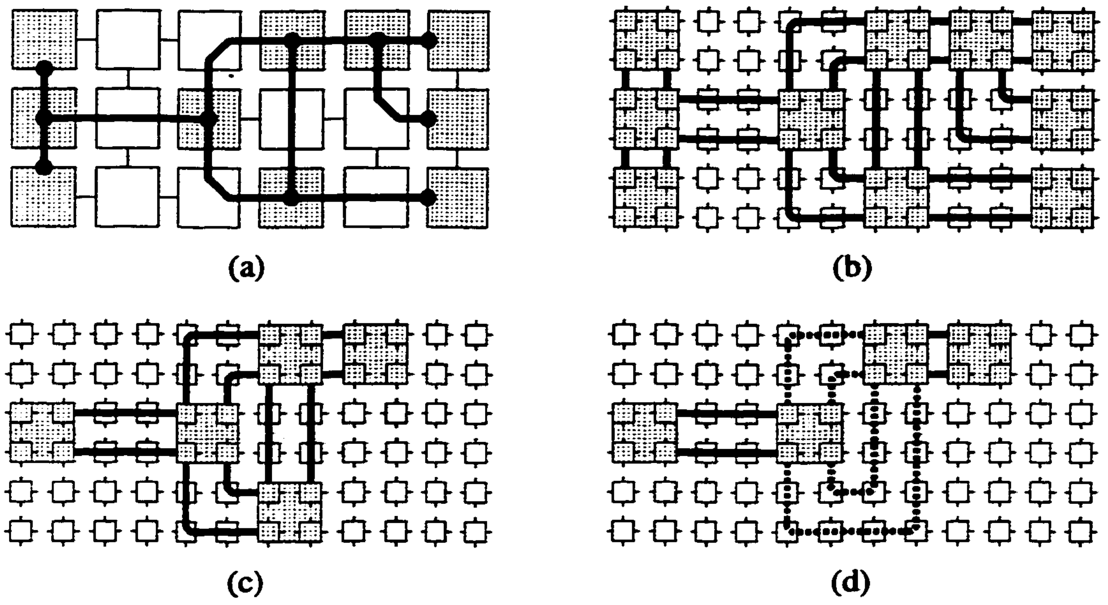


Figure 4.7: LR-Mesh simulating an R-Mesh (first part): a) R-Mesh non-linear graph; b) LR-Mesh replication of edges of the non-linear graph; c) Raked graph; d) Distilled graph.

$\mathcal{Z}$ constructs a *raked graph* by removing each chain of linear nodes that connects a terminal node to a non-linear node (Figure 4.7(c)). The reason for this step is to eliminate port configurations like those of processors (0,1) and (1,2) in Figure 4.3, for which a 2 × 2 group lacks sufficient width to directly embed the connections for constructing a pseudo-Euler tour in subsequent steps.

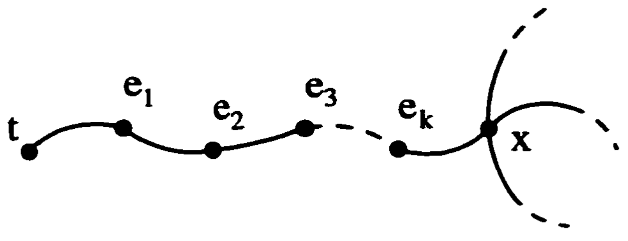


Figure 4.8: Raking chains of linear nodes in Step 3.

Consider a terminal node t, connected via linear nodes $e_1, e_2, \ldots, e_k$ to non-linear node x (see Figure 4.8). First, $\mathcal{Z}$ configures groups as in Step 1. Then, node t transmits a predefined signal (for example a '1') to notify $e_1, e_2, \ldots, e_k$ that they will be raked up. This signal also informs node x that it needs to store the resulting bus data after performing the write cycle on the bus that connects the chain of linear nodes. During the write cycle, any writing port associated with nodes $t, e_1, e_2, \ldots, e_k, x$ writes to the bus, the bus resolves the concurrent writes, and node x (having been alerted by the previous signal) picks up the resulting data from the bus. (After the pseudo-Euler tour construction, Step 10 will incorporate this data while handling other writes.)

Step 4 - Constructing the distilled graph: In this step, $\mathcal{Z}$ transforms the raked graph into the *distilled graph* by flagging maximal chains of linear nodes as edges. This transformation allows $\mathcal{Z}$ to speed up the construction of the spanning tree, since long chains of linear nodes (now edges) can be flagged to be part of the spanning tree in the following steps. Only the non-linear and the newly generated terminal nodes of the raked graph can be nodes in the distilled graph. (Figure 4.7(d) shows a distilled graph; notice how the linear node changed into an edge.)

Step 5 - Initiating spanning tree construction: The goal of Steps 5 to 8 is to generate a spanning tree for each connected component of the distilled graph. In Step 5, $\mathcal{Z}$ starts the spanning tree construction by merging edges of the distilled graph. This step generates a set of partial trees. Subsequent steps will merge these trees iteratively to complete the spanning tree construction. For simplicity, we will explain the procedure for a single component and refer to the partial trees in the component as the *forest*.

Only groups of processors of $\mathcal{Z}$ that represent nodes of the distilled graph are active during Step 5, we refer to them as *active nodes*. Step 5 proceeds as follows. The active nodes exchange their indices with each other if they are adjacent, that is, if they are connected by an edge in the distilled graph (a chain of linear nodes in the raked graph). Each active node chooses its neighboring active node with smallest index as its *parent*, only if this index is smaller than its own index. Then $\mathcal{Z}$ flags or selects the edge between each parent and "child" as part of the new spanning tree.

At this point of the simulation, the active nodes do not have any internal connections in their ports. So, the edges selected in this step are not connected to each other, even though two selected edges have a common active node. Figure 4.7(d) shows the first two edges (in bold) of the spanning tree. The dotted edges remain unselected in this step. Notice that each edge consists of one outgoing bus and one incoming bus. An active node incident with a selected edge marks itself as a *root* if its index is smaller than the indices of all its neighboring active nodes connected to it through selected edges. Each root identifies itself by comparing its index against the indices of its neighbors.

Remark: The terms "parent" and "child" do not indicate a directed tree. They just give a convenient form to describe the direction in which $\mathcal{Z}$ performs grafting.

Step 6 - Constructing the initial pseudo-Euler tour: This step constructs a pseudo-Euler tour of each tree of the forest obtained in Step 5.

Claim: Embedding in $\mathcal{Z}$ the equivalent group configuration of each active node connected to selected edges (of the distilled graph) generates a set of Euler tours.

Proof: Each selected edge consists of two independent parallel buses (one outgoing bus and one incoming bus). Notice that the resulting graph is connected, since each partial tree in the distilled graph is connected and the configurations Figure 4.4(e)-(g) always connect together all the selected edges that represent a partial tree. Notice also

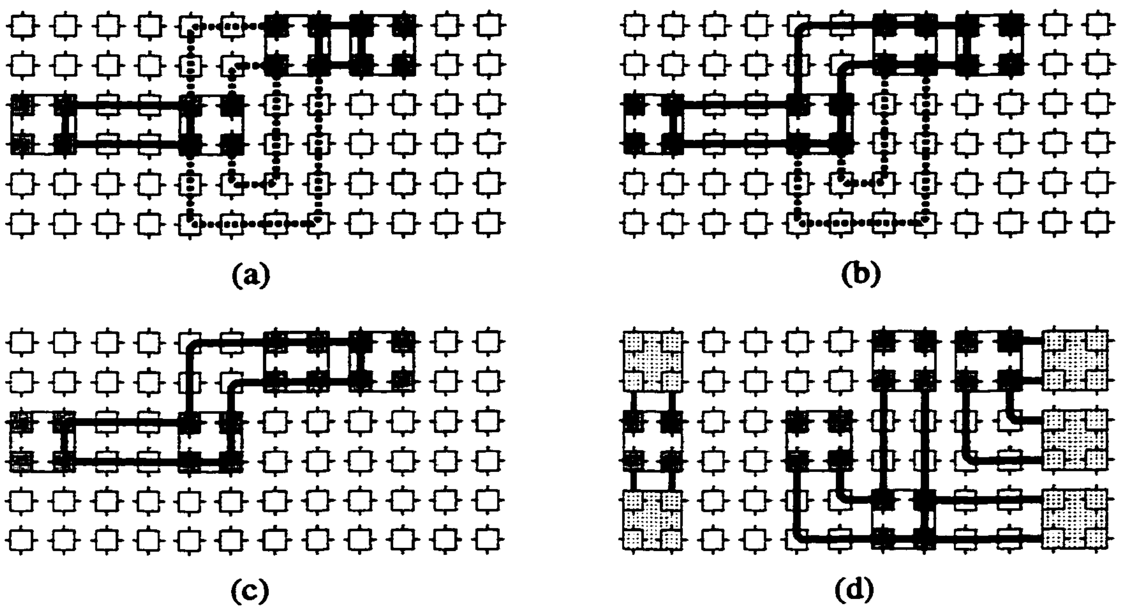


Figure 4.9: LR-Mesh simulating an R-Mesh (second part): a) Growing a spanning tree and constructing pseudo-Euler tours; b) Pseudo-Euler tour of the spanning tree of the distilled graph; c) Pseudo-Euler tour of the spanning tree after incorporating data of unselected edges; d) Broadcasting final values to raked segments.

that the degree of each port (number of buses connected to a port) is two, therefore the bus that connect the ports must be a cycle. ■

Since each partial tree of the forest has only one root, the corresponding root group can remove one of the internal port connections to avoid forming a cycle in the Euler tour (see Figure 4.9(a)), thus forming a "pseudo-Euler tour". The root node broadcasts its index to all nodes of its tree; this index is the *label* of the tree. Figure 4.9(a) shows the pseudo-Euler tours generated in Step 6.

Steps 7 and 8 below form the body of the iterative process. $\mathcal{Z}$ repeats them $2(\log N + 1)$ times to constructs a spanning tree of the distilled graph. The input to Step 7 is always a set of trees (pseudo-Euler tours). The output of Step 8 is a smaller set of larger trees.

Step 7 - Grafting trees: This step merges partial trees (pseudo-Euler tours) in the forest by a *graft operation*. The effect of this step is to connect subtrees of the spanning tree by using unselected edges. Step 7 proceeds as follows.

Each active node in each partial tree of the forest looks for a potential new parent into which to graft. The selecting active node and the new parent must (1) be in different trees, (2) be connected by an unselected edge, and (3) have different tree labels, and the label of the potential parent must be smaller than that of the selecting active node. These conditions ensure an acyclic structure using only connections of the distilled graph. Selecting a parent just involves information exchange on buses of $\mathcal{Z}$. By Lemma 4.1, the root of each tree (pseudo-Euler tour) determines whether any active node in its tree has identified a potential parent, and, if so, chooses one such active node (there may be several). This node will perform the grafting operation and the root will close its cuts. Also, by Lemma 4.1, the root of a tree determines if some other tree has grafted into it; the root needs this information to establish whether or not its tree is a *rejected tree*.

The root of the tree resulting from the grafts above is the root of a tree with smallest index among the trees that comprise the new tree. Figure 4.9(b) shows the resulting pseudo-Euler tour after merging the two small pseudo-Euler tours. The algorithm incorporates one of the two unselected edges into the spanning tree.

Example: Figure 4.10 shows a graft operation between two trees (pseudo-Euler tours). Let s be a selecting active node that has determined p to be its new parent. Let r_s and r_p be the roots of the trees of s and p, respectively. The idea is to cut the pseudo-Euler tours at s and p and join them at the cut. Active nodes s and p inform their respective parents about the graft operation, then r_s closes the break, and r_p broadcasts its label to all the new nodes in its tree.

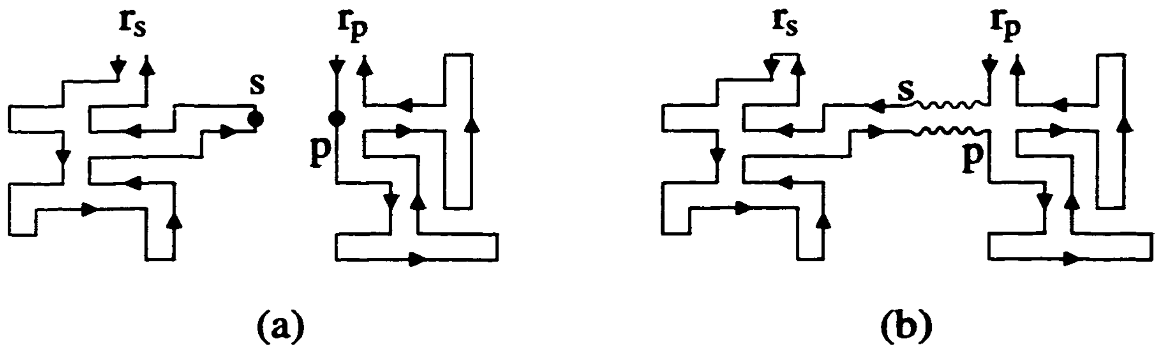


Figure 4.10: Grafting operation: a) Trees of selected edges represented by their pseudo-Euler tours; b) Grafting the left tree onto the right tree.

Step 8 - Grafting rejected trees: Consider the situation in Figure 4.11 where several low labeled trees have edges only to one high labeled tree. If $\mathcal{Z}$ uses only Step 7 in the iterations, then it is possible for the high labeled tree to graft sequentially onto the low labeled trees, starting with the tree with label N and continue with the trees $N-1$, $N-2$, and so on. This could result in $O(N^2)$ simulation time. We avoid this situation by grafting trees that have not been involved in a graft operation in Step 7.

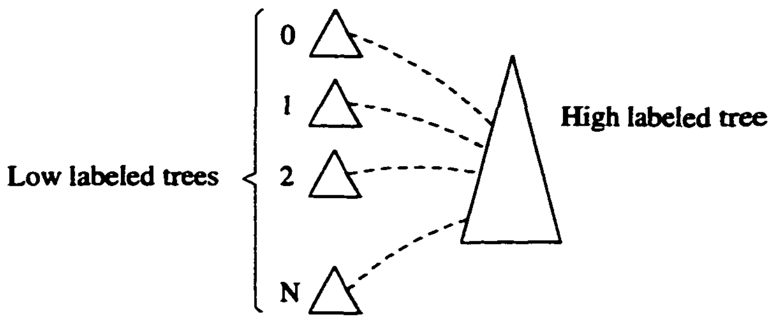


Figure 4.11: The worst case scenario for grafting trees occurs when the high labeled tree on the right grafts onto the low labeled trees in the left following a descending order.

Step 8 forces each such rejected tree to graft onto some neighboring tree. For example, in Figure 4.11, if the high labeled tree grafted onto the tree with label N, then Step 8 forces trees 0 to $N-1$ (the rejected trees) to graft onto the high labeled tree. To graft the rejected trees, $\mathcal{Z}$ performs the same procedure as Step 7, except without requiring the new parent to have a lower tree label.

Claim: Step 8 does not create cycles.

Proof: Since the label of a rejected tree is always smaller than any of the labels of its neighbor trees, two rejected trees cannot be neighbors. This implies that no rejected tree can graft onto another rejected tree, so the grafting of Step 8 does not form cycles. ■

Since each rejected tree always has at least one neighbor tree, it is always possible to graft the rejected tree onto some non-rejected neighbor.

At this point, the $2(\log N + 1)$ iterations of Steps 7 and 8 are completed and there is a spanning tree for each connected component of the distilled graph.

Step 9 - Handling writes on unselected edges: This step accounts for edges not included in the spanning tree (pseudo-Euler tour). Using Lemma 4.1, $\mathcal{Z}$ chooses a leader between the two active nodes at the ends of the edge. $\mathcal{Z}$ performs a write cycle on the bus representing the edge; each group of processors (including the active nodes on the ends) that simulates a writer processor writes its data to the bus, then the leader reads from the bus and stores the value. The leader will write this value to the pseudo-Euler tour during the writing cycle of Step 10.

Step 10 - Simulation on pseudo-Euler tour: In this step, $\mathcal{Z}$ simulates the communication among processors of $\mathcal{Q}$. Each group of processors in the pseudo-Euler tour that simulates a writer processor of $\mathcal{Q}$ now writes its data (that includes the effects of raked linear chains in Step 3 and unselected edges in Step 9) to the pseudo-

Euler tour bus, and all the groups on the pseudo-Euler tour obtain a single value by letting the bus to resolve the concurrent writes.

Step 11 - Conveying information to raked readers: Step 10 generated the final bus data for each bus of $\mathcal{Z}$. This information is available only to groups of processors included in the spanning tree. In Step 11, $\mathcal{Z}$ conveys the final bus data to the remaining groups of processors (raked linear chains and unselected edges). $\mathcal{Z}$ first broadcasts the bus data to unselected edges that were processed in Step 9, then $\mathcal{Z}$ repeats this action with the linear chains that were raked in Step 3.

This completes the transformation of all non-linear buses of the $N \times N$ Common CRCW R-Mesh, $\mathcal{Q}$, into acyclic linear buses on the $2N \times 2N$ Common CRCW LR-Mesh, $\mathcal{Z}$. We next derive the running time of the algorithm, then extend the simulation to (1) permit other concurrent write rules for $\mathcal{Q}$, (2) reduce the size of $\mathcal{Z}$ to $N \times N$, and (3) modify $\mathcal{Z}$ to use only exclusive writes.

4.2.2 Simulation Running Time

The leader election in Step 2 runs in $O(\log N)$ time (Lemma 4.2); all the remaining steps run in constant time. The iterative procedure involving Steps 7 and 8 reduces the number of possible trees (pseudo-Euler tours) by at least a factor of 2 in each iteration (explained below). This is because Step 8 guarantees that each tree is involved in at least one grafting. Since it is possible to have $O(N^2)$ trees for the same component after Step 6, the algorithm executes Steps 7 and 8 $O(\log N)$ times. Overall, the algorithm runs in $O(\log N)$ time.

Assume that tree T_i has label α_i, where $0 \leq i < N^2$. Each tree T_i must belong to one and only one of the following three disjoint sets of trees:

1. Set of all T_i whose neighbor trees have labels smaller than α_i.

2. Set of all T_i whose neighbor trees have labels smaller and greater than α_i.

3. Set of all T_i whose neighbor trees have labels greater than α_i.

Step 7 assures that all the trees in the first and second sets graft into some tree with a smaller label. Partition the third set into two disjoint subsets: the first subset includes the trees that were grafted onto by trees with larger labels (these trees contains the root group of the new tree); the second subset includes all the rejected trees (these are the only trees that have not been subject to a graft operation in Step 7). Step 8 ensures that each rejected tree grafts onto some non-rejected tree. So, Steps 7 and 8 involve every tree in a grafting operation and successfully reduce the number of trees in the forest by at least half.

4.2.3 Allowing Other Write Rules in $\mathcal{Q}$

In the simulation described above, $\mathcal{Z}$ and $\mathcal{Q}$ use the COMMON rule. If $\mathcal{Q}$ uses the COLLISION rule, $\mathcal{Z}$ can simulate $\mathcal{Q}$ by modifying Steps 3, 9, and 10. First, $\mathcal{Z}$ uses leader election (Lemma 4.1, since the buses are acyclic) to select one writer processor. Then, the leader broadcasts its index to all the processors in the bus. If other writers are present on the bus, all of them broadcast a collision symbol; otherwise, the leader broadcasts its data. $\mathcal{Z}$ performs a similar procedure if $\mathcal{Q}$ uses the COLLISION$^+$ rule, the difference is that the leader broadcasts its data rather than its index. If other writers are present on the bus with different data, they broadcast a collision symbol. When $\mathcal{Q}$ uses the PRIORITY rule, $\mathcal{Z}$ resolves concurrent writes in Steps 3, 9, and 10 using priority resolution in $O(\log N)$ time. This time does not alter the overall execution time of the algorithm. The procedure for the ARBITRARY rule (which is less restrictive than PRIORITY) is the same.

4.2.4 Reducing the Size of $\mathcal{Z}$

$\mathcal{Z}$ is four times bigger than $\mathcal{Q}$. Using the scaling simulation of Ben-Asher *et al.* [4], scale the $2N \times 2N$ Common CRCW LR-Mesh down to an $\frac{N}{2} \times \frac{N}{2}$ Collision$^+$ CRCW LR-Mesh. This reduction in size is necessary to allow the simulating LR-Mesh to get rid of the concurrent writes. Since the scaling simulation for the LR-Mesh uses the Collision$^+$ rule, we need to perform this size reduction before removing from the LR-Mesh the capability of using concurrent writes.

Remark: Since the buses of $\mathcal{Z}$ are acyclic (except for the non-writing cycles of Step 1) and the scaling simulation [4] does not create any cyclic linear bus, the buses of the $\frac{N}{2} \times \frac{N}{2}$ are also acyclic (except for those in Step 1).

4.2.5 Exclusive Write for $\mathcal{Z}$

We now simplify the simulating LR-Mesh, $\mathcal{Z}$, to use only exclusive writes. Trahan *et al.* [50] proved for the RMBM reconfigurable model that the CREW version can simulate the Common, Collision, or Collision$^+$ CRCW version in constant time, utilizing the ability to perform leader election. We follow a similar procedure to show that a CREW LR-Mesh can implement bus linearization. Specifically, we prove that a CREW LR-Mesh can simulate in constant time the Collision$^+$ LR-Mesh that uses only acyclic buses. This method is similar to the one discussed in Section 4.2.3 that simulates the Collision$^+$ rule on a CRCW Common LR-Mesh. Since we use Lemmas 4.1 and 4.2, we increase the size of the simulating machine by a factor of four to accomodate the double bus structure.

To simulate the Collision$^+$ rule, $\mathcal{Z}$ uses Lemma 4.1 (which runs with exclusive writes) to select a leader among the writer groups. The leader broadcasts its data to all groups in the bus. If some writer groups hold different data, then $\mathcal{Z}$ uses Lemma 4.1

again to find a leader to broadcast a collision symbol. During the simulation, Steps 3, 9, and 10 use concurrent writes, so $\mathcal{Z}$ can handle them with the above procedure in constant time. All the remaining steps, except Steps 1 and 2, execute on acyclic linear buses and do not require concurrent writes. In Step 1, $\mathcal{Z}$ may construct cyclic buses, but no processor writes on them. In Step 2, $\mathcal{Z}$ handles the cycles using Lemma 4.2, which can also be implemented using exclusive writes. The $O(\log N)$ time to execute Step 2 does not alter the overall running time of the algorithm, since all the other steps also run in $O(\log N)$ time.

So, using exclusive writes, an $N \times N$ LR-Mesh can simulate an $N \times N$ R-Mesh using the Common, Collision, Collision$^+$, Priority, or Arbitrary rules in $O(\log N)$ time, proving Theorem 4.3.

4.3 Scaling Simulations

In Section 1.2, we introduced the notion of a scaling simulation, which adapts an algorithm instance designed to run on a model of arbitrary size to run on a smaller model without significant loss of efficiency. This section applies bus linearization to construct improved scaling simulations for the R-Mesh and the FR-Mesh. In both cases, the simulating machine is a CREW LR-Mesh. Before doing so, we briefly recount current results for scaling different versions of the R-Mesh. The result we present in the followin section improves over all previous R-Mesh scaling simulations.

4.3.1 R-Mesh Scaling Simulation

For ease of explanation, we describe the scaling simulation as a sequence of three phases, but, in fact, the three phases roll into one for execution:

Phase 1. Simulation of an $N \times N$ CRCW R-Mesh on an $N \times N$ CREW LR-Mesh,

Phase 2. Simulation of an $N \times N$ CREW LR-Mesh on a $\frac{P}{2} \times \frac{P}{2}$ CRCW LR-Mesh, and

Phase 3. Simulation of a $\frac{P}{2} \times \frac{P}{2}$ CRCW LR-Mesh on a $P \times P$ CREW LR-Mesh.

The first phase uses bus linearization to get rid of non-linear buses. The second phase (optimally) scales down the simulating LR-Mesh. Finally, the third phase refines this result so that the simulating LR-Mesh uses only exclusive writes.

This simulation allows the powerful and flexible algorithm design permitted by the CRCW model with arbitrary bus structure, while using a simple bus structure more feasible to implement since it just requires exclusive writes.

Phase 1

Use the bus linearization procedure of Section 4.2 to simulate each step of an $N \times N$ Common, Collision, Collision$^+$, Arbitrary, or Priority CRCW R-Mesh on an $N \times N$ CREW LR-Mesh in $O(\log N)$ time.

Phase 2

This phase uses the scaling simulation of Ben-Asher *et al.* [4] to scale the $N \times N$ CREW LR-Mesh of Phase 1 down to a $\frac{P}{2} \times \frac{P}{2}$ Collision$^+$ CRCW LR-Mesh. Therefore, with Phase 1, we obtain the following.

- For any $P < N$, any step of an $N \times N$ Common, Collision, Collision$^+$, Arbitrary, or Priority CRCW R-Mesh can be simulated on a $\frac{P}{2} \times \frac{P}{2}$ Collision$^+$ CRCW LR-Mesh in $O\left(\frac{N^2}{P^2} \log N\right)$ time.

Phase 3

This phase simplifies the simulating $\frac{P}{2} \times \frac{P}{2}$ LR-Mesh to use only exclusive writes. Use the procedure presented in Section 4.2.5 to transform the COLLISION$^+$ CRCW buses to CREW buses. This procedure requires a $P \times P$ CREW LR-Mesh and runs in constant time. Combining this result with the one of Phase 2, we obtain the following theorem.

Theorem 4.4 *For any $P < N$, any step of an $N \times N$* COMMON, COLLISION, COLLISION$^+$, ARBITRARY, *or* PRIORITY *CRCW R-Mesh can be simulated on a $P \times P$ CREW LR-Mesh in $O\left(\frac{N^2}{P^2} \log N\right)$ time.* ■

Although Matias and Schuster [32] also simulated the general R-Mesh via the LR-Mesh, their simulation is randomized and quite different from the one proposed in this paper. Their simulation computes connected components in two stages. In the first stage, they obtained the connected components of each sub-mesh of size $P \times P$ of the $N \times N$ R-Mesh, and used it in the second stage to compute the connected components on a graph with $O\left(\frac{N^2}{P}\right)$ nodes and $O\left(\frac{N^2}{P}\right)$ edges. The connected components algorithm in the second stage uses an LR-Mesh simulation of a randomized PRAM algorithm.

On the other hand, the main part of our scaling simulation is the bus linearization procedure, which transforms the simulated R-Mesh bus configuration into an equivalent LR-Mesh bus configuration of nearly the same size (that later can be scaled down optimally). Our simulation is deterministic and the input is a graph with $O(N^2)$ nodes and $O(N^2)$ edges. Another important difference is that, to attain the stated overhead in the simulation of Matias and Schuster, the write rule for the simulating machine must be ARBITRARY, which is difficult to implement in a bus; when its simulation uses the COLLISION rule, the simulation overhead is not constant (see Table 1.1). In contrast, our simulation uses only exclusive writes.

4.3.2 FR-Mesh Scaling Simulation

In Chapter 3, we developed a scaling simulation for the FR-Mesh. Though most steps of this scaling simulation run on an LR-Mesh, some parts require processors to internally connect all four of their ports (this is not permitted on an LR-Mesh) while handling a $P \times P$ sized "window" of the FR-Mesh. Given Theorem 4.3, a CREW LR-Mesh can now simulate the connection pattern of the simulating FR-Mesh window in $O(\log P)$ time. Since the above FR-Mesh simulation requires the simulating LR-Mesh to find this equivalent configuration a constant number of times per window, the simulation overhead of this new simulation is still $O(\log P)$ (see Table 1.1).

The following corollary expresses this result.

Corollary 4.5 *For any $P < N$, any step of an $N \times N$* COMMON, COLLISION, COLLISION$^+$, ARBITRARY, *or* PRIORITY *CRCW FR-Mesh can be simulated on a $P \times P$ CREW LR-Mesh in $O\left(\frac{N^2}{P^2} \log P\right)$ time.*

This matches the overhead of the previous scaling simulation that used the more powerful COMMON CRCW FR-Mesh as the simulating model.

4.4 Simulation of R-Mesh by PR-Mesh

The Pipelined Reconfigurable Mesh or PR-Mesh [45] is a special type of reconfigurable mesh that uses optical buses. It can configure its port partitions to form linear buses as in an LR-Mesh, but with no cycles. In addition, each optical bus can perform multiple one-to-one communications in constant time by using pipelining. Figure 4.12 shows a

1×4 PR-Mesh. It has two optical waveguides for addressing and one for transmitting messages. Each of these waveguides consists of two bus segments (upper and lower) connected at one of the ends, forming a directional U-shaped bus. Processors use the upper bus segment to write and the lower segment to read. There are fixed and conditional delays in the addressing waveguides to allow processors to select the destinations of their messages. When more than one processor sends a message to the same destination, the destination accepts the first message that arrives, thus solving the conflict by a PRIORITY rule, where the processor nearest to the "U-turn" has the highest priority.

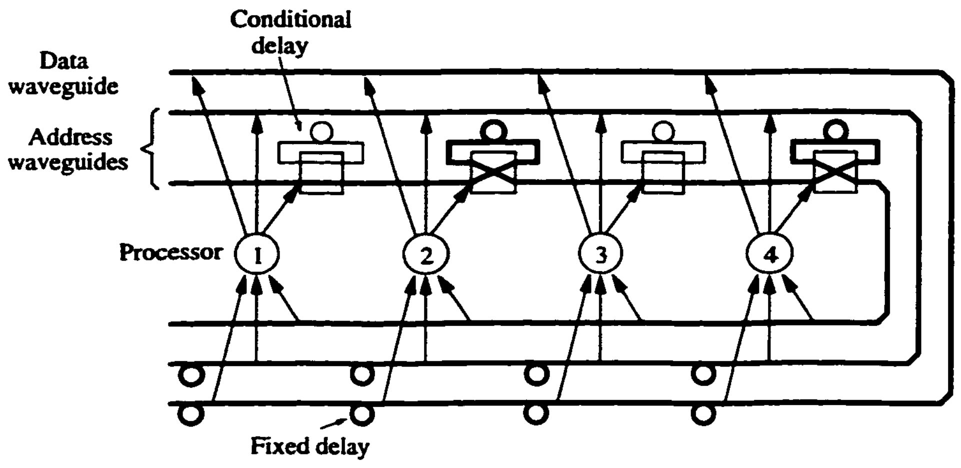


Figure 4.12: 1×4 PR-Mesh.

A $P \times P$ PR-Mesh can simulate each step of a $P \times P$ CREW LR-Mesh with no cycles in constant time as follows. Scale the LR-Mesh down to a $\frac{P}{2} \times \frac{P}{2}$ LR-Mesh. Then, replicate the connections of this new LR-Mesh on the PR-Mesh, creating a double bus structure to decide which of the two end-processors connects the upper and lower segments. Finally, use only one of these two buses to broadcast the written value. (Earlier papers [41, 45] also used broadcasting on the linear buses of the

PR-Mesh to simulate the linear buses of the LR-Mesh, though they did not explicitly address the leader election problem.)

Corollary 4.6 *Any step of an $N \times N$ CREW LR-Mesh can be simulated on an $N \times N$ PR-Mesh in $O(\log N)$ time.*

Remark: The running time of the above simulation is the time to cut cycles of the LR-Mesh. For an acyclic LR-Mesh, the above simulation runs in constant time.

Combining Corollary 4.6 with Theorem 4.3, Theorem 4.4, and Corollary 4.5, we obtain the following.

Corollary 4.7 *Any step of an $N \times N$* COMMON, COLLISION, COLLISION$^+$, ARBITRARY, *or* PRIORITY *CRCW R-Mesh can be simulated on an $N \times N$ PR-Mesh in $O(\log N)$ time.*

Corollary 4.8 *For any $P < N$, any step of an $N \times N$* COMMON, COLLISION, COLLISION$^+$, ARBITRARY, *or* PRIORITY *CRCW R-Mesh can be simulated on a $P \times P$ PR-Mesh in $O\left(\frac{N^2}{P^2} \log N\right)$ time.*

Corollary 4.9 *For any $P < N$, any step of an $N \times N$* COMMON, COLLISION, COLLISION$^+$, ARBITRARY, *or* PRIORITY *CRCW FR-Mesh can be simulated on a $P \times P$ PR-Mesh in $O\left(\frac{N^2}{P^2} \log P\right)$ time.*

Since the PR-Mesh is simulating a CREW LR-Mesh, then at most one processor at a time broadcasts data on each bus. For this reason, a restricted model of the PR-Mesh with no fixed and conditional delays with only one addressing waveguide rather than two suffices to simulate the CREW LR-Mesh. The reason for the existence of the second waveguide in the PR-Mesh is to be able to transmit messages to selected destinations. Our simulation does not use this feature, so a simpler model is enough. Consequently, we can readily extend the simulation to work on other reconfigurable models with optical buses, as described below.

Bourgeois and Trahan [7] proved that the classes of languages accepted in constant time with polynomial number of processors by the PR-Mesh, the APPBS [17], and the AROB [41] are the same. Since the PR-Mesh is a restricted version of the AROB, the bus linearization method also works for the AROB with the same overhead as for the PR-Mesh.

The APPBS is different from the PR-Mesh; this model uses switches to connect processors to buses, and these switches allow only four configurations. As a result, the APPBS cannot end a bus in the middle of the mesh. Bourgeois and Trahan [7] presented a simulation of a cycle-free LR-Mesh using an APPBS that runs in constant time. Bus linearization applies to the APPBS holding the same overhead as for the PR-Mesh.

Thus, the results of Corollaries 4.6, 4.7, 4.8, and 4.9 also apply to the AROB and APPBS.

Chapter 5

Simulation of DR-Mesh by LR-Mesh

This chapter deals with the problem of running algorithms designed for directed reconfigurable models, specifically the directed R-Mesh (DR-Mesh), on undirected models, specifically the LR-Mesh. Ben-Asher *et al.* [3] proved that an $N \times N$ LR-Mesh can simulate each step of an $N \times N$ directed LR-Mesh (DLR-Mesh) in constant time. The reverse simulation also runs in constant time, but the DLR-Mesh uses $2N \times 2N$ processors to simulate an $N \times N$ LR-Mesh. In this simulation, they assumed a directed model that is more restricted than the one we use in this chapter (see Section 5.1). They also proved that a $2N \times 2N$ DR-Mesh can simulate each step of an $N \times N$ R-Mesh in constant time.

Ben-Asher *et al.* [5] proved that the class of languages accepted by a DR-Mesh (resp., R-Mesh) in constant time with polynomial number of processors is equivalent to the class NL (resp., SL) of languages accepted in non-deterministic logarithmic space (resp., symmetric logarithmic space) on a Turing machine. Since it is widely conjectured that $SL \subset NL$, it is not likely that the R-Mesh can simulate a step of the DR-Mesh in constant time even with a super-polynomial increase in the number of processors. On the other hand, simulating an R-Mesh on a DR-Mesh is straightforward, since the DR-Mesh possesses all the features of an R-Mesh.

Trahan *et al.* [48] proved that each step of an $N \times N$ DR-Mesh can be simulated by an $O(N^4) \times O(N^4)$ R-Mesh in $O(\log N)$ time. This simulation is fast, but requires too many processors. To perform the simulation, they constructed a graph where the ports are nodes, then, by finding the transitive closure, they determined the destinations of messages written to ports. We present a simulation of a CRCW DR-Mesh on a CRCW LR-Mesh that follows the same approach as theirs, but runs more efficiently. Both the simulated and simulating machines use the same concurrent write rule, one of COMMON, COLLISION, or COLLISION$^+$; later we will extend this result by restricting the simulating LR-Mesh to use only exclusive writes. In our simulation, the simulating LR-Mesh, a model weaker than the R-Mesh, has $O\left(N \times N \times \frac{N}{\log N}\right)$ processors in three dimensions (or $O\left(\frac{N^2}{\log N} \times \frac{N^2}{\log N}\right)$ processors in two dimensions). The simulation runs in $O\left(\log^2 N\right)$ time.

Section 5.1 describes the DR-Mesh. Section 5.2 defines the basic terminology used in the simulation. Section 5.3 gives a general description of the simulation, while Sections 5.4 and 5.5 detail the phases of the simulation, then Section 5.6 proves its correctness. Finally, Section 5.7 refines the DR-Mesh simulation to run on an exclusive write simulating model and on a model with pipelined optical buses.

5.1 The DR-Mesh

The structure of the DR-Mesh differs from the R-Mesh, in that the DR-Mesh has two oppositely directed buses to connect each pair of neighboring processors (see Figure 5.1), rather than one undirected bus as in the R-Mesh. The data on a directed bus propagates in only one direction. Each processor has four output ports (black circles in Figure 5.1) connected to outgoing buses and four input ports (white circles in Figure 5.1) connected to incoming buses. The internal connection between ports can be any partition of the set of ports. This assumption allows 4140 different port partitions (the R-Mesh has only 15). Figure 5.1 shows a 3×5 DR-Mesh.

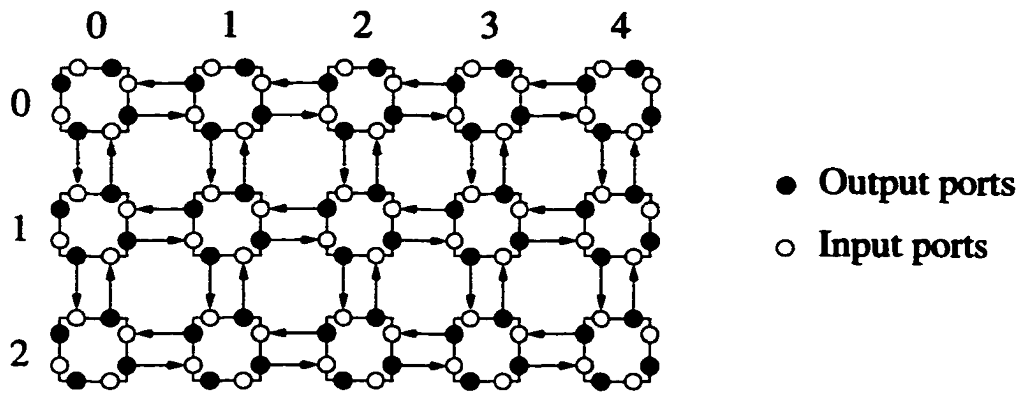
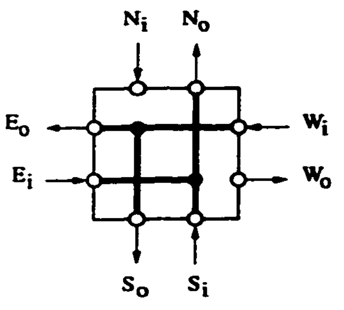


Figure 5.1: 3×5 DR-Mesh.

Information propagation in the DR-Mesh: Consider the port configuration in the example of Figure 5.2(a). Assume that information α arrives at port W_i. Since ports W_i, E_o, and S_o are connected together, information α propagates to ports E_o and S_o. Notice in the transitive closure matrix of Figure 5.2(b) (that for this example is the same as the adjacency matrix) that columns E_o and S_o have '1' entries in their intersections with row W_i, this means that both E_o and S_o are reachable from W_i.

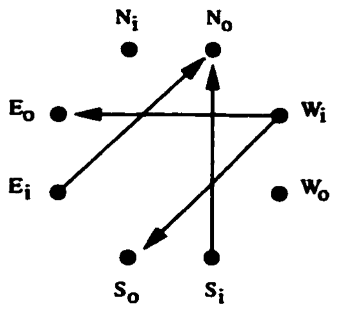
Now, assume that port S_i receives information β. This information propagates only to port N_o and not to port E_i, even though the three of them are in the same block of the partition. Notice that the transitive closure of Figure 5.2(b) indicates that port E_i is not reachable from S_i (there is no path between S_i and E_i, as shown in the transitive closure graph of Figure 5.2(c)), since there is a '0' in the corresponding entry.



(a)

$$\begin{array}{c|cccccccc}
 & N_i & N_o & E_i & E_o & W_i & W_o & S_i & S_o \\
\hline
N_i & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
N_o & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
E_i & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
E_o & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
W_i & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
W_o & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
S_i & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
S_o & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}$$

(b)



(c)

Figure 5.2: Representation of connections of a DR-Mesh processor: a) Port configuration for a DR-Mesh processor; b) Transitive closure matrix of the ports; c) Transitive closure graph of the ports.

5.2 DR-Mesh Simulation Terminology

Let $\mathcal{Q}$ denote an $N \times N$ CRCW DR-Mesh (the simulated machine) and $\mathcal{Z}$ a $16N \times 16N \times \frac{N}{\log N}$ CRCW LR-Mesh (the simulating machine). Let both $\mathcal{Z}$ and $\mathcal{Q}$ have the same concurrent write rule, one of COMMON, COLLISION, or COLLISION$^+$. Let $\tau(i)$ (a *simulated tile*) denote a $2^i \times 2^i$ sub-DR-Mesh of the simulated machine $\mathcal{Q}$. Let $T(i)$ (a *simulating tile*) denote the corresponding $16\,(2^i) \times 16\,(2^i)$ sub-LR-Mesh of the simulating machine $\mathcal{Z}$ that simulates $\tau(i)$. DR-Mesh $\mathcal{Q}$ and LR-Mesh $\mathcal{Z}$ partition into $\left(\frac{N}{2^i}\right)^2$ tiles of size $2^i \times 2^i$ and $16(2^i) \times 16(2^i)$, respectively, for each $0 \leq i \leq \log N$. Note that $\tau(i)$ and $T(i)$ are generic symbols for tiles of size $2^i \times 2^i$ and $16(2^i) \times 16(2^i)$, rather than particular tiles of that size. A tile $\tau(i)$, where $1 \leq i \leq \log N$, contains four sub-tiles of size $2^{i-1} \times 2^{i-1}$. Denote these sub-tiles by $\tau_1(i-1)$, $\tau_2(i-1)$, $\tau_3(i-1)$, and $\tau_4(i-1)$ (see Figure 5.3). Let an *internal port* of tile $\tau(i)$ be a port that communicates between two sub-tiles within tile $\tau(i)$. Let an *external port* of tile $\tau(i)$ be a port that communicates with a neighboring tile $\tau(i)$. Figure 5.3 shows tile $\tau(1)$ comprising four sub-tiles $\tau_1(0)$, $\tau_2(0)$, $\tau_3(0)$, and $\tau_4(0)$. It also shows the internal and external ports of $\tau(1)$.

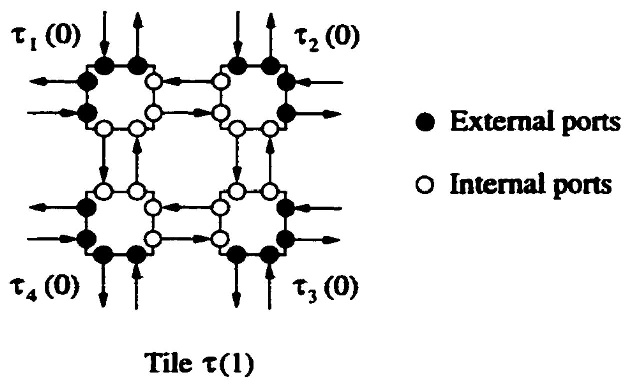


Figure 5.3: Tile $\tau(1)$, its four sub-tiles $\tau_1(0), \ldots, \tau_4(0)$, and its internal and external ports.

Let $A(i)$ denote the *adjacency matrix* of the internal and external ports of a tile $\tau(i)$. An entry in the adjacency matrix is '1' if and only if the two corresponding ports are neighbors. Different tiles $\tau(i)$ have different matrices $A(i)$. The size of matrix $A(i)$ is $16(2^i) \times 16(2^i)$, where the number of columns and rows corresponds to the total number of internal and external ports in tile $\tau(i)$. Let $A^*(i)$ denote the *transitive closure matrix* of $A(i)$. Let $A^+(i)$ denote the *reduced transitive closure matrix* of $A^*(i)$. Obtain $A^+(i)$ by removing from $A^*(i)$ rows and columns assigned to internal ports, so the size of matrix $A^+(i)$ is $8(2^i) \times 8(2^i)$. We use this reduction to keep the size of the transitive closure proportional to the size of the tile; otherwise, by including all the ports within the tile in the transitive closure, its size will grow in terms of the square of the size of the tile. Each processor $p_{k,\ell,0}$ of a tile $T(i)$, where $0 \leq k, \ell < 16(2^i)$ (resp., $0 \leq k, \ell < 8(2^i)$), holds the entry of row k and column ℓ of the matrix $A^*(i)$ (resp., $A^+(i)$). Let *Dout*(i) denote the set of bus data leaving the external output ports of tile $\tau(i)$. The elements of *Dout*(i) are the results of the contributions of all writes that originate within tile $\tau(i)$. Let *Din*(i) denote the set of bus data entering the external input ports of tile $\tau(i)$; this data could result from writes originating anywhere in the DR-Mesh (including within the tile in question).

5.3 DR-Mesh Simulation Description

To simulate an $N \times N$ DR-Mesh, the simulating LR-Mesh uses $16N \times 16N \times \frac{N}{\log N}$ processors. In the first two dimensions, a group of 16×16 processors of the simulating machine, $\mathcal{Z}$, is responsible for a single processor of the simulated machine, $\mathcal{Q}$. The processors in the third dimension increase the speed of the simulation in operations such as matrix multiplication and data movement. The simulation reduces to the problem of determining connectivity of the ports of the simulated machine, that is, determining all possible destinations for data written at any output port of the simulated machine. Although the idea is similar to that of Trahan *et al.* [48], we use the processors much more efficiently by using a divide-and-conquer strategy. The simulation consists of two phases. The first phase uses the algorithm *Going_Out* which we describe next.

The objective of algorithm *Going_Out* is to obtain the set of data *Dout*$(\log N)$ and the reduced transitive closure $A^{+}(\log N)$ for the simulated DR-Mesh, $\mathcal{Q}$. The algorithm divides $\mathcal{Q}$ into four sub-meshes of the same size. Then, it solves the problem recursively to obtain *Dout*$(\log N - 1)$ and $A^{+}(\log N - 1)$ for each sub-mesh. Finally, the algorithm combines the matrices $A^{+}(\log N - 1)$ of the four sub-meshes to generate the transitive closure matrix $A^{*}(\log N)$. Using this matrix and the set of data *Dout*$(\log N - 1)$ of each of the four sub-meshes, the algorithm calculates *Dout*$(\log N)$. It is straightforward to obtain $A^{+}(\log N)$ by removing specific columns and rows from $A^{*}(\log N)$. The set *Dout*$(\log N)$ represents the information that leaves $\mathcal{Q}$, which is irrelevant for our simulation; on the other hand, the sets $Dout(i)$, for $1 \leq i < \log N$, are fundamental to find the final bus data at each port of $\mathcal{Q}$.

The second part of the simulation (algorithm *Going_In*) starts by splitting an $N \times N$ tile into four $\frac{N}{2} \times \frac{N}{2}$ sub-tiles. It employs the transitive closure matrix $A^*(\log N)$ that relates the ports at the borders of the four $\frac{N}{2} \times \frac{N}{2}$ sub-tiles to combine the data arriving at the external ports of the $N \times N$ tile with data $Dout_j(\log N - 1)$ generated within each sub-tile. The objective of this procedure is to determine the final data that reaches each input port of the sub-tiles of size $\frac{N}{2} \times \frac{N}{2}$. The algorithm iterates this procedure and stops when it reaches tiles of size 1×1. The bus data at the input ports of 1×1 tiles is the final bus data that includes the effect of all writing in the simulated machine.

5.4 Algorithm Going_Out

Figure 5.4 shows the recursive algorithm *Going_Out*. It consists of $\log N$ recursion levels. The inputs for the i^{th} level of recursion are the reduced transitive closure matrices $A_j^+(i-1)$ and the set of bus data $Dout_j(i-1)$, for each $1 \leq j \leq 4$, from the four component sub-tiles $\tau_j(i-1)$ of tile $\tau(i)$. One of the outputs is the matrix $Dout(i)$, which represents the bus data at the external output ports of $\tau(i)$; the other output is the reduced transitive closure matrix $A^+(i)$. LR-Mesh $\mathcal{Z}$ uses matrix $A^*(i)$ to calculate $Dout(i)$. Using the configuration of a tile $\tau(0)$ (a single processor), the corresponding tile T_0 can construct the transitive closure $A^*(0) = A^+(0)$ of $\tau(0)$ in constant time (for example, see Figure 5.2). Initially, $Dout(0)$ is just the data written by ports N_o, S_o, W_o, E_o. Figure 5.4 shows a pseudo-code for procedure *Going_Out*. We next discuss subroutines within this procedure.

```
Procedure Going_Out ($\tau(i)$)    /* Determines matrices $Dout(i)$ and $A^+(i)$ */
    if $i = 0$ then
        return $Dout(0)$ and $A^+(0)$
    else
        Divide $\tau(i)$ into four sub-tiles $\tau_1(i-1)$, $\tau_2(i-1)$, $\tau_3(i-1)$, and $\tau_4(i-1)$
        for $j \longleftarrow 1$ to 4 pardo
            Going_Out ($\tau_j(i-1)$)
        $A^*(i) \longleftarrow$ Find_A* $(A_1^+(i-1), A_2^+(i-1), A_3^+(i-1), A_4^+(i-1))$
        $Dout(i) \longleftarrow$ Find_Dout $(A^*(i), Dout_1(i-1), \ldots, Dout_4(i-1))$
        $A^+(i) \longleftarrow$ Find_A+ $(A^*(i))$
        return $Dout(i)$ and $A^+(i)$
end
```

Figure 5.4: Pseudo-code for algorithm *Going_Out*.

5.4.1 Procedure Find_A*

This procedure generates $A^*(i)$, the transitive closure of the internal and external ports of tile $\tau(i)$, given the matrices $A_j^+(i-1)$, for each $1 \leq j \leq 4$. Procedure *Find_A** consists of three stages.

Stage 1 - Moving matrices $A_j^+(i-1)$: Move the matrices $A_2^+(i-1)$ and $A_4^+(i-1)$ from tiles $T_2(i-1)$ and $T_4(i-1)$, respectively, to the main diagonal of $T(i)$ (see Figure 5.5). These matrices are part of the adjacency matrix $A(i)$, which is completed in Stage 2. The movement of matrices in Stage 1 assigns the task of representing a port (internal or external) of $\tau(i)$ to each row and column of processors of $T(i)$.

Using the processors of the third dimension, $\mathcal{Z}$ performs Stage 1 in $\lceil \frac{(16)2^i \log N}{N} \rceil$ steps; this is a constant if $1 \le i \le \log N - \log\log N$.

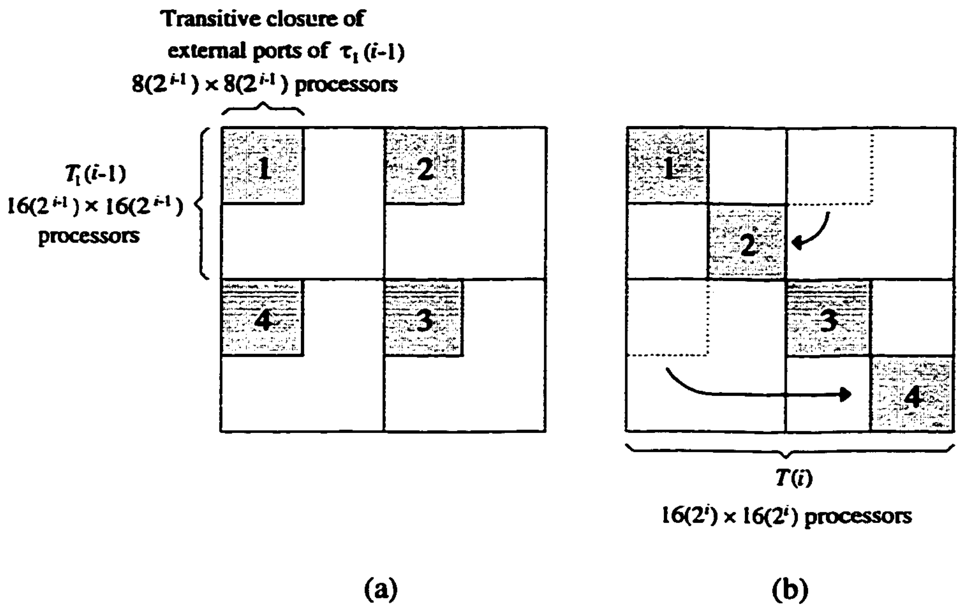

Transitive closure of
external ports of $\tau_1(i\text{-}1)$
$8(2^{i-1}) \times 8(2^{i-1})$ processors
$T_1(i\text{-}1)$
$16(2^{i-1}) \times 16(2^{i-1})$
processors
1
2
3
4
$T(i)$
$16(2^i) \times 16(2^i)$ processors
(a)
(b)

Figure 5.5: Moving matrices $A_2^+(i-1)$ and $A_4^+(i-1)$: a) Initial location of matrices $A_2^+(i-1)$ and $A_4^+(i-1)$ in $T(i)$; b) Final location of these matrices.

Stage 2 - Constructing matrix $A(i)$: Each processor of $T(i)$ that does not have an entry of $A(i)$ generates an entry of the adjacency matrix $A(i)$ as explained below. The entry is '1' if and only if a processor is located at any of the following positions (see Figure 5.3).

1. The intersection of any row representing a north output port in tile $\tau_4(i-1)$ (resp., $\tau_3(i-1)$) and any column representing a south input port of tile $\tau_1(i-1)$ (resp., $\tau_2(i-1)$).

2. The intersection of any row representing a south output port in tile $\tau_1(i-1)$ (resp., $\tau_2(i-1)$) and any column representing a north input port of tile $\tau_4(i-1)$ (resp., $\tau_3(i-1)$).

3. The intersection of any row representing an east output port in tile $\tau_1(i-1)$ (resp., $\tau_4(i-1)$) and any column representing a west input port of tile $\tau_2(i-1)$ (resp., $\tau_3(i-1)$).

4. The intersection of any row representing a west output port in tile $\tau_2(i-1)$ (resp., $\tau_3(i-1)$) and any column representing an east input port of tile $\tau_1(i-1)$ (resp., $\tau_4(i-1)$).

These conditions are straightforward to check, so $\mathcal{Z}$ executes Stage 2 in constant time.

Stage 3 - Constructing matrix $A^*(i)$: Compute the transitive closure matrix $A^*(i) = (I + A(i))^{2^{(i+4)}}$, where I is the $16(2^i) \times 16(2^i)$ identity matrix [19].

Tile T_i (using the processors of the third dimension) applies the method of repeated squaring to calculate $A^*(i)$, where $1 \le i \le \log N$. This method uses $i+4$ iterations, so tile T_i performs $i+4$ Boolean matrix multiplications. Tile T_i computes a Boolean matrix multiplication in $\lceil \frac{(16)2^i \log N}{N} \rceil$ steps, which is constant if $i \le \log N - \log\log N$.

5.4.2 Procedure Find_Dout

Procedure *Find_Out* obtains the bus data, *Dout*(i), generated within the tile $\tau(i)$, and determines how this data propagates to the external output ports of $\tau(i)$. Figure 5.6 shows how Algorithm *Going_Out* propagates bus data *Dout*(i) to external ports for three different levels of recursion. The inputs for this procedure are the transitive closure matrix, $A^*(i)$, and the bus data, *Dout*$_j(i-1)$, from each sub-tile $\tau_j(i-1)$, for each $1 \le j \le 4$. Procedure *Find_Dout* consists of the following steps.

1. Each processor in $T(i)$ configures its ports as crossover.

2. Each processor in the leftmost column of $T(i)$ whose row represents an output

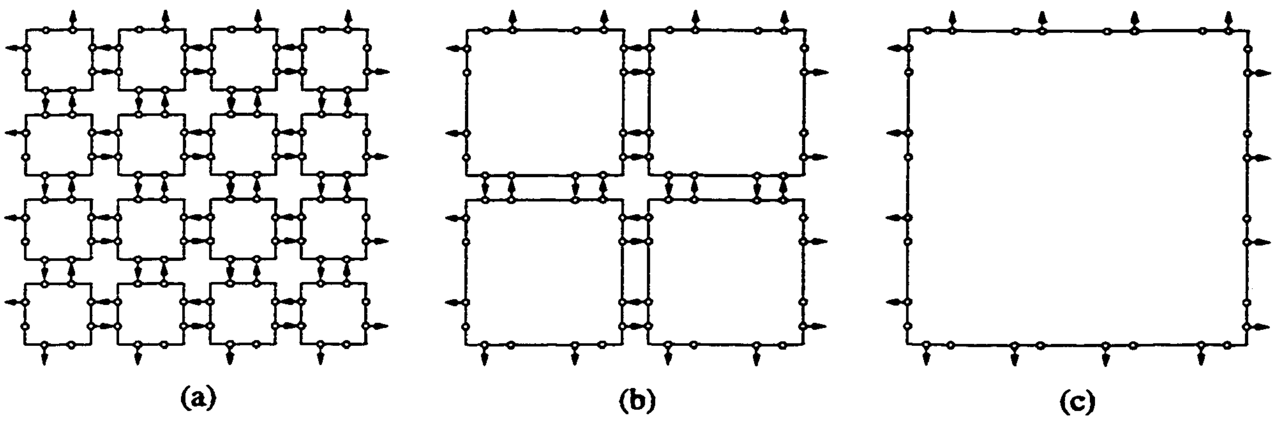


Figure 5.6: Algorithm *Going_Out* propagates bus data *Dout*(i) (shown by arrows) to external ports; Figures (a) to (c) show this propagation for different levels of recursion.

port (from some sub-tile $\tau_j(i-1)$) writes the data corresponding to that port from $Dout_j(i-1)$ on its horizontal bus.

3. Each processor in $T(i)$ that holds a '1' entry in the matrix $A^*(i)$ reads from its horizontal bus and writes that value to its vertical bus (forming "paths" between ports represented by horizontal and vertical buses). Concurrent writes may occur in this step. $\mathcal{Z}$ handles concurrent writes by using the same write rule (COMMON, COLLISION, or COLLISION$^+$) as $\mathcal{Q}$.

4. Each processor in the first row of $T(i)$ whose column represents an external output port reads its vertical bus and stores the value. These values represent the set *Dout*(i).

$\mathcal{Z}$ executes Procedure *Find_Dout* in constant time.

5.4.3 Procedure Find_A$^+$

This procedure generates the matrix $A^+(i)$. Obtain this matrix from the transitive closure matrix $A^*(i)$ by removing entries in processors of columns and rows that represent internal processors of $\tau(i)$ and compacting the remaining columns and rows to the first $8(2^i)$ column and row processors of $T(i)$. This is a routing problem similar to Stage 1 of procedure *Find_A**.

Using the processors of the third dimension, $\mathcal{Z}$ performs procedure *Find_A*$^+$ in $\lceil\frac{(16)2^i\log N}{N}\rceil$ steps, which is a constant if $1 \le i \le \log N - \log\log N$.

Execution time. The execution time of algorithm *Going_Out* is due mainly to procedure *Find_A**, which consists of three stages. For $1 \le i \le \log N - \log\log N$, $\mathcal{Z}$ performs Stages 1 and 2 in constant time and Stage 3 in $O(i)$ time. Thus, $\mathcal{Z}$ completes recursion levels 1 to $\log N - \log\log N$ in

$$\sum_{i=1}^{\log N - \log\log N} i = O\left(\log^2 N\right) \text{ time.}$$

For $i > \log N - \log\log N$, $\mathcal{Z}$ performs Stage 1 in $O\left(\frac{2^i\log N}{N}\right)$ time, Stage 2 in constant time, and Stage 3 in $O\left(\frac{(i)2^i\log N}{N}\right)$ time. Thus, $\mathcal{Z}$ completes recursion levels $\log N - \log\log N$ to $\log N$ in

$$\sum_{i=\log N - \log\log N}^{\log N} \frac{(i)2^i\log N}{N} = O\left(\log^2 N\right) \text{ time.}$$

Overall, $\mathcal{Z}$ executes algorithm *Going_Out* in $O\left(\log^2 N\right)$ time.

5.5 Algorithm Going_In

Algorithm *Going_Out* comprises Phase 1 of the simulation of a DR-Mesh by an LR-Mesh. Phase 1 collects information from individual processors in tiles $\tau(i)$ and propagates them outward to the whole DR-Mesh at the level of tile borders. Phase 2 distributes the information generated by Phase 1 down to individual processors. Phase 2 consists of Algorithm *Going_In* which we discuss next.

Algorithm *Going_In* proceeds in a reverse fashion with respect to *Going_Out*. The inputs to the i^{th} level of recursion of *Going_In* are the following:

1) Tile $\tau(i)$,

2) Set of bus data $Din(i)$,

3) Set of bus data $Dout_j(i-1)$, for each $1 \leq j \leq 4$, and

4) Transitive closure matrix $A^*(i)$.

Remark: Since algorithm *Going_Out* computed $Dout_j(i-1)$ and $A^*(i)$ for every possible i, each tile $\tau(i)$ holds these values.

```
Procedure Going_In ($\tau(i), Din(i)$)    /* Determines $Din_j(i-1)$ for each sub-tile $\tau_j(i-1)$ */
    if $i = 0$ then
        return
    else
        $Din_1(i-1), \ldots, Din_4(i-1) \leftarrow$ Find_Din ($Din(i), A^*(i), Dout_1(i-1), \ldots, Dout_4(i-1)$)
        for $j \leftarrow 1$ to 4 pardo
            Going_In ($\tau_j(i-1), Din_j(i-1)$)
end
```

Figure 5.7: Pseudo-code for algorithm *Going_In*.

The outputs for each level of recursion are the sets of data $Din_j(i-1)$, for each $1 \leq j \leq 4$. The final objective of algorithm *Going_In* is to determine the set of bus data $Din(0)$ for each tile $\tau(0)$.

A fundamental part of algorithm *Going_In* is the procedure *Find_Din*, which we will describe next. Initially, $i = \log N$ and each element of the set $Din(\log N)$ is null, because tile $\tau(\log N)$ does not have any neighbors and no bus data enters the tile through its external ports. The inputs $A^*(i)$ and $Dout_j(i-1)$, for $1 \leq j \leq 4$, were calculated by algorithm *Going_Out*, so each tile possesses these inputs. Procedure *Find_Din* consists of the following steps.

1. Each processor of $T(i)$ configures its ports as crossover.

2. Each processor in the leftmost column of $T(i)$ whose row represents an external input port or an internal output port of $\tau(i)$ writes on its horizontal bus the bus datum of that port. ($Din(i)$ contains the bus data for external input ports of $\tau(i)$, and $Dout_j(i-1)$ for the internal output ports of $\tau(i)$.)

3. Each processor in $T(i)$ holding a '1' entry in the matrix $A^*(i)$ reads from its horizontal bus and writes that value to its vertical bus. Concurrent writes may occur in this step and $\mathcal{Z}$ handles them by the same write rule (COMMON, COLLISION, or COLLISION$^+$) as $\mathcal{Q}$.

4. Each processor of $T(i)$ whose column represents an external or internal input port reads its vertical bus and stores the value. (These values comprise the set of bus data $Din_j(i-1)$.)

Figure 5.8(a) shows how $Din(i)$ (represented by black arrows) enters tile $\tau(i)$. Figure 5.8(b) shows how Algorithm *Find_Din* combines $Din(i)$ (black arrows) with $Dout(i-1)$ (represented by white arrows) to find $Din(i-1)$. Finally, Figure 5.8(c) shows $Din(i-1)$ for each of the four sub-tiles.

$\mathcal{Z}$ executes procedure *Find_Din* in constant time and algorithm *Going_In* in $O(\log N)$ time.

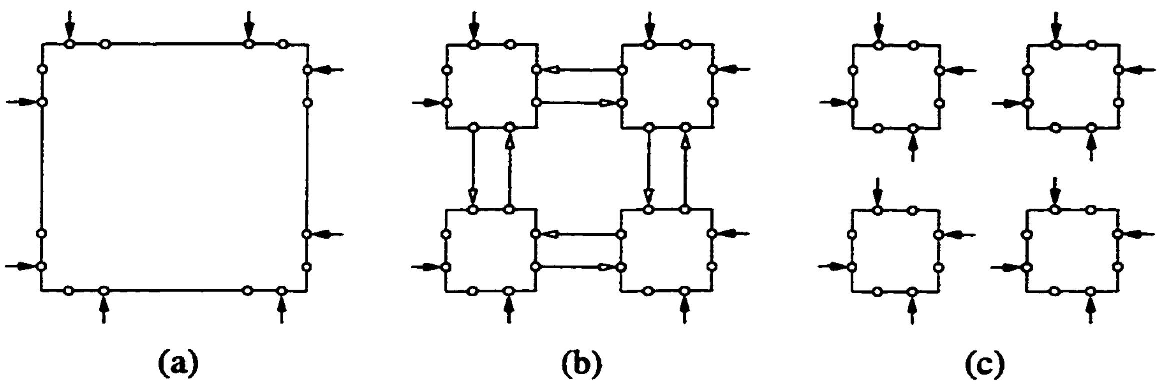


Figure 5.8: Procedure *Find_Din*: a) Data $Din(i)$ (black arrows) entering tile $\tau(i)$; b) Combining data $Din(i)$ and $Dout(i{-}1)$ (white arrows); c) Resulting data $Din(i{-}1)$ entering tiles $\tau(i-1)$.

5.6 Algorithm Correctness

This section proves the correctness of the DR-Mesh simulation algorithm. Specifically, we will prove the correctness of algorithms *Going_Out* and *Going_In*.

Lemma 5.1 *During Algorithm Going_Out, the set of bus data* $Dout(i)$*, for any* $0 \leq i \leq \log N$*, represents the contribution of all writes generated within tile* $\tau(i)$ *that propagate to neighboring tiles.*

Proof: We use induction on the level of recursion i. Our induction hypothesis is that for any $0 \leq i \leq \log N$, any value written by an output port X inside (not necessarily on the border of) tile $\tau(i)$ appears in $Dout(i)$ for external output port Z of tile $\tau(i)$, if X has a path to Z within tile $\tau(i)$.

The basis of the induction is when $i = 0$; at this point, tile $\tau(0)$ is a single processor and the set of bus data, $Dout(0)$, that leave the tile comprises just the data written by the four output ports of the processor.

Assume that the induction hypothesis holds for all values of i, where $0 \leq i < k$. We will show that the induction hypothesis also holds for $i = k$. By definition, all border ports have their data included in $Dout(k)$, so consider non-border port X. If X has no path to a border port, then its data is irrelevant. Suppose that there is a path from port X to border port Z within tile $\tau(k)$.

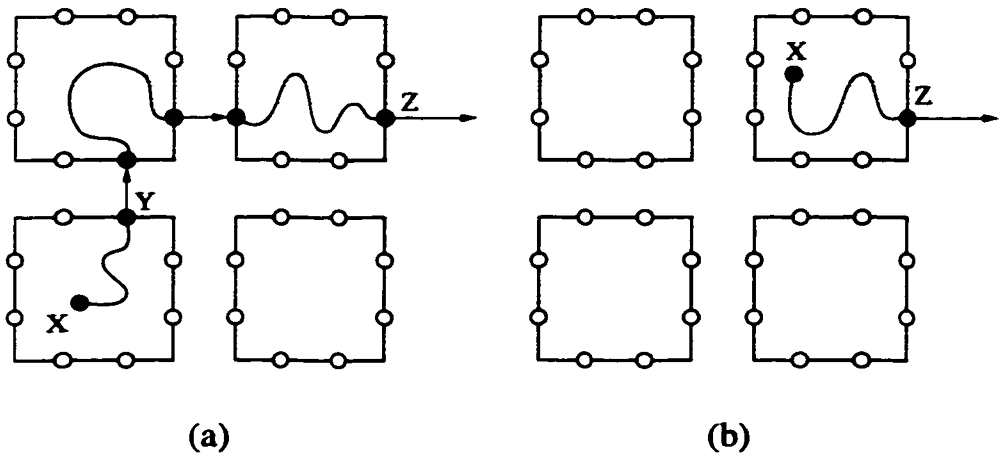


Figure 5.9: Algorithm *Going_Out* ensures that any writing produced by arbitrary port X inside tile $\tau_j(k-1)$ reaches the external output port Z of tile $\tau(k)$ if there is a path between X and Z: a) The path from X to Z crosses two or more sub-tiles; b) The path from X to Z never leaves the sub-tile.

Case 1: X is inside sub-tile $\tau_r(k-1)$, Z is in sub-tile $\tau_s(k-1)$ (r may be equal to s), and the path from X to Z crosses two or more sub-tiles (see Figure 5.9(a)). Since there is a path between X and Z within tile $\tau(k)$, there must be a sub-path contained in $\tau_r(k-1)$ between X and some output port Y of sub-tile $\tau_r(k-1)$. By the induction hypothesis, a value written by X appears in Y and also in $Dout_r(k-1)$. Since there is a path from Y to Z, then the transitive closure matrix, $A^*(k)$, has a '1' entry in the intersection of row Y and column Z. Procedure *Find_Dout*, using the information provided by $A^*(k)$, propagates data (from the set $Dout_j(k-1)$) from horizontal buses that represent output ports of sub-tiles $\tau_j(k-1)$ to vertical buses that represent external output ports of tile $\tau(k)$. Top row processors read the vertical buses and store the bus data that represents the set $Dout(k)$. Any write that appears at port Y also appears at port Z and, hence, in the set $Dout(k)$.

Case 2: X, Z, and the path from X to Z are in the same sub-tile (Figure 5.9(b)). By the induction hypothesis, a value written by X appears in $Dout_j(k-1)$ at port Z for some j, where $1 \leq j \leq 4$. The transitive closure matrix, $A^*(k)$, has a '1' entry in the intersection of row Z and column Z. Using the same reasoning as in the Case 1, procedure *Find_Dout* propagates the data that leaves Z from the set $Dout_j(k-1)$ to the set $Dout(k)$. ■

Lemma 5.2 *The set of bus data, $Din(i)$, generated by Algorithm Going_In, for any $1 \leq i \leq \log N$, represents the final bus data that arrive at the external input ports of tile $\tau(i)$.*

Proof: We use induction on the level of recursion i. Our induction hypothesis is that for any $1 \leq i \leq \log N$, the values in $Din(i)$ that appear at the external input ports of tile $\tau(i)$ contain the effects of all writes inside and outside the tile.

The basis of the induction is when $i = \log N$; for this case, the bus data at each external input port of tile $\tau(\log N)$ is null because tile $\tau(\log N)$ has no neighboring tiles.

Assume that the induction hypothesis holds for all values of i, where $k \leq i \leq \log N$. We will show that the induction hypothesis also holds for $i = k-1$. Tile $\tau(k)$ comprises four sub-tiles $\tau_j(k-1)$, for each $1 \leq j \leq 4$. The bus data $Din_s(k-1)$ at the external input ports of sub-tile $\tau_s(k-1)$ must have originated by one of the following cases.

Case 1: The bus datum comes from some output port of a neighboring tile $\tau(k)'$ and arrives at an external input port, A, of tile $\tau(k)$ (see Figure 5.10(a)). Port A is also an external input port of tile $\tau_r(k-1)$. Assume that there exists a path in $\tau(k)$ from port A to some input port C in sub-tile $\tau_s(k-1)$ (see Figure 5.10(a)). By the induction hypothesis, the bus datum at port A appears on $Din(k)$. Since there is a path between port A and port C, then there is a '1' entry in the intersection of the horizontal bus that represents port A and the vertical bus that represents port C in the transitive closure matrix $A^*(k)$. Procedure *Find_Din* of algorithm *Going_In* propagates the bus datum at port A (provided by $Din(k)$) through the horizontal bus that represents port A, then through the vertical bus that represents port C, and stores the bus datum in $Din_s(k-1)$.

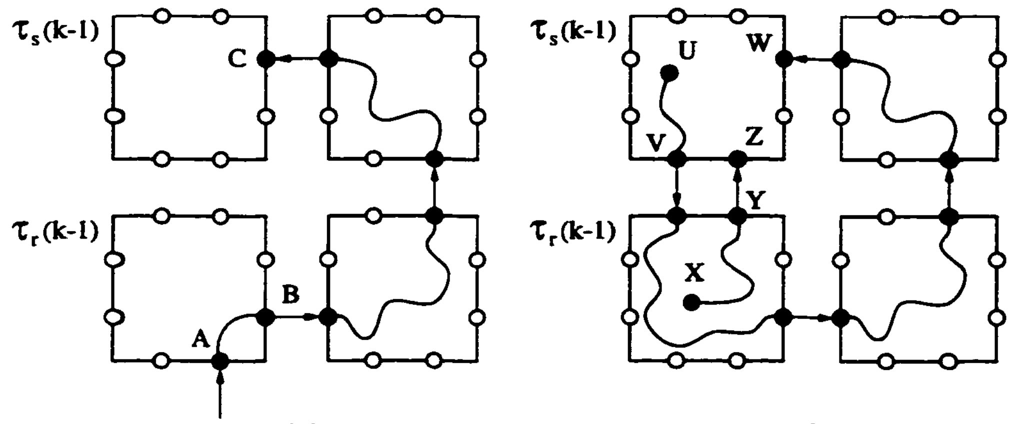


Figure 5.10: Algorithm *Going_In* calculates the final bus data at input ports of each tile $\tau_j(k-1)$: a) A bus datum generated outside tile $\tau(k)$ arrives at a border port; b) Bus data generated inside tile $\tau(k)$.

Case 2: The bus datum comes from some output port inside some sub-tile $\tau_r(k-1)$, where $s \neq r$. An output port X inside (not necessarily on the border) of sub-tile $\tau_r(k-1)$ writes a bus datum to its bus. Assume that there is a path in $\tau(k)$ from port X to some input port Z in sub-tile $\tau_s(k-1)$ (see Figure 5.10(b)). By Lemma 5.1, this bus datum propagates to some output port Y at the border of sub-tile $\tau_r(k-1)$ and shows up in $Dout_r(k-1)$. Since there is a path between Y and Z, using the same argument as in Case 1, procedure *Find_Din* ensures that the bus datum written by X shows up in $Din_s(k-1)$.

Case 3: The bus datum comes from some output port inside sub-tile $\tau_s(k-1)$. An output port U inside (not necessarily on the border) of sub-tile $\tau_s(k-1)$ writes a bus datum to its bus. Assume that there is a path that starts at port U, leaves sub-tile $\tau_s(k-1)$ at port V, and returns to sub-tile $\tau_s(k-1)$ at port W (see Figure 5.10(b)). By Lemma 5.1, the bus datum written by U propagates to output port V at the border of sub-tile $\tau_s(k-1)$ and shows up in $Dout_s(k-1)$. Since there is a path between V and W, using the same argument as in Case 1, procedure *Find_Din* ensures that the bus datum written by U shows up in $Din_s(k-1)$.

So, algorithm *Going_In* correctly calculates the final bus data at the input port of each tile $\tau(0)$. ■

Theorem 5.3 *Any step of an $N \times N$* Common, Collision, *or* Collision$^+$ *CRCW DR-Mesh can be simulated on an $O\left(N \times N \times \frac{N}{\log N}\right)$* Common, Collision, *or* Collision$^+$ *CRCW LR-Mesh in $O\left(\log^2 N\right)$ time.* ■

5.7 Simulation Improvements

This section presents some improvements to the DR-Mesh simulation.

Two-dimensional LR-Mesh. Vaidyanathan and Trahan [52] designed a procedure to transform an $A \times B \times C$ three-dimensional LR-Mesh into a $6BC \times (7A + AB)$ two-dimensional LR-Mesh. We use this procedure to transform the simulating LR-Mesh.

Corollary 5.4 *Any step of an $N \times N$* Common, Collision, *or* Collision$^+$ *CRCW DR-Mesh can be simulated on an $O\left(\frac{N^2}{\log N} \times \frac{N^2}{\log N}\right)$* Common, Collision, *or* Collision$^+$ *CRCW LR-Mesh in $O\left(\log^2 N\right)$ time.* ■

Exclusive Writes. The simulated DR-Mesh and the simulating LR-Mesh in the present simulation assume the Common, Collision, or Collision$^+$ rules. In Chapter 4, we proved that a CREW LR-Mesh that uses only acyclic buses can simulate a Common, Collision, or Collision$^+$ LR-Mesh in constant time (Theorem 4.1). In the present simulation, the simulating LR-Mesh uses only acyclic buses, so we can replace it by one that uses only exclusive writes without altering the execution time of the simulation.

Corollary 5.5 *Any step of an $N \times N$* Common, Collision, *or* Collision$^+$ *CRCW DR-Mesh can be simulated on an $O\left(\frac{N^2}{\log N} \times \frac{N^2}{\log N}\right)$ CREW LR-Mesh in $O\left(\log^2 N\right)$ time.* ■

DR-Mesh on PR-Mesh. In Chapter 4, we proved that an $O(N \times N)$ PR-Mesh can simulate in constant time an $N \times N$ CREW LR-Mesh that uses only acyclic buses (Corollary 4.6). In the simulation presented in this chapter, we use a three-dimensional LR-Mesh to speed up the execution of certain operations, like data movement and Boolean matrix multiplication. A PR-Mesh performs these types of operations more efficiently because of its ability to transmit multiple messages on a single bus by using pipeline. Combining Theorem 5.3 and Corollary 5.5 with PR-Mesh data movement abilities, we obtain the following.

Corollary 5.6 *Any step of an $N \times N$* Common, Collision, *or* Collision$^+$ *CRCW DR-Mesh can be simulated on an $O\left(N \times \frac{N^2}{\log N}\right)$ PR-Mesh in $O\left(\log^2 N\right)$ time.* ■

Notice how the PR-Mesh uses $O\left(\frac{N^3}{\log N}\right)$ processors in two dimensions to perform the simulation. The LR-Mesh uses the same number of processors, but in three dimensions.

Chapter 6

Summary and Future Work

The general aim of this research is to provide a better understanding about scaling simulations of reconfigurable models, specifically of the R-Mesh and some of its variants. Many of the techniques in this dissertation can be used for other reconfigurable models as well.

In Chapter 3, we have identified a new R-Mesh restriction, called the FR-Mesh. For this model, we have designed a strong scaling simulation with overhead $\log P$ for a $P \times P$ simulating machine. By integrating the strong scaling simulation of the FR-Mesh and the optimal scaling simulation of the LR-Mesh, we have identified a new class of algorithms (called separable algorithms) with strong scaling simulations that accommodate solutions to a wide range of problems.

We have also studied the effect of different concurrent write rules for the simulated and simulating models, such as COMMON, COLLISION, COLLISION^{+}, ARBITRARY, and PRIORITY, and have established that the simulation overhead for the FR-Mesh is due only to leader election.

The FR-Mesh scaling simulation also leads to an improved (weak) scaling simulation for the R-Mesh. Its simulation overhead is $O\left(\log P \log \frac{N}{P}\right)$ (the simulation overhead of the previous fastest scaling simulation for the R-Mesh was $O\left(\log N \log \frac{N}{P}\right)$).

In this scaling simulation, a part of the simulation overhead is due to leader election, as in the FR-Mesh simulation. Thus, any improvement in techniques to perform leader election will immediately translate to a further reduction of the overheads for scaling simulations of the FR-Mesh, the R-Mesh, and separable R-Mesh algorithms.

In Chapter 4, we have presented bus linearization, a procedure that transforms non-linear buses into acyclic linear buses. Using bus linearization, we simulate a CRCW R-Mesh (for various write rules) on a CREW LR-Mesh. This procedure gives an algorithm designer the liberty of using buses of arbitrary shape, while automatically translating the algorithm to run on a simpler platform.

We have presented two important applications for bus linearization. The first is a further improvement in the simulation overhead for the R-Mesh. This overhead of $O(\log N)$ is even smaller than the one presented in Chapter 3. Moreover, the simulating model in this scaling simulation is an LR-Mesh, a model weaker than the R-Mesh. Furthermore, the LR-Mesh uses only exclusive writes, while in all previous simulations the simulating machine always used concurrent writes. The second application of bus linearization transforms R-Mesh algorithms to run on reconfigurable models with pipelined optical buses such as the PR-Mesh, APPBS, and AROB.

In Chapter 5, we have presented a simulation of a CRCW DR-Mesh on a CREW LR-Mesh. This $O\left(\log^2 N\right)$-time simulation, that uses $O\left(N \times N \times \frac{N}{\log N}\right)$ processors in three dimensions (or $O\left(\frac{N^2}{\log N} \times \frac{N^2}{\log N}\right)$ processors in two dimensions), is big improvement over the previous best simulation that uses $O(N^8)$ processors to run in $O(\log N)$ time. Our simulation also runs in $O\left(\log^2 N\right)$ time on models with pipelined optical buses using only $O\left(N^2 \times \frac{N}{\log N}\right)$ processors in two dimensions.

Future Work: Open problems include investigating whether or not the R-Mesh and the FR-Mesh have optimal scaling simulation. So far, all approaches have resulted in weak scaling simulations for the R-Mesh and in a strong scaling simulation for the FR-Mesh.

Another open problem is to improve the time complexity for leader election on Common, Collision, or Collision$^+$ CRCW R-Mesh or in CREW R-Mesh. One possible way to accomplish this goal is by using randomization.

Another research direction is to improve the overhead of specific algorithms. For example, designing R-Mesh algorithms that use a limited number of linear connections at each step. An algorithm such as this will scale down with low overhead, since the overhead on the scaling simulation of Section 4.3.1 depends on the number of non-linear connections.

An interesting problem is to identify new variants of the R-Mesh that admit scaling simulations with low overhead. A simpler scaling simulation for the LR-Mesh will also be beneficial, since many of the results in this dissertation are based on the LR-Mesh scaling simulation.

Exploiting knowledge of the structure of a particular algorithm for scaling purposes, is another research direction. For example, Jang and Prasanna [22] have proved that for any $1 \leq T \leq \sqrt{N}$, an $\frac{N}{T} \times \frac{N}{T}$ R-Mesh can sort N elements in $O(T)$ time. Similar results may be found in [46].

Bibliography

[1] H. M. Alnuweiri, "Constant-Time Algorithms for Image Labeling on a Reconfigurable Network of Processors," *IEEE Trans. Parallel Distrib. Systems*, vol. 5, no. 3, (1994), pp. 320–326.

[2] H. M. Alnuweiri, "Parallel Constant-Time Connectivity Algorithms on a Reconfigurable Network of Processors," *IEEE Trans. Parallel Distrib. Systems*, vol. 6, no. 1, (June 1995), pp. 105–110.

[3] Y. Ben-Asher, D. Gordon, and A. Schuster, "Optimal Simulations in Reconfigurable Arrays," Technion Israel Institute of Technology, Technical Report 716, (Feb. 1992).

[4] Y. Ben-Asher, D. Gordon, and A. Schuster, "Efficient Self Simulation Algorithms for Reconfigurable Arrays," *J. Parallel Distrib. Comput.*, vol. 30, no. 1, (1995), pp. 1–22.

[5] Y. Ben-Asher, K.-J. Lange, D. Peleg, and A. Schuster, "The Complexity of Reconfiguring Network Models," *Info. and Comput.*, vol. 121, no. 1, (1995), pp. 41–58.

[6] Y. Ben-Asher, D. Peleg, R. Ramaswami, and A. Schuster, "The Power of Reconfiguration," *J. Parallel Distrib. Comput.*, vol. 13, (1991), pp. 139–153.

[7] A. G. Bourgeois and J. L. Trahan, "Relating Two-Dimensional Reconfigurable Meshes with Optically Pipelined Buses," manuscript, (1999).

[8] J. Bruck, L. De Coster, N. Dewulf, C.-T. Ho, and R. Lauwereins, "On the Design and Implementation of Broadcast and Global Combine Operations Using the Postal Model," *IEEE Trans. Parallel Distrib. Systems*, vol. 7, no. 3, (Mar. 1996), pp. 256–265.

[9] G. -H. Chen, B. -F. Wang, and C. -J. Lu, "On the Parallel Computation of the Algebraic Path Problem," *IEEE Trans. Parallel Distrib. Systems*, vol. 3, (1992), pp. 251–256.

[10] K.-L. Chung, "Image Template Matching on Reconfigurable Meshes," *Parallel Proc. Letters*, vol. 6, no. 3, (1996), pp. 345–353.

[11] H. ElGindy and L. Wetherall, "A Simple Voronoi Diagram Algorithm for a Reconfigurable Mesh," *IEEE Trans. Parallel Distrib. Systems*, vol. 8, (1997), pp. 1133–1142.

[12] J. A. Fernández-Zepeda, J. L. Trahan, and R. Vaidyanathan, "Scaling the FR-Mesh under Different Concurrent Write Rules," *Proc. World Multiconference on Systemics, Cybernetics and Informatics*, (1997), pp. 437–444.

[13] J. A. Fernández-Zepeda, R. Vaidyanathan, and J. L. Trahan, "Scalability of the Fusing-Restricted Reconfigurable Mesh," *Proc. 8th IASTED Int'l. Conf. Par. Distrib. Comput. and Sys.*, (1996), pp. 467–471.

[14] J. A. Fernández-Zepeda, R. Vaidyanathan, and J. L. Trahan, "Scaling Simulation of the Fusing-Restricted Reconfigurable Mesh," *IEEE Trans. Parallel Distrib. Systems*, vol. 9, no. 9, (Sep. 1998), pp. 861–871.

[15] J. A. Fernández-Zepeda, R. Vaidyanathan, and J. L. Trahan, "Improved Scalability Simulations of the General Reconfigurable Mesh," *Proc. 6th Reconfigurable Architecture Workshop*. LNCS vol. 1586, April 1999, pp. 616-624.

[16] J. Gunnels, C. Lin, G. Morrow, and R. van de Geijn, "A Flexible Class of Parallel Matrix Multiplication Algorithms," *Proc. 12th Int'l. Par. Processing Symp. & 9th IEEE Symp. Par. Distrib. Processing*, (1998), pp. 110–116.

[17] Z. Guo, "Optically Interconnected Processor Arrays with Switching Capability," *J. Parallel Distrib. Comput.*, vol. 23, (1994), pp. 314–329.

[18] T. Hayashi, K. Nakano, and S. Olariu, "An $O((\log\log n)^2)$ Time Algorithm to Compute the Convex Hull on Reconfigurable Meshes," *IEEE Trans. Parallel Distrib. Systems*, vol. 9, no. 12, (1998), pp. 1167–1179.

[19] J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley Publishing Co., 1992.

[20] J.-W. Jang, M. Nigam, V. K. Prasanna, and S. Sahni, "Constant Time Algorithms for Computational Geometry on the Reconfigurable Mesh," *IEEE Trans. Parallel Distrib. Systems*, vol. 8, (1997), pp. 1–12.

[21] J.-w. Jang and V. K. Prasanna, "An Optimal Multiplication Algorithm on Reconfigurable Mesh," *IEEE Trans. Parallel Distrib. Systems*, vol. 8, no. 5, (1997), pp. 521–532.

[22] J.-w. Jang and V. K. Prasanna, "An Optimal Sorting Algorithm on Reconfigurable Mesh," *J. Parallel Distrib. Comput.*, vol. 25, no. 1, (1995), pp. 31–41.

[23] T.-W. Kao, S.-J. Horng, and Y.-L. Wang, "An O(1) Time Algorithms for Computing Histogram and Hough Transform on a Cross-bridge Reconfigurable Array of Processors," *IEEE Trans. System, Man and Cybernetics*, vol. 25, no. 4, (1995), pp. 681–687.

[24] R. M. Karp and V. Ramachandran, "Parallel Algorithms for Shared-Memory Machines," in *Handbook of Theoretical Computer Science, vol. A: Algorithms and Complexity*, J. van Leeuwen, ed., MIT Press, (1990), pp. 869–941.

[25] T. H. Lai and M.-J. Sheng, "Constructing Euclidean Minimum Spanning Trees and All Nearest Neighbors on Reconfigurable Meshes," *IEEE Trans. Parallel Distrib. Systems*, vol. 7, no. 8, (Aug. 1996), pp. 806–817.

[26] H. Li and Q. F. Stout, "Reconfigurable SIMD Massively Parallel Computers," *IEEE Proceedings*, vol. 79, no. 4, (Apr. 1991), pp. 429–443.

[27] K. Li, Y. Pan, and S. Q. Zheng, *Parallel Computing Using Optical Interconnections*, Kluwer Academic Publishers, Boston, MA, 1998.

[28] R. Lin and S. Olariu, "Reconfigurable Buses with Shift Switching: Concepts and Applications," *IEEE Trans. Parallel Distrib. Systems*, vol. 6, no. 1, (Jan. 1995), pp. 93–102.

[29] R. Lin, S. Olariu, J. L. Schwing, and J. Zhang, "Sorting in $O(1)$ Time on an $n \times n$ Reconfigurable Mesh," *Proc. Plenary Address Proc. EWPC*, (1992), pp. 16–27.

[30] M. Maresca, "Polymorphic Processor Arrays," *IEEE Trans. Parallel Distrib. Systems*, vol. 4, no. 5, (May 1993), pp. 490–506.

[31] M. Maresca and P. Baglietto, "Transitve Closure and Graph Component Labeling on Realistic Processor Arrays Based on Reconfigurable Mesh Network," *Proc. IEEE Int'l. Conf. on Comp. Design: VLSI in Comp. and Proc.*, (1991), pp. 229–232.

[32] Y. Matias and A. Schuster, "Fast, Efficient Mutual and Self Simulations for Shared Memory and Reconfigurable Mesh," *Parallel Algorithms and Architectures*, vol. 8, (1996), pp. 195–221.

[33] R. Miller, V. K. Prasanna-Kumar, D. Reisis, and Q. Stout, "Parallel Computations on Reconfigurable Meshes," *IEEE Trans. Comput.*, vol. 42, no. 6, (June 1993), pp. 678–692.

[34] M. M. Murshed and R. P. Brent, "Algorithms for Optimal Self-Simulation of Some Restricted Reconfigurable Meshes," *Proc. 2nd Int'l. Conf. Computational Intelligence and Multimedia Applic.*, (1998), pp. 734–744.

[35] K. Nakano, "Efficient Summing Algorithms for a Reconfigurable Mesh," *Proc. IPPS 94 Workshop on Reconfigurable Architectures.*

[36] K. Nakano, "Prefix-Sums Algorithms on Reconfigurable Meshes," *Parallel Proc. Letters*, vol. 5, no. 1, (1995), pp. 23–35.

[37] K. Nakano, "A Bibliography of Published Papers on Dynamically Reconfigurable Architectures," *Parallel Proc. Letters*, vol. 5, no. 1, (Mar. 1995), pp. 111–124.

[38] M. Nigam and S. Sahni, "Sorting n Numbers on $n \times n$ Reconfigurable Meshes with Buses," *J. Parallel Distrib. Comput.*, vol. 23, (1994), pp. 37–48.

[39] N. Nisan and A. Ta-Shma, "Symmetric Logspace is Closed Under Complement," *Proc. 27th ACM Symp. Theory of Computing*, (1995), pp. 140–146.

[40] Y. Pan and K. Li, "Linear Array with a Reconfigurable Pipelined Bus System: Concepts and Applications," *Informations Sciences – An International Journal*, vol. 106, (1998), pp. 237–258.

[41] S. Pavel and S. G. Akl, "On the Power of Arrays with Optical Pipelined Buses," *Proc. Int'l. Conf. Par. Distr. Proc. Techniques and Appl.*, (1996), pp. 1443–1454.

[42] S. Sahni, "Computing on Reconfigurable Bus Architectures," in *Computer Systems & Education*, Balakrishnan *et al.* (Eds.), Tata McGraw-Hill Publishing Co., New Delhi, 1994, pp. 386–398.

[43] Y. Shiloach and U. Vishkin, "An $O(\log N)$ Parallel Connectivity Algorithm," *Journal of Algorithms*, vol. 3, (1982), pp. 57–67.

[44] J. L. Trahan, A. G. Bourgeois, Y. Pan, and R. Vaidyanathan, "Optimally Scaling Permutation Routing on Reconfigurable Arrays with Optically Pipelined Buses," *Proc. 13th Int'l. Par. Process. Symp. & 10th Symp. Par. Distr. Process.*, (1999), pp. 233–237.

[45] J. L. Trahan, A. G. Bourgeois, and R. Vaidyanathan, "Tighter and Broader Complexity Results for Reconfigurable Models," *Parallel Proc. Letters*, vol. 8, (1998), pp. 271–282.

[46] J. L. Trahan, C-m. Lu, and R. Vaidyanathan, "Scalable Reconfigurable Mesh Algorithms for Matrix Operations with Integer and Floating Point Inputs," manuscript, 1998.

[47] J. L. Trahan, Y. Pan, R. Vaidyanathan, and A. G. Bourgeois, "Scalable Basic Algorithms on a Linear Array with a Reconfigurable Pipelined Bus System," *Proc. 10th ISCA Int'l. Conf. Par. Distr. Comput. Sys.*, (1997), pp. 564–569.

[48] J. L. Trahan, R. Vaidyanathan, and A. G. Bourgeois, "LRN Simulation of RN and RN Simulation of DRN," manuscript, June 1997.

[49] J. L. Trahan, R. Vaidyanathan, and C.P. Subbaraman, "Constant Time Graph Algorithms on the Reconfigurable Multiple Bus Machine," *J. Parallel Distrib. Comput.*, vol. 46, (1997), pp. 1–14.

[50] J. L. Trahan, R. Vaidyanathan, and R. K. Thiruchelvan, "On the Power of Segmenting and Fusing Buses," *J. Parallel Distrib. Comput.*, vol. 34, no. 1, (Apr. 1996), pp. 82–94.

[51] J. L. Trahan and R. Vaidyanathan, "Relative Scalability of the Reconfigurable Multiple Bus Machine," *Proc. Workshop Reconfigurable Arch. and Algs.*, 1996.

[52] R. Vaidyanathan and J. L. Trahan, "Optimal Simulation of Multidimensional Reconfigurable Meshes by Two-dimensional Reconfigurable Meshes," *Information Processing Letters*, vol. 47, (no. 5, Oct. 1993), pp. 267–273.

[53] U. Vishkin, "Structural Parallel Algorithmics," *Proc. Int'l. Colloq. Automata, Languages and Programming*, (1991), pp. 363–380.

[54] B. F. Wang and G. H. Chen, "Constant Time Algorithms for the Transitive Closure and Some Related Graph Problems on Processor Arrays with Reconfigurable Bus Systems," *IEEE Trans. Parallel Distrib. Systems*, vol. 1, no. 4, (Oct. 1990), pp. 500–507.

Vita

José Alberto Fernández Zepeda was born in Mexico City on June 26, 1966. He received the degree of Ingeniero Mecánico Electricista and the degree of Maestro en Ingeniería Eléctrica from the Universidad Nacional Autónoma de México in 1991 and 1994, respectively. He is currently a doctoral student in Computer Engineering and a teaching assistant in the Department of Electrical and Computer Engineering at Louisiana State University. His current research interests include reconfigurable based-bus architectures and design and analysis of parallel algorithms. He will receive the degree of Doctor of Philosophy in December, 1999.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Jose Alberto FERNANDEZ ZEPEDA

Major Field: Electrical Engineering

Title of Dissertation: Scaling Simulations of Reconfigurable Meshes

Approved:

[illegible]
Major Professor and Chairman

[illegible]
Dean of the Graduate School

EXAMINING COMMITTEE:

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

Date of Examination:

October 12, 1999