Fast, Efficient Mutual and Self Simulations for Shared Memory and Reconfigurable Mesh

Yossi Matias
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill NJ 07974
matias@research.att.com

Assaf Schuster
Dept. of Computer Science
Technion, Technion City
Haifa, ISRAEL 32000
assaf@cs.technion.ac.il

Abstract

This paper studies relations between the parallel random access machine (PRAM) model, and the reconfigurable mesh (RMESH) model, by providing mutual simulations between the models. We present an algorithm simulating one step of an $(n \lg\lg n)$-processor CRCW PRAM on an $n \times n$ RMESH with delay $O(\lg\lg n)$ with high probability. We use our PRAM simulation to obtain the first efficient self-simulation algorithm of an RMESH with general switches: An algorithm running on an $n \times n$ RMESH is simulated on a $p \times p$ RMESH with delay $O((n/p)^2 + \lg n \lg\lg p)$ with high probability, which is optimal for all $p \leq n/\sqrt{\lg n \lg\lg n}$. Finally, we consider the simulation of RMESH on the PRAM. We show that a $2 \times n$ RMESH can be optimally simulated on a CRCW PRAM in $\Theta(\alpha(n))$ time, where $\alpha(\cdot)$ is the slow-growing inverse Ackermann function. In contrast, a PRAM with polynomial number of processors cannot simulate the $3 \times n$ RMESH in less than $\Omega(\lg n / \lg\lg n)$ expected time.

1 Introduction

The parallel random access machine (PRAM) model of computation is the most widely used model for the design and analysis of parallel algorithms (see, e.g. [27, 23, 43]). The PRAM model consists of a number of processors operating in lock-step and communicating by reading and writing locations in a shared memory.

A reconfigurable mesh, or RMESH, is composed of a partitionable bus in the shape of a mesh, connecting a set of processors which operate synchronously, and a set of memory modules. Thus, an n by n RMESH, or n-RMESH, consists of n processors, n memory modules, and an interconnection network consisting of an n by n mesh whose nodes are switches; each switch can reconfigure dynamically to connect arbitrary subsets of its adjacent edges. As a result of local switching, the global bus is partitioned into subbuses. In this way a large number of network configurations are possible, each supporting a certain communication pattern. The RMESH may use dynamic switching between these configuration patterns in order to accelerate computation.

The RMESH has begun to attract many researchers due to its promising prospects for efficient hardware implementation [30]. Indeed, the model captures features of many commercial and experimental machines, which offer flexible communication patterns. The increasing interest in the RMESH has resulted in a rapidly expanding volume of algorithmic and theoretical results; see for example [40, 4, 3, 5]. It has been shown that the RMESH can solve some fundamental problems very fast. In fact, it can solve some of them faster than what is possible with the CRCW PRAM. For instance, an n-RMESH can find the parity of n^2 bits in constant time [29], which is significantly faster than the $\Theta(\lg n/\lg\lg n)$ time required to solve the problem on a CRCW PRAM when using n^c processors, for any constant $c > 0$ [2].

1.1 Simulating PRAM on RMESH

We address the following question: can *any* PRAM algorithm be adapted to an efficient algorithm on RMESH with a small slowdown? A related question is whether one step of a PRAM can be simulated effectively on an RMESH with a small delay. A simple bi-section bandwidth argument implies that simulating an n-processor PRAM on a p-RMESH requires delay $T(n,p) = \Omega(n/p)$. Typically, the delay can be expressed as $T(n,p) = f(n,p)\cdot(n/p) + d(n,p)$, for some functions $f = f(n,p)$ and $d = d(n,p)$. We denote f as the (multiplicative) *overhead* of the simulation, and d as the *additive delay* of the simulation. If $f(n,p) = O(1)$ then the simulation is said to be *efficient*. Note that an efficient simulation algorithm is optimal for $p = O(n/d)$. Therefore, our goal is to obtain an efficient simulation of an n-processor CRCW PRAM algorithm on a p-RMESH with the smallest possible additive delay.

A good simulation result has an appeal from the algorithmic point of view. The popularity of the PRAM, which is arguably due to its "programming" simplicity, resulted in a reach literature of fast and efficient algorithms. These algorithms make it an attractive target for simulations on more feasible machines. Indeed, a fast, efficient simulation would translate any efficient PRAM algorithm into an efficient RMESH algorithm which is slower than the PRAM algorithm by a factor of at most the delay of the simulation.

The problem of simulating a PRAM on the RMESH was studied by Wang and Chen [54] and by Ben-Asher *et al.* [4]. However, the simulation algorithms given in these papers are not efficient. In [54] and [4] it is shown how to simulate an n-processor PRAM having M memory cells by the $n \times M$ RMESH, with $O(1)$ delay. For any $p \leq n$ this implies, at best, a simulation on a p-RMESH whose delay is $O((M/n) \cdot (n/p)^2)$, which is $O((M/p) \cdot (n/p))$, implying a multiplicative overhead of $f = O(M/p)$. As M may be considerably larger than p, this simulation is far from being efficient.

The difficulty in simulating shared memory on the RMESH is partly due to the distributed memory nature of the latter. There are studies that concentrate on models for distributed memory. A relatively well studied model is the distributed memory machine (DMM) in which the interconnection network is abstracted away, and assumed to be a fully connected network. Another related model, in which the interconnection network is replaced by optical communication, is the optical communication parallel computer (OCPC). The simulations of the PRAM model on the DMM and on the OCPC models were considered in a sequence of papers [48, 37, 28, 25, 52, 47, 16, 26, 17, 21], obtaining efficient simulations with doubly-logarithmic delay. In this work we extend these simulation results to obtain an efficient simulation of an $(n \lg\lg n)$-processor PRAM on an n-RMESH with an additive delay of $O(\lg\lg n)$ with high probability. This is the first efficient simulation of a PRAM on an RMESH.

As a direct application of the simulation we conclude that any algorithm for the n-processor PRAM can be implemented on a p-RMESH, for $p = n/\lg\lg n$, with $O(\lg\lg n)$ slowdown with high probability. The efficiency of the simulation implies that many PRAM algorithms (e.g. [27, 23, 43]) can be automatically adapted to efficient RMESH algorithms. The rather small delay implies a small slowdown; in particular, fast, efficient PRAM algorithms (e.g. [34]) imply fast, efficient RMESH algorithms. We illustrate the usefulness of this rather general result by deriving a fast, efficient implementation for the fundamental problem of building and supporting the parallel dictionary data structure on the RMESH.

1.2 Self-simulation on RMESH

One of the most important features of parallel models is the *self-simulation* property. When a machine is self-simulating, a smaller machine of size p can simulate a step of a larger one of size N in $O(N/p)$ steps. This makes the programming task considerably easier: it implies that an algorithm designer may conveniently assume that the machine at hand is as large as required by the algorithm. On the other hand, without self-simulation algorithm, an algorithm designer must specifically address the entire range of machine sizes, since an algorithm given for a particular machine size does not uniformly scale to all range of sizes, in general. Suppose that a program takes τ steps on a large machine of size N. When the program is executed on a smaller machine of size p, the compiler may use the self-simulation property to produce an efficient algorithm for the smaller machine, which works in $O(\tau N/p)$ steps.

Consider the simulation of an N-RMESH on a p-RMESH, and consider a step in which each switch of the N-RMESH (except for the rightmost column) sends some arbitrary message to the right while reading its left port, and disconnecting its left, up and down ports. There are approximately N^2 message transmissions, hence in simulating this (and other) simple communication pattern on the p-RMESH the delay must be $\Omega((N/p)^2)$. Therefore, for a self-simulation algorithm that takes time $T(N,p) = f(N,p)\cdot(N/p)^2 + d(N,p)$, we denote f as the (multiplicative) *overhead* of the simulation, and d as the *additive delay* of the simulation. As for the PRAM simulation, the self-simulation is said to be *efficient* if $f(N,p) = O(1)$, and we note that an efficient self-simulation algorithm is optimal for $p = O(N/\sqrt{d})$. Therefore, our goal is to obtain an efficient self-simulation of an N-RMESH on a p-RMESH with the smallest possible additive delay.

The self-simulation problem of larger two-dimensional arrays by smaller ones was addressed in [3]. Optimal self-simulation was obtained for two variants of the RMESH: the one with the simplest switches assumes a switch may only connect/disconnect the column ports (the upper port with the lower port) and the row ports (the left port with the right port). The buses are thus either horizontal or vertical, hence this type is called HV-RMESH. Another variant allows to connect/disconnect only pairs of ports (but unlike the HV-RMESH, any pair of nodes). Thus the buses configured are paths of edges, and this type is called the Linear-RMESH. (The Linear-RMESH is the model assumed for the PRAM simulation.) It was shown in [3] that both the HV-RMESH and the Linear-RMESH exhibit optimal self-simulation.

The most general and strongest variant of the RMESH which we consider, denoted as the General-RMESH, allows the switches to connect/disconnect any subset of their ports (rather than just pairs of them). Hence a bus may consist of any connected sub-graph. In [5, 45] it was shown that the set of problems computable in constant time on a polynomial size Linear-RMESH is exactly the set of problems computable by a logspace Turing machine, whereas the corresponding set for the General-RMESH contains exactly all the problems that are computable by a logspace Turing machine having a symmetric logspace oracle. Thus the General-RMESH is expected to be more powerful than the Linear-RMESH. We note that there is a fairly simple constant time connected components algorithm on the General-RMESH [53], while no such equivalent algorithm is known for the polynomial size Linear-RMESH.

Previous to this work, only a non-efficient self-simulation was known for the General-RMESH: an N-RMESH was simulated by a p-RMESH in $O((N/p)^2 \lg N \lg(N/p))$ slowdown [54, 3], which implies an overhead of $f = O(\lg N \lg(N/p))$.

By using our PRAM simulation, we obtain here the first efficient self-simulation algorithm for the General-RMESH. The self-simulation is efficient and has an additive delay of $d = O(\lg N \lg\lg p)$ w.h.p. for the General-RMESH when the simulating RMESH has the ARBITRARY bus-conflict resolution. For a simulating RMESH with the COLLISION bus-conflict resolution, the self-simulation has an overhead of $f = O(\lg p)$ and the same additive delay. Furthermore, our simulating algorithm is implemented on the Linear-RMESH. Thus, the self-simulation can also be interpreted as a simulation of the strong General-RMESH on the weaker Linear-RMESH.

For completeness we mention that some work was previously carried in the direction of simulating general networks using a *larger* RMESH. It was shown that any constant-degree reconfiguring network may be simulated with no slowdown by an RMESH, paying a quadratic blow-up of the number of switches [5, 45]. This result was improved for the case of a k-dimensional RMESH which is simulated by the $O(n^{k-1})$-RMESH with constant slowdown [50].

1.3 Simulating a $2 \times n$ RMESH on the PRAM

In the opposite direction, we consider the simulation of a $2 \times n$ RMESH on the PRAM. We show that using such a network, only a slight (nearly-constant) speedup can be obtained compared to the CRCW PRAM. Specifically, we show that a step of a $2 \times n$ RMESH can be simulated on CRCW PRAM in $O(\alpha(n))$ time, using an optimal (i.e., $n/\alpha(n)$) number of processors, where $\alpha(n)$ is the inverse Ackermann function (this is an extremely slow-growing function and is at most 4 for all conceivable n). We also show that this work-optimal simulation result is best possible: $\Omega(\alpha(n))$ time is required when simulating a $2 \times n$ RMESH, or even a $1 \times n$ RMESH, on an n-processor CRCW PRAM. Constant time simulation can also be obtained by using $nI_i(n)$ processors, for any constant $i > 0$, where $I_i(n)$ is the inverse of a function at the i-th level of the primitive recursive hierarchy (e.g., $I_3(n) = \lg^* n$).

The tight simulation result implies that any $\omega(\alpha(n))$ lower bound for the CRCW PRAM translates into a non-trivial lower bound for the $2 \times n$ RMESH, which is similar to the CRCW PRAM lower bound up to an $\alpha(n)$ time factor, or up to a constant time factor if slight increase in the number of PRAM processors does not affect the PRAM time lower bound. As an immediate corollary, the CRCW PRAM lower bound of Beame and Hastad [2] implies that the *parity* problem cannot be solved faster than $\Omega(\lg n/\lg\lg n)$ time on a $2 \times n^k$ RMESH, for any constant k; this particular lower bound has been improved to $\Omega(\lg n)$ by the independent work of MacKenzie [33] (but only for the $2 \times n$ RMESH). The same lower bound obviously holds also for the more general *compaction* problem, which is the problem of moving the contents of m distinguished elements, given in an array of size n, into an array B of size m. New lower bounds are also derived for a relaxed version of the problem, the *linear compaction* problem, in which the output array B may be of size $O(m)$: $\Omega(\lg^* n/\alpha(n))$ expected time for a randomized algorithm, and $\Omega(\lg\lg n/\alpha(n))$ time for a deterministic algorithm, both for the $2 \times n$ RMESH; the first is derived by the CRCW PRAM lower bound of MacKenzie [32], and the second by the CRCW PRAM lower bound of Chaudhuri [8]. Other new lower bounds for the

$2 \times n$ RMESH include an $\Omega(\lg \lg n)$ time lower bound for merging two sequences of length n, both for deterministic algorithms [44] and for randomized algorithms [19], and an $\Omega(\lg \lg n)$ time lower bound for computing the minimum [51, 38, 6].

The simulation of a $2 \times n$ RMESH by a PRAM is obtained by showing that this problem is equivalent, up to a constant factor, to the so-called *nearest-zeros* problem. We then use known results on the complexity of computing the nearest-zeros problem on the CRCW PRAM. Our technique is related to those used in the independent works of Condon *et al.* [10] and Lin *et al.* [31]. These works present simulations for considerably weaker RMESH model, namely the HV-RMESH, which only allows a limited subset of the switch configurations allowed in our model: only buses that are either totally horizontal or totally vertical are possible. As noted above, the power of restricted models such as the HV-RMESH is quite limited. For instance, while there exists a constant time algorithm for the *parity* problem on n bits, which runs on a $3 \times n$ RMESH; no such algorithm is possible even on the $n \times n$ HV-RMESH.

The rest of the paper is organized as follows. The next section provides models definitions and other preliminaries. The simulation of the DMM and PRAM models on the RMESH are given in Section 3. The self-simulation algorithm for the RMESH is given in Section 4. Section 5 describes the simulation of the $2 \times n$ RMESH on the PRAM. Conclusions are given in Section 6.

The results of Section 3 were described in in [46]. The results of Section 5 were described in preliminary form in [35].

2 Preliminaries

2.1 Models definitions

The PRAM model: A Parallel Random Access Machine (PRAM) consists of synchronous processors that have constant-time access into a shared memory of arbitrary size. Standard PRAM models can be distinguished by their rules regarding concurrent access to shared memory locations. These rules are generally classified into the *exclusive* read/write (ER/EW) rule in which each location can be read or written by at most one processor in each PRAM step, and the *concurrent* read/write (CR/CW) rule in which each location can be read or written by any number of processors in each PRAM step. These two rules can be applied independently to reads and writes; the resulting models are denoted in the literature as the EREW, CREW, ERCW, and CRCW PRAM models. In this paper we consider the Concurrent Read Concurrent Write (CRCW) PRAM model in which an arbitrary number of processors can access the same memory cell at the same time in both read and write steps. There are several sub-models that differ in the conflict resolution rule in case two or more processors are trying to write at the same time to the same memory cell, including: the PRIORITY, in which the processor with the lowest *id* succeeds; the ARBITRARY, in which an arbitrary processor succeeds; and the COLLISION, in which a special collision data symbol is written in the memory cell. The PRIORITY CRCW PRAM is stronger than the ARBITRARY CRCW PRAM in the sense that one step of an n-processor machine of the second model can be simulated in constant time by an n-processor machine of the first model. Similarly, the ARBITRARY CRCW PRAM is stronger than the COLLISION CRCW PRAM. Formal definitions and more details can be found in, e.g., [27, 23, 43]. Whenever it will be important for the discussion, we will state explicitly which of the sub-models we assume.

The DMM model: A Distributed Memory Machine (DMM) consists of n synchronous processors $p_0, p_1, \ldots, p_{n-1}$ that communicate via a distributed memory consisting of n memory modules $M_0, M_1, \ldots, M_{n-1}$. Every module has a communication window, and a collection of memory cells. In each DMM step, a processor may perform some internal computations or request access to a memory cell in one of the memory modules. Out of the set of requests made by processors trying to access a certain module, the module answers only one (arbitrarily), and displays $\langle Request, Answer\rangle$ in its communication window. Every processor with a request for this module that reads the window at this step, can get the answer. From the viewpoint of the processors, a communication window functions like a shared memory cell of a CRCW PRAM. Similarly to the CRCW PRAM, the DMM may have sub-models that defer according to their conflict resolution in case of concurrent write. The sub-models can be identified as for the CRCW PRAM: PRIORITY, ARBITRARY, and COLLISION. Unless noted otherwise in the text, we will assume as default the ARBITRARY DMM.

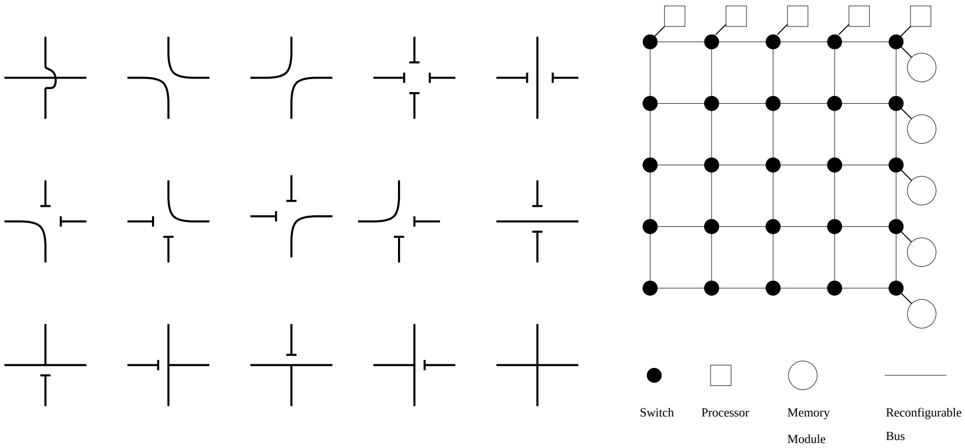

Figure 1: Left: All allowable switch configurations. Right: The 5×5 reconfigurable mesh.

The reconfigurable mesh (RMESH) model: An RMESH consists of an $n \times m$ array of switches, $(0,0)$ at the upper left corner and $(n-1, m-1)$ at the lower right corner. A 5×5 RMESH is depicted in Figure 1. The switches may take one of several configurations, out of which they select locally in each step. This is carried by the following sub-steps. **(1)** The switch selects a *configuration*. Allowable configurations connect subsets of ports (see Figure 1), where a set of connected ports acts as if it is hardwired connected. This induces a partition of the mesh edges into a set of connected components, each of which is called a *bus*. (Each node may be part of up to 4 connected components.) **(2)** Several of the switches connected by a bus may use it to transmit a message. **(3)** All the switches connected by a bus may read it; a message transmitted on a bus is assumed to arrive at the same time step to all its switches. **(4)** A constant time comparison may be taken by every switch.

All the nodes in the mesh have the switching capability, and each node can compare two elements in constant time, store a constant number of elements, and broadcast data on the buses and read data from the buses. There are two distinguished types of nodes: *processors* and *modules*. The processors are the nodes located at the top row of the RMESH (sometimes referred to as $p_0, \ldots, p_{n-1}$). In addition to the switching capabilities, they have computational power corresponding to the power of the PRAM or the DMM processors. The modules are the nodes located at the rightmost column of the RMESH (sometimes referred to as $M_0, \ldots, M_{n-1}$). In addition to the switching capabilities, each module has a collection of memory cells in which it may store data and which it can read in $O(1)$ time. The input is always assumed to be given to the processors. In the sequel we denote the $n \times n$ RMESH as the n-RMESH.

As for the concurrent write PRAM models, when several transmitters attempt to transmit concurrently then some conflict resolution policy should be applied. We say that the bus has three possible states: *speak* – when some message is heard on the bus, *idle* – when no message is transmitted on the bus, and *error* – when several messages are concurrently transmitted on the bus and some "noise" is heard as a result, so that all transmitted information is lost. In this work we are interested in two main conflict resolution policies: ARBITRARY – in which out of several messages that are concurrently transmitted on the bus one of them is arbitrarily elected and heard by the receivers, and COLLISION – in which an error bus-state occurs whenever more than a single message is transmitted on the bus.

Following [5, 3], we consider a hierarchy of RMESH models with respect to the possible connectivity patterns that may be taken by the switches. The model with the simplest switches assumes a switch may only connect/disconnect the column ports (the upper port with the lower port) and the row ports (the left port with the right port). The buses are thus either horizontal or vertical; hence this type is called HV-RMESH. A somewhat stronger model allows to connect/disconnect only pairs of ports. The buses configured are paths of edges, and this type is called the Linear-RMESH. The most general and strongest variant of the RMESH which we consider, the General-RMESH, allows the switches to connect/disconnect any subset of their ports (rather than just pairs of them). Hence a bus may consist of any connected sub-graph.

Discussion Intuitively, the definition of the RMESH given above may be viewed as follows: There are n processors and n memory modules which are interconnected by an $n \times n$ reconfigurable mesh. The RMESH serves merely as a routing device, whose local switches have a very limited processing power. We emphasize here that all results in this paper hold for this model. In particular, this indicates that for the DMM and PRAM simulation results in Section 3 the amount of simulating hardware is linear, except for the routing device, which (in accordance with AT^2 lower-bounds) requires a quadratic blow-up [46]. On the other hand it is instructive to note that all results in the paper also hold for the stronger model, in which each of the nodes of the RMESH consists of a full-fledged processor and a memory module. In particular, this is the case for the self-simulation algorithm of Section 4 and the simulation of a $2 \times n$-rmesh by the PRAM in Section 5.

2.2 Basic procedures for the RMESH

We make use of the following tools.

Sorting in the RMESH. Sorting n elements located at the top row of an n-RMESH, can be done in constant time by using the sorting algorithm suggested in [4, 24, 41]. Using self-simulation results for the Linear-RMESH [3] it is also possible to sort in constant time if the number of elements is higher than n, say cn for some constant c.

Bus splitting. One of the most basic techniques in computing with the RMESH is called *bus splitting.* Consider a single row of the RMESH, and suppose some arbitrary subset of its switches store input values, a single value at each switch. Then, in a single step, the rightmost switch in the row can have one of these values. This is done as follows: The switches which do not have input values connect their edges, while the others disconnect and transmit their values to the right. Clearly the rightmost input value (if exists) will be read by the rightmost switch of the row. Similarly, bus splitting may be carried on a column.

Convention. The following terminology is used to denote the probability of randomized events: We say that an event occurs *with high probability (w.h.p.)* when it occurs with probability $1 - n^{-c}$, for any constant $c > 0$.

3 Simulating the DMM and the PRAM models on the RMESH

A PRAM step is simulated on the RMESH by first simulating it on a DMM, and then simulating the DMM on the RMESH. We start by providing a constant time simulation of DMM on RMESH. Using known PRAM simulation on the DMM, we derive a simulation of PRAM on the RMESH. This simulation is improved by providing a simple constant time hashing algorithm on the RMESH, replacing a PRAM hashing algorithm. Finally, the step-simulation is extended to a full PRAM program simulation.

Unless otherwise explicitly stated, we assume in this section that the conflict resolution of the buses is COLLISION and the switching configurations are Linear.

3.1 Basic simulations

We show how a step of an n-processor DMM can be simulated on an n-RMESH in constant time. In a DMM step each of the processors issues one of the following:

- A read request, $\mathsf{read}(x, M_i)$, where M_i is the module from which the contents of memory location x, denoted by $c(x)$, has to be read.
- A write request, $\mathsf{write}(c(x), x, M_i)$, where $c(x)$ is to be written to memory location x in module M_i.

The simulation of a DMM step:

1. Each of the processors at the top row transmits its request to all nodes belonging to its column. Node (i, j) receiving a request to module M_i, stores the request.
2. In row i of the mesh, for all $0 \leq i \leq n-1$: Using row broadcasting and bus-splitting, M_i receives the right-most request addressed to it. If it is a write request, $\mathsf{write}(c(x), x, M_i)$, M_i updates the contents of the memory location x in the module to $c(x)$, and broadcasts x to all nodes of the row. If it is a read request for key x, then in the next step M_i transmits the request and the value back to the row.
3. Any node (i, j) which reads a read request $\mathsf{read}(x, M_i)$, compares x with the key x' of the request of p_j. If the request of p_j is for module M_i, and $x = x'$, then it sends the received value to p_j.
4. Any processor p_i, $0 \leq i \leq n-1$, makes the internal computations taken by processor p_i of the DMM at this step.

Clearly, every module in the above algorithm responds to the request issued by the processor with the lowest *id*. This is a PRIORITY conflict resolution, and in particular, it is stronger than the ARBITRARY conflict resolution. The above algorithm can be summarized by the following lemma.

Lemma 3.1 (DMM simulation) *One step of an n-processor* DMM *with* PRIORITY *conflict resolution can be simulated in constant time on the* COLLISION *n-*RMESH.

We use the simulation results due to Karp, Luby and Meyer auf der Heide.

Lemma 3.2 ([26]) *An $(n \lg\lg n \lg^* n)$-processor* EREW PRAM *and an $(n \lg\lg n)$-processor* ARBITRARY CRCW PRAM *can be simulated by an n-processor* DMM *with delay $O(\lg\lg n \lg^* n)$ w.h.p.*

Combining Lemma 3.1, and Lemma 3.2 we get:

Corollary 3.3 *An $(n \lg\lg n \lg^* n)$-processor* EREW PRAM *and an $(n \lg\lg n)$-processor* ARBITRARY CRCW PRAM *can be simulated on a* COLLISION *n-*RMESH *with $O(\lg\lg n \lg^* n)$ delay per step w.h.p.*

3.2 Faster, efficient simulation of the CRCW PRAM on the RMESH

In this section we improve the simulations of PRAM on the RMESH. We show:

Theorem 3.4 *One step of an $(n \lg\lg n)$-processor* ARBITRARY CRCW PRAM *can be simulated in $O(\lg\lg n)$ time w.h.p. on a* COLLISION *Linear n-*RMESH.

The simulation algorithm is based on the algorithms by Karp *et al.* [26].

Before proceeding further, we mention a crucial data type for our application. A *lookup table* is a data structure that stores a multi-set S of size n in a $O(n)$ memory size and enables n *lookup queries* to be executed in constant time, using n processors. A *hashing algorithm* computes a lookup table for a given multi-set S.

The following lemma is implicit in [26]:

Lemma 3.5 *One step of an $(n \lg\lg n)$-processor* ARBITRARY CRCW PRAM *can be simulated on an n-processor* DMM *by $O(\lg\lg n)$ calls to a hashing algorithm.*

The algorithm that realizes Lemma 3.5 is the basis for the algorithm that realizes the CRCW PRAM simulation of Lemma 3.2; the $(\lg^* n)$ overhead and the $(\lg^* n)$ factor in the delay of the latter simulation algorithm are due to the application of a CRCW PRAM hashing algorithm [20], adapted to the DMM. As a result we have:

Corollary 3.6 *One step of an $(n \lg\lg n)$-processor* ARBITRARY CRCW PRAM *can be simulated on a* COLLISION *n-*RMESH *by $O(\lg\lg n)$ calls to a hashing algorithm.*

Proof. By Lemma 3.5 and Lemma 3.1. ∎

A hashing algorithm takes $\Theta(\lg^* n)$ steps on the DMM. A hashing algorithm can be executed, however, in constant time on the n-RMESH, as described next. We will then proceed to describe the resulting improved simulation algorithms.

Hashing in constant time on the RMESH

The PRAM simulation on the RMESH uses a lookup table data structure, in which hashed elements are stored.

Lemma 3.7 *Let $c > 0$ be a constant. A lookup table for a multi-set of cn elements can be constructed in constant time by a Linear* COLLISION *n-*RMESH*. Subsequently, for every constant $c' > 0$, a batch of $c'n$ queries (lookup, delete or insert) can be executed in constant time, as long as the number of elements is $O(n)$.*

Proof. The lookup table is represented in an array $H[1..cn]$, where c elements of the array are stored at each memory module M_i. The constant time construction of the lookup table is described in the following procedure.

Algorithm Static Table:

1. *Sorting:* Sort the input elements.
2. *Selecting a representative:* For each sequence of equi-valued elements, all but the left-most element are removed.
3. *Building the table:* Every processor p_i sends its elements (which were not removed) to module M_i.

Subsequently, a batch of n lookup queries, one per processor, are executed by letting each switch (k, j) compare the k-th request with the at most c elements in module M_j.

Algorithm Static Lookup:

1. Each processor broadcasts its query element to all the nodes in its column.
2. Repeat the following for $i = 1$ to c:
   (a) Each module broadcasts its i-th element to all the nodes in its row.
   (b) Each node compares the two elements received (one from the processor and one from the module). If they match, the node notifies the processor in its column.

Both the construction of the table and the execution of n lookup instructions in parallel are completed in $O(1)$ time.

A lookup operation for $c'n$ elements may be executed in $O(1)$ time, simply by repeating Algorithm Static Lookup c' times. Whenever the number of stored elements is $O(n)$, a batch of $O(n)$ insert or delete operations can be executed in constant number of steps. ∎

We note that the only usage of the sorting algorithms above is for electing a leader out of every subset of non-distinct elements to be stored in the hashing table.

Theorem 3.4 follows by Lemma 3.7 and Corollary 3.6.

3.3 Simulating full PRAM programs

We now show that simulating a single step of a PRAM implies a simulation of a full PRAM program with a slowdown factor being the same as the delay of one step w.h.p. Thus the efficient simulation of one PRAM step translates into the efficient simulation of full PRAM algorithm. In fact, this extension is necessary even for claiming an efficient simulation of a single PRAM step, for the following reason. Note that Theorem 3.4 deals with simulating a step of an n-processor PRAM on p-RMESH, where $p = n/\lg\lg n$. Given a p-RMESH, for an arbitrary $p \leq n/\lg\lg n$, the n-processor PRAM step is implemented on a $(p\lg\lg p)$-processor PRAM in $O(n/p\lg\lg p)$ time, using the self-simulation property of the PRAM. Each of the $O(n/p\lg\lg p)$ steps are then simulated on the p-RMESH, using the step simulation of Theorem 3.4. Thus, in order to simulate the n-processor PRAM step, we need a simulation algorithm that deals with several steps.

Theorem 3.8 *Let $\mathcal{A}$ be an algorithm that runs on an n-processor* CRCW PRAM *in time T, using memory of size M, such that $T \leq n^k$ and $M \leq n^k$ for some constant k. Then Algorithm $\mathcal{A}$ can be adapted to run on an p-*RMESH *in time $O(T(n/p + \lg\lg p))$ w.h.p.*

Proof. The simulation which realizes Lemma 3.5, and hence Theorem 3.4, is based on mapping the memory locations into the memory modules, by using several hash functions that are selected at random from an appropriate class of hash functions. With high probability, the hash functions satisfy certain properties that are required in order to carry on the simulation algorithm. If the performance of the algorithm degrades due to an unluckily bad selection of the hash functions, then a new set of hash functions are selected and the algorithm is started afresh. As shown in [26], the probability that the simulation takes more than $c\lg\lg p$ time, for some constant c, due to bad selection of hash functions, can be made $n^{-\ell}$ for an arbitrary large constant ℓ (the constant c may depend on ℓ).

Let $\epsilon > 0$ be some constant, and assume first that $p \geq n^{1-\epsilon}$. The n-processor PRAM algorithm of T steps is self-simulated on a $(p\lg\lg p)$-processor PRAM to yield an algorithm running in $T' = T\lceil n/p\lg\lg p\rceil \leq n^{k+\epsilon}/\lg\lg p$ steps. Each of these steps is simulated on the p-RMESH in $O(\lg\lg p)$ time w.h.p., using Theorem 3.4. The probability that the simulation of *any* of the T' steps takes longer than $c\lg\lg p$ time is at most $n^{k+\epsilon-\ell}$. By selecting a sufficiently large $\ell > k+\epsilon$, the total running time of the simulated algorithm is $T'\lg\lg p = O(T(n/p + \lg\lg p))$ w.h.p.

It remains to consider the case $p \leq n^{1-\epsilon}$. The simulation algorithm for this case is in fact simpler, and is based on an algorithm due to Kruskal, Rudolph and Snir [28, Thm 4.9], in which the memory locations are mapped into the memory modules using a single polynomial hash function. For $p \leq n^{1-\epsilon}$, and an arbitrary constant ℓ, there is a constant c' (which is a function of ℓ and ϵ) such that an n-PRAM step can be simulated on a p-RMESH in time $c'n/p$ with probability $1 - n^{\ell}$. The probability that the simulation of *any* of the T PRAM steps takes longer than $c'n/p$ is at most $n^{k-\ell}$. By selecting a sufficiently large $\ell > k$, the total running time of the simulated algorithm is $O(Tn/p)$ w.h.p. ∎

3.4 A direct application: a parallel dictionary on the RMESH

As a direct application of Theorem 3.4 we obtain a construction of a parallel dictionary on the RMESH. A *parallel dictionary* is a data structure which supports the instructions *insert, delete,* and *lookup queries*, for elements that are drawn from some finite universe $U = [0, \ldots, p-1]$. It is a standard and basic data type (see, e.g., [1]). Both lookup tables and parallel dictionaries are useful tools for parallel algorithms that use a large space. Previously, no fast, efficient implementation of the dictionary algorithm on the RMESH was known.

A parallel dictionary handles one batch of operations at a time. Each batch consists of an array of elements and an instruction: insert, delete, or lookup. The parallel dictionary processes a batch using several processors.

We use the following theorem:

Lemma 3.9 ([20]) *There exists a dictionary implementation on an n-processor* ARBITRARY CRCW PRAM*, with the following features, all w.h.p.*

- *(a) At all times, the total space used by the dictionary is linear in the number of elements currently stored in the dictionary.*
- *(b) Any batch of $n \lg^* n$ elements is processed in $O(\lg^* n)$ time.*
- *(c) A lookup instruction for a batch of n elements is processed in constant time.*

Now the following theorem is implied directly by the PRAM simulation algorithms on the n-RMESH and by the above theorem. When the size of the dictionary is linear in n, then the implementation may use the constant time hashing procedures directly. The detailed description of how all this is carried may be found in [46].

Theorem 3.10 *Let S be the set of elements in the dictionary, let $\epsilon > 0$ be chosen arbitrarily small, and let b be a sufficiently large constant. A parallel dictionary can be implemented on the n-*RMESH *with the following features:*

1. *Any batch of m arbitrary instructions is processed in $O(m/n + \lg\lg n \lg^* n)$ time w.h.p.*

2. *A lookup instruction for a batch of $O(n \lg\lg n)$ elements is processed in $O(\lg\lg n)$ time w.h.p..*

3. *If $|S| + m \leq bn$, ($|S|$ is the number of elements in the dictionary) then any batch of m instructions is processed in $O(1)$ time.*

4 Self-Simulation of the RMESH

In this section we give an important application of the PRAM simulation. Somewhat surprisingly, we are able to use PRAM simulation in order to improve previous self-simulation results for the RMESH.

We consider the simulation of the $N \times N$ General-RMESH by the $p \times p$ Linear-RMESH. Since the General-RMESH is stronger than the Linear-RMESH, the simulation immediately implies self-simulation algorithm with the same complexities for both the General-RMESH and the Linear-RMESH. For brevity we call them both RMESH in the sequel; it should be understood that the specific model is either the General-RMESH or the Linear-RMESH, depending if it is the simulated or the simulating model, respectively.

In the sequel we say that the self-simulation algorithm terminates in a certain number of steps w.h.p. when the probability of non-termination can be made as small as $N^{-\Omega(1)}$.

One of the notions we use extensively is that of a *window*. A submesh of the N-RMESH is called a window, when it is one of the $(N/p)^2$ $p \times p$ submeshes in a partitioned N-RMESH (assuming, for simplicity, that N/p is integral). Fig. 2 below illustrates a partition of the mesh into windows. A window is always "aligned to a boundary of size p", where the *boundary* of a window consists of all the switches from which edges exit the window (or from which there is a direction with no edge at all). The boundary of a window is composed out of four *facets* in the intuitive way (each corner switch belongs to two facets). Every window W has a unique label $\mathsf{label}(W)$ of size $2(\lg N - \lg p)$ bits. On the boundary of each window, every switch has a unique local id of size $2 + \lg p$ bits. For simplicity we assume these quantities to be integral numbers.

Folding the mesh

The following function is useful for mapping a mesh of size N into a mesh of size p. Let $p \leq x \leq N$; then

$$\mathsf{fold}(x) = \begin{cases} x \bmod p & \text{if } x \text{ div } p \text{ is even} \\ p - 1 - (x \bmod p) & \text{otherwise} \end{cases}$$

We use $(r, c) \longrightarrow (\mathsf{fold}(r), \mathsf{fold}(c))$ for mapping a switch (r, c) of an N-RMESH into the p-RMESH. Informally, this has the same effect as that of folding a large page of paper several times into a square of size $p \times p$. A point on the p-sized square "simulates" all points of the folded page that are stabbed when pushing a pin at this point.

The advantage of the folding mapping in self-simulation algorithms for the RMESH was first observed in [3]. It is implied by the fact that when the p-RMESH simulates the p-submeshes one by one, boundary points of p-submeshes which are adjacent in the folded mesh are mapped to the same switch of the p-RMESH. Hence, the existence of a bus which crosses from one p-submesh to the other may be observed by this switch.

4.1 Algorithm outline

The idea of the self-simulation algorithm is to simulate the windows one by one, performing the local buses communication, while collecting connectivity data for buses which cross window boundaries. We note that the simulation of these global, inter-window buses is what makes the self-simulation rather difficult. The connectivity data is collected and kept through the windows boundaries. The folding function implies an assignment of processors to boundary nodes which is suitable for the processing of this data, and its representation as an appropriate graph connectivity problem. An edge in the graph represents a pair of two adjacent boundary nodes from adjacent windows which are connected in the configuration of the simulated RMESH (e.g., nodes x and x' in Fig. 2). A connected component of the graph corresponds to a global bus in the simulated RMESH.

The connected components of this graph are computed, along with the message that is heard on each of the corresponding buses. This connected components computation is where we use our efficient PRAM simulation results from the previous section. Specifically, the PRAM simulation, together with a known PRAM algorithm for connected component computation, are used to derive an appropriate connected component algorithm for the simulating RMESH. We note that although the derived connected component algorithm on the RMESH is not efficient in general, its usage in our context enables an efficient self-simulation algorithm.

Finally, the message computed for each connected component is distributed to the appropriate boundary nodes at each window. Once again, the computations within windows are simulated one-by-one, where the information at the boundary nodes is disseminated to the internal nodes by regular RMESH steps.

4.2 The self-simulation algorithm

The self-simulation algorithm consists of three main phases and two intermediate ones.

Phase (1): Sweep

In this phase we scan the windows one-by-one in an arbitrary order, where windows are mapped to the p-RMESH according to the folding mapping as described above. A window W is simulated in a single step by the simulating p-RMESH. Any switch which is transmitting in the original mesh transmits and all switches connected by the bus B store the resulting message $\mathsf{message}(B)$. When a bus does not exit the window, its simulation is completed in this step. We thus proceed to consider only boundary-crossing buses. In parallel, for every bus B in W we elect a label $\mathsf{label}(B)$ as follows: We first elect a "leader" $\mathsf{leader}(B)$ out of the switches on the boundary of W which are also connected to B. Note there is at least one such switch. If the conflict resolution of the buses is ARBITRARY (or stronger) then this can be done in a single step. Else, if the conflict resolution is COLLISION then electing a leader can be done by standard methods in $\text{Sizeof}(\mathsf{id}) = 2 + \lg p$. We next set the label of the bus to be the concatenation of the identity of the elected leader with the label of the window, i.e., $\mathsf{label}(B) = \langle \mathsf{label}(W), \mathsf{id}(\mathsf{leader}(B))\rangle$.

Summarizing, for each window we elect labels in parallel for all the buses, which takes $O(1)$ steps in the ARBITRARY model and $O(\lg p)$ in the COLLISION model. All switches on the boundary of the bus B store the information $[\mathsf{label}(B), \mathsf{message}(B)]$, called the *identifier* of B. Since there are $(N/p)^2$ windows, every boundary switch of the simulating p-RMESH collects up to $(N/p)^2$ bus identifiers, one per window. More accurately, if a switch is at the point (r', c'), then it stores the identifiers that occur at each boundary point of a window which represents a point (r, c) of the original mesh such that $\mathsf{fold}(r, c) = (r', c')$. These identifiers are then moved to the memory modules, so that the i-th module stores the i-th identifier from each facet of the boundary of every window.

Intermediate-Phase (1.5): Creating the Connectivity Graph

We define a graph G as follows. The nodes of G are the bus labels, $\mathsf{label}(B)$, i.e., each node represents a global bus B at a certain window W. The edges of G are all the pairs of nodes $\langle \mathsf{label}(B), \mathsf{label}(B')\rangle$ such that $\mathsf{fold}(\mathsf{leader}(B)) = \mathsf{fold}(\mathsf{leader}(B'))$ and B is connected to B' in the simulated mesh. Thus, there is an edge for each pair of connected buses B, B' whose identifiers are stored in the same switch. For each edge we also assign a message $\mathsf{conflict}(\mathsf{message}(B), \mathsf{message}(B'))$, that is determined as a function of the messages of the respective buses.

The function $\mathsf{conflict}$ depends on the nature of the messages $\mathsf{message}(B)$ and $\mathsf{message}(B')$, and on the collision rule of the simulated RMESH: For the ARBITRARY RMESH, $\mathsf{conflict}$ receives the value of either $\mathsf{message}(B)$ or $\mathsf{message}(B')$, with arbitrary choice, unless exactly one of the messages is non-idle, in which case it is the same as the non-idle message. For the COLLISION RMESH, $\mathsf{conflict}$ receives the collision symbol #, unless exactly one of the messages is non-idle, in which case it is the same as the non-idle message.

Suppose that an edge $\langle \mathsf{label}(B), \mathsf{label}(B') \rangle$ is computed by the processor of switch $\mathsf{fold}(\mathsf{leader}(B))$ $(= \mathsf{fold}(\mathsf{leader}(B'))$, in a constant number of operations. Since there are $(N/p)^2$ windows, computing the edges for all pairs of adjacent buses and storing them in the modules of the p-RMESH takes $O((N/p)^2)$ steps.

Examples for the process of constructing G are given in Fig. 2.

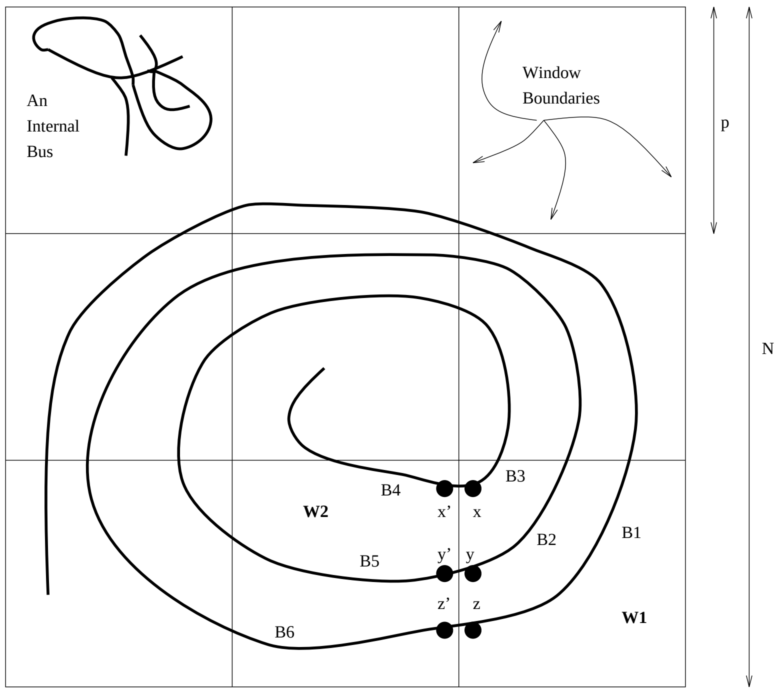

Figure 2: An example for the construction of the connectivity graph G. Here $N/p = 3$ so the simulated mesh is split into 9 windows. The upper left window completely contains a somewhat complicated bus which is simulated in a single step of Phase (1). In contrast, the simulation of the lower concentric bus requires the collection of connectivity data from all windows. For example, when simulating W_1 and W_2, x, y and z are simulated by the same switches simulating x', y' and z' respectively (i.e., $\mathsf{fold}(x) = \mathsf{fold}(x')$, $\mathsf{fold}(y) = \mathsf{fold}(y')$ and $\mathsf{fold}(z) = \mathsf{fold}(z')$). Thus, during Phase (1) switch $\mathsf{fold}(x)$ (or its corresponding memory module) stores both identifiers of $B3$ and $B4$, and during Intermediate-Phase (1.5) it (or the corresponding processor) combines them into an edge of the connectivity graph G. Similarly switches $\mathsf{fold}(y)$ and $\mathsf{fold}(z)$ create edges in G which represent the connections of $B2 - B5$ and $B1 - B6$, respectively.

Phase (2): Connected Components

Since each window has at most $4p$ bus labels, and there are $(N/p)^2$ windows, the total number of bus labels, and hence, the total number of nodes in G, is at most $4N^2/p$. Moreover, the total number of edges in G is at most $4N^2/p$. The connectivity of G defines the connectivity of the buses across windows in the simulated RMESH. We therefore determine the bus connectivity of the N-RMESH by computing connected components on the graph G.

We note that the connected component algorithm from [40] requires the adjacency matrix to be given at specific switches of the RMESH, and that the one from [3] requires the input to be of approximately the square root of the size of the mesh. Thus the available connected component algorithms for RMESH are not suitable to our goal. Our approach is to take a PRAM algorithm for computing connected components, and simulate it on the p-RMESH using the PRAM simulation algorithm from Section 3.

We use a CRCW PRAM algorithm by Gazit [18] for computing the connected components of a given graph G on v nodes and m edges on $n = (v+m)/\lg v$ processors, in $T = O(\lg v)$ time w.h.p. In our case, $v = O((N/p)^2)$ and $m = O(N^2/p)$. Using Theorem 3.8, Gazit's algorithm can be simulated on the p-RMESH in time $O(T(n/p + \lg\lg p))$, which is $O(\lg N \lg\lg p + (N/p)^2)$ w.h.p.

Intermediate-Phase (2.5)

Once connected components are determined, the message for each bus is determined by joining together $\mathsf{message}(B)$ as described above for all edges in the connectivity graph G. This is done for all segments of the same bus by extracting the messages from the input edges formed in Phase (1). We avoid tiresome details of this procedure by simulating again a PRAM algorithm for this task (which takes $O(1)$ steps on a CRCW PRAM with N^2/p processors).

Reversing some previous data movements, we make the edges move back to the modules where they were formed in Intermediate-Phase (1.5). Note that these are point-to-point communication operations which may be "reversed" by forming the same buses and switching transmitters and receivers. Finally, by reversing Intermediate-Phase (1.5) itself, we attach to each identifier of a bus segment the evaluated message of its full bus.

Phase (3): BackSweep

Note the identifiers together with the evaluated messages now reside in the same modules where they were formed. Thus, it is easy to move them to switches of the p-RMESH which simulate window-boundary switches and which were elected as leaders during Phase (1).

The p-RMESH now (logically) scans window by window of the simulated N-RMESH while making the leaders of the buses broadcast the evaluated message on their corresponding bus-segments. The readers of the simulated segments read the broadcasted messages. This clearly takes one step per simulated window.

4.3 Summary

The whole self-simulation algorithm takes $O(\lg N \lg\lg p + (N/p)^2)$ steps w.h.p. on the RMESH with buses having ARBITRARY collision resolution, and $O(\lg N \lg\lg p + (N/p)^2 \lg p)$ steps w.h.p. on the RMESH with COLLISION resolution. Thus, for the ARBITRARY model the self-simulation is efficient with an additive delay of $O(\lg N \lg\lg p)$. For the COLLISION model the self-simulation has an overhead of $O(\lg p)$. Recall that previous (deterministic) self-simulation algorithms have a multiplicative overhead of $O(\lg N \lg(N/p))$ [3].

Theorem 4.1 *The self-simulation of a General N-RMESH on a Linear p-RMESH can be performed w.h.p. in*

- $O(\lg N \lg\lg p + (N/p)^2)$ *steps on the* ARBITRARY *model, and in*
- $O(\lg N \lg\lg p + (N/p)^2 \lg p)$ *steps on the* COLLISION *model.*

5 Simulating the $2 \times n$ General-RMESH on the PRAM

We show in this section that simulating one step of a $2 \times n$ General-RMESH on a CRCW PRAM is equivalent, up to a constant factor, to computing the nearest-zeros problem on the CRCW PRAM. We assume here that the PRAM write conflict resolution policy is the same as that of the simulated RMESH, i.e., it is either ARBITRARY or COLLISION. For most of our discussion an even weaker sub-model may be assumed, the COMMON CRCW PRAM, in which all processors trying to write to the same memory cell are writing the same value. For the rest of this section an RMESH always denotes a General-RMESH.

We first define the *nearest-zeros problem*, then proceed to show an efficient reduction to this problem.

The Nearest-Zeros Problem

Let the function $I_i(n)$ be the inverse of a function at the i-th level of the primitive recursive hierarchy, and the function $\alpha(n)$ be the inverse Ackermann function. (Thus, $I_2(n) = \lg n$, $I_3(n) = \lg^* n$, and $I_{\alpha(n)}(n) = \alpha(n)$, cf. [12, pp. 451-453].)

Given an array $A[1..n]$ of 0's and 1's, the *nearest-zeros* problem is to find for each index $i = 1, \ldots, n$ the nearest locations on its right and on its left in A which contain a 0 bit. Berkman and

Vishkin [7] and Ragde [42] considered this problem and presented very fast parallel algorithms, and recently, Chaudhuri and Radhakrishnan [9] provided a matching lower bound:

Lemma 5.1 ([7, 42, 9]) *The parallel time complexity for solving the nearest-zeros problem on the* CRCW PRAM *is:*

(i) $O(\alpha(n))$*, using* $n/\alpha(n)$ *processors;*

(ii) $O(i)$*, for any* $i = 1, \ldots, \alpha(n)$*, using* $nI_i(n)$ *processors; and*

(iii) $\Omega(\alpha(n))$*, when using* n *processors.*

5.1 Reducing the simulation problem to the nearest-zero problem

Lemma 5.2 *The simulation of a step of a* $2 \times n$ RMESH *on a* CRCW PRAM *can be reduced in constant time to the nearest-zeros problem.*

Proof. A step of a $2 \times n$ RMESH consists of two sub-steps: First, the configuration is determined by letting each node select a subset of its neighbors. Then, a subset of the processors (the "writers") write their values which reach all nodes in their respective buses. The simulation algorithm will implement these two sub-steps on the CRCW PRAM, using an array M for communication: $M[i]$ is used for the i-th connected component, with the connected components labeled in some arbitrary manner.

The simulation algorithm is based on the following general steps:

- Compute connected component: let each connected component (a bus) have a distinct label from $[1..O(n)]$ and let each node know to which connected component it belongs.
- Let each processor that represents a writer of the i-th connected component write its value into $M[i]$.
- Let each processor belonging to connected component i read $M[i]$.

The only non-trivial part is the computation of connected components. We proceed to show that this part can be reduced with constant-time overhead to the nearest-zeros problem.

Recall that a configuration on a $2 \times n$ mesh is a sub-graph G of the $2 \times n$ mesh that consists of edge-disjoint buses. We show first how to reduce the connected components problem into one in which all buses are vertex-disjoint ("vertex disjoint configuration"). Formally, in the description of the reduction we use the following terminology. We say that the graph consists of *vertex disjoint* *components* if each vertex belongs to at most one connected component. Let $D = (V, E_D)$ and $D' = (V, E_{D'})$ be two graphs with the same set of vertices V. Then $D \cup D' = (V, E_D \cup E_{D'})$ denotes the graph with the same set of vertices V and a set of edges that is the union of the respective edge-sets. Similarly, $D \cap D' = (V, E_D \cap E_{D'})$.

A reduction to vertex-disjoint configurations

Let an *extreme edge* of some bus be an edge which is disconnected by at least one of its adjacent vertices (e.g., a cycle has no extreme edges, a path has two extreme edges). Recall that the decisions whether to connect an edge are taken locally by the switch, hence an edge can be extreme for a bus even when both its adjacent vertices have other edges belonging to that (or to another) bus.

Consider configurations G_v, G_e and G_o based on the extreme edges of each bus. G_v is a configuration consisting of all extreme edges which are vertical. G_e (resp. G_o) is a configuration consisting of all extreme edges which are horizontal, and which connect nodes i and $i+1$ (of either of the top or bottom rows) where i is even (resp. odd). Let G' be the configuration $G \setminus \{G_v \cup G_e \cup G_o\}$, so each of G', G_e, G_o and G_v consists of vertex disjoint components. Let S be the set of nodes in $G' \cap \{G_v \cup G_e \cup G_o\}$, i.e., the set of nodes which are not at the endpoint of some path in G, and which belong to extreme edges of G. Figure 3 gives an example for an input configuration and its four subgraphs which consist of vertex disjoint components.

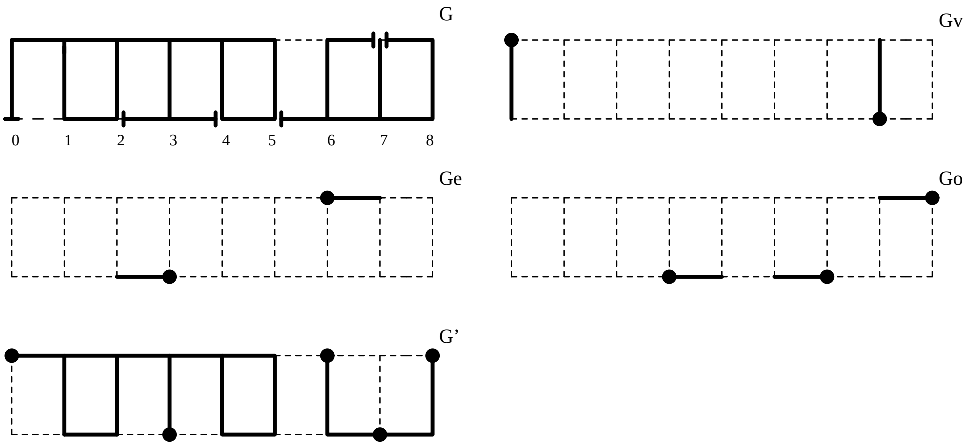

Figure 3: An input configuration G and its four subgraphs G', G_o, G_e and G_v, which consist of vertex disjoint components. Vertices in S are emphasized as black circles.

We now show a simulation of the $2 \times n$ mesh by itself, where all the configurations taken during the simulation are vertex-disjoint. A step with configuration G is simulated in seven steps involving the configurations G_v, G_e, G_o, G', G_o, G_e and G_v (correspondingly). In each of the first three steps, the writers are the writers of G which are included in the configuration that is taken, and the readers are the nodes from S. In the fourth step, the writers are all the writers of G and those nodes in S which read some message during the first three steps. The readers are all nodes in S and all readers from G. During the fifth, sixth and seventh steps, nodes from S write the value they read during the fourth step. It is straightforward to verify the correctness of the above reduction.

The rest of the proof deals with computing connected components in a sub-graph with vertex-disjoint configuration.

Let nodes that are connected in the sub-graph with a vertical edge be denoted as a *pair*. The main observation which enables our simulation is the following fact which can be easily verified.

Proposition 5.3 *In G', if two pairs belong to the same connected component then all pairs in between belong to the same component.*

Proposition 5.3 implies that between two pairs of the same connected component, there cannot be a pair which belongs to a different connected component. Thus, any other component between such pairs must consist of a horizontal path only; denote such component as a *straight component.* Denote as *right path* and *left path* the parts of a component to the right of its rightmost pair, and to the left of its leftmost pair, respectively (the right and left paths may be of length 0). Assume that straight components as well as right and left paths are deleted from the given sub-graph. Then, between any two components there is a vertical cut. Our algorithm will therefore consist of the following steps:

Step 1. Identify all the straight components, label each by its rightmost node, and delete them from the sub-graph.

Step 2. Identify all the right (resp. left) paths, find for each a "leader": its leftmost (resp. rightmost) node, and delete them from the sub-graph.

Step 3. Compute for each component its rightmost pair, and label it by the index of this pair.

Step 4. Label each right or left path by the label of its leader.

Implementation of Step 1. Consider only nodes in the top row. Let $R[i]$ (resp. $L[i]$) be 1 if node i is connected with node $i+1$ (resp. $i-1$), and 0 otherwise. A node is in a straight component if and only if the nodes represented by its nearest-zero indices from its right in R and from its left in L

are both disconnected from their neighbors in the bottom row. Similarly, the straight components from the bottom row can be identified and labeled.

Implementation of Step 2. Consider only nodes in the top row. Let $R[i]$ (resp. $L[i]$) be 1 if node i is connected with node $i+1$ (resp. $i-1$), and 0 otherwise. A node is in a right path if and only if the node represented by its nearest-zero index from its right in R is disconnected from its neighbor in the bottom row, and the node represented by its nearest-zero index from its left in L is connected to its neighbor in the bottom row. The left path is found and labeled in a similar way, and so are the right and left paths in the bottom row.

Implementation of Step 3. Let $R[i]$ be 1 if either the top or bottom nodes in location i are connected with their neighbors in location $i+1$, and 0 otherwise. The right nearest-zero i indicates the rightmost pair in the connected component for which any of the nodes in location i may belong. The correctness of this implementation is implied by the fact that the straight components and the right paths were removed.

Implementation of Step 4. Trivial.

The lemma follows. ∎

Remark. (1) The simulation algorithm of Lemma 5.2 shows that simulating one step of a $2 \times n$ RMESH on a CRCW PRAM is as easy (up to a constant factor) as computing connected components of a sub-graph of the $2 \times n$ mesh on the CRCW PRAM. These problems are in fact equivalent, up to a constant factor, since connected components of such graphs can be easily computed on the $2 \times n$ RMESH. (2) All the steps of the algorithm described in the proof of Lemma 5.2, except for the nearest-zeros computations, can be implemented on the Concurrent Read Exclusive Write (CREW) PRAM in constant time. This implies an $O(\lg n)$ time simulation of the $2 \times n$ RMESH on an $(n/\lg n)$-processor CREW PRAM. We note that since the OR function can be computed in constant time on the $1 \times n$ RMESH, simulating the $2 \times n$ RMESH on any number of CREW PRAM processors takes $\Omega(\lg n)$ expected time [11, 15].

5.2 On the optimality of the simulation and its applications

Lemma 5.2 shows that simulating one step of $2 \times n$ RMESH is as easy as computing nearest-zeros, both on the CRCW PRAM. The following lemma shows that computing nearest-zeros is as easy as simulating one step of $1 \times n$ RMESH, both on the CRCW PRAM as well. Its proof is by an easy algorithm that is similar to an algorithm used in [39] to compute the OR function, and to an algorithm used in [49] for the more general 'neighbor localization' problem.

Lemma 5.4 *The nearest-zeros problem can be solved in two steps on a $1 \times n$ RMESH.*

Proof. We represent the input array for the nearest-zeros problem by a $1 \times n$ RMESH, as follows. Node i is connected to node $i+1$ if the i-th bit in the input array is 1, and it is disconnected otherwise. In the first step, each node connected to its left neighbor but disconnected from its right neighbor, writes its index. Each node records the value it reads as the index of the right nearest zero. In the second step, each node connected to its left neighbor but disconnected from its right neighbor, writes its index. Each node records the value it reads as the index of the left nearest zero. ∎

Remark:. A similar representation enables an easy two-step reduction, of simulating a $1 \times n$ RMESH step on a CRCW PRAM, into the nearest-zeros problem.

Lemma 5.2 implies that any upper bound for the nearest-zeros problem on the CRCW PRAM will translate into an upper bound for simulating one step of a $2 \times n$ RMESH as well. Lemma 5.4 implies that any lower bound for the nearest-zeros problem on the CRCW PRAM will hold for the simulation of one step of a $1 \times n$ RMESH as well. (Note that this lower bound only applies to step-by-step simulations.)

By Lemma 5.2, Lemma 5.4, and Lemma 5.1 we obtain:

Theorem 5.5 *The parallel time complexity for simulating one step of a $2 \times n$ General-*RMESH *on a* CRCW PRAM *is:*

(i) $O(\alpha(n))$, using $n/\alpha(n)$ processors;

(ii) $O(i)$, for any $i = 1, \ldots, \alpha(n)$, using $nI_i(n)$ processors; and

(iii) $\Omega(\alpha(n))$, when using n processors.

Deriving lower bounds for the $2 \times n$ RMESH

As a corollary from Theorem 5.5 we obtain:

Corollary 5.6 *Consider a problem $\mathcal{P}$ of size n and let $T(n,q)$ be a time lower bound on computing $\mathcal{P}$ on a q-processor* CRCW PRAM. *Then, $T' = \Omega(T(n,q))$ is a time lower bound on computing $\mathcal{P}$ on a $2 \times (q/I_i(n))$* RMESH, *and $T'/\alpha(n)$ is a time lower bound on computing $\mathcal{P}$ on a $2 \times q$* RMESH.

Proof. Let $\mathcal{A}$ be an algorithm that computes $\mathcal{P}$ in time $T_{\mathcal{A}}$ on a $2 \times q$ RMESH. Using the simulation algorithm of Theorem 5.5, the problem $\mathcal{P}$ can be computed on a q-processor CRCW PRAM in time $O(T_{\mathcal{A}}\alpha(n))$, and on a $qI_i(n)$-processor CRCW PRAM in time $O(T_{\mathcal{A}})$. ∎

Corollary 5.6 immediately implies the new lower bounds that are mentioned in the introduction.

6 Conclusions

We have presented the first efficient simulations of the PRAM on the RMESH. We establish the usefulness of the PRAM simulation by means of two applications. The first application is the efficient implementation of the dictionary data structure, which was not previously considered on the RMESH. It is mostly obtained in a straightforward manner from our simulation result and a known PRAM algorithm. The second is a first efficient self-simulation algorithm for the RMESH with general switches. Indeed, the number of problems whose solution may take advantage of our simulation is abundant.

It is well known that the RMESH may be used to solve some problems faster than traditional models of parallel computation such as the PRAM. This accelerated computation is attributed to the utilization of network configurations as a computational resource: There are exponential number of configurations, each of which induces a different communication pattern. Despite the above, it may seem that intensive data-manipulation problems are solved on the PRAM much faster than on the RMESH due to the PRAM's increased memory bandwidth. Our randomized simulation is targeted at this set of problems, and we believe it will prove useful whenever one of them is considered.

We gave tight bounds for the complexity of simulating one step of the $2 \times n$ RMESH on the CRCW PRAM. The simulation result shows that the power of reconfiguration may be significant only for $r \times n$ RMESH, for $r \geq 3$. More specifically, lower bounds for CRCW PRAM translate to similar lower bounds on the $2 \times n$ RMESH, up to a factor of $\alpha(n)$. Thus, we obtain as a corollary several new lower bounds for the $2 \times n$ RMESH.

Postscript

Recently, Czumaj, Meyer auf der Heide, and Stemann [13] provided improved simulation algorithms of the EREW PRAM on the DMM. They showed that an $(n \lg\lg\lg n \lg^* n)$-processor EREW PRAM can be simulated on an n-processor DMM with $O(n \lg\lg\lg n \lg^* n)$ delay. Combined with Lemma 3.1, this implies a similar simulation result of the EREW PRAM on an n-RMESH. Specifically, we obtain that an $(n \lg\lg\lg n \lg^* n)$-processor EREW PRAM can be simulated on an n-processor COLLISION RMESH with $O(\lg\lg\lg n \lg^* n)$ delay.

As a result, the self-simulation of an N-RMESH on a p-RMESH, given in Theorem 4.1, can be improved to $O(\lg N \lg\lg\lg p \lg^* p + (N/p)^2 \lg p)$ steps on the COLLISION RMESH. This can be obtained by using the algorithm presented in Section 4, with the exception of replacing Gazit's CRCW PRAM connectivity algorithm by the EREW PRAM algorithm of Halperin and Zwick [22].

Very recently, Czumaj *et al.* [14] improved the results of Section 3 for the ARBITRARY RMESH, showing that an n-processor CRCW PRAM can be simulated on an ARBITRARY n-RMESH in constant time w.h.p. (This, however, does not imply improvement in simulating a CRCW PRAM on the

COLLISION RMESH.) As a result, using the algorithm presented in Section 4, the self-simulation of an N-RMESH on a p-RMESH, given in Theorem 4.1, can be improved to $O(\lg N + (N/p)^2)$ time w.h.p. on the ARBITRARY RMESH; i.e., with constant overhead and $O(\lg N)$ additive delay. An interesting open problem is to obtain a similar improvement for the COLLISION RMESH.

Finally, since sorting can be computed in constant time on the RMESH, any EREW PRAM simulation on the RMESH can be adapted to a CRCW PRAM simulation on the RMESH with the same complexities, up to a constant factor [14] (cf. [36]). By the EREW PRAM simulation mentioned above, this implies an efficient simulation of a CRCW PRAM on an n-processor COLLISION RMESH with $O(\lg\lg\lg n \lg^* n)$ delay w.h.p. An interesting open problem is to obtain an efficient, constant time simulation of the CRCW PRAM on the COLLISION RMESH (as for the ARBITRARY RMESH).

The above remarks can be summarized in the following theorems:

Theorem 6.1 (simulating PRAM on RMESH)
We have,

- *One step of an $(n \lg\lg\lg n \lg^* n)$-processor* CRCW PRAM *can be simulated in $O(\lg\lg\lg n \lg^* n)$ time w.h.p. on a* COLLISION *Linear n-*RMESH.
- *One step of an n-processor* CRCW PRAM *can be simulated in $O(1)$ time w.h.p. on a* ARBITRARY *Linear n-*RMESH *[13].*

Theorem 6.2 (self-simulating RMESH)
*The self-simulation of a General N-*RMESH *on a Linear p-*RMESH *can be performed w.h.p. in*

- *$O(\lg N + (N/p)^2)$ steps on the* ARBITRARY *model; and in*
- *$O(\lg N \lg\lg\lg p \lg^* p + (N/p)^2 \lg p)$ steps on the* COLLISION *model.*

Acknowledgments

We thank Galit Shemesh for participating in some of the research reported in Section 3. We thank Artur Czumaj, Friedhelm Meyer auf der Heide, and Volker Stemann for helpful discussions.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1983.

[2] P. Beame and J. Håstad. Optimal bounds for decision problems on the CRCW PRAM. *Journal of the ACM*, 36(3):643–670, July 1989.

[3] Y. Ben-Asher, D. Gordon, and A. Schuster. Optimal simulations in reconfigurable arrays. In *Proc. 1st European Symposium on Algorithms*, September 1993. To appear JPDC.

[4] Y. Ben-Asher, D. Peleg, R. Ramaswami, and A. Schuster. The power of reconfiguration. *Journal of Parallel and Distributed Computing*, 13(2), October 1991. Special issue on Massively Parallel Computation.

[5] Y. Ben-Asher, D. Peleg, and A. Schuster. The complexity of reconfiguring networks models. In *Proc. of the Israel Symposium on the Theory of Computing and Systems*, May 1992. To appear Information and Computation.

[6] O. Berkman, Y. Matias, and P.L. Ragde. Triply-logarithmic upper and lower bounds for minimum, range minima, and related problems with integer inputs. In *Proc. 3rd Workshop on Algorithms and Data Structures, Springer LNCS 709*, pages 175–187, 1993.

[7] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.

[8] S. Chaudhuri. A lower bound for linear approximate compaction. In *Proc. 2nd Israel Symp. on Theory of Comp. and Sys.*, pages 25–32, 1993.

[9] S. Chaudhuri and J. Radhakrishnan. The complexity of parallel prefix problems on small domains. In *Proc. 33rd IEEE Symp. on Foundations of Computer Science*, 1992.

[10] A. Condon, R.E. Ladner, J. Lampe, and R. Sinha. Complexity of sub-bus mesh computations. Technical Report 93-10-02, Dept. of CS&EE, U. of Washington, 1993. To appear in *SIAM Journal on Computing.*

[11] S.A. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing*, 15:87–97, 1986.

[12] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1990.

[13] A. Czumaj, F. Meyer auf der Heide, and V. Stemann. Shared memory simulations with triple-logarithmic delay. In *Proc. European Symp. on Algorithms*, Corfu, September 1995. (To appear).

[14] A. Czumaj, F. Meyer auf der Heide, and V. Stemann. Simulating shared memory in real time: on the computation power of reconfigurable meshes. In *Workshop on Reconfigurable Architectures*, Santa Barbara, March 1995.

[15] M. Dietzfelbinger, M. Kutyłowski, and R. Reischuk. Exact lower time bounds for computing boolean functions on CREW PRAMs. *Journal of Computer and System Sciences*, 48(2):231–254, 1994.

[16] M. Dietzfelbinger and F. Meyer auf der Heide. How to distribute a dictionary in a complete network. In *Proc. 22nd ACM Symp. on Theory of Computing*, pages 117–127, 1990.

[17] M. Dietzfelbinger and F. Meyer auf der Heide. Simple, efficient shared memory simulations. In *5th ACM Symp. on Parallel Algorithms and Architectures*, pages 110–119, 1993.

[18] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM Journal on Computing*, 20(6):1046–1067, December 1991.

[19] M. Gereb-Graus and D. Krizanc. The complexity of parallel comparison merging. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, pages 195–201, 1987. Also *SIAM J. Comput.*, to appear.

[20] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, pages 698–710, October 1991.

[21] L.A. Goldberg, Y. Matias, and S.B. Rao. An optical simulation of shared memory. In *6th ACM Symp. on Parallel Algorithms and Architectures*, pages 257–267, June 1994.

[22] S. Halperin and U. Zwick. An optimal randomized logarithmic time connectivity algorithm for the EREW PRAM. In *6th ACM Symp. on Parallel Algorithms and Architectures*, pages 1–10, July 1994.

[23] J. JáJá. *Introduction to Parallel Algorithms.* Addison-Wesley Publishing Company, Inc., 1992.

[24] J. Jang and V.K. Prasanna. An optimal sorting algorithm on reconfigurable mesh. In *Proc. 6th Inter. Parallel Processing Symp.*, pages 130–137, March 1992.

[25] A.R. Karlin and E. Upfal. Parallel hashing: An efficient implementation of shared memory. *Journal of the ACM*, 35(4):876–892, 1988.

[26] R.M. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 318–326, 1992.

[27] R.M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 869–941. North-Holland, Amsterdam, 1990.

[28] C.P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71:95–132, 1990.

[29] H. Li and Q. F. Stout. Reconfigurable SIMD massively parallel computers. *Proc. of the IEEE*, 79(4):429–443, April 1991.

[30] H. Li and Q.F. Stout. *Reconfigurable massively parallel Computers.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[31] R. Lin, S. Olariu, J.L. Schwing, and J. Zhang. Simulating enhanced meshes, with applications. *Parallel Proocessing Letters*, 3(1):59–70, May 1993.

[32] P.D. MacKenzie. Load balancing requires $\Omega(\lg^* n)$ expected time. In *Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms*, pages 94–99, January 1992.

[33] P.D. MacKenzie. A separation between reconfigurable mesh models. In *Proc. 7th Int. Parallel Processing Symp.*, pages 84–88, 1993.

[34] Y. Matias. *Highly Parallel Randomized Algorithmics.* PhD thesis, Tel Aviv University, Tel Aviv 69978, Israel, 1992.

[35] Y. Matias and A. Schuster. On the power of the $2 \times n$ reconfigurable mesh. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1993.

[36] Y. Matias and U. Vishkin. A note on reducing parallel model simulations to integer sorting. In *Proc. 9th Int. Parallel Processing Symp.*, pages 208–212, 1995.

[37] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339–374, 1984.

[38] F. Meyer auf der Heide and A. Wigderson. The complexity of parallel sorting. In *Proc. 26th IEEE Symp. on Foundations of Computer Science*, pages 532–540, 1985.

[39] R. Miller, V.K. Prasanna-Kumar, D.I. Reisis, and Q.F. Stout. Image computations on reconfigurable VLSI arrays. In *Proc. of the Conference on Vision and Pattern Recognition*, pages 925–930, 1988.

[40] R. Miller, V.K. Prasanna-Kumar, D.I. Reisis, and Q.F. Stout. Parallel computations on reconfigurable meshes. *IEEE Trans. Comput.*, 42(6):678–692, June 1993.

[41] M. Nigam and S. Sahni. Sorting n numbers on $n \times n$ reconfigurable meshes with buses. In *Proc. of Intl. Parallel Processing Symposium*, pages 174–181, April 1993.

[42] P.L. Ragde. The parallel simplicity of compaction and chaining. *Journal of Algorithms*, 14:371–380, 1993.

[43] J.H. Reif, editor. *A Synthesis of Parallel Algorithms.* Morgan-Kaufmann, San Mateo, CA, 1993.

[44] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988.

[45] A. Schuster. *Dynamic Reconfiguring Networks for Parallel Computers: Algorithms and Complexity Bounds*. PhD thesis, Hebrew University, Jerusalem, Israel, August 1991.

[46] G. Shemesh. Upper and lower bounds for dynamically reconfiguring networks. Master's thesis, July 1993.

[47] A. Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, pages 20–25, 1989.

[48] E. Upfal. Efficient schemes for parallel communication. *Journal of the ACM*, 31(3):507–517, 1984.

[49] R. Vaidyanathan. Sorting on PRAMs with reconfigurable buses. *Information Processing Letters*, 42:203–208, 1992.

[50] R. Vaidyanathan and J.L. Trahan. Optimal simulation of multidimensional reconfigurable meshes by two-dimensional reconfigurable meshes. *Information Processing Letters*, 47:267–273, October 1993.

[51] L.G. Valiant. Parallelism in comparison problems. *SIAM Journal on Computing*, 4:348–355, 1975.

[52] L.G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 18, pages 944–971. Elsevier Science Publishers B.V., 1990.

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

[54] B. -F. Wang and G. -H. Chen. Two-dimensional processor array with a reconfigurable bus system is at least as powerful as CRCW model. *Information Processing Letters*, 36(1):31–36, October 1990.