The Reconfigurable Mesh: Programming Model, Self-Simulation, Adaptability, Optimality, and Applications

Mohammad Manzur Murshed

A thesis submitted for the degree of
Doctor of Philosophy at
The Australian National University

November 1999

Typeset in Palatino by TEX and LATEX 2ε.

Except where otherwise indicated, this thesis is my own original work and has not been submitted for any other degree.

Mohammad Manzur Murshed

26 July 1999

Dedicated to my parents for all their love and inspiration.

Acknowledgments

I would like to express my sincerest gratitude and profound indebtness to my supervisors Professor R. P. Brent, Professor E. V. Krishnamurthy, and Dr. M. Hegland for their constant guidance, insightful advice, helpful criticism, valuable suggestions, commendable support, and endless patience towards the completion of this thesis. I feel very proud to have worked with them and without their inspiring enthusiasm and encouragement this work could not have been completed.

My special thanks to Dr. J. B. Millar for his constant support in financing my conference activities. I would also like to thank Dr. B. B. Zhou for his valuable time in reviewing the final manuscript.

The Computer Sciences Laboratory at the Australian National University has provided an excellent environment for my research. I spent many enjoyable hours with department members and fellow students. I would like to thank all the staff at the Computer Sciences Laboratory who have been extremely supportive through their friendship and encouragement. I particularly thank Ms. M. Moravec for her prompt administrative help.

The support I have received from the Australian national University in the form of scholarships and various professional development programs is gratefully acknowledged.

Many thanks are due to my wonderful wife Srabony for her love, care, patience, and understanding during this work. I would also like to thank my little boy Sharon for keeping my inquisitiveness alive by constantly asking questions at home.

Last but by no means least, I am thankful to God for the talents and abilities I was given that made it possible to write this thesis.

Abstract

This thesis contributes to the acceptance of the reconfigurable mesh as the architecture of the next generation massively parallel computer by focusing on programming, algorithmic, scaling, and optimality issues. A new programming model is defined for 3-dimensional reconfigurable meshes which is capable of reusing programs as subroutine calls in different axis-orientations within restricted regions. This programming model is supported by a new serial simulator. By exploiting two unique properties of the maximal contour, a number of constant time algorithms are developed to compute the contour of maximal elements of a set of planar points on ordinary as well as restricted and unrestricted reconfigurable meshes of various dimensions. A new generic self-simulation algorithm is developed which can self-simulate some restricted models of the reconfigurable mesh with asymptotically optimal slowdown and for which the constant factor associated with the optimal slowdown is much less than that of the existing self-simulation algorithms. Self-simulation is then devalued as an efficient strategy for solving the problem of scaling down algorithms by showing that even with optimal slowdown, the resultant algorithms fail to remain AT^2 optimal when a large reconfigurable mesh is self-simulated on a smaller mesh for which AT^2 optimal algorithms exist. The idea of developing adaptive algorithms, which can run on reconfigurable meshes of variable sizes and aspect ratios, without compromising AT^2 optimality, is introduced as an alternative strategy for solving the problem of scaling down algorithms. Another frontier is opened up in the study of adaptive algorithms by showing an example where a new efficient optimal algorithm on the ordinary mesh is extracted from an adaptive algorithm on the reconfigurable mesh.

Notation

The notation used in this thesis is similar to that used in most technical texts and should not present any difficulty for readers with a modest technical background.

In accordance with most recent computer science texts, we use the following notations to describe the complexity bounds of algorithms:

I. $f(N) = O(g(N))$ denotes the fact that there exist constants $c > 0$ and N_0 such that $f(N) \leq cg(N)$ for all $N \geq N_0$.

II. $f(N) = \Omega(g(N))$ denotes the fact that there exist constants $c > 0$ and N_0 such that $f(N) \geq cg(N)$ for all $N \geq N_0$.

III. $f(N) = \Theta(g(N))$ denotes the fact that there exist constants $c_1 > 0$, $c_2 > 0$, and N_0 such that $c_1 g(N) \leq f(N) \leq c_2 g(N)$ for all $N \geq N_0$.

IV. $f(N) = o(g(N))$ denotes the fact that for any value of $c > 0$, there exist a value of N_0 such that $f(N) < cg(N)$ for all $N \geq N_0$.

For example, if $f(N) = 7N$ and $g(N) = N^2/3$, then $f(N) = O(N)$, $g(N) = \Omega(N^2)$, $g(N) = \Theta((f(N))^2)$, and $f(N) = o(g(N))$.

Contents

Chapter 1

Introduction

It is well known that interprocessor communications and simultaneous memory accesses often act as bottlenecks in present-day parallel computers. Bus systems have been introduced recently to a number of parallel architectures to address these issues. Among these parallel architectures, the *reconfigurable mesh* (Section 2.2) has drawn much attention because of its underlying mesh topology which is considered as one of the simplest models of parallel computing because of the locality of the communication and the regularity of the design. This makes it ideally suitable for VLSI embedding.

The fundamental advantage of the reconfigurable mesh over the ordinary mesh-connected computer (Section 2.1) is the ability of any processor of the reconfigurable mesh to act as an integral part of the bus system besides its normal computational functions. In an ordinary mesh, a processor is connected to its neighbouring processors with fixed bus segments, and in unit time a processor can send a message only to its neighbours. The case is quite different for the reconfigurable mesh where a processor can reconfigure itself to act as switch to connect mutually exclusive subsets of the fixed buses. Therefore, the reconfigurable mesh allows a message to propagate through several processors in unit time.

Consider any possible communication path between two arbitrary processors PE_i and PE_j of a reconfigurable mesh. If each processor along the path, except processors PE_i and PE_j, acts as a switch to connect the fixed bus segments connected to the processor along the path, processor PE_i can transmit a message to processor PE_j in unit time regardless of the number of processors along the path. This key assumption of unit-time delay in message propagation leads in achieving $O(1)$ communication di-

ameter (Definition 2.2) for the reconfigurable mesh. This has been exploited by many researchers to develop constant time algorithms on this particular architecture.

Although the reconfigurable mesh appears to be a powerful contestant for the next generation of massively parallel computers, most of the research works on it is still at the pen and paper stage. A commercially viable parallel computer based on the reconfigurable mesh architecture depends on solving the following problems arising, ironically, from the strength of reconfiguration:

- Maresca [58] has expressed his concern that the reconfigurable mesh is so flexible and powerful that it has turned out to be nearly impossible to derive high level programming models preserving such flexibility and power.
- Consider the *problem of scaling down algorithms* where an algorithm written on a virtual mesh of size $\alpha P \times \beta Q$ is to be executed on a physically available mesh of size $P \times Q$, where α and β are integers greater than 1. Self-simulation is the obvious way to address this problem. A simulation program takes the responsibility to execute each step of the original algorithm on the physical mesh which self-simulates the virtual mesh through some sort of processor mapping. In ordinary "non-reconfigurable" mesh architecture, self-simulation can be done with optimal slowdown $\alpha\beta$ by letting each processor of the physical mesh self-simulate $\alpha\beta$ processors. Self-simulation in the reconfigurable mesh architecture is not so obvious [31]. The difficulty stems mainly from the fundamentally different way in which some computations are performed, exploiting the strength of reconfigurability in a manner known as the *configurational computing* [115]. A configurational computation, typical to many algorithms, is as follows:

  1 Processors configure themselves locally to establish a global pattern of buses interconnecting the processors;

  2 A designated processor issues a special signal at a fixed position in the bus structure;

  3 The processors deduce an answer depending on where the signal arrives;

  Note that the above configuration computation does not involve any computational functionalities of the processor.

It is still an open problem whether self-simulation of the reconfigurable mesh is possible with optimal slowdown unless the reconfiguring capability of the processors is severely restricted. It is now widely accepted that an additional polylogarithmic factor is inherent in the slowdown of self-simulating the unrestricted reconfigurable mesh [4].

The aim of this thesis is to contribute to the acceptance of the reconfigurable mesh as the architecture of the next generation massively parallel computer. This thesis thus focuses on the following seemingly loosely connected aims relevant to the reconfigurable mesh architecture:

- To develop a programming model, capable of reusing programs, for 3-dimensional reconfigurable meshes by means of a suitable programming language, and to construct a serial simulator to test programs written in that language. Some serial simulation works [68, 99, 119] have been published for 2-dimensional reconfigurable meshes but the extent of the programming models developed there is in the vicinity of merely visualisation to aid in evaluation and debugging of algorithms. These programming models lack the capability of reusing programs, a key feature of any successful programming language.

- To develop constant time algorithms to compute the contour of maximal elements of a set of planar points [24] on reconfigurable meshes of various dimensions. This is an important computational geometry problem and so far no work has been done on reconfigurable meshes. The motivation for this work is the work of Jang *et al.* [37] where a non-recursive generic algorithm (Section 5.1) is used to solve a number of computational geometry problems in constant time on reconfigurable meshes.

- To develop efficient optimal self-simulation algorithms for some restricted reconfigurable mesh models. Self-simulation algorithms [4] with optimal slowdown have already been developed for some restricted reconfigurable meshes. We believe that these self-simulation algorithms have high constants associated with the highest order terms in their slowdown functions. Also, the simplicity of configurational computing is sacrificed. We also believe that the self-simulation

algorithms in [4] are unnecessarily complex if applied to self-simulate many fundamental algorithms like the prefix-sum computation. Our primary goal is to develop simple self-simulation algorithms, preserving the beauty of configurational computing and having lower constants associated with the highest order terms in their slowdown functions. This may require even more restricted reconfigurable mesh models if not achievable for the restricted models used in [4].

- Self-simulation is not necessarily the only way to solve the problem of scaling down algorithms. An alternative solution lies in refining, if necessary redesigning, the algorithm so that it can adapt to reconfigurable meshes of various sizes and aspect ratios. We explore this idea of *self-scalable* algorithms with a comparative study against the conventional self-simulation method. To our knowledge, self-scalable algorithms have not been studied before as a prospective solution to the problem of scaling down algorithms on the reconfigurable mesh. Similar idea has recently been studied on linear arrays with reconfigurable pipelined bus systems [111].

For the sake of completeness of this introductory chapter, the main contributions of the thesis are outlined below:

i) Defining a programming model for 3-dimensional reconfigurable meshes by means of a new programming language, *Reconfigurable Mesh Parallel C* (RMPC), which has the unique capability of reusing programs in different axis-orientations within restricted regions.

ii) Developing a full-scale serial simulator, *Reconfigurable Mesh SIMulator* (RMSIM), which can execute RMPC programs.

iii) Introducing two unique properties of the maximal contour of a set of planar points which are exploited to develop efficient optimal parallel algorithms to compute maximal contours on the linear array, the ordinary mesh, and the reconfigurable mesh of various dimensions.

iv) Presenting two new restricted models of the reconfigurable mesh, the *Monotonic-Bus* (MB) model and the *Piecewise-Monotonic-Bus* (PMB) model, where restrictions are imposed on the global characteristics of the buses configured.

v) Developing a simple generic *self-simulation* algorithm which can self-simulate the MB model optimally and the PMB model asymptotically optimally. Although our algorithm self-simulates more restricted models, the constant associated with the optimal slowdown is much lower than that of any of the previously developed self-simulation algorithms for restricted reconfigurable meshes.

vi) Devaluing the self-simulation technique as an efficient method of scaling down algorithms by pointing out that the resultant algorithms are not necessarily AT^2 optimal when AT^2 optimal algorithms are self-simulated on reconfigurable meshes even with optimal slowdown.

vii) Introducing the idea of developing *adaptive* algorithms, as an alternative method to self-simulation for scaling down algorithms. The adaptive algorithms can run on reconfigurable meshes of variable sizes and aspect ratios while maintaining AT^2 optimality.

viii) Developing adaptive algorithms for sorting items and computing the contour of maximal elements of a set of planar points.

ix) Conjecturing that in developing adaptive algorithms, it is sufficient to configure buses whose lengths are bounded solely by the parameter which represents how much the mesh is filled with data initially. To support our conjecture our adaptive algorithms are successfully transformed on the *constrained* reconfigurable mesh where buses of at most a fixed length are allowed to be formed.

x) Arguing that the study of adaptive algorithms on reconfigurable meshes for solving any specific problem will lead in developing new efficient algorithms on mesh-connected networks—reconfigurable or ordinary ones. This argument is supported by extracting a new AT^2 optimal maximal contour algorithm on the ordinary mesh, from our adaptive maximal contour algorithm. The new

algorithm has lower constant associated with the highest order term in the complexity function than the existing optimal algorithm.

As mentioned in the beginning of this chapter, this thesis tries to weave a thread across a number of partly-related and partly-independent themes presented in the core chapters. Literature review, therefore, is distributed among the relevant chapters to retain better cohesion. The thesis is organised as follows:

In Chapter 2 we present background material for the entire thesis. This includes the computational models of the ordinary mesh, the reconfigurable mesh, the linear array, and the reconfigurable linear array. We also provide detailed classification criteria along with various models of the ordinary mesh as well as the reconfigurable mesh. This chapter also discusses configurational computing and the relative power of the reconfigurable mesh with an idealistic parallel computational model PRAM.

In Chapter 3 we define a new programming model for the general 3-dimensional reconfigurable mesh model. The model is expressed by means of a new programming language, named RMPC, which is further supported by a serial simulator, named RMSIM, to simulate parallel algorithms written in RMPC on a 3-dimensional reconfigurable mesh. The work of this chapter was presented in the *International Symposium on Audio, Video, Image Processing and Intelligent Applications*, Baden-Baden, Germany, 1998 [75].

In Chapter 4 we provide a background in sorting on mesh-connected networks including the linear array, the ordinary mesh, and the reconfigurable mesh. We concentrate mainly on optimal algorithms in the sense of limitations imposed by communication diameter as well as AT^2 measure. Sorting algorithms discussed in this chapter are used extensively in most of the algorithms developed in Chapters 5, 7, and 8.

In Chapter 5 we present two unique properties of the contour of the maximal elements of a set of planar points which can be exploited to develop efficient parallel maximal contour algorithms. This chapter discusses optimal maximal contour algorithms on the linear array and the ordinary mesh. Three optimal maximal contour algorithms are developed here on reconfigurable meshes of various dimensions. These algorithms were published in [74] and also presented in the *International Conference on*

Parallel and Distributed Systems, Seoul, Korea, 1997 [71].

In Chapter 6 we discuss self-simulation algorithms, for a number of restricted as well as the "unrestricted" general reconfigurable mesh models, where the simulation involves the problem of computing the connected components of graphs. This chapter also defines two new restricted reconfigurable mesh models where restrictions are imposed on the global, rather than the local, characteristics of the buses allowed to be formed. We further develop a new generic self-simulation algorithm, avoiding any computation of connected components, which can self-simulate the new models with asymptotically optimal slowdown, and for which the constant factor associated with the optimal slowdown is much less than that of the algorithms which exploit computations of connected components. This generic self-simulation algorithm was presented in the *2nd International Conference on Computational Intelligence and Multimedia Applications*, Gippsland, Australia, 1998 [73]. Finally we devalue the self-simulation technique as an efficient method of scaling down algorithms by pointing out that the resultant algorithms are not necessarily AT^2 optimal when AT^2 optimal algorithms are self-simulated on reconfigurable meshes even with optimal slowdown.

In Chapter 7 we introduce the idea of developing *adaptive* algorithms, as an alternative method to self-simulation for scaling down algorithms. The adaptive algorithms can run on reconfigurable meshes of variable sizes and aspect ratios while maintaining AT^2 optimality. Two adaptive sorting algorithms and an adaptive maximal contour algorithm are developed using the framework of a generic adaptive algorithm. Two of these algorithms were presented in the *10th IASTED International Conference on Parallel and Distributed Computing and Systems*, Las Vegas, U.S.A., 1998 [72] and the remaining one will be presented in the *International Symposium on Intelligent Multimedia and Distance Education*, Baden-Baden, Germany, 1999 [70]. In this chapter we also propose a conjecture stating that it is sufficient to configure buses of length $O(k)$ in an arbitrary adaptive algorithm where k represents how much of the mesh is filled with data initially, and the conjecture is then supported by transforming our adaptive algorithms on k-constrained reconfigurable meshes. This work was presented in the *International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, U.S.A., 1999 [69].

In Chapter 8 we develop a new AT^2 optimal maximal contour algorithm on the ordinary mesh, based on our adaptive maximal contour algorithm developed in Chapter 7. The new algorithm has lower constant associated with the highest order term in the complexity function than the existing optimal algorithm. This opens a new frontier in the study of adaptive algorithms. This work was presented in the *International Conference on Computer and Information Technology*, Dhaka, Bangladesh, 1998 [76].

In Chapter 9 conclusions are given with reference to possible future extensions of the results.

Chapter 2

Background

The aim of this chapter is to define the computational model of the reconfigurable mesh computer which is the fundamental parallel architecture on which this thesis is based. To support our ideas and results on the reconfigurable mesh, we also make extensive use of the ordinary mesh and the linear array of processors throughout the thesis. This chapter, therefore, includes discussions on these mesh-connected computers as well.

Definitions of some fundamental properties of a network of processors (not necessarily mesh-connected), which are widely used throughout the thesis, are given below:

Definition 2.1 *The* bandwidth *of a network is the maximum number of bits that can be transmitted over any bus segment at a time. Although no upper limit exists (besides the physical limitation) for the bandwidth of a network $\mathcal{N}$ of P processors, it is a common practice to assume the bandwidth of $\mathcal{N}$ must be $\geq \log P$, number of bits necessary to distinguish P processors. This is also referred to as the* minimum bandwidth *requirement of a network.*

Definition 2.2 *Consider a network $\mathcal{N}$ of M processors where processors are numbered from 0 to $M-1$. Let $d_{i,j}$ denote the* distance *between the pair processors PE_i and PE_j which is the smallest number of wires (fixed bus segments) that have to be traversed in order to get from processor PE_i to processor PE_j for all $0 \leq i, j < M-1$. The* communication diameter *of $\mathcal{N}$ is $\max_{\forall i \forall j} d_{i,j}$.*

The communication diameter of a network often gives a lower bound on the time which it takes to perform a calculation where information in a processor might have

to be used by some distant processor.

Definition 2.3 *The* bisection width *of a network is the minimum number of wires that must be removed from it so that the network becomes two disjoint subnetworks with identical (within 1) number of processors.*

The bisection width of a network is also a critical factor in determining the time with which the network can perform a calculation where the data contained in one half of the network may be needed by the other half before the calculation can be completed.

The organisation of this chapter is as follows. In Section 2.1 we describe the computational model of the ordinary mesh parallel computer. Some models of the ordinary mesh and a brief definition of the linear array of processors are also included in Section 2.1. In Section 2.2 we give details of the reconfigurable mesh parallel computer by defining its general computational model, discussing classification criteria, presenting various models, discussing the power of reconfigurable computing, and defining the reconfigurable array of processor.

2.1 The Ordinary Mesh

The *mesh-connected computer* (*mesh*) is one of the simplest models of parallel computers. The locality of the communication, the simplicity of the interconnection pattern, and the regularity of the design make it ideally suited to VLSI implementation, easy to program and to scale up.

In many cases, throughout the thesis, we term the mesh as *ordinary* to distinguish it from the reconfigurable mesh.

A mesh of size $r \times c$ is a parallel computer with rc processors which are arranged in a $r \times c$ lattice. Let $PE_{i,j}$ denote the processor at row i and column j of a mesh of size $r \times c$ for all $0 \leq i < r$, $0 \leq j < c$ and let processor $PE_{0,0}$ reside in the north-western corner. Every processor $PE_{i,j}$, for all $0 \leq i < r$, $0 \leq j < c$, is connected, via unit-time communication links, to its four neighbours, processors $PE_{i\pm1,j\pm1}$, assuming they exist (Figure 2.1).

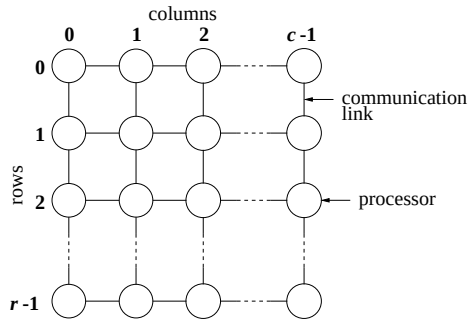

Figure 2.1: An ordinary mesh of size $r \times c$.

Each processor has a fixed number of registers (words) and can perform standard arithmetic and logical operations in unit time. Each processor can also send or receive a word of data from each of its neighbour in unit time.

The minimum bandwidth, the communication diameter, and the bisection width of an ordinary mesh of size $r \times c$ are $\log rc$, $r + c - 2$, and $\min(r, c)$ or $\min(r, c) + 1$ respectively.

In Section 2.1.1 we classify the ordinary mesh, on the directionality of the communication links, which is extensively used in the discussion of algorithms and their complexity bounds in Section 4.3.

2.1.1 Ordinary Mesh Models

Three models of the ordinary mesh, proposed so far based on the directionality of the communication links, are as follows:

Bidirectional Model: Two neighbouring processors can exchange data simultaneously over the communication link connected between these processors. To achieve this it is assumed that there are two links between every pair of neighbouring processors.

Unidirectional Model: In this model two neighbouring processors cannot exchange data simultaneously. Therefore, swapping of data between two neighbouring processors takes at least two units of time.

Strict Unidirectional Model: In this unidirectional model, all processors that simul-

taneously transfer data to a neighbouring processor do so to the same neighbour, i.e., all active processors transfer data to their north neighbour, or all to their south neighbour, etc.

2.1.2 The Linear Array

The easiest way to express the *linear array* is to define it as a dimensionally restricted ordinary mesh. A linear array of N processors is an ordinary mesh of size $1 \times N$ or $N \times 1$. In the first case, each processor has at most its east and west neighbours; while in the second case, each processor has at most its north and south neighbours.

2.2 The Reconfigurable Mesh

This section is organised as follows. In Section 2.2.1 we define the general computational model. Classification criteria, based on which various models of the reconfigurable mesh are defined, are discussed in Section 2.2.2. In Section 2.2.3 we present various models of the reconfigurable mesh in brief. The comparative power of the reconfigurable mesh architecture is evaluated in Section 2.2.4. In Section 2.2.5 we define the reconfigurable linear array.

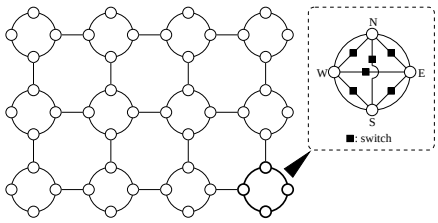

Figure 2.2: A reconfigurable mesh of size 3×4.

2.2.1 The General Computational Model

The *reconfigurable mesh* is primarily a two-dimensional mesh of processors connected by reconfigurable buses. In this parallel architecture, a processor element is placed at the grid points as in the usual mesh connected computers. Processors of a reconfigurable mesh of size $X \times Y$ are usually denoted by $PE_{i,j}$, $0 \leq i < X$, $0 \leq j < Y$. Let processor $PE_{0,0}$ reside in the north-western corner of the mesh. As in the ordinary

mesh, each processor is connected to at most four neighbouring processors through fixed bus segments connected to four I/O ports **E** & **W** along dimension *x* and **N** & **S** along dimension *y*. These fixed bus segments are building blocks of larger bus components which are formed through switching, decided entirely on local data, of the six internal switches (see Figure 2.2) between the four I/O ports of each processor. The fifteen possible interconnections of I/O ports through switching are shown in Figure 2.3. The connection patterns are represented as $\{p_1, p_2, \ldots\}$, where each of p_i represents a group of switches connected together such that $\bigcup_{\forall i} p_i = \{\mathbf{N}, \mathbf{E}, \mathbf{W}, \mathbf{S}\}$. For example, {**E**,**W**,**NS**} represents the connection pattern with ports **N** and **S** connected and ports **E** and **W** unconnected.

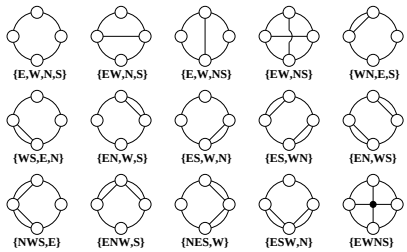

Figure 2.3: All 15 Possible interconnections between the four I/O ports of a processor in the reconfigurable mesh.

A reconfigurable mesh operates in the single instruction multiple data (SIMD) mode. Besides the reconfigurable switches, each processor has a computing unit with a fixed number of local registers. The processors of a reconfigurable mesh operate synchronously and a single step of a reconfigurable mesh is composed of the following four substeps in sequence:

BUS substep: Every processor switches the internal connectors between I/O ports by local decision.

WRITE substep: Along each bus, one or more processors on the bus transmit a message of length bounded by the bandwidth of the fixed bus segments as well as the switches. These processors are called the *speakers*.

READ substep: Some or all the processors connected to a bus read the message transmitted by a single speaker. These processors are called the *readers*.

COMPUTE substep: A constant-time local computation is done by each processor.

The minimum bandwidth, the communication diameter, and the bisection width of a reconfigurable mesh of size $X \times Y$ are $\log XY$, 1, and $\min(X,Y)$ or $\min(X,Y)+1$ respectively.

2.2.2 Classification Criteria

Various reconfigurable mesh models, as discussed in Section 2.2.3 are usually classified by the following key criteria:

Width: It refers to the data width of the processors and thus also refers to the register capacity and bus/switch bandwidth. The two classes of models which have been proposed are *bit* and *word* models. Consider a reconfigurable mesh of size $X \times Y$. In the word model [64–66] the width is assumed to be $O(\log XY)$, a word, while in the bit model [34] the width is considered to be $O(1)$. Thus in the word model, every processing element has a fixed number of $O(\log XY)$-bit registers and $O(\log XY)$-bit ALU and the bandwidths of the switches as well as buses are assumed to be $O(\log XY)$ bits. In the bit model 1-bit ALUs are used on $O(1)$-bit registers and the switches and buses operate with only $O(1)$ bit bandwidth.

Delay: One critical factor in the complexity analysis of reconfigurable algorithms is the time needed to propagate a message over a bus. In the *unit-time delay* model (most common) it is assumed that in any configuration any message can be transmitted along any bus in constant time, regardless of the bus length. This assumption, based on which a large number of algorithms with constant time complexity are developed, is theoretically false, as the speed of signals carrying information is bounded by the speed of light. This partially explains why reconfigurable meshes have not gained wide acceptance initially. Recently some VLSI implementations of reconfigurable meshes have demonstrated that the broadcast delay, though not a constant, is nevertheless relatively small in terms of machine cycles. For example, only 16 machine cycles are required to broadcast on a 10^6 processor YUPPIE (Yorktown Ultra Parallel Polymorphic Image

Engine) [53, 59]. GCN (Gated-Connection Network) [104] has even shorter delays by adopting precharged circuits. Broadcast delay can further be reduced by using optical fibre for the reconfigurable bus system and electrically controlled directional coupler switches as proposed in [6].

In the *log-time delay* model [66] it is assumed that each broadcast takes $\Theta(\log s)$ steps to reach all the processors connected to a bus, where s is the maximum number of switches in a minimum switch path between two processors connected on the bus. Although this assumption sacrifices the property of unit communication diameter, it is also not realistic in terms of the speed of light.

Bus Access: At each step, a bus may take one of the following three state:

- *Idle*, no processor transmits;
- *Speak*, there is only one speaker;
- *Ambiguous*, there are more than one speakers.

In the most common *exclusive-write* model, the *ambiguous* state is considered to be an *error* state. The *common-write* model [12, 66], handles the *ambiguous* state in a different manner. It allows multiple processors to simultaneously broadcast to the same bus so long as they all broadcast the same message. Otherwise the *ambiguous* state is considered to be an *error* state. In both the models it is assumed that the *error* state is detectable by the processors and the bus carries arbitrary values. The *concurrent-write* model [120] assumes no *ambiguous* state at all. It also allows multiple processors to simultaneously broadcast to the same bus and the bus carries the *wired-or* of all the messages.

Bus Direction: The buses of reconfigurable meshes are generally assumed to be undirected[1] in the sense that data can move in both direction. In [4, 5, 8, 58] a different model[2] has been proposed where each undirected bus is replaced by two directed buses of opposite directions. It can be easily seen that this directed model can simulate any configuration of the undirected mesh. The opposite is

[1] Resembling the unidirectional mesh model.
[2] Resembling the bidirectional mesh model.

not true [8]: a directed graph cannot be simulated by an undirected graph as signals will sometimes have to propagate backwards. For example, any undirected graph that is used to simulate the directed graph $x \rightarrow z$, $y \rightarrow z$, must make x and y connected, even though they were not initially connected.

Connection Patterns: Each processor can set the connection between its four ports based on local data. There are a total of 15 different connection patterns possible as shown in Figure 2.3. Different models have been proposed in [4–6, 31, 73, 116] which differ mainly in the number of allowed connection patterns. In [56], the reconfigurable mesh is divided into the *cross-over* and the *non-cross-over* models based on whether the connection pattern $\{\mathbf{NS}, \mathbf{EW}\}$ is allowed or not. In some instances the *cross-over* model have been shown to be more powerful than the *non-cross-over* model.

Mesh Dimension: A reconfigurable mesh is usually assumed to be two dimensional. But reconfigurable meshes of higher dimensions can also be constructed in a similar way. For example, in a 3-dimensional reconfigurable mesh of size $X \times Y \times Z$, processors are denoted by $PE_{i,j,k}$, $0 \leq i < X-1$, $0 \leq j < Y-1$, $0 \leq k < Z-1$. Each processor of a 3-dimensional reconfigurable mesh has two additional ports **U** and **D** along dimension z.

Throughout the thesis, we assume the following assumptions and notations, if not stated otherwise:

- A reconfigurable mesh is a 2-dimensional undirected word model with unit-time delay and exclusive-write bus access where each processor is allowed to configure any of the fifteen possible interconnections.
- We have used the labels "b:", "w:", "r:", and "c:" to denote BUS, WRITE, READ, and COMPUTE substeps respectively of a step in an algorithm on the reconfigurable mesh. For example see Algorithm 5.4.
- The word *mesh* is used to refer to both an ordinary mesh and a reconfigurable mesh as long as no ambiguity arises.

- For the sake of convenience, we have used the notation $PE_{*,i_2,i_3,\ldots,i_k}$ to denote the set of processors $\forall i_1 : PE_{i_1,i_2,i_3,\ldots,i_k}$. Similarly $PE_{*,i_2,*,i_4,\ldots,i_k}$ denotes the set of processors $\forall i_1 \forall i_3 : PE_{i_1,i_2,i_3,\ldots,i_k}$.

2.2.3 Reconfigurable Mesh Models

In this section we present a number of reconfigurable mesh models which have appeared in the literature.

2.2.3.1 The PARBS Model

The Processor Array with a Reconfigurable Bus System (PARBS) [116] is the most general and powerful model of the reconfigurable mesh. In fact, this is the *de facto* model we use throughout the thesis. PARBS is defined for 2- and 3-dimensions with each processor having 4 and 6 ports respectively. Processors are connected to a grid-shaped reconfigurable bus systems. Any configuration of this bus system, that is derivable by properly establishing the local connection among the ports within each processor, is allowed. PARBS assumes *unit-time* delay, *word* data width and *exclusive write* on the bus.

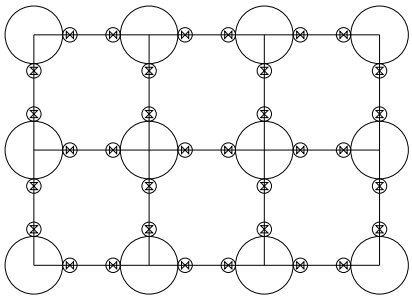

Figure 2.4: A 3×4 RMESH.

2.2.3.2 The RMESH Model

Reconfigurable MESH (RMESH) [65, 66] is a 2-dimensional mesh of processors which differ from 2-dimensional PARBS in the placement of switches. In 2-dimensional PARBS, six switches are used to connect four I/O ports internally in all possible 15

configurations. But in the RMESH model, all the four ports are always connected internally and four switches are used, one with every port, simply to connect/disconnect the fixed bus segment attached to a port as shown in Figure 2.4. Any two adjacent processors can now be connected together if both the processors connect the switch attached to the fixed bus segment between them while they can disconnected from each other if any one of the processors disconnect its switch. Among the fifteen configurations in Figure 2.3, the patterns {**EW**,**NS**}, {**ES**,**WN**}, and {**EN**,**WS**} are not achieveable in RMESH.

Li and Stout [54] have conjectured that PARBS is more powerful than RMESH by showing the performance difference of the computation of exclusive-or (XOR) on PARBS and RMESH models. On an $n \times n$ PARBS where each processor is holding a single bit, the XOR of all the bits can be computed in $O(1)$ time but an RMESH of the same size takes $O(\log\log n)$ time. The reason it is conjectured that PARBS is more powerful than RMESH is the fact that RMESH does not allow *cross-over* connection pattern {**NS**,**EW**}. Mackenzie [56] has proved a lower bound of $\Omega(\log n)$ for computing parity of n bits on an RMESH of size $k \times n$ where $0 < k < n$; while Li and Stout [54] have shown that the problem can be solved in constant time on a PARBS of size $k \times n$ where $k \geq 3$.

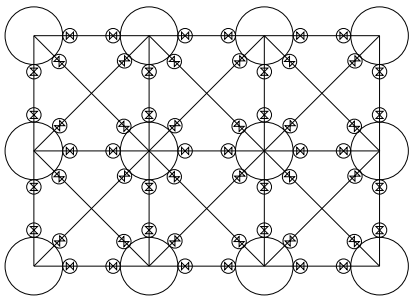

Figure 2.5: An 8-connected RMESH of size 3×4.

To make RMESH as powerful as PARBS, Shi *et al.* [103] have suggested an 8-connected RMESH, as shown in Figure 2.5, in place of usual 4-connected RMESH. Shi *et al.* [103] have also shown that 8-connected RMESH is equivalent to a PARBS by demonstrating that a PARBS can be simulated by an 8-connected RMESH and an 8-connected RMESH can be simulated by a PARBS without any increase in time com-

plexity.

2.2.3.3 The HV-RM Model

In the HV-RM model [4], buses are formed either along rows (horizontally) or along columns (vertically), but may not contain fixed bus segments from both dimensions. The HV-RM model thus allows a processor to configure only the following four patterns: {**E**,**W**,**N**,**S**}, {**EW**,**N**,**S**}, {**E**,**W**,**NS**}, and {**EW**,**NS**}.

2.2.3.4 The LRM Model

In the LRM model [4], a processor may partition the fixed bus segments connected to it into any combination of connected pairs and singletons. Hence buses are only linear (or just a cycle), i.e., a fixed bus segment is attached to at most one other fixed bus segment at each end and the global configuration is a partition of the network into a set of edge-disjoint linear buses. Thus among the fifteen configurations in Figure 2.3, the patterns {**NWS**,**E**}, {**ENW**,**S**}, {**NES**,**W**}, {**ESW**,**N**}, and {**EWNS**} are not allowed.

2.2.3.5 The FR Model

The FR model [31] is a restricted version of the general model that allows only two of the fifteen configurations in Figure 2.3, the *fusing* pattern {**EWNS**} and the *cross-over* pattern {**EW**,**NS**}. Because of the above restriction in interconnecting ports of a processor, it may be assumed without loss of generality that each processor of an FR mesh has only two ports, the *vertical port* (**NS**) and the *horizontal port* (**EW**).

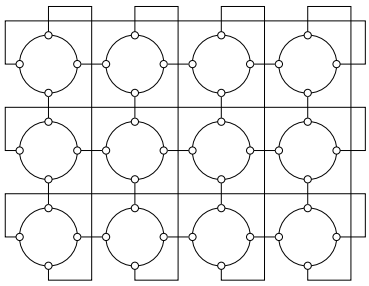

Figure 2.6: The Polymorphic Torus Network.

2.2.3.6 The PTN Model

The Polymorphic Torus Network [53, 59] is identical to the 2-dimensional PARBS architecture except that the rows and columns of the underlying mesh wrap around as shown in Figure 2.6.

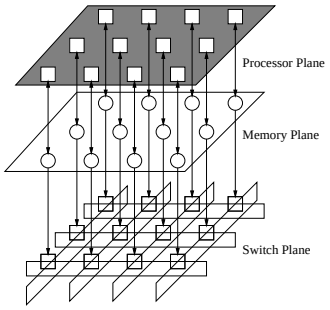

Figure 2.7: The Polymorphic Processor Array.

2.2.3.7 The PPA Model

The logical architecture of the Polymorphic Processor Array (PPA) [58], as shown in Figure 2.7, consists of a stack of three planes, respectively called processor plane, memory plane and switch plane. The processor and memory planes are 2-dimensional arrays of processors and registers respectively. The switch plane is a torus of switch boxes. PPA assumes *unit-time* delay and *word* data width. To make PPA more realistic and cost-effective, buses/switches are made directional and switch box implements only the interconnections between opposite ports as shown in Figure 2.8. Except for the directional buses, PPA is very similar to the HV-RM mesh model.

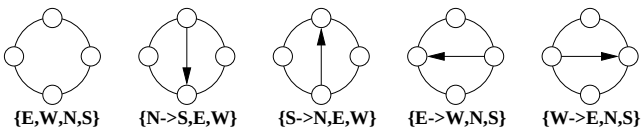

Figure 2.8: Possible 5 directional interconnections between the four I/O ports of a switch box in PPA.

2.2.3.8 The CAAPP Model

Content Addressable Array Parallel Processor (CAAPP) [120] is a square-grid array of 1-bit serial processors intended to perform low-level image processing tasks. Each processing element is linked through a four-way (**E**,**W**,**N**,**S**) switch which allows certain types of long-distance communication to take place quickly. One of the means of communication among CAAPP processors involves the *Coterie Network* (Figure 2.9) which is very similar to the RMESH in allowing specific connection patterns. CAAPP allows multiple processors to write to the same isolated processor group, coterie, at the same time and the collision on the bus is resolved by the logical OR of the output bits.

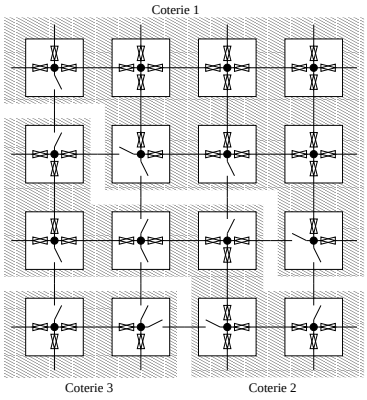

Figure 2.9: The Coterie Network.

2.2.3.9 The Bit Model

Jang *et al.* proposed a Bit Model [34] of the reconfigurable mesh which can simulate most of the word based models of the reconfigurable mesh in asymptotically the same amount of time using the same VLSI area. Each processor consists of six bit-level switches, local bit storage and a 1-bit ALU. The switches can realize all of the possible 15 connection patterns shown in Figure 2.3. For obvious reason the buses can carry only $O(1)$ bits of data.

2.2.3.10 The *k*-Constrained Model

It has already been stated in Section 2.2.2 that *unit-time delay* is a theoretically false assumption. To account for this, the *k*-constrained reconfigurable mesh is proposed in [10, 11] where buses of length at most *k* are allowed. Here *k* is assumed to be a constant. The *k*-constrained reconfigurable mesh is quite different from the *log-time delay* model. It can be argued that the *k*-constrained reconfigurable mesh model is asymptotically no faster than the ordinary mesh, since for large M and N, a *k*-constrained reconfigurable mesh of size $M \times N$ is at best *k* times faster than an ordinary mesh of the same size.

2.2.4 Power of the Reconfigurable Mesh

The power of the reconfigurable mesh is discussed here along two different perspectives: configurational computing (Section 2.2.4.1) and relative strength of the reconfigurable mesh over PRAM architecture (Section 2.2.4.2).

2.2.4.1 Configurational Computing

Certainly communication diameter 1, in assuming *unit-time delay* broadcast, and the reconfigurability of processors into bus segments are the key powers of the reconfigurable mesh. Based on this, a large number of highly efficient algorithms have appeared in the literature with attainable optimal time complexities on mesh topology. Constant time algorithms have been developed for sorting, routing, and ranking [7, 10, 20, 26, 30, 36, 40, 44, 63, 83, 85, 97, 105, 118], computing arithmetic [27, 39, 79, 86, 87, 93, 94], solving computational geometry and graph problems [10, 13, 17, 18, 29, 37, 42, 48, 49, 64, 66, 81, 116], image processing [12, 22, 38, 41, 64, 66], and solving various problems of interest [15, 19, 21, 78]. Reconfiguration of buses plays such an important role in these algorithms that Wang [115] uses a special name *Configurational Computing* to denote the inherent strategy of these algorithms where computation is done, as far as possible, exploiting network configurations rather than performing arithmetic computations. The following example illustrates configurational computing:

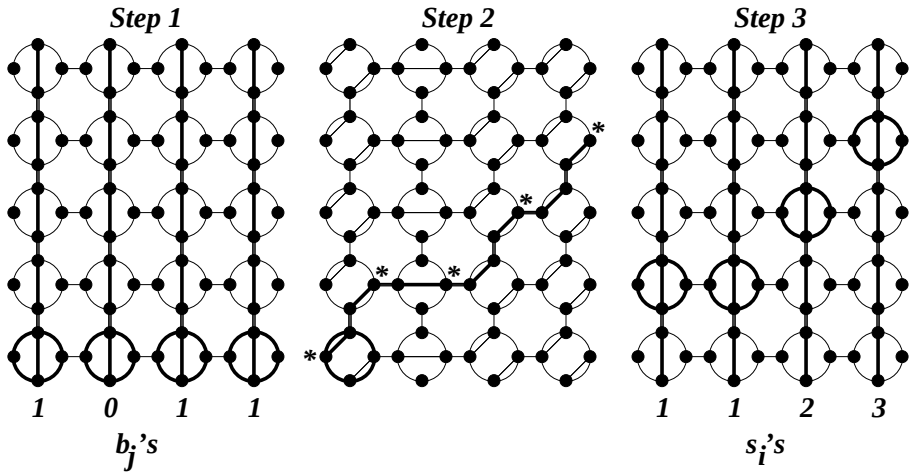

Figure 2.10: Three steps [79] in computing prefix-sum of 4 bits on a reconfigurable mesh of size 5×4.

Given a binary sequence, b_j, $0 \leq j < N$, the *prefix-sum computation* is to compute, $\forall i : 0 \leq i < N$, $s_i = b_0 + b_1 + \cdots + b_i$. In Figure 2.10 prefix-sum of the binary sequence {1, 0, 1, 1} is computed on a reconfigurable mesh of size 5×4. The speakers and the buses which carry the transmitted messages are shown in thick lines. Bits b_j's are assumed to be distributed one bit per processor on the first row. In step 1, all the processors on the first row broadcast b_j's along the column. In step 2, all the processors that have just received the bit '1', assume {**ES**,**NW**} configuration while the rest of the processors assume {**EW**,**N**,**S**} configuration. Now, the left most processor on the first row transmits a special message '*' through port **W** and all the processors read in port **E**. In step 3, only the processors which have just read in the special message '*', exactly one processor per column, transmit their row-index values along the columns to the first row.

Note that in the above configurational computing of prefix-sum, no arithmetic addition is performed.

2.2.4.2 Comparison With PRAM

In the literature, the power of reconfigurable mesh is usually compared with that of an idealistic parallel computational model, widely known as the Parallel Random-Access Machine (PRAM) [33]. A PRAM can contain an arbitrary number of synchronous pro-

cessors, each of which is identified by a unique index. All the processors have unit-time access to an arbitrary size shared memory. There are several variations of the PRAM model based on the assumptions regarding the handling of the simultaneous access of several processors to the same location of the shared memory. The Exclusive Read Exclusive Write (EREW) PRAM does not allow any simultaneous access to a single memory location. The Concurrent Read Exclusive Write (CREW) PRAM allows simultaneous access for reading only. Access to a location for a read or a write simultaneously is allowed in the Concurrent Read Concurrent Write (CRCW) PRAM. The three principal varieties of CRCW PRAMs are differentiated by how concurrent writes are handled. The *common* CRCW PRAM allows concurrent writes only when all processors are attempting to write the same value. The *arbitrary* CRCW PRAM allows an arbitrary processor to succeed. The *priority* CRCW PRAM assumes that the indices of the processors are linearly ordered and allows the one with the minimum index to succeed. Obviously the CREW is at least as powerful as the EREW, and the CRCW is the most powerful model.

Simulating a priority CRCW PRAM of k processors and m memory locations in constant time by a 2-dimensional reconfigurable mesh with the number of processors polynomially bounded by k and m, Wang and Chen [117] (and independently Ben-Asher *et al.* [6]) have established the following lemma:

Lemma 2.1 *A 2-dimensional reconfigurable mesh is at least as powerful as a priority CRCW PRAM.* ■

The claim in Lemma 2.1 can further be improved as follows:

Lemma 2.2 *A 2-dimensional reconfigurable mesh is more powerful than a priority CRCW PRAM.*

Proof. As Lemma 2.1 is established, this proof needs only to show that a priority CRCW PRAM is not as powerful as a 2-dimensional reconfigurable mesh which can easily be concluded from the following two facts. 1) parity of n bits can be found in constant time on a 2-D RM [86]; 2) to solve the same problem on a priority CRCW

PRAM with polynomially bounded number of processors requires $\Omega\left(\frac{\log n}{\log\log n}\right)$ time [33, pp. 513]. ■

Often, reconfigurable meshes seem to run $O(\log N)$ times faster than the CREW PRAMs and $O\left(\frac{\log N}{\log\log N}\right)$ times faster than the CRCW PRAMs.

A detailed hierarchy of powers of the PRAM and reconfigurable bus-based models can be found in [5, 109, 112].

2.2.5 Reconfigurable Linear Arrays

A reconfigurable linear array of N processors is a reconfigurable mesh of size $1 \times N$ or $N \times 1$. In the first case, each processor has at most its east and west neighbours; while in the second case, each processor has at most its north and south neighbours.

Chapter 3

A New Programming Model for the Reconfigurable Mesh

In this chapter we define a new programming model for the general 3-dimensional reconfigurable mesh model which is expressed by means of a new programming language. The language, named RMPC, is further supported by a serial simulator, named RMSIM, to simulate parallel algorithms written in RMPC on a 3-dimensional reconfigurable mesh. In the introductory Section 3.1, we discuss background of the programming model as well as serial simulation and visualisation of reconfigurable meshes. The programming model of RMPC is then developed in Section 3.2. In Section 3.3 we present the technical details of the serial simulator RMSIM.

3.1 Introduction

Research on the reconfigurable mesh has concentrated mainly on the development of computation models and on the implementation of experimental systems. The experimental systems YUPPIE [53, 59] and GCN [104] have mainly focused on the efficient implementation of the hardware supporting reconfigurability, and have not produced any programming model.

The lack of a programming model makes the development of algorithms on reconfigurable meshes very difficult; the algorithms are usually formulated as sequences of steps involving the manipulation of switches that control reconfiguration and neither automatic validation nor simulation can be done. A programming model supporting a high level language and a simulator would allow for the automatic validation of the

algorithms and for the automatic performance evaluation of the programs.

Maresca [58] has expressed his concern that the general reconfigurable mesh is so flexible and powerful that it has turned out to be difficult to derive high level programming models preserving such flexibility and power. Maresca, therefore, has pruned the flexibility and power of the general reconfigurable mesh in defining a new reconfigurable mesh architecture, Polymorphic Processor Arrays (PPA) (see Section 2.2.3.7), for which a programming model has been proposed as a basis for the design of a parallel programming language, called PPC (Polymorphic Parallel C), and a compiler/simulator has been implemented [58]. PPC is a well defined programming model but it addresses only issues concerning solely PPA.

Serial simulation of reconfigurable meshes has recently attracted considerable attention in the form of developing efficient visualisation systems for algorithms on 2-dimensional reconfigurable meshes. Sasada [99] has developed a visualisation system in C which can simulate algorithms written in assembly language which is not very user friendly. To overcome this limitation, Watanabe *et al.* [119] have developed a visualisation system in C which can simulate algorithms written in a C-like language. Recently Miyashita *et al.* [68] have presented a visualisation system, written in Java and therefore, it can be executed on various platforms, which also accepts algorithms written in a C-like language.

The primary goal of all the three serial simulation papers [68, 99, 119] was to develop efficient visualisation systems for algorithms on 2-dimensional general reconfigurable mesh model so that algorithms can be constructed, evaluated, and verified. Each of the visualisation systems in [68, 99, 119] has to define a specific programming model of the reconfigurable mesh to represent parallel algorithms on 2-dimensional reconfigurable meshes in the system. But the scope of these programming models remains very limited in the vicinity of merely visualisation. These programming models thus lack the capability of reusing programs, not related to visualisation but a key feature of any successful programming language. Moreover no light has been shed on the simulation of multi-dimensional reconfigurable meshes.

Recently, Ben-Asher *et al.* [4–6] have proposed a systematic approach to expressing algorithms on reconfigurable meshes where each step of an algorithm is divided

into four substeps in sequence as discussed in Section 2.2.1. This has motivated us to define a programming model of the general reconfigurable mesh and to write a serial simulator, RMSIM (Reconfigurable Mesh SIMulator) to support the model. The programming model is expressed by means of a programming language, called RMPC (Reconfigurable Mesh Parallel C).

RMPC and RMSIM are not merely limited only by 2-dimensional reconfigurable meshes. RMPC is specially designed to write programs on a 3-dimensional reconfigurable mesh of any size. To support RMPC, RMSIM can simulate a 3-dimensional reconfigurable mesh of size limited to only by the memory of the serial computer it uses. It is true that RMSIM cannot simulate any arbitrary d-dimensional mesh but it considers important issues concerning simulation of multi-dimensional meshes by covering the 3-dimensional mesh model so that future extension can be made if necessary. In defining RMPC, we concentrate on making the effort of transforming algorithms into equivalent programs straightforward and easy. RMPC is also designed to facilitate reusing of programs and in this direction we introduce the idea of executing a program in different axis-orientations and/or within restricted regions here for the first time (except for author's paper [75]).

The main limitation of RMSIM is in its visualisation capabilities. To aid in visualisation and debugging, RMSIM is only capable of generating a LaTeX picture of any planar segment of the 3-dimensional mesh, at any step, while executing a program. The reason for not providing any 3-dimensional visualisation tool lies in keeping the complexity of writing RMSIM at minimum and also in realising that the complete power of a 3-dimensional reconfigurable mesh has been rarely used in the literature [22]. In most of the cases [6, 20, 63, 118] a weak 3-dimensional model, named *mesh-of-meshes*, has been used in developing algorithms on 3-dimensional reconfigurable meshes and a 2-dimensional visualisation tool is sufficient for debugging in such cases.

Each processor of a 3-dimensional reconfigurable mesh has six ports, **N**, **S**, **E**, **W**, **U**, **D**, which can participate simultaneously in configuring dynamic interconnections. A *mesh-of-meshes* is a weak 3-dimensional reconfigurable mesh where buses are only allowed to be configured on either an XY-plane or a YZ-plane or a ZX-plane as shown in Figure 3.1. In a mesh-of-meshes, a processor is thus allowed to interconnect only

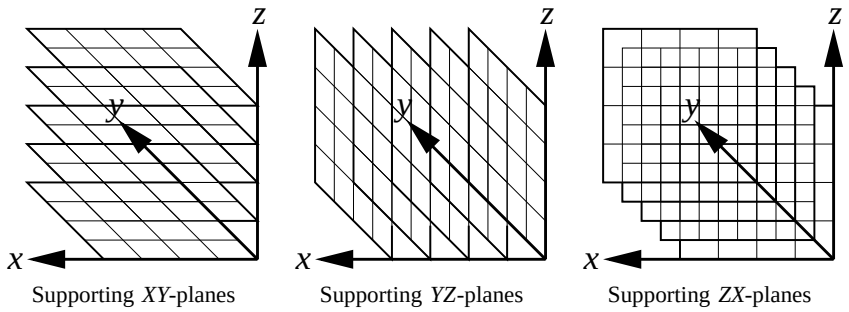

Figure 3.1: A mesh-of-meshes of size $5 \times 5 \times 5$.

the ports on the same plane and all the possible fifteen interconnections (Figure 2.3) among each set of ports {**E**, **W**, **N**, **S**} on XY-plane, {**E**, **W**, **U**, **D**} on XZ-plane, and {**N**, **S**, **U**, **D**} on YZ-plane are allowed.

3.2 The Programming Model: RMPC

In this section, we define a programming model for the reconfigurable mesh-of-meshes model by means of a programming language, called *Reconfigurable Mesh Parallel C (RMPC)*. RMPC is an extension of ANSI C. Like C, RMPC is designed to be small. RMPC programs, therefore, are assumed to be dependent extensively on library programs which are not the integral part of the core of the language. Basic primitives for global initialisation, data manipulation, data communication, and reusing of programs are provided in the form of library functions.

In Section 3.2.1 we provide the construct of the RMPC language. The internal data structures and data manipulation primitives are presented in Section 3.2.2. Basic primitives for data communication and some new concepts in reusing of programs are discussed in Section 3.2.3 and Section 3.2.4 respectively.

3.2.1 The Program Construction

The construction of RMPC language in EBNF (Extended Backus Naur Form) [2] is given in Figure 3.2. *C-string*, *C-variable-declaration*, *C-identifier*, and *C-statement* in Figure 3.2 represent an array of characters, a variable declaration (may be comma separated), an identifier, and a statement respectively which are valid in ANSI C. An

program-file	::=	{ *program* | *subprogram-file* }
subprogram-file	::=	"`::input`" *file-name* "`\n`"
file-name	::=	*C-string*
program	::=	"`::`" *prog-name* "`\n`"
		{ *C-variable-declaration*"`\n`" }
		statements
prog-name	::=	*C-identifier*
statements	::=	[*Start-of-program-statement*]
		[*beGin-of-step-statement*]
		essential-statements
		[*Finish-of-step-statement*]
		{ *essential-statements* }
		[*End-of-program-statement*]
essential-statements	::=	*Bus-statement*
		Write-statement
		Read-statement
		[*Compute-statement*]
Start-of-program-statement	::=	"`S::`" *statement* "`\n`"
beGin-of-step-statement	::=	"`G::`" *statement* "`\n`"
Finish-of-step-statement	::=	"`F::`" *statement* "`\n`"
End-of-program-statement	::=	"`E::`" *statement* "`\n`"
Bus-statement	::=	"`B::`" *statement* "`\n`"
Write-statement	::=	"`W::`" *statement* "`\n`"
Read-statement	::=	"`R::`" *statement* "`\n`"
Compute-statement	::=	"`C::`" *statement* "`\n`"
statement	::=	*C-statement*
		| "{" *C-statement* {*statement*} "}"

Figure 3.2: The construction of RMPC language.

RMPC program is essentially a collection of BUS, WRITE, READ, and COMPUTE substeps defined in Section 2.2.1. It is recommended that the statement tags `S::`, `G::`, etc. should appear at the beginning of a line. It is also apparent from Figure 3.2 that all types of statements need not necessarily be present in an RMPC program.

Although all types of statements appear to be very similar except for the statement tags `S::`, `G::`, etc., a programmer should be aware of the importance of these statements from the execution sequencing point of view. The *Start-of-program-statement* and the *End-of-program-statement* are executed at the start and at the end of a program execution respectively. A *Bus-statement* is responsible for reconfiguring buses and thus it should use the `Bus()` primitive as explained in Section 3.2.3. Similarly a *Write-statement*(*Read-statement*) is responsible for executing a WRITE(READ) substep and therefore, it should use the `Write()`(`Read()`) primitive (see Section 3.2.3). A *Compute-statement* is optional which is responsible for a COMPUTE substep. The *beGin-of-step-statement* and the *Finish-of-step-statement* are executed at the start and at the end of each lot of *essential-statements*.

Any of the above statements can be *simple* or *complex*. As in C, a complex statement is recursively defined as a sequence of simple or complex statements enclosed in a pair of curly braces. A statement can also be empty which is represented by a single "`;`" only.

The *Start-of-program-statement*, the *End-of-program-statement*, the *beGin-of-step-statement* and the *Finish-of-step-statement* provides debugging entries during program execution. The *Start-of-program-statement* is also intended to be used in initialising the dimension of the simulated mesh and number of registers available to each processor in RMSIM. The following initialisation primitive is provided:

- `SetGlobalDim(Nx, Ny, Nz, regN, write_mode, filename)` – sets a virtual reconfigurable mesh of size `Nx` × `Ny` × `Nz` with `regN` number of registers per processor and `write_mode` = (`exclusive|common|concurrent`) bus write mode and opens `filename` file to write any LaTeX pictures generated during the execution.

Once initialisation is done, `Nx`, `Ny`, and `Nz` are considered as constants which are

usually referred as the *dimensional constants*.

An RMPC program file can contain more than one programs and one of the programs must be named `::main` from which the execution starts as in C and the remaining programs act as subprograms. It is also possible to include additional programs by the `input` command as long as exactly one program is named `::main`. An example of a complete RMPC programs set, divided into two separate files `rank.rpc` and `main.rpc`, is presented in Figure 3.3.

3.2.2 Data Structures and Constants

RMPC assumes data width of the reconfigurable mesh to be the `sizeof(double)`. The registers are considered as `double` variables and the bandwidth of the buses and ports are assumed to be the data width. It is programmer's responsibility to interpret the content of a register otherwise, if other simple data types e.g., `int`, `char`, etc. are to be used. In a similar way complex data types can also be handled.

Three constants `x`, `y`, and `z` are available to the programmer as the 3-dimensional Cartesian address of the processor executing the program. As a reconfigurable mesh is operated in the single-instruction-multiple-data (SIMD) mode, a programmer should assume that the constants `x`, `y`, and `z` will be replaced by values i, j, and k respectively before a program is executed on the processor $PE_{i,j,k}$, for all $0 \leq i <$ `Nx`, $0 \leq j <$ `Ny`, and $0 \leq k <$ `Nz`.

An RMPC program also assumes that it is executed in a restricted region (Figure 3.5) of the original mesh of size `Nx` $\times$ `Ny` $\times$ `Nz`, defined by the *boundary constants* `Sx`, `Sy`, `Sz`, `Ex`, `Ey`, and `Ez`, which includes only the processors $PE_{i,j,k}$, $\min(\texttt{Sx},\texttt{Ex}) \leq i \leq \max(\texttt{Sx},\texttt{Ex})$, $\min(\texttt{Sy},\texttt{Ey}) \leq j \leq \max(\texttt{Sy},\texttt{Ey})$, $\min(\texttt{Sz},\texttt{Ez}) \leq k \leq \max(\texttt{Sz},\texttt{Ez})$. This strategy enables more than one program to be executed simultaneously in different parts of the mesh which also helps in program reusability as discussed in Section 3.2.4.2.

Registers are indexed from 0 to `regN` -1. To enforce data hiding, registers are accessed only through the following two data access primitives:

- `SetReg(reg, val)` – sets the content of register `reg` with `val`.

```
::PrefixSum
B:: Bus("NS", "E", "W", "U", "D", "");
W:: if(y == Sy && z == Sz) Write(N, GetReg(0));
R:: Read(N, 0);

B:: if(GetReg(0) == 0) Bus("EW", "N", "S", "U", "D", "");
    else Bus("WN", "ES", "U", "D", "", "");
W:: if(x == Sx && y == Sy && z == Sz)
      if(GetReg(0) == 0) Write(E, 99);
      else Write(N, 99);
R:: Read(E, 1);

B:: if(z == Sz) Bus("NS", "E", "W", "U", "D", "");
W:: if(z == Sz && GetReg(1) == 99) Write(S, y-Sy);
R:: if(y == Sy && z == Sz) Read(S, 1);

::Rank
B:: Bus("NS", "E", "W", "U", "D", "");
W:: if(y == Sy && z == Sz) Write(N, GetReg(0));
R:: Read(N, 0);

B:: Bus("EW", "N", "S", "U", "D", "");
W:: if(x == y && z == Sz) Write(E, GetReg(0));
R:: Read(E, 1);
C:: { SetReg(2, GetReg(0));
      if(GetReg(0) > GetReg(1)) SetReg(0, 1);
      else SetReg(0, 0); }

B:: ;
W:: ;
R:: ;
C:: if(y == Sy && z == Sz) {
        Call(PrefixSum, YZ_X, Ey, Sy, Sz, Ez, x, x);
        SetReg(0, GetReg(2)); }
```

File: `rank.rpc`

```
::input "rank.rpc"
::main
int n = 4, m = n-1;
::S SetGlobalDim(n, n, n, 3, exclusive, "rank.tex");

::B ;
::W ;
::R ;
::C if(y == Sy && z == Sz) {
        switch( x ) {
           case 0: SetReg(0, 14); break;
           case 1: SetReg(0, 7); break;
           case 2: SetReg(0, 15); break;
           case 3: SetReg(0, 3); } }

::E Call(Rank, XY_Z, 0, m, 0, m, 0, m);
```

File: `main.rpc`

Figure 3.3: An RMPC program to compute the ranks of 4 distinct numbers (14,7,15,3) using algorithms in [118].

- `GetReg(reg, val)` – gets the content of register `reg` into storage denoted by `val`.

Six ports of a processor are denoted, as usual, by aliases `E`, `W`, `N`, `S`, `U`, and `D` and six different axis-orientations (see Section 3.2.4) are denoted by aliases `XY_Z`, `YX_Z`, `YZ_X`, `ZY_X`, `ZX_Y`, and `XZ_Y`.

3.2.3 Data Communication

Data communication in RMPC is supported by the following four primitives:

- `Bus(set_1, set_2, ...)` – selects the connection pattern {`set_1`, `set_2`, ...} as the interconnection of I/O ports. There can at most be six parameters and each parameter is a string of characters drawn from the alphabet {**E**, **W**, **N**, **S**, **U**, **D**} such that $\bigcup_{\forall i} \texttt{set\_}i = \{\mathbf{E}, \mathbf{W}, \mathbf{N}, \mathbf{S}, \mathbf{U}, \mathbf{D}\}$ and $\forall i : \forall j \neq i : \texttt{set\_}i \cap \texttt{set\_}j = \{\}$.
- `Write(prt, val)` – writes the `val` on the bus connected to the port `prt`.
- `Read(prt, reg)` – reads the content of the bus connected to the port `prt` into register `reg`..
- `Error(prt)` – returns 1 if the bus connected to the port `prt` is in *error* state; returns 0 otherwise.

3.2.4 Program Reusage

Like most of the programming languages, RMPC is capable of using external programs into the current program through its `Call()` primitive. This is one of the strength of RMPC which puts it ahead of other programming models for the reconfigurable mesh in [58, 68, 99, 119]. Many algorithms on reconfigurable meshes have appeared in the literature containing references to other published algorithms which has simplified the description of these algorithms significantly. The capability of reusing programs enables a programmer not only to convert these algorithms into programs in similar straightforward fashion but also write new programs based on modular design.

We introduce here two new concepts in reusing RMPC programs. In Section 3.2.4.1 we show a way of reusing a program in different axis-orientation and in Section 3.2.4.2 we show the mechanism of reusing a program in a restricted region.

3.2.4.1 Axis-Orientation Mapping

Let the `PrefixSum` program in Figure 3.3 be available to us which can compute the prefix sums of some binary numbers stored in the processors $PE_{*,0,0}$ using the *XY*-plane of processors $PE_{*,*,0}$. Now, consider that we are solving a problem on reconfigurable mesh which requires at some stage to compute the prefix sums of some binary numbers stored in the processors $PE_{0,*,0}$ using the *YZ*-plane of processors $PE_{0,*,*}$. Can this prefix sum computation be done reusing the `PrefixSum` program?

Yes, RMPC is capable of the above operation by its unique axis-orientation mapping which, in this particular case, will map *x*-axis to *y*-axis, *y*-axis to *z*-axis, and *z*-axis to *x*-axis, in other words, after the mapping *y*-, *z*-, and *x*-axes will be assumed as *x*-, *y*-, and *z*-axes respectively, before applying the `PrefixSum` program.

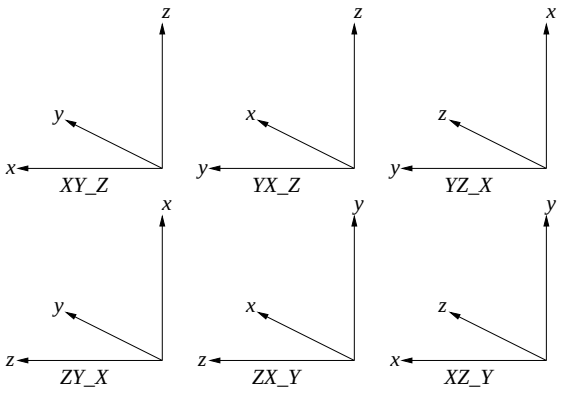

Figure 3.4: Possible six axis-orientations.

Any RMPC program assumes the natural *XY_Z* to be the axis-orientation. But through the `Call()` primitive, a program can be executed in any of the possible six axis-orientations as shown in Figure 3.4. RMPC also allows nesting of the `Call()` primitive and the axis-orientation is selected accordingly. Suppose the current axis-orientation and the requested axis-orientation are *YZ_X* and *XZ_Y* respectively, RMPC will then select *YX_Z* as the resultant axis-orientation.

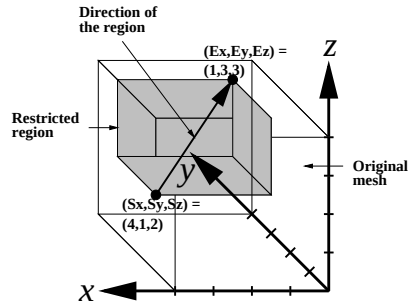

Figure 3.5: A restricted region in a reconfigurable mesh of size $5 \times 5 \times 5$.

The `Rank` program, in Figure 3.3, of computing the ranks of n distinct numbers on a reconfigurable mesh of size $n \times n \times n$ further exemplifies the axis-orientation mapping. Let the number n_i be stored in $PE_{i,0,0}$, $0 \leq i < n$. Now a row broadcast after a column broadcast is done to distribute the numbers in the XY-plane so that $PE_{i,j,0}$, $0 \leq i, j < n$, receives the pair (n_i, n_j) and then produces 1, if $n_i > n_j$, or 0, otherwise. Ranks are now computed by simply `Call()`ing the program `PrefixSum` in YZ_X axis-orientation on every YZ-planes to add the comparison values along each column.

3.2.4.2 Region Mapping

The power of program reusability through axis-orientation mapping cannot be realised completely if no way is allowed to `Call()` a program in a restricted region of the original reconfigurable mesh e.g., in the `Rank` program in Figure 3.3, each `Call()` of the program `PrefixSum` uses a specific 2-dimensional YZ-plane rather than using the entire 3-dimensional mesh.

As mentioned in Section 3.2.2, an RMPC program always assumes that it is executed in a restricted region of the original mesh of size $\mathtt{Nx} \times \mathtt{Ny} \times \mathtt{Nz}$, defined by the boundary constants `Sx`, `Sy`, `Sz`, `Ex`, `Ey`, and `Ez` as shown in Figure 3.5. Hence, defining a restricted region in program `Call()`ing is as simple as assigning $\mathtt{Sx} = \mathtt{sx}$, $\mathtt{Ex} = \mathtt{ex}$, and so on. By default, it is assumed in an RMPC program that $\mathtt{Sx} = 0$, $\mathtt{Ex} = \mathtt{Nx} - 1$, $\mathtt{Sy} = 0$, $\mathtt{Ey} = \mathtt{Ny} - 1$, $\mathtt{Sz} = 0$, and $\mathtt{Ez} = \mathtt{Nz} - 1$.

The restricted region, defined by the boundary constants, has a *direction*, pointing from Cartesian co-ordinate $(\mathtt{Sx}, \mathtt{Sy}, \mathtt{Sz})$ to $(\mathtt{Ex}, \mathtt{Ey}, \mathtt{Ez})$, which also plays an important role in program reusage as evident in the RMPC programs in Figure 3.3. The

program `PrefixSum`, in default *XY_Z* axis-orientation, computes and then stores the prefix sum $\sum_{j=0}^{i} b_j$ in processor $PE_{i,0,0}$, for all $i : 0 \leq i < \texttt{Nx}$, where the binary value b_j is initially stored in processor $PE_{j,0,0}$, for all $j : 0 \leq j < \texttt{Nx}$. Let the sum $\sum_{j=0}^{\texttt{Nx}-1} b_j$ be called the *final sum*. Now, the program `Rank` reuses the program `PrefixSum` in *YZ_X* axis-orientation within restricted region of distinct *YZ*-planes where the direction of the regions is chosen opposite to the normal to allow the final sums to be stored in the processors $PE_{*,0,0}$ in stead of the processors $PE_{*,\texttt{Ny}-1,0}$.

Finally, we provide the program reusage primitive as follows:

- `Call(prog-name, axis-ori, sx, ex, sy, ey, sz, ez)` – executes the program `prog-name` in `axis-ori` axis-orientation within the region consisting the processors $PE_{i,j,k}$, $\min(\texttt{sx},\texttt{ex}) \leq i \leq \max(\texttt{sx},\texttt{ex})$, $\min(\texttt{sy},\texttt{ey}) \leq j \leq \max(\texttt{sy},\texttt{ey})$, $\min(\texttt{sz},\texttt{ez}) \leq k \leq \max(\texttt{sz},\texttt{ez})$ with the direction from $(\texttt{sx},\texttt{sy},\texttt{sz})$ to $(\texttt{ex},\texttt{ey},\texttt{ez})$.

3.3 The Serial Simulator: RMSIM

RMSIM is a serial simulator written in ANSI C which can simulate a 3-dimensional reconfigurable mesh of variable size. RMSIM supports the RMPC language by providing an execution mechanism to execute programs written in RMPC. In this section we provide a brief description of the technical issues concerned in the development of RMSIM. This software, along with a reasonable amount of technical details, is freely available by `ftp://cslab.anu.edu.au/pub/Manzur/RMSIM`.

In Section 3.3.1 we present some important data structures we use in developing RMSIM. Implementation of the axis-orientation mapping and the region mapping is discussed in Section 3.3.2. In Section 3.3.3, some technical issues regarding execution of an RMPC program on RMSIM are provided. We discuss the debugging and visualisation facilities of RMSIM in Section 3.3.4.

3.3.1 Data Structure

Each processor of the simulated mesh is represented by an instance of the following data structure `PE`:

```
typedef struct {
   double *reg;
   unsigned char port[6];
   double port_val[6];
   unsigned char port_flag[6];
} PE;
```

In the above data structure, `port_val` stores the values carried by the buses connected to the ports and `port_flag` is used to store various decision flags required to provide necessary functionalities. The simulated mesh is thus represented as `PE ***RM;`

The second important data structure is the `Call()` stack which is necessary to preserve the *state of the system* while execution is transferred to another program via the `Call()` primitive. RMPC demands each processor to be equipped with its own internal `Call()` stack, but as RMSIM is a serial simulator, a global `Call()` stack is sufficient for the simulation. The state of the system includes the constants `Sx`, `Sy`, `Sz`, `Ex`, `Ey`, and `Ez`, the current axis-orientation, and the step and the substep in which the `Call()` is made.

3.3.2 Necessary Mappings

The axis-orientation mapping and the region mapping of RMPC language is simulated in RMSIM through the mapping of the dimensional constants, the boundary constants, the processors, and the ports of each processor. The mapping of the dimensional constants, the boundary constants, and the processors is guided entirely by the requested axis-orientation in the `Call()` primitive. The mapping of ports, however, depends on both the requested axis-orientation as well as the direction of the requested restricted region in the `Call()` primitive.

3.3.3 Serial Execution Order

RMPC assumes that a program will be executed on all the processors in parallel. But being a serial simulator, RMSIM assumes the following order in accessing processors sequentially:

Loop along 3rd axis
  Loop along 2nd axis
    Loop along 1st axis

Actual axes to be considered in place of 1st, 2nd, and 3rd axes are resolved according to the current axis-orientation. In *ZX_Y* axis-orientation 1st, 2nd, and 3rd axes are taken as *z*, *x*, and *y* axes. The step of each loop is either 1 or -1 depending on associated boundary constants. For example, the region defined in Figure 3.5 will generate the following loop structure if the current axis-orientation is *XY_Z*:

for z = 2 to 3 step 1
  for y = 1 to 3 step 1
    for x = 4 to 1 step -1

Although it is possible to take the advantage of the above serial order in designing RMPC programs on RMSIM, it is highly undesirable and therefore, not recommended, as it destroys the principle of parallel computing.

3.3.4 Debugging and Visualisation Facilities

RMSIM generates run time error codes while executing a program if a problem occurs e.g., the `Call()` primitive requests a restricted region which is not confined within the dimension of the simulated mesh. Besides this standard technique, RMSIM is also equipped with visualisation tools to generate LaTeX pictures of the bus configurations, along any plane of the simulated mesh, at any step of program execution.

The generated pictures are scalable and can show the content of at most two registers of each processor. The buses which carry data are drawn in thick lines to differentiate these from the buses carrying no data.

Figures 3.6–3.11 are generated by RMSIM while executing the RMPC programs in Figure 3.3 where the caption of a figure mentions the step number of the program at the end of which the figure is captured.

As stated in Section 3.1, RMSIM is not equipped with any visualisation tool to generate 3-dimensional pictures of the simulated mesh. The reason for not providing any 3-dimensional visualisation tool lies primarily in keeping the complexity of writing RMSIM at minimum. But we also realise that 2-dimensional visualisation tools are sufficient for the mesh-of-meshes model on which most of the algorithms developed so far on 3-dimensional reconfigurable meshes can be adapted directly without any modification.

3.4 Conclusions

In this chapter we have defined a new programming model for the general 3-dimensional reconfigurable mesh model which is expressed by means of a new programming language, called RMPC (Reconfigurable Mesh Parallel C). We have also presented some technical issues involved in the development of a serial simulator, RMSIM (Reconfigurable Mesh SIMulator), which can execute any RMPC program on a simulated 3-dimensional reconfigurable mesh.

The main purpose of defining a new programming model for the reconfigurable mesh is to provide facilities for reusing programs, like subroutine calls, which are not available in the existing programming models in [58, 68, 99, 119]. We have thus introduced the idea of executing a program in different axis-orientations and/or within restricted regions, which we believe are considered here for the first time (except for author's paper [75]).

A 2-dimensional visualisation tool has been developed as an integral part of RMSIM to assist in program debugging and validation.

In defining RMPC, we have concentrated on making the effort of transforming

algorithms into equivalent programs straightforward and easy. Yet most of the algorithms on reconfigurable meshes, discussed or developed in this thesis, are presented in algorithmic form rather than using RMPC constructs because we believe that an algorithmic presentation gives the reader a clearer intuitive understanding than a formal RMPC-based presentation.

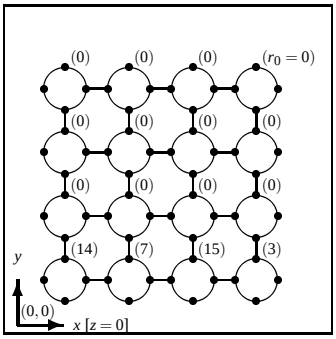

Figure 3.6: PROG: Main, STEP: 0

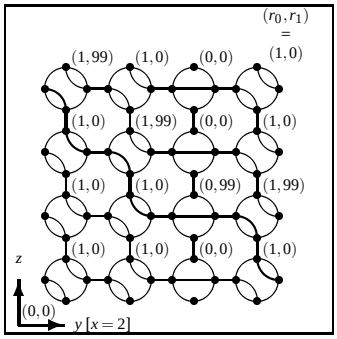

Figure 3.9: PROG: PrefixSum, STEP: 1

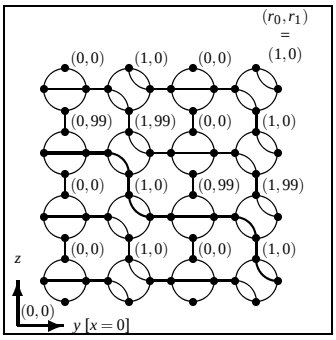

Figure 3.7: PROG: PrefixSum, STEP: 1

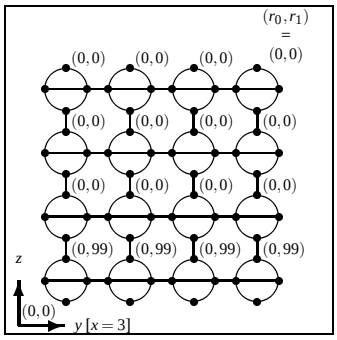

Figure 3.10: PROG: PrefixSum, STEP: 1

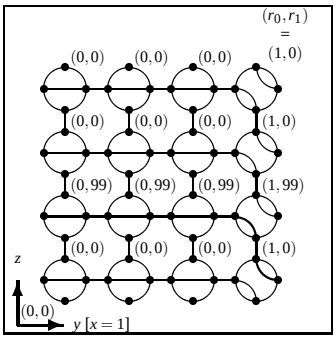

Figure 3.8: PROG: PrefixSum, STEP: 1

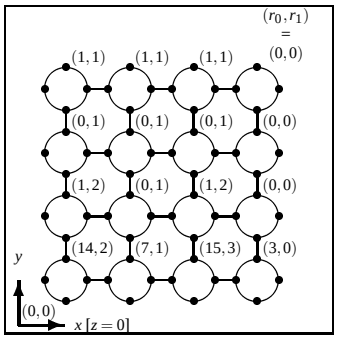

Figure 3.11: PROG: Rank, STEP: 2

Chapter 4

Sorting on Mesh-Connected Networks

The aim of this chapter is to provide a background in sorting on mesh-connected networks including linear arrays, ordinary meshes, and reconfigurable meshes. We concentrate mainly on optimal algorithms in the sense of limitations imposed by communication diameter as well as AT^2 measure.

Some general assumptions on sorting as well as the capability of the networks are discussed in the next introductory section. In Section 4.2 optimal sorting on linear arrays of processors is addressed. A number of optimal sorting algorithms on ordinary mesh are discussed in Section 4.3. In Section 4.4 we review AT^2 optimal constant time algorithms on reconfigurable meshes including meshes with higher dimension.

4.1 Introduction

The problem of sorting needs no introduction. Undoubtedly, sorting is one of the most widely researched topics. In this thesis, sorting plays a significant role in the development of maximal contour algorithms on ordinary as well as reconfigurable meshes (Chapter 5) and in the establishment of the idea of adaptive algorithms for reconfigurable meshes (Chapter 7).

So much work has already been done on sorting that it is beyond the scope of this thesis to cover all but a small part of it. Thus we consider only some specific related topics of parallel sorting on inter-connected networks of processors. Many sorting algorithms have so far been developed for these networks with a significant

consideration being the input and output of data. Tasks involved with input and output of data are not discussed in this chapter as the networks are considered to be filled with data, to be sorted, at the start of any sorting algorithm.

To avoid unnecessary complication, the length of each item to be sorted on a network will be assumed to be at most the bandwidth of the network. If necessary, the algorithms can easily be extended to sort lengthy multi-item records as long as the length of the *key* is within the above limit. Moreover, the items to be sorted are also assumed to be distinct without any loss of generality as identical keys can be made distinct by introducing a secondary key.

4.2 Sorting on Linear Arrays

Consider sorting of N items on a linear array of N processors where each processor has exactly one item in a scrambled order at the start. Without any loss of generality, it is assumed that, after sorting, each processor will again contain exactly one item in a linear sorted order of processors such that the smallest item resides in the leftmost processor and the largest item resides in the rightmost processor.

The lower bound of sorting on linear arrays is discussed in the next section. Section 4.2.2 presents a sorting algorithm which almost achieves the lower bound.

4.2.1 Lower Bounds

In the worst case, the smallest item might start in the rightmost processor and thus we would need $N-1$ steps to move it into the leftmost processor. Thus sorting of N items on a linear array of N processors with bidirectional links requires at least $N-1$ steps. Now, for the strict unidirectional model this lower bound can further be improved to $2N-2$ based on the scenario where the smallest and the largest items are residing into the rightmost and the leftmost processors respectively.

4.2.2 Odd-Even Transposition Sorting

Odd-even transposition sorting algorithm [51], which takes exactly N steps, almost achieves the lower bound discussed above for bidirectional model. The algorithm

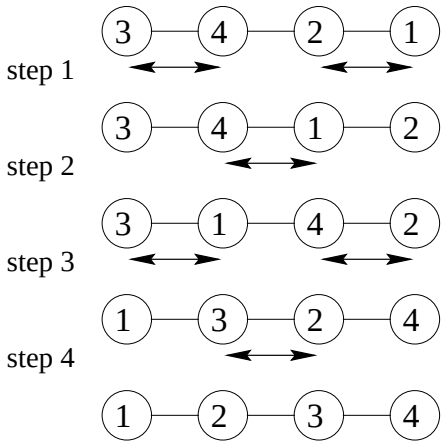

Figure 4.1: Odd-even transposition sort.

is quite simple. At the odd steps, the items of processors 1 and 2, 3 and 4, etc. are compared and, if necessary, these items are also swapped so that the smaller items ends up in the left of the processor pairs. The same operations are performed for processors 2 and 3, 4 and 5, etc. at even steps. For example, see Figure 4.1.

Theorem 4.1 *N items on a linear array of N processors with bidirectional links, where each processor has exactly one item, can be sorted in exactly N steps.*

Proof. See [51, Section 1.6]. ■

If the unidirectional model is considered, performing comparison as well as swapping items residing in a pair of neighbouring processors requires two steps, instead of one. Hence follows:

Theorem 4.2 *N items on a linear array of N processors with unidirectional links, where each processor has exactly one item, can be sorted in exactly* $2N$ *steps.* ■

4.3 Sorting on Ordinary Meshes

Consider sorting MN items on an ordinary mesh of size $M \times N$ where each processor has exactly one item (in scrambled order) at the start. Also suppose that, after sorting, each processor will again contain exactly one item in a prescribed order of processors.

There is no single natural ordering of the processors of a mesh for sorting. The order of the processors for sorting should not be confused with the Cartesian addressing of the processors. In fact, any one-to-one mapping from $\{1,2,\ldots,M\} \times \{1,2,\ldots,N\}$ onto $\{1,2,\ldots,MN\}$ can be used as the ordering of processors such that after the sorting is done, the jth smallest (ascending lexicographical order) or largest (descending lexicographical order) item will reside in the processor which is mapped to the index j according to the above mapping. Clearly the order of the processors should be prescribed in advance, independent of the input data.

A number of orders of processors for sorting have so far been introduced in the literature of which *row-major*, *column-major*, *snake-like-row-major*, and *snake-like-column-major* orders are the most common. For examples of these orders of processors, see Figure 4.2.

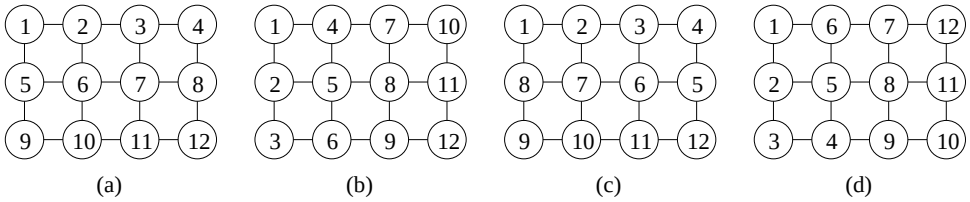

Figure 4.2: Order of processors for sorting in ascending order on mesh. (a) row-major, (b) column-major, (c) snake-like-row-major, and (d) snake-like-column-major orders.

In the next section the lower bound of sorting on ordinary meshes is discussed. An optimal sorting algorithm by Schnorr and Shamir and an improved algorithm by Nigam and Sahni are presented in Section 4.3.2. In section 4.3.3 we discuss Leighton's optimal columnsort algorithm. Rotatesort, an optimal sorting algorithm by Marberg and Gafni, is presented in Section 4.3.4.

4.3.1 Lower Bounds

Whatever the order of processors in the final state of sorting, instances can be found where two items, initially residing at diagonally opposite corner processors, have to be transposed during the sorting. It is very easy to conclude that even for such a simple transposition we need at least $M+N-2$ and $2M+2N-4$ steps for the bidirectional and the strict unidirectional model respectively.

Theorem 4.3 *Sorting of MN items on an ordinary mesh of size $M \times N$, where the final output can reside in any arbitrary order of processors, cannot be done in less than $M+N-2$ steps with bidirectional links and $2M+2N-4$ steps with strict unidirectional links.* ■

Not surprisingly, optimal sorting on ordinary meshes have been studied extensively [51, 52, 57, 82, 84, 92, 101, 107]. In many of the above cases, sorting has been studied primarily on square meshes. For consistency, we have adapted those works for rectangular meshes. Thompson and Kung [107] have developed a sorting algorithm with complexity $2M+4N+o(M+N)$ on the strict unidirectional mesh model. Leighton [52] has presented his $7M+4N$ order *columnsort* algorithm on a bidirectional mesh where $M \geq 2(N-1)^2$. Schnorr and Shamir [101] have published a sorting algorithm with complexity $M+2N+o(M+N)$ where $M^2 > N$. In [101] Schnorr and Shamir also claimed that sorting in row-major final order on the bidirectional mesh model, even with no penalty for excessive computational and/or storage, requires at least $M+2N-o(M+N)$ steps.

Very recently, Nigam and Sahni [84] have disproved the lower bound of [101] by presenting an $M+N+o(M+N)$ order sorting algorithm on the same powerful mesh model used in [101]. Nigam and Sahni [84] have also disproved Schnorr and Shamir's lower bound even on the basic bidirectional mesh model with limited computational power and storage by developing a sorting algorithm with complexity $M+1.5N+o(M+N)$. In [101] Nigam and Sahni have also considered the strict and the non-strict unidirectional mesh models.

All the above algorithms proceed by dividing the mesh into submeshes, working on the submeshes recursively in parallel, then combining the results in some fashion. In [57] Marberg and Gafni have presented an optimal $7M+7N$ order *rotatesort* algorithm on the bidirectional mesh involving transformations (sorting and rotation) alternatively along only rows and columns where $M \geq N^{1/2}$.

For the sake of completeness, the algorithms developed by Schnorr and Shamir [101], Nigam and Sahni [84], Leighton [51, 52] and Marberg and Gafni [57] are discussed briefly in the following sections.

Whenever the expression $N^{\alpha/\beta}$, where α and β are integers and $\alpha < \beta$, is encountered hereafter in this chapter it is assumed to be an integer for the sake of simplicity in presenting algorithms.

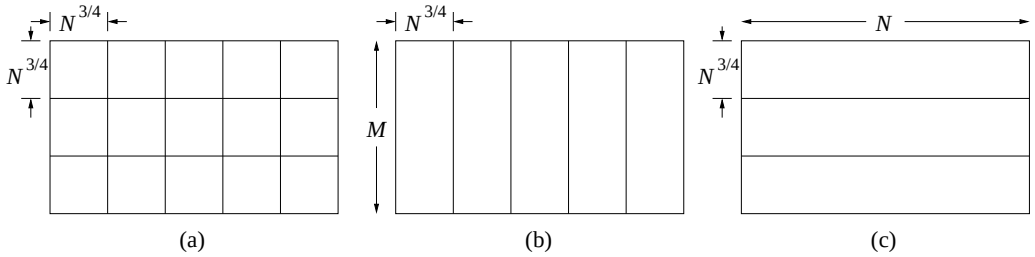

Figure 4.3: Definition of blocks (a), vertical slices (b), and horizontal slices (c) in the sorting algorithm of Schnorr and Shamir.

4.3.2 Schnorr and Shamir's Algorithm

Schnorr and Shamir [101] have assumed the simplest bidirectional mesh model where each processor has very limited computational power and storage. The order of processors in the final output is assumed to be the snake-like-row-major order. To keep the algorithm correct for the entire range of values of M and N, it is also assumed that $N \leq M^2$. We further assume that $M = 2^{4s}$, $N = 2^{4t}$, and $4s \geq 3t$, which implies $M \geq N^{3/4}$, for the sake of simplicity in presentation.

Algorithm 4.1 Schnorr and Shamir's Algorithm [101]

1 *Sort all the blocks (Figure 4.3(a)) in snake-like-row-major order;*

2 *Permute the columns so that the $N^{3/4}$ columns in each block are distributed evenly among the $N^{1/4}$ vertical slices; (For example of vertical slices, see Figure 4.3(b).)*

3 *Sort all the blocks in snake-like-row-major order;*

4 *Sort all the columns of the mesh downwards;*

5 *Collectively sort blocks 1 and 2, blocks 3 and 4, etc., of each vertical slice in snake-like-row-major order;*

6 Collectively sort blocks 2 and 3, blocks 4 and 5, etc., of each vertical slice in snake-like-row-major order;

7 Sort all the rows of the mesh into alternating left-to-right and right-to-left order;

8 Perform $2N^{3/4}$ steps of the odd-even transposition sort (Section 4.2.2) along the snake;

Lemma 4.4 *Sorting MN items on a bidirectional mesh of size $M \times N$, $N \leq M^2$, in snake-like-row-major final order, can be done in $M + 2N + o(M+N)$ steps.*

Proof. See the complexity analysis of the above algorithm in [101]. ■

In spite of its asymptotic optimality, the above algorithm of Schnorr and Shamir is not likely to be very practical, since for moderate values of N the low order term, $O(N^{3/4})$, remains significant.

In [101], Schnorr and Shamir not only have presented the above algorithm but also have claimed the complexity of their algorithm to be very close to the lower bound. In the process of claiming this, they have assumed a stronger bidirectional mesh model where each processor can have unlimited computational power and storage and then they have argued that any row-major (normal or snake-like) sorting algorithm must take $M + 2N - o(M+N)$ steps.

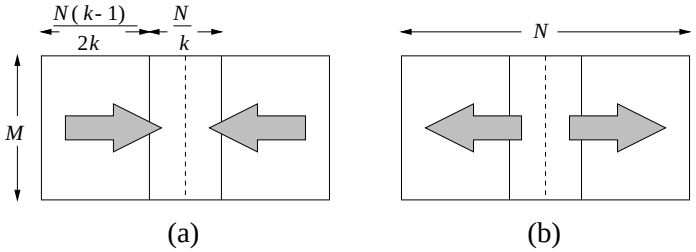

Figure 4.4: Folding (a) and unfolding (b) of data in the sorting algorithm of Nigam and Sahni.

Nigam and Sahni [84] have successfully disproved the lower bound claim of Schnorr and Shamir by developing an algorithm which requires less steps. Nigam

and Sahni have assumed the same stronger bidirectional mesh model used in establishing the lower bound of Schnorr and Shamir. For some integer k, a power of 2, the data are folded (Figure 4.4(a)) to the N/k columns in the centre so that each column of processors contains k successive columns of data. Some steps of Schnorr and Shamir's algorithm have then be modified to sort MN items on the central submesh of size $M \times N/k$ where at the start and at the end of sorting each processor contains k items. Finally the data are unfolded as in Figure 4.4(b).

Lemma 4.5 *Sorting MN items on a bidirectional mesh model of size $M \times N$, with no penalty for excessive computation and storage, can be done in $M + N(1 + 1/k) + o(M + N)$ steps.*

Proof. See [84]. ■

If $k = \Omega(N)$ then Nigam and Sahni's algorithm takes only $M + N + o(M + N)$ steps which is extremely close to the general lower bound of the bidirectional mesh model in Theorem 4.3.

As Nigam and Sahni [84] have assumed a strong bidirectional mesh model where each processor can store up to k items, their sorting algorithm is not applicable to mesh models where computing and storage capability of each processor is very limited as assumed in Algorithm 4.1 by Schnorr and Shamir.

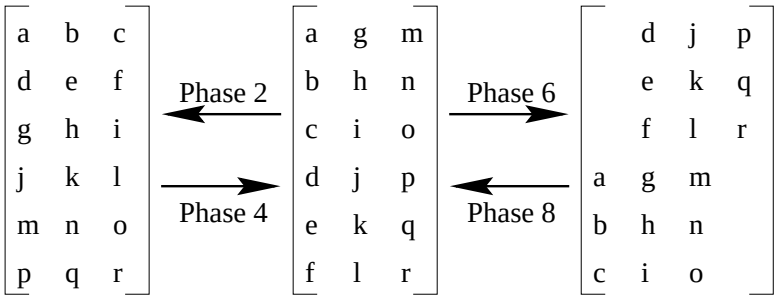

Figure 4.5: Permutation of matrix in various phases of columnsort. For simplicity, we have chosen a 6×3 matrix which does not satisfy the $M \geq 2(N-1)^2$ constraint.

4.3.3 Leighton's *Columnsort* Algorithm

In the original columnsort algorithm by Leighton [52], an $M \times N$ matrix of items is sorted in column-major order. It can be shown that if $M \geq 2(N-1)^2$ and $M \bmod N = 0$ then the following eight phases are sufficient to sort the items:

Algorithm 4.2 Columnsort [52]

1 *Sort all the columns downward;*

2 *Transpose (Figure 4.5) the matrix by picking up the items in column-major order and setting them in row-major order, preserving the shape of the matrix;*

3 *Sort all the columns downward;*

4 *Reverse the permutation applied in phase 2 (Figure 4.5);*

5 *Sort all the columns downward;*

6 *Shift all the items $M/2$ positions as shown Figure 4.5;*

7 *Sort all the columns downward;*

8 *Shift all the items back $M/2$ positions as shown Figure 4.5;*

The above algorithm can be ported into a mesh of size $M \times N$ by keeping the extra column at phases 6, 7, and 8 in the last column. A careful scrutiny reveals that the above algorithm can be executed on the $M \times N$ bidirectional mesh model in $7M + 4N$ steps.

In [51, pp. 261], Leighton has improved his columnsort algorithm with resulting complexity $6M + 4N$. In phase 5, columns are alternately sorted in downward and upward directions. The requirement of an extra column is eliminated by replacing phases 6 and 7 with two steps of odd-even transposition sort along each row in phase 6 only.

4.3.4 Marberg and Gafni's *Rotatesort* Algorithm

Most of the sorting algorithms on meshes involve recursion on submeshes as a natural divide-and-conquer technique. In the quest for sorting algorithms without involving any recursion on submeshes, Scherson *et al.* [100] (and independently Sado and Igarashi [98]) have published the following algorithm where the rows and columns are alternately sorted:

Algorithm 4.3 Shearsort [100]

1 Repeat for $\lceil \log M \rceil + 1$ *times the following:*

1.1 Sort all the rows alternately to the right and to the left;

1.2 Sort all the columns downward;

The above sorting algorithm requires $(M+N)(\lceil \log M \rceil + 1)$ steps which is clearly non-optimal.

Schnorr and Shamir [101] have modified the shearsort algorithm to achieve a sublogarithmic nearly-optimal algorithm *revsort* by introducing cyclic rotation of rows in addition to sorting of rows.

Marberg and Gafni [57] have developed their algorithm *rotatesort* where a constant number of linear transformations is made alternately to rows and columns. Like revsort, rotatesort also uses cyclic rotation of rows in addition to sorting of rows. Interestingly a cyclic rotation of rows can easily be emulated by sorting of rows. Hence, it can be claimed that rotatesort involves only sorting of rows and columns in alternate steps. For simplicity, it is assumed in defining the rotatesort that $M = 2^s$ and $N = 2^{2t}$, where $s \geq t$.

Algorithm 4.4 Rotatesort [57]

1 Balance each vertical slice of size $M \times N^{1/2}$ *(Figure 4.6(b)) in parallel:*

1.1 Sort all the columns downward;

1.2 Rotate each row of the slice i, $(i \bmod N^{1/2})$ positions to the right;

1.3 Sort all the columns downward;

2 Unblock the entire mesh:

2.1 Rotate each row i, $(iN^{1/2} \bmod N)$ positions to the right;

2.2 Sort all the columns downward;

3 Balance each horizontal slice of size $N^{1/2} \times N$ (Figure 4.6(c)) in parallel:

3.1 Sort all the columns of the slice downward;

3.2 Rotate each row of the slice i, $(i \bmod N)$ positions to the right;

3.3 Sort all the columns of the slice downward;

4 Repeat phase 2;

5 Do the following steps for three times:

5.1 Sort all the rows alternately to the right and to the left;

5.2 Sort all the columns downward;

6 Sort all the rows to the right to obtain row-major order, or alternately rows in opposite directions to obtain snake-like-row-major order;

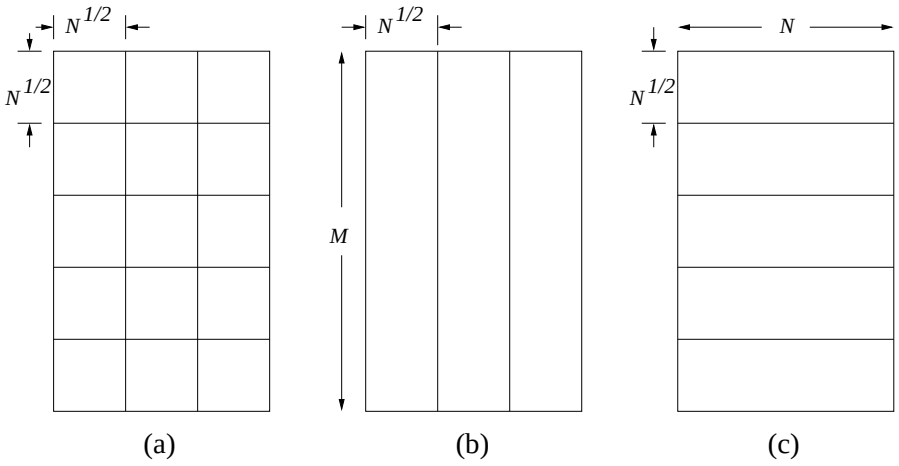

Figure 4.6: Definition of blocks (a), vertical slices (b), and horizontal slices (c) in the rotatesort algorithm.

It is easy to conclude that for the bidirectional mesh model the above algorithm requires $7M + 7N$ steps.

4.4 Sorting on Reconfigurable Meshes

Consider sorting of N items on an ordinary mesh of size $N \times N$ with unidirectional links where each processor in the bottom row contains an item at the start and at the end. It is very easy to argue that even though the bisection-width of the mesh, N, matches number of items to be sorted, we cannot expect to sort these numbers in less than $N - 1$ steps, which is the minimum communicational distance between the leftmost and the rightmost processors at the bottom row.

Now the *information content* of sorting N items is $\Omega(N)$ [113, pp. 58]. Let us assume that sorting N items on an area of $N \times N$ takes $O(T)$. By Ullman [113, pp. 56], the AT^2 lower bound of the above sorting problem is $\Omega(I^2(n)) = \Omega(N^2)$ from which we can easily conclude that AT^2 optimal sorting of the above configuration should take constant time, i.e., $O(1)$ steps, which is far beyond the reach of any of the ordinary mesh models.

The communication diameter of any reconfigurable mesh of any arbitrary size is 1 as discussed in Section 2.2.1. Many researchers [6, 18, 20, 36, 40, 44, 55, 63, 80, 83, 85, 87, 88, 97, 118] have exploited this fundamental property of reconfigurable mesh to design efficient sorting algorithms.

Consider sorting $n \leq MN$ items on a reconfigurable mesh of size $M \times N$ where $M \geq N$. If we are interested in developing AT^2 optimal algorithms with complexity $O(1)$ then according to equation (7.1, pp. 104) we get the following:

$$MN \times 1^2 = n^2 \times \frac{M}{N} \Rightarrow n = N \ .$$

This suggests that to develop constant time AT^2 optimal algorithms to sort N items, we must consider a reconfigurable mesh of size at least $N \times N$. A number of such algorithms have already been published and for the sake of completeness some of these algorithms will be discussed in Sections 4.4.1–4.4.4, grouped according to the basic methodologies. Many non-optimal sorting algorithms have also been developed

on reconfigurable meshes but these will be intentionally omitted from our discussion to keep the thesis brief and precise.

The following two results have played a significant role in the development of constant time sorting algorithms:

Lemma 4.6 *N items on a reconfigurable mesh of size $MN \times N$, $M \leq N$, can be sorted in $O(\log N)$ time if $M = 1$, and in $O(\log N / \log M)$ time if $M > 1$.*

Proof. See [55]. ■

Lemma 4.7 *Given N items on the bottom row of a reconfigurable mesh of size $N \times N$, any permutation of these items can be done in constant time provided the destination of an item is known to the processor containing the item.*

Proof. See [40]. ■

Some constant time sorting algorithms where columnsort is adapted are discussed in next section. In Section 4.4.2 a constant time adaptation of rotatesort is presented. We discuss some constant time sorting algorithms based on bucket and/or radix sort in Section 4.4.3. In Section 4.4.4 we present constant time sorting algorithms on higher dimensional reconfigurable meshes.

4.4.1 Algorithms Based on *Columnsort*

Jang and Prasanna [40] have adapted Leighton's columnsort to develop a constant time sorting algorithm on reconfigurable mesh. The N items are considered to be arranged in a virtual matrix of size $N^{3/4} \times N^{1/4}$. Now it is assumed that the leftmost $N^{3/4}$ processors at the bottom row contains the first column of the virtual matrix, the next $N^{3/4}$ processors at the bottom row contains the second column of the virtual matrix, and so on. Now, phases 1, 3, 5, and 7 are done in constant time by Lemma 4.6 using a submesh of size $N \times N^{3/4}$ for each column. Phases 2, 4, 6, and 8 can be considered as pure permutation problems and thus can also be done in constant time by Lemma 4.7.

Later Nigam and Sahni [83] have adapted Leighton's columnsort to develop a constant time sorting algorithm with a smaller number of broadcasts. They assume

a different virtual matrix of size $\frac{1}{2}N^{3/4} \times 2N^{1/4}$. The sorting of each column of $\frac{1}{2}N^{3/4}$ items in phases 1, 3, 5, and 7 is performed by implementing a second columnsort in a submesh of size $N \times \frac{1}{2}N^{3/4}$. The $\frac{1}{2}N^{3/4}$ items in each submesh are assumed to be arranged in another virtual matrix of size $N^{1/2} \times \frac{1}{2}N^{1/4}$.

4.4.2 Algorithms Based on *Rotatesort*

In [83], Nigam and Sahni have also adapted Marberg and Gafni's rotatesort algorithm to design a constant time sorting algorithm. Here, the N items are considered to be arranged in a virtual matrix of size $N^{1/2} \times N^{1/2}$. During the row phases, the N items are stored in row-major order. During the column phases, the items are stored in column-major order in the bottom row of the mesh. Obviously these rearrangements are purely permutations of the N items and hence can be done in constant time by Lemma 4.7. Now, by Lemma 4.6, sorting of $N^{1/2}$ columns or $N^{1/2}$ rows of $N^{1/2}$ items each can easily be done in $O(1)$ time using $N^{1/2}$ submeshes of size $N \times N^{1/2}$ each in parallel. Again, cyclic rotation of a row can be considered as a permutation problem and therefore, such rotations of $N^{1/2}$ rows of $N^{1/2}$ items each can be performed in constant time using $N^{1/2}$ submeshes of size $N^{1/2} \times N^{1/2}$ each.

4.4.3 Algorithms Based on Bucket Sort and Radix Sort

A few constant time sorting algorithms have been developed on reconfigurable meshes by adapting "bucket sort" [45] and/or "radix sort" [45]. In [55], Lin *et al.* have considered bucket sorting by multi-selection in designing their algorithm. First the $iN^{2/3}$-th, $1 \leq i \leq N^{1/3}$, smallest items amongst the N given items are computed in $O(1)$ time. Once these $iN^{2/3}$-th smallest items are found then $N^{1/3}$ implicit buckets of exactly $N^{2/3}$ items each can easily be obtained in constant time. It is now a straightforward operation to sort the $N^{2/3}$ items in every bucket using a submesh of size $N \times N^{2/3}$ in $O(1)$ time by Lemma 4.6.

Kapoor *et al.* [43] have pointed out that a large constant factor is associated with the above constant time sorting algorithm in [55] because the multi-selection problem is solved completely before the original unsorted list is divided into smaller lists for

sorting. A modified version of the sorting algorithm in [55] has thus been presented in [43] with smaller constant factor.

Olariu and Schwing [85] have presented a constant time sorting algorithm based on bucket sorting by a novel deterministic sampling scheme. N items are divided into $N^{3/8}$ groups of exactly $N^{5/8}$ items each. Each of these groups is sorted in parallel using a submesh of size $N \times N^{5/8}$ in $O(1)$ time. A sample A of $N^{3/8}$ items is drawn by taking the largest item in each group. The rank, among the entire given N items, of each item $\in A$ is computed in constant time using a submesh of size $N^{1/4} \times N$. Let the sample A in sorted order be $q_1, q_2, \ldots, q_{N^{3/4}}$. Now, $N^{3/8}$ implicit buckets are obtained by using $q_{N^{3/8}}, q_{2N^{3/8}}, \ldots, q_{N^{3/4}}$ as the selection pivots in constant time. In has been proved by Olariu and Schwing that such a bucketing will produce buckets of size at most $2N^{5/8}$ items. It is now a straightforward operation to sort each bucket using a submesh of size $N^{5/8} \times N$ in $O(1)$ time by adapting Lemma 4.6.

In [87], Olariu *et al.* have published a constant time sorting algorithm which is a hybrid between bucket sort and radix sort. They have considered sorting of integers in the range 0 to $N^c - 1$ for an integer constant c. An integer $a < N^c$ can be written in radix $N^{1/2}$ as follows:

$$a = a_{2c-1}N^{(2c-1)/2} + \cdots + a_3N^{3/2} + a_2N + a_1N^{1/2} + a_0$$

where $0 \leq a_0, a_1, \ldots, a_{2c-1} < N^{1/2}$.

In the first iteration all the given integers are separated into $N^{1/2}$ buckets of length at most N according to the value of a_{2c-1} in the radix $N^{1/2}$ representation of each integer. Cardinality of each bucket is computed in parallel using a submesh of $N^{1/2} \times N$ in constant time. All the N integers are then rearranged in bucket-major order preserving the order of items within a bucket.

A similar iteration is done using a_i, $2c - 1 > i \geq 0$, in the radix $N^{1/2}$ representation of each integer. Clearly the complexity of this algorithm is $O(c)$, i.e., O(1).

The following lemma follows from any of the above constant time algorithms:

Theorem 4.8 *Sorting of N items in a row of a reconfigurable mesh of size $N \times N$ can be done in $O(1)$ time.* ■

4.4.4 Sorting on Multi-Dimensional Reconfigurable Meshes

Chen and Chen [20] have considered sorting of N items in a hyperplane of processors of a k-dimensional reconfigurable mesh of size $N^{1/(k-1)} \times N^{1/(k-1)} \times \cdots N^{1/(k-1)}$ where k is an integer constant, independent of N. Leighton's columnsort has been adapted to solve this problem. N items are assumed to be arranged in a virtual matrix of size $N^{(k-2)/(k-1)} \times N^{1/(k-1)}$. Each column of $N^{(k-2)/(k-1)}$ items is recursively sorted on a $k-1$-dimensional submesh of size $N^{1/(k-1)} \times N^{1/(k-1)} \times \cdots N^{1/(k-1)}$ in parallel. The recursion stops at the case when $N^{1/(k-1)}$ items are to be sorted on a submesh of size $N^{1/(k-1)} \times N^{1/(k-1)}$ which can be done in constant time by Theorem 4.8. It can be easily verified that permutation of N items can be done in constant time on a k-dimensional reconfigurable of mesh of size $N^{1/(k-1)} \times N^{1/(k-1)} \times \cdots N^{1/(k-1)}$.

Leighton's columnsort remains valid on the above virtual matrix as long as $N^{(k-2)/(k-1)} \geq 2(N^{1/(k-1)} - 1)^2$, i.e., $k \geq 4$. Even for $K \geq 4$, the claim of constant time complexity of the above method depends on the availability of a constant time algorithm on a reconfigurable mesh of size $N^{1/2} \times N^{1/2} \times N^{1/2}$ to sort N items on a plane of processors.

In fact Chen and Chen [20] have developed the abovesaid sorting algorithm. First they have adapted the columnsort algorithm on a reconfigurable mesh of size $r \times s \times r$ to sort N items in constant time, where $rs = N$, $r \bmod s = 0$, $r \geq 2(s-1)^2$, and N items are physically arranged in an $r \times s$ matrix of processors $PE_{*,*,0}$. This algorithm is then further adapted, without any slowdown, to a reconfigurable mesh of size $r/m \times sm \times r/m$, where $2m^3 \leq r$, $m^2 \leq r/s$, and N items are now virtually arranged in an $r \times s$ matrix where each column of items is stored in m columns of the submesh $PE_{*,*,0}$. Therefore, we have the following theorem assuming $r = 2N^{2/3}$, $s = \frac{1}{2}N^{1/3}$, and $m = 2N^{1/6}$:

Theorem 4.9 *Sorting of N items in a plane of a reconfigurable mesh of size $N^{1/2} \times N^{1/2} \times N^{1/2}$ can be done in $O(1)$ time.* ■

Hence, follows the following theorem:

Theorem 4.10 *Sorting of N items in a hyperplane of processors of a k-dimensional reconfigurable mesh of size $N^{1/(k-1)} \times N^{1/(k-1)} \times \cdots N^{1/(k-1)}$, $k \geq 4$, can be done in $O(4^k)$ time.* ■

Nigam and Sahni [83] and Jang and Prasanna [35] have also claimed that their constant time sorting algorithms on 2-dimensional reconfigurable meshes can be extended to sorting on multi-dimensional reconfigurable meshes in a similar fashion.

Now a k-dimensional reconfigurable of mesh of size $N^{1/(k-1)} \times N^{1/(k-1)} \times \cdots N^{1/(k-1)}$ can be laid out in $\Omega(N^2)$ area according to Lemma 6.1. Hence the above algorithms on multi-dimensional reconfigurable meshes are all AT^2 optimal.

4.5 Conclusions

In this chapter we have discussed optimal sorting algorithms on linear arrays, ordinary meshes and reconfigurable meshes.

In Chapter 7 we introduce the concept of developing adaptive algorithms for reconfigurable meshes which will remain AT^2 optimal even if the size and aspect ratio of the meshes are not fixed. Sorting is used as the first problem to establish this idea. A significant portion of results including the algorithms presented in this chapter are used in developing adaptive AT^2 optimal sorting algorithms on reconfigurable meshes. The developments have demanded not only the existing optimal sorting algorithms on reconfigurable mesh but also optimal sorting algorithms on ordinary meshes and linear arrays.

We have given the results for sorting on ordinary meshes to the detail of getting the exact constant associated with the highest order term in the complexity analysis in order to compare two optimal maximal contour algorithms in Chapter 8.

In the next chapter we develop optimal algorithms on reconfigurable meshes to compute the contour of maximal elements of a given set of planar points. These algorithms extensively use various results for sorting on reconfigurable meshes described in this chapter.

Chapter 5

Computing $\mathcal{M}$ -Contour on Mesh-Connected Networks

In this chapter we present two unique properties (Lemmas 5.1 and 5.2) of the contour of the maximal elements of a set of planar points which can be exploited to develop efficient parallel maximal contour algorithms. These properties are indeed used here to develop optimal maximal contour algorithms on mesh-connected networks e.g., linear arrays, ordinary meshes, and reconfigurable meshes. A generic constant time algorithm to solve a large number of computational geometry problems is discussed in Section 5.1. In Section 5.2 we define the maximality problems and give some lower bounds. Optimal maximal contour algorithms on linear arrays and ordinary meshes are discussed in Sections 5.3 and 5.4 respectively. In Section 5.5 we develop three constant time maximal contour algorithms on reconfigurable meshes of various dimensions.

5.1 Introduction

Computational geometry problems are a recurring theme in a large number of contexts in computer science. It comes as no surprise, therefore, that these problems received a great deal of attention [96, 108]. In particular, computational geometric algorithms on the reconfigurable mesh and its variants have been widely proposed in the literature [10, 13, 14, 18, 28, 29, 37, 48, 49, 66, 77, 81, 89].

Computational geometry problems, in general, can be solved best by divide-and-conquer strategy. Jang *et al.* [37] have recently developed a number of constant

time computational geometry algorithms on the reconfigurable mesh by applying the divide-and-conquer strategy. In [37] it has been realised that to achieve constant time complexity, the number of levels, in recursively dividing the problems into smaller subproblems, must be a constant. Moreover the atomic subproblems must be solved in constant time in parallel and merging of subsolutions at each level must also be done in constant time. In fact Jang *et al.* [37] have proposed the following non-recursive generic algorithm, where the problem is divided into subproblems only once, in developing new constant time algorithms to solve computational geometry problems of size N on a reconfigurable mesh of size $N \times N$:

Algorithm 5.1 Generic Constant Time Algorithm [37]

1 *Divide a given problem of size N into N^{ε} subproblems of size $N^{1-\varepsilon}$, where $0 < \varepsilon < 1$. Solve each subproblems in constant time, using a mesh of size at most $N \times N^{1-\varepsilon}$;*

2 *Merge the solutions to the N^{ε} subproblems in constant time using the entire mesh;*

Using the above technique Jang *et al.* [37] have developed constant time algorithms to solve convex hull, smallest enclosing box, triangulation, all nearest neighbour, two-set dominance counting, and three dimensional maxima problems.

In this chapter we explore one further computational geometry problem from a similar point of view. The problem is to compute the contour of the maximal elements of a given set of planar points (see Section 5.2). We present three constant time algorithms to compute the contour of the maximal elements of N planar points on the reconfigurable mesh. The first algorithm in Section 5.5.1 employs a reconfigurable mesh of size $N \times N$ while the second one in Section 5.5.2 uses a 3-dimensional reconfigurable mesh of size $N^{1/2} \times N^{1/2} \times N^{1/2}$. In Section 5.5.3 the second algorithm is then further extended on a k-dimensional reconfigurable mesh of size $N^{1/(k-1)} \times N^{1/(k-1)} \times \cdots \times N^{1/(k-1)}$, where $k \geq 4$.

Whenever necessary $N^{1/(k-1)}$ is assumed to be an integer where k is the dimension of the reconfigurable mesh under consideration. This implies $k \leq \log_2 N + 1$.

In Chapter 7 we use the problem of computing the contour of the maximal elements of a given set of planar points to establish our idea of developing *adaptive* algorithms. In this process we use optimal algorithms to solve the above problem on linear array of processors as well as ordinary meshes. For the sake of completeness, these optimal algorithms are discussed in Sections 5.3 and 5.4.

5.2 Problem Definition

In this section we formally introduce maximality problems on a partially ordered set. The next section defines the problem of computing the $\mathcal{M}$ -contour of a set of planar points. In Section 5.2.2 maximality is defined for multi-dimensional space. An alternative definition of maximality is given in Section 5.2.3. In Section 5.2.4 we present some lower bounds on $\mathcal{M}$ -contour computations.

5.2.1 $\mathcal{M}$ -Contour of a Set of Planar Points

Let the planar point at coordinate (i, j) be defined as $P(i, j)$. For any point p, let $x(p)$ denote the x-coordinate and $y(p)$ denote the y-coordinate of p, e.g., $x(P(i, j)) = i$ and $y(P(i, j)) = j$.

Definition 5.1 *A point p* dominates *a point q (denoted by $q \prec p$) if and only if $x(q) \leq x(p)$ and $y(q) \leq y(p)$. (The relation "$\prec$" is naturally called* dominance.*)*

Let S be a finite set of planar points. To simplify the exposition of our algorithms, the points in S are assumed to be distinct.

Definition 5.2 *A point $p \in S$ is* maximal *if and only if there is no point $q \in S$ such that $p \prec q$ and $p \neq q$.*

The definition above actually defines maximality w.r.t. the northeast(**NE**) direction as depicted in Figure 5.2. The definitions of maximality w.r.t. other directions are then obvious.

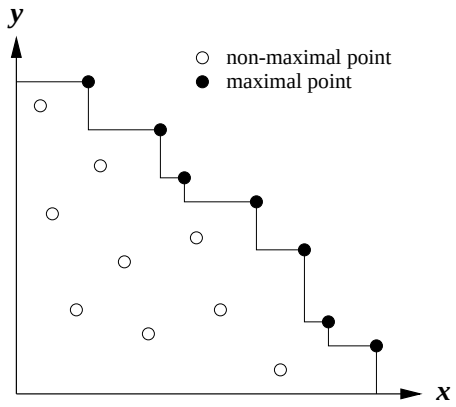

Figure 5.1: Maximal contour of a set of planar points.

Definition 5.3 *Let the set of all the maximal elements of S be denoted by $m(S)$. We are interested in the contour spanned by the maximal elements of S, called the $\mathcal{M}$ -*contour *of S which can be obtained by sorting the maximal elements in ascending order of their x-coordinates (see Figure 5.1). Let $\tilde{m}(S)$ denote the $\mathcal{M}$ -contour of S. Then both $m(S)$ and $\tilde{m}(S)$ represent the same set except that $\tilde{m}(S)$ is ordered.*

Computation of maximal elements is important in solving the *Largest Empty Rectangle Problem* [1, 16, 23, 25, 90, 91, 95] where a rectangle R, and a number of planar points $S \in R$, are given and the problem is to compute the largest rectangle $r \subseteq R$ that contains no point in S and whose sides are parallel to those of R. If R is divided into four quadrants then the maximal elements w.r.t. the northeast(**NE**), northwest(**NW**), southwest(**SW**), and southeast(**SE**) directions as depicted in Figure 5.2 remain the only candidates to be the supporting elements of the empty rectangles lying in all the four quadrants. The largest empty rectangle problem arises naturally in a number of applications including VLSI layout design [3], design rule checking [106], routing and testing [121], among many others.

We now present two important properties of the maximal contours, based on which efficient algorithms are developed in this chapter and in Chapters 7 and 8.

Lemma 5.1 *Every $\mathcal{M}$ -contour is sorted in descending order of the y-coordinates.*

Proof. Suppose the contrary holds. Then there exists at least one pair of maximal elements p and q such that $y(p) < y(q)$ while $x(p) \leq x(q)$, which contradicts with the

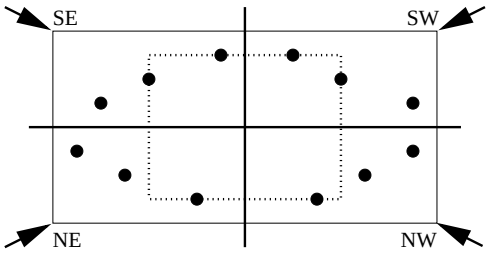

Figure 5.2: Importance of maximal elements in computing largest empty rectangle.

assumption that point p is maximal. ■

For any set P of planar points, let functions $min_x(P)$ and $max_x(P)$ denote the *minimum* and *maximum* x-coordinates in the set respectively. Let two more functions $min_y(P)$ and $max_y(P)$ be defined similarly w.r.t. the y-coordinate.

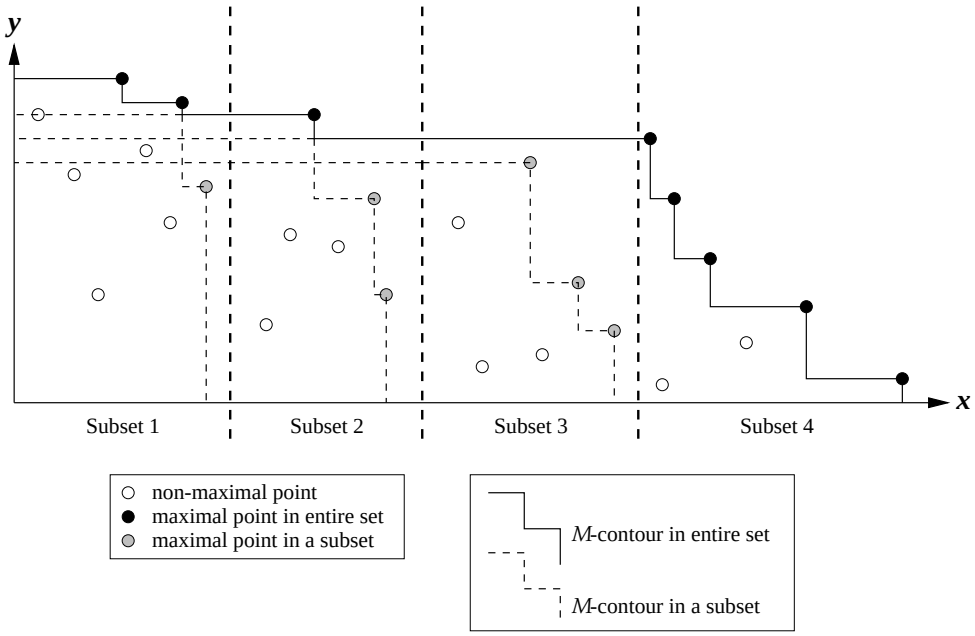

Figure 5.3: A property of $\mathcal{M}$ -contour which aids in parallelisation.

Lemma 5.2 *Given K sets $S_0, S_1, \ldots S_{K-1}$ of planar points such that*

$$max_x(S_t) \leq min_x(S_{t+1}), \;\; t = 0, \ldots, K-2$$

then, for every $p \in m(S_i)$, $i = 0, \ldots, K-1$,

$$p \in m\left(\bigcup_{t=0}^{K-1} S_t\right)$$

if and only if

$$y(p) > max_y(m(S_j)), \text{ for all } j = i+1, \ldots, K-1.$$

Proof. The *sufficiency* part can be proved by arranging a contradiction of Lemma 5.1. To prove the *necessity* part we consider, for some $i : 0 \leq i < K$, a point $p \in m(S_i)$ such that $p \notin m(\bigcup_{t=0}^{K-1} S_t)$. Then by the definition of maximality there exists some $q \in \bigcup_{t=i+1}^{K-1} S_t$ such that $p \prec q$, i.e., $y(p) \leq y(q)$. ■

The property of $\mathcal{M}$ -contour described in Lemma 5.2 has been illustrated in Figure 5.3 with $K = 4$.

5.2.2 Definition in Multi-Dimensional Space

The definition of dominance can be extended to a set S^k of k-dimensional points. Let a point at coordinate $(j_1, j_2, \ldots, j_k)$ be defined as $P(j_1, j_2, \ldots, j_k)$. Again, for any point p, let $x_i(p)$ denote the coordinate along the i-th axis. Thus for $1 \leq i \leq k$, $x_i(P(j_1, j_2, \ldots, j_k)) = j_i$.

Definition 5.4 *In k-dimensional space, a point* p dominates *a point* q *(denoted by* $q \prec p$*) if and only if* $x_i(q) \leq x_i(p)$ *for all* i, $1 \leq i \leq k$.

The definition of maximality in k-dimensional space remains the same as Definition 5.2:

Definition 5.5 *A point* $p \in S^k$ *is* maximal *if and only if there is no point* $q \in S$ *such that* $p \prec q$ *and* $p \neq q$.

The concept of contour is not applicable for $k \geq 3$ as the maximal points can only be partially ordered.

5.2.3 An Alternative Definition

The problem of finding maximal points in k-dimensional space is also known as the problem of finding the maxima of a set of k-dimensional vectors [46, 47, 122] as given below:

Let $U_1, U_2, \ldots, U_k$ be totally ordered sets and let V be a set of k-dimensional vectors in the Cartesian product $U_1 \times U_2 \times \cdots \times U_k$. For any vector $v \in V$, let $x_i(v)$ denote the i-th component of v. A partial ordering $\prec$ is defined on V in a natural way, i.e., for $u, v \in V$, $v \prec u$ if and only if $x_i(v) \leq x_i(u)$ for all $1 \leq i \leq k$. For $v \in V$, v is defined to be a *maximal element* of V if there does not exist $u \in V$ such that $v \prec u$ and $v \neq u$.

5.2.4 Lower Bound

Theorem 5.3 *The problem of sorting distinct numbers can be transformed into an $\mathcal{M}$ -contour problem of the same size.*

Proof. Consider sorting of N distinct integers $d_1, d_2, \ldots d_n$. Construct a set S of N planar points $\{P(d_1, -d_1), P(d_2, -d_2), \ldots, P(d_n, -d_n)\}$.

Lemma 5.4 *Each point $\in S$ is a maximal element of S.*

Proof. Let the point $P(d_i, -d_i) \in S$ dominate the point $P(d_j, -d_j) \in S$ for some i and j. Then we find $d_i \geq d_j$ and $-d_i \geq -d_j$ which contradict unless $i = j$ which is not considered. ■

Let the $\mathcal{M}$ -contour of S w.r.t. the northeast(**NE**) direction be the ordered set $\{P(d_{i_1}, -d_{i_1}), P(d_{i_2}, -d_{i_2}), \ldots, P(d_{i_n}, -d_{i_n})\}$ where $(i_1, i_2, \ldots, i_n)$ is a permutation of the set $\{1, 2, \ldots, n\}$. Obviously we can conclude that $d_{i_1} < d_{i_2} < \cdots < d_{i_n}$, i.e., the sorting order of the integers can be inferred from the $\mathcal{M}$ -contour of S. Hence the proof of Theorem 5.3 is completed. ■

Therefore, we can conclude that the time complexity for computing the contour of the maximal elements of N planar points is $\Omega(N \log N)$ on a serial computer and the AT^2 lower bound of an $\mathcal{M}$ -contour problem of size N is $\Omega(N^2)$. Kung *et al.* [47] have

given a direct proof of an $\Omega(N \log N)$ lower bound on a serial computer for finding the maxima of N k-dimensional vectors for any $k \geq 2$.

Dehne [24] has published an AT^2 optimal algorithm for solving an $\mathcal{M}$ -contour problem of size N on an ordinary mesh of size $N^{1/2} \times N^{1/2}$ in $O(N^{1/2})$ time. In Chapter 8 we present a new optimal $\mathcal{M}$ -contour algorithm on ordinary meshes which has better coefficient with the highest order term in the complexity than that in the complexity of Dehne's algorithm. We develop three constant time $\mathcal{M}$ -contour algorithms on reconfigurable meshes of various dimensions in Section 5.5. Using Lemma 6.3 of optimal simulation of a multi-dimensional reconfigurable mesh by a 2-dimensional reconfigurable mesh, it can easily be shown that all the three algorithms in Section 5.5 are AT^2 optimal. An adaptive optimal $\mathcal{M}$ -contour algorithm on reconfigurable meshes is presented in Chapter 7. In Chapter 7 we also extend our results on constrained reconfigurable meshes.

To avoid unnecessary complication, we assume that the bandwidth of any network considered in this chapter is wide enough to transmit two coordinates of a point simultaneously.

5.3 Computing $\mathcal{M}$ -Contour on Linear Arrays

Consider computing of the $\mathcal{M}$ -contour of a set S of N planar points on a linear array of N processors where each processor contains exactly one point at the start and at the end. We are interested in sorting the points w.r.t. the x-coordinate in ascending order from leftmost to the rightmost processors with the maximal elements tagged. Hence the output preserves the order of the maximal elements in the $\mathcal{M}$ -contour of the set but these maximal elements are not necessarily contained in consecutive processors.

By Theorem 5.3 in the light of Section 4.2, it can be concluded that the above computing must take at least $N-1$ steps if the communication links are bidirectional and $2N-1$ steps if the links are strictly unidirectional. An upper bound of the same order is achieved in the following algorithm by applying Lemma 5.2 with $K = N$:

Algorithm 5.2 Computing $\mathcal{M}$ -contour on a Linear Array

1 Sort S w.r.t. the x-coordinate of the points in ascending order from leftmost to the rightmost processors;

2 Each processor sets the variable M to be the y-coordinate of the point it contains;

3 Processor PE_{N-1} tags the point in it as a maximal element;

4 For $i = N-1, N-2, \ldots, 1$ do the following:
For $j = 0, 1, \ldots, i-1$ do the following in parallel:
Processor PE_j updates M with that of processor PE_{j+1};
If $M \geq$ the y-coordinate of the point then
Processor PE_j tags the point in it as a maximal element;

Theorem 5.5 *$\mathcal{M}$ -contour of N planar points on a linear array of N processors, where each processor has exactly one point, can be done in exactly $2N-1$ steps if the communication links are bidirectional and in exactly $3N-1$ steps if the links are strictly unidirectional.*

Proof. Phase 1 of Algorithm 5.2 takes exactly N steps by Theorem 4.1 if the communication links are bidirectional and $2N$ steps by Theorem 4.2 if the communication links are strictly unidirectional. Both phases 2 and 3 can be done in constant time without any communication. Phase 4 takes exactly $N-1$ steps. ■

5.4 Computing $\mathcal{M}$ -Contour on Ordinary Meshes

Consider computing of the contour of maximal elements of a set S of N planar points on an ordinary mesh of size $N^{1/2} \times N^{1/2}$ where each processor contains exactly one point at the start and at the end. We are interested in sorting the points w.r.t. the x-coordinate in snake-like order with the maximal elements tagged. Hence the output preserves the order of the maximal elements in the $\mathcal{M}$ -contour of the set but these maximal elements are not necessarily contained in consecutive processors.

In [24] Dehne has developed an optimal algorithm to compute the $\mathcal{M}$ -contour of N planar points on an $N^{1/2} \times N^{1/2}$ mesh. Dehne has applied Lemma 5.2 with $K = 2$ for computing the maximal elements using the well-known binary divide-and-conquer approach. For the sake of simplicity we assume $S = 2^q$ for some integer q.

Algorithm 5.3 Dehne's Algorithm [24]

1 *Sort S in snake-like order w.r.t. the x-coordinate of the points;*

2 *Divide S into two disjoint subsets L and R of equal size with $x(l) \leq x(r)$ for all $l \in L$ and $r \in R$;*

3 *Shift all points of L and R, respectively, to the left and right half of the mesh;*

4 *Recursively compute $m(L)$ and $m(R)$ in parallel;*

5 *Set $m(S) \leftarrow m(R)$;*

6 *Get $max_y(m(R))$ in the right half of the mesh;*

7 *Broadcast $max_y(m(R))$ to all the processors in the left half of the mesh;*

8 *For all $l \in m(L)$: if $y(l) \geq max_y(m(R))$ then set $m(S) \leftarrow m(S) \cup \{l\}$;*

9 *Only at the first level of recursion, sort all the points in snake-like order w.r.t. the x-coordinate of the points to get $\tilde{m}(S)$;*

In [24] Dehne has shown that the complexity of Algorithm 5.3 satisfies

$$T(N) = T\left(\frac{N}{2}\right) + c\sqrt{N}$$

where c is a constant and thus Dehne concludes $T(N) = O(\sqrt{N})$.

Theorem 5.6 *The $\mathcal{M}$ -contour of N planar points on an ordinary mesh of size $N^{1/2} \times N^{1/2}$ can be computed in $O(N^{1/2})$ time.* ■

Dehne has not provided any scheme of dividing the mesh into halves, which we show, in Chapter 8, plays an important role in achieving optimality as shown in the

above analysis. In Chapter 8 we carry out detailed analysis of Dehne's algorithm to obtain the minimum possible coefficient of the highest order term in the complexity which the Big-Oh expression hides. There we also develop a new optimal algorithm to compute the $\mathcal{M}$ -contour of N planar points on an $N^{1/2} \times N^{1/2}$ mesh and we compare these two asymptotically optimal order algorithms by evaluating the exact coefficient of the highest order term in the complexity and the new algorithm comes out far better than Dehne's algorithm.

5.5 Computing $\mathcal{M}$ -contour on Reconfigurable Meshes

Given a binary sequence, b_j, $j = 0, 1, \ldots, N-1$, the *and* computation is to compute $b_0 \wedge b_1 \wedge \cdots \wedge b_{N-1}$. Similarly the *or* computation computes $b_0 \vee b_1 \vee \cdots \vee b_{N-1}$. These computations play a significant role in developing our algorithms in this section.

Lemma 5.7 *Given a binary sequence of length N in the only row of a reconfigurable mesh of size* $1 \times N$*, both the* and *and the* or *of the elements in the sequence can be computed in* $O(1)$ *time.*

Proof. The proof adapts the technique of *bus splitting* [66]. First consider the *and* computation. Every processor connects port **E** with **W** if the processor carries a 1; otherwise it disconnects the ports. Now a specific message # is written to port **E** of the rightmost processor. If this message is read from port **W** of the leftmost processor then the result is 1; otherwise the result is 0.

The *or* computation can be done similarly. Every processor connects port **E** with **W** if the processor carries a 0; otherwise it disconnects the ports. Now a specific message # is written to port **E** of the rightmost processor. If this message is read from port **W** of the leftmost processor then the result is 0; otherwise the result is 1. ■

We develop three constant time algorithms for computing the $\mathcal{M}$ -contour of a set of N planar points. The first algorithm MAXIMAL1, in Section 5.5.1, uses a 2-dimensional reconfigurable mesh of size $N \times N$ while the second algorithm MAXIMAL2, in Section 5.5.2, requires a 3-dimensional reconfigurable mesh of size $N^{1/2} \times$

$N^{1/2} \times N^{1/2}$. The third algorithm MAXIMAL3, in Section 5.5.3, is an extension of algorithm MAXIMAL2 to higher dimensions. An algorithm similar to algorithm MAXIMAL1 appears in [37] to compute three dimensional maxima. In fact algorithm MAXIMAL1 can also be used to compute multi-dimensional maxima in constant time.

5.5.1 A Brute Force Algorithm

Consider computing the $\mathcal{M}$ -contour of N planar points on a 2-dimensional reconfigurable mesh. Using the same arguments as in Section 4.4 and Theorem 5.3 it can be proved that to develop constant time AT^2 optimal algorithm to solve the above problem, the size of the reconfigurable mesh must be at least $N \times N$.

We develop an AT^2 optimal constant time algorithm on a square mesh in a very straightforward way. It is not difficult to realise that a brute force algorithm can be designed to compute the $\mathcal{M}$ -contour of N planar points using $O(N^2)$ comparisons. We distribute these comparisons among N^2 processors to achieve constant time complexity.

N planar points are given in the row of processors $PE_{*,0}$. These points, after sorting, are distributed over the reconfigurable mesh through column and row broadcast in such a way that each column of processors $PE_{i,*}$, $0 \leq i, < N$, computes the dominance of all other points over the i-th point. Then each column computes the logical *and* of the previous dominance decision to assert whether the point represented by that column is a maximal point or not. As all the points are already sorted, the $\mathcal{M}$ -contour is obtained simply by following this sorted sequence. The detailed description of the algorithm is given below.

Algorithm 5.4 MAXIMAL1

Precondition: *Registers r_0 and r_1 hold x- and y-coordinates respectively.*

Postcondition: *Register r_2 holds the decision of maximality.*

1 *Sort the given N points in the row of processors $PE_{*,0}$ in ascending order of register r_0, i.e., in ascending order of the x-coordinates. The sorted list also*

resides in the row of processors $PE_{*,0}$ in ascending row-major order;

2 *b: Every processor connects port* **N** *with* **S**;

w: Every processor $\in PE_{*,0}$ writes register r_0 to port **N**;

r: Every processor reads port **N** *into register r_0;*

3 *b: Every processor connects port* **N** *with* **S**;

w: Every processor $\in PE_{*,0}$ writes register r_1 to port **N**;

r: Every processor reads port **N** *into register r_1;*

4 *b: Every processor connects port* **E** *with* **W**;

w: Every processor $PE_{i,i}$, $0 \leq i < N$, writes register r_0 to port **E**;

r: Every processor reads port **E** *into register r_2;*

5 *b: Every processor connects port* **E** *with* **W**;

w: Every processor $PE_{i,i}$, $0 \leq i < N$, writes register r_1 to port **E**;

r: Every processor reads port **E** *into register r_3;*

6 *b: Every processor $PE_{i,j}$, $0 \leq i, j < N$, does the following:*

If $i \neq j$ and $P(r_0, r_1) \prec P(r_2, r_3)$ then

disconnect all the ports;

Else

connect port **N** *with* **S**;

w: Every processor $\in PE_{*,N-1}$ writes an arbitrary constant # to port **N**;

r: Every processor $\in PE_{*,0}$ reads port **S** *into register r_2;*

c: Every processor $\in PE_{*,0}$ does the following:

If $r_2 = \#$ then

set register $r_2 = 1$;

Else

set register $r_2 = 0$;

Theorem 5.8 *Given N planar points in a row of processors, the $\mathcal{M}$ -contour of these points can be obtained in $O(1)$ time using a reconfigurable mesh of size $N \times N$.*

Proof. Phase 1 of algorithm MAXIMAL1 can be computed in constant time using Lemma 4.8. Phases 2–5 require $O(1)$ time. Phase 6 is an elaboration of computing the *and* operation on a binary sequence and thus requires constant time by Lemma 5.7. ■

5.5.2 A Divide-and-Conquer Algorithm

In this section we consider computing the $\mathcal{M}$ -contour of N planar points on a 3-dimensional reconfigurable mesh of size $N^{1/2} \times N^{1/2} \times N^{1/2}$. We use an $N^{1/2}$-ary divide-and-conquer approach to compute the $\mathcal{M}$ -contour. N points are given in the plane of processors $PE_{*,*,0}$. These points are sorted in order of x-coordinate to divide them into $N^{1/2}$ disjoint sets of length $N^{1/2}$ each. This division complies with the first condition in Lemma 5.2. Now the $\mathcal{M}$ -contour of the i-th smaller set is computed using the 2-dimensional submesh of processors $PE_{i,*,*}$, $0 \leq i < N^{1/2}$. Merging of the solutions of these smaller problems is then done by carefully utilising Lemma 5.2. Lemma 5.1 helps in getting the max_y of each smaller $\mathcal{M}$ -contour in constant time (phase 3). The i-th max_y is then distributed over the plane of processors $PE_{*,*,i}$ (phases 4–6). Every point in each smaller $\mathcal{M}$ -contour then computes its overall maximality using the processors along the z-axis (phase 7–8). The detailed description of the algorithm is given below.

Algorithm 5.5 MAXIMAL2

Precondition: *Registers r_0 and r_1 hold x- and y-coordinates respectively.*

Postcondition: *Register r_2 holds the decision of maximality.*

1 *Sort the given N points in the plane of processors $PE_{*,*,0}$ in ascending order of register r_0, i.e., in ascending order of the x-coordinates. The sorted list also resides in the plane of processors $PE_{*,*,0}$ in ascending column-major order;*

2 *For every column i, $0 \leq i < N^{1/2}$, the $\mathcal{M}$ -contour of the $N^{1/2}$ points residing in the i-th column of processors $PE_{i,*,0}$ is computed using the algorithm MAXIMAL1 on the 2-dimensional submesh of processors $PE_{i,*,*}$. Here phase 1 of algorithm MAXIMAL1 should be ignored;*

3 *b: Every processor* $\in PE_{*,*,0}$ *does the following:*
If $r_2 = 0$ *then*
connect port **N** *with* **S**;
Else
disconnect all the ports;

w: Every processor $\in PE_{*,*,0}$ *does the following:*
If $r_2 = 1$ *then*
write register r_1 *to port* **S**;

r: Every processor $\in PE_{*,0,0}$ *reads port* **S** *into register* r_3;

4 *b: Every processor* $\in PE_{*,0,*}$ *connects port* **U** *with* **D**;

w: Every processor $\in PE_{*,0,0}$ *writes register* r_3 *to port* **U**;

r: Every processor $PE_{i,0,i}$, $0 \leq i < N^{1/2}$, *reads port* **U** *into register* r_3;

5 *b: Every processor* $\in PE_{*,0,*}$ *connects port* **E** *with* **W**;

w: Every processor $PE_{i,0,i}$, $0 \leq i < N^{1/2}$, *writes register* r_3 *to port* **E**;

r: Every processor $\in PE_{*,0,*}$ *reads port* **E** *into register* r_3;

6 *b: Every processor connects port* **N** *with* **S**;

w: Every processor $\in PE_{*,0,*}$ *writes register* r_3 *to port* **N**;

r: Every processor reads port **N** *into register* r_3;

7 *b: Every processor connects port* **U** *with* **D**;

w: Every processor $\in PE_{*,*,0}$ *writes register* r_1 *to port* **U**;

r: Every processor reads port **U** *into register* r_4;

8 *b: Every processor* $PE_{i,j,k}$, $0 \leq i, j, k < N^{1/2}$, *does the following:*
If $k > i$ *and* $r_4 \leq r_3$ *then*
disconnect all the ports;
Else
connect port **U** *with* **D**;

w: Every processor $\in PE_{*,*,N^{1/2}-1}$ *writes an arbitrary constant # to port* **U**;

r: Every processor $\in PE_{*,*,0}$ *reads port* **D** *into register* r_4;

c: Every processor $\in PE_{*,*,0}$ *does the following:*
If $r_2 = 1$ *and* $r_4 \neq \#$ *then*

set register $r_2 = 0$;

Theorem 5.9 *Given N planar points in a plane of processors, the $\mathcal{M}$ -contour of these points can be obtained in $O(1)$ time using a 3-dimensional reconfigurable mesh of size $N^{1/2} \times N^{1/2} \times N^{1/2}$.*

Proof. Phase 1 of algorithm MAXIMAL2 can be computed in constant time using Theorem 4.9. By Theorem 5.8 phase 2 can also be done in constant time. It is obvious that the remaining steps require $O(1)$ time. ■

5.5.3 Extension to Higher Dimensions

Based on Lemma 5.1 and Lemma 5.2, the $\mathcal{M}$ -contour problem can be solved in a recursive way. We derive the $\mathcal{M}$ -contour algorithm on a k-dimensional reconfigurable mesh from the $\mathcal{M}$ -contour algorithm on a $(k-1)$-dimensional reconfigurable mesh. The size of the k-dimensional reconfigurable mesh adopted here is $N^{1/(k-1)} \times N^{1/(k-1)} \times \cdots \times N^{1/(k-1)}$. The algorithm developed for k-dimensional reconfigurable mesh is very similar to algorithm MAXIMAL2. In fact the technique applied in algorithm MAXIMAL2 is simply extended to higher dimensions. This extension is made possible by the availability of the sorting algorithm stated in Theorem 4.10. The detailed description of the algorithm is given below.

Algorithm 5.6 MAXIMAL3

Precondition: *Registers r_0 and r_1 hold x- and y-coordinates respectively.*

Postcondition: *Register r_2 holds the decision of maximality.*

1 *Sort the given N points in the hyperplane of processors $PE_{*,*,\ldots,*,0}$ in ascending order of register r_0, i.e., in ascending order of the x-coordinates. The sorted list also resides in the hyperplane of processors $PE_{*,*,\ldots,*,0}$;*

2 *For every $(k-2)$-flat i, $0 \leq i < N^{1/(k-1)}$, the $\mathcal{M}$ -contour of the $N^{(k-2)/(k-1)}$ points residing in the i-th $(k-2)$-flat of processors $PE_{i,*,\ldots,*,0}$ is computed on the $(k-1)$-dimensional submesh of processors $PE_{i,*,\ldots,*,*}$. If $k-1 \geq 4$ we simply recursively use the algorithm MAXIMAL3 else algorithm MAXIMAL2 is used. Here phase 1 of algorithms MAXIMAL2 and MAXIMAL3 should be ignored;*

3 *Using $k-2$ bus splittings based on register r_2, one along each axis except the 1st and the k-th axes, the max_y of the $\mathcal{M}$ -contour of the i-th subset, which is the first maximal element according to Lemma 5.1, is read into register r_3 of the processor $PE_{i,0,\ldots,0,0}$, $0 \leq i < N^{1/(k-1)}$;*

4 *Using a single broadcast along the k-th axis transfer the content of register r_3 of the processor $PE_{i,0,\ldots,0,0}$ into register r_3 of the processor $PE_{i,0,\ldots,0,i}$, $0 \leq i < N^{1/(k-1)}$;*

5 *Using $k-1$ broadcasts, one along each axis except the k-th axis, transfer the content of register r_3 of the processor $PE_{i,0,\ldots,0,i}$ into the register r_3 of the hyperplane of processors $PE_{*,*,\ldots,*,i}$, $0 \leq i < N^{1/(k-1)}$;*

6 *b: Every processor connects ports along the k-th axis;*

w: Every processor $\in PE_{*,*,\ldots,*,0}$ writes register r_1 to the top port along the k-th axis;

r: Every processor reads the top port along the k-th axis into register r_4;

7 *Using a single bus splitting along the k-th axis, based on registers r_3 and r_4, compute the overall maximality decision into register r_2 of all the processors $\in PE_{*,*,\ldots,*,0}$;*

Theorem 5.10 *Given N planar points in a hyperplane of processors, the $\mathcal{M}$ -contour of these points can be obtained in $SORT(N,k)+O(k^2)$ time using a k-dimensional reconfigurable mesh of size $N^{1/(k-1)} \times N^{1/(k-1)} \times \cdots \times N^{1/(k-1)}$, $k \geq 4$ where $SORT(N,k)$ denotes the time required by the sorting algorithm used in phase 1 of algorithm MAXIMAL3.*

Proof. Phase 1 of algorithm MAXIMAL3 is used only once. Phases 3–7 take only $O(k)$ time. Considering the recursion in phase 2, the overall complexity of algorithm MAXIMAL3 then can be expressed as $SORT(N,k)+O(k)+O(k-1)+\cdots+O(1) \cong SORT(N,k)+O(k^2)$. ■

From Theorem 4.10, $SORT(N,k) = O(4^k)$. Thus, Theorem 5.10 gives constant time complexity for any fixed $k \geq 4$.

5.6 Conclusions

In this chapter we have developed AT^2 optimal constant time algorithms to solve $\mathcal{M}$ -contour problems on 2-dimensional, 3-dimensional, and k-dimensional reconfigurable meshes, where $k \geq 4$. To our knowledge this problem on the reconfigurable mesh has been examined here for the first time (except for the author's papers [71, 74]).

Our algorithm on 2-dimensional reconfigurable meshes can also be easily adapted to compute maximal elements of a set of points in multi-dimensional space AT^2 optimally. It is still an open problem whether AT^2 optimal algorithm exists for solving the above problem on higher dimensional reconfigurable meshes.

In Chapter 7 we develop an AT^2 optimal adaptive $\mathcal{M}$ -contour algorithm to support our idea. Optimal $\mathcal{M}$ -contour algorithms presented in this chapter on linear arrays, ordinary meshes, and reconfigurable meshes play significant roles in the development of the above algorithm.

Self-simulation of various reconfigurable meshes is the topic of the next chapter.

Chapter 6

Self-Simulation of Reconfigurable Meshes

Can the reconfigurable mesh be the basis for the design of next generation of massively parallel computers? Perhaps the answer significantly depends on solving the most fundamental *problem of scaling down algorithms*: Given an algorithm which is designed for a large reconfigurable mesh, can it be executed efficiently on a smaller reconfigurable mesh? As mentioned in Chapter 1, the scaling down problem can be solved in more than one way of which *self-simulation* appears to be the obvious one where a simulation program takes the responsibility to execute each step of the original algorithm on the smaller mesh which self-simulates the larger mesh through some sort of processor mapping.

The aim of this chapter is to develop efficient optimal self-simulation algorithms on some restricted reconfigurable mesh models. A literature review on self-simulation of reconfigurable meshes is presented in the next section. In Section 6.2 we establish that it is sufficient to consider self-simulating only for 2-dimensional reconfigurable meshes. The problem of self-simulation and some relevant terminologies are defined in Section 6.3. In Section 6.4 we discuss a number of processor mapping schemes used in self-simulating various reconfigurable mesh models. Self-simulation algorithms based on contraction mapping are included in Section 6.5. In Section 6.6 we discuss a number of self-simulation algorithm where simulation is transformed into the problem of computing the connected components of graphs. In Section 6.7 we develop a simple generic self-simulation algorithm SIMPLE using only window traversal and avoiding any computation of connected components. Two new reconfigurable mesh

models are also proposed in this section and it is shown that algorithm SIMPLE can provide optimal (in some cases asymptotically optimal) self-simulation for these models. In Section 6.8 we devalues the concept of self-simulation of reconfigurable meshes by showing that even with optimal slowdown, the resultant algorithms fail to remain AT^2 optimal when a large reconfigurable mesh is self-simulated on a smaller mesh for which AT^2 optimal algorithms exist.

6.1 Introduction

Although efficient self-simulation carries crucial responsibilities in making the reconfigurable mesh widely acceptable, very few works have so far been directed on the reconfigurable mesh along this direction. Maresca and Li [60] have shown for the first time that optimal (see Definition 6.4) self-simulation is not possible for the general model of reconfigurable meshunder the contraction mapping (Section 6.4.1) of processors. Maresca [58] has then proposed a new architecture *polymorphic processor arrays*, which is identical to the HV-RM model, for which optimal self-simulation exists.

Ben-Asher *et al.* [4] have addressed the self-simulation issues of reconfigurable meshes with in-depth and rigorous analysis for the first time. They have developed optimal self-simulation algorithms for the HV-RM and the LRM models. A weak (see Definition 6.4) self-simulation algorithm for the general model has also been developed in [4].

Fernández-Zepeda *et al.* [31] have developed a strong (see Definition 6.4) self-simulation algorithm for the FR model. They have also presented a weak self-simulation algorithm for the general model with smaller slowdown than that of the weak self-simulation algorithm in [4]. Very recently, Matsumae and Tokura [62] have developed a new weak self-simulation algorithm for the general model with slowdown similar to that of the algorithm in [31].

In Section 6.7.2 we propose two new reconfigurable mesh models MB and PMB, within the LRM model, where restrictions are imposed on the global characteristics of bus reconfigurations. In Section 6.7.1 we present a simple generic self-simulation algorithm which remains optimal and asymptotically optimal while self-simulating

the MB and the PMB models respectively.

Some works have been carried in the direction of simulating higher dimensional reconfigurable mesh using a larger 2-dimensional reconfigurable mesh. It has been shown that any constant degree reconfigurable mesh can be simulated with no slow-down by a 2-dimensional reconfigurable mesh, paying by a quadratic blow-up of the number of processors, which in fact is the lower bound [102, 114].

Matias and Schuster [61] have proposed a randomised self-simulation algorithm for the general model with optimal slowdown. Recently self-simulation algorithms on undirected reconfigurable networks are presented in [9].

6.2 Dimensional Choice in Self-Simulation

Definition 6.1 *Let M_1 and M_2 denote two parallel computational models. Model M_1 is considered* as powerful as *model M_2 if and only if each step in M_2 with P processors can be simulated in constant time by M_1 with $O(P^{\epsilon})$ processors where ϵ is a small constant ≥ 1. Model M_1 is considered* more powerful than *model M_2 if and only if M_1 is as powerful as M_2 but M_2 is not as powerful as M_1.*

In this section we establish that it is sufficient self-simulating only 2-dimensional reconfigurable meshes. One of the most attractive features of a 2-dimensional reconfigurable mesh, as opposed to a d-dimensional reconfigurable mesh, where $d > 2$, is that it has a simple and straightforward VLSI layout. In Section 6.2.1 it is established that not only a 2-dimensional reconfigurable mesh is as powerful as a d-dimensional reconfigurable mesh but also any d-dimensional reconfigurable mesh, where $d > 2$, can be simulated by a 2-dimensional reconfigurable mesh within a constant factor of VLSI area. In Section 6.2.2 we show that a 2-dimensional reconfigurable mesh is more powerful than a 1-dimensional reconfigurable mesh (reconfigurable linear array).

6.2.1 2-Dimensional vs Higher Dimensional Meshes

Lemma 6.1 *For any constant $d \geq 2$, a d-dimensional reconfigurable mesh of size $P_0 \times P_1 \times \cdots \times P_{d-1}$, where $P_{d-1} \geq P_0, P_1, \ldots, P_{d-2}$, requires $\Omega\left(\prod_{i=0}^{d-2} {P_i}^2\right)$ VLSI layout area.*

Proof. See [114]. ■

It is not hard to realise why the above theorem holds. The bisection width of a d-dimensional mesh of size $N \times N \times \cdots \times N$ is at least N^{d-1} [51, pp. 223–226] and the area needed to lay out a network with bisection width B is $\Omega(B^2)$ [50, pp. 67].

Based on the above theorem a lower bound on the number of processors required by a 2-dimensional reconfigurable mesh to simulate a higher dimensional reconfigurable mesh in constant time can be given as follows:

Corollary 6.2 *Any 2-dimensional reconfigurable mesh that simulates a d-dimensional reconfigurable mesh of size $P_0 \times P_1 \times \cdots \times P_{d-1}$, where $d > 2$ is a constant and $P_{d-1} \geq P_0, P_1, \ldots, P_{d-2}$, in constant time must have $\Omega\left(\prod_{i=0}^{d-2} {P_i}^2\right)$ processors.*

This lower bound has indeed been achieved, within a constant factor if P_0 and P_{d-1} are assumed to be the largest two dimensions and of the same order, by Vaidyanathan and Trahan [114] as shown in the following lemma:

Lemma 6.3 *For any constant $d > 2$, a d-dimensional reconfigurable mesh of size $P_0 \times P_1 \times \cdots \times P_{d-1}$ can be simulated by a 2-dimensional reconfigurable mesh of size*

$$\left(2d \prod_{i=1}^{d-1} P_i\right) \times \left((2d+1)P_0 + \sum_{i=0}^{d-2} \prod_{j=0}^{i} P_j\right)$$

in constant time.

Proof. See [114]. ■

In [102] Schuster has presented a similar simulation which requires $\Theta\left(\prod_{i=0}^{d-1} {P_i}^2\right)$ processors.

Based on the above discussion we may conclude the following:

Theorem 6.4 *A 2-dimensional reconfigurable mesh is as powerful as a d-dimensional reconfigurable mesh, for any constant $d > 2$.* ■

6.2.2 2-Dimensional vs 1-Dimensional Meshes

By showing that a priority CRCW PRAM with P^2 processors can simulate 1-dimensional reconfigurable array of P processors in constant time [114], Vaidyanathan and Trahan have shown that

Lemma 6.5 *A priority CRCW PRAM is at least as powerful as a 1-dimensional reconfigurable array.* ■

Hence, the following theorem can be concluded from Lemmas 2.2 and 6.5:

Theorem 6.6 *A 2-dimensional reconfigurable mesh is more powerful than a 1-dimensional reconfigurable array.* ■

Based on Theorems 6.4 and 6.6, it can be easily concluded that it is sufficient to self-simulate only 2-dimensional reconfigurable meshes.

6.3 Definitions and Terminologies

Let $\mathrm{RM}_{A\times B}$ denote a reconfigurable mesh of A rows and B columns.

Definition 6.2 *The self-simulation problem of reconfigurable mesh is to step-by-step simulate* $\mathrm{RM}_{M\times N}$ *(simulated mesh) by* $\mathrm{RM}_{P\times Q}$ *(simulating mesh) where* $P \leq M$, $Q \leq N$, *and the computing power of the processors and the bus bandwidth (not less than* $\log MN$*) are assumed to be equivalent in both the meshes. Moreover it is assumed that a processor of the simulating mesh has* $\Theta\left(\left\lceil\frac{M}{P}\right\rceil\left\lceil\frac{N}{Q}\right\rceil\right)$ *times more registers (memory locations) than those of a processor of the simulated mesh and this will be referred to as the* optimal space *assumption of self-simulation.*

To simplify the exposition $\frac{M}{P}$ and $\frac{N}{Q}$ are assumed to be integers. If the memory requirement of the simulating mesh is bounded as defined in the above definition then the *slowdown* remains as the key issue.

Definition 6.3 *We say that the simulating mesh simulates the simulated mesh with* slowdown *S if the result for any arbitrary algorithm $\mathcal{A}$ on the simulated mesh is achieved through the execution of a step-by-step simulation algorithm $\mathcal{B}$ on the simulating mesh in which each step of $\mathcal{A}$ is simulated with at most S steps.*

Clearly, the slowdown of simulating $\text{RM}_{M\times N}$ by $\text{RM}_{P\times Q}$ will be $\Omega\left(\frac{M}{P}\frac{N}{Q}\right)$ as the work of MN processors will be performed on only PQ processors.

Definition 6.4 *For any $P \leq M$ and $Q \leq N$, let $\text{RM}_{P\times Q}$ simulates $\text{RM}_{M\times N}$ with slowdown $O\left(\frac{M}{P}\frac{N}{Q}f(M,N,P,Q)\right)$ where $f(M,N,P,Q) \geq 1$.*

1. *The above simulation is* optimal *if and only if $f(M,N,P,Q) = O(1)$.*
2. *The above simulation is* strong *if and only if $f(M,N,P,Q)$ is a function of only P and Q.*
3. *The above simulation is* weak *if and only if it is neither optimal nor strong.*

An ordinary mesh can always be self-simulated optimally as the links of a processor are used only for exchanging data among neighbouring processors. It is still an open problem whether optimal self-simulation exists for the general reconfigurable mesh model where the links of a processor can act as computing elements besides their normal data transmission usage.

6.4 Processor Mapping

Mapping of processors of the simulated mesh onto the processors of the simulating mesh plays a significant role in self-simulation. In Sections 6.4.1–6.4.3 three types of mapping used so far in the literature are discussed.

Let the processors of the simulated mesh $\text{RM}_{M\times N}$ and the simulating mesh $\text{RM}_{P\times Q}$ be denoted by the matrices $R[0:M-1,0:N-1]$ and $S[0:P-1,0:Q-1]$ respectively. Let $R(x,y)$, $0 \leq x < M$ and $0 \leq y < N$, denote the processor at the intersection of row x and column y of the simulated mesh. Similarly let the processor at the intersection of row x and column y of the simulating mesh be denoted by $S(x,y)$, $0 \leq x < P$ and $0 \leq y < Q$.

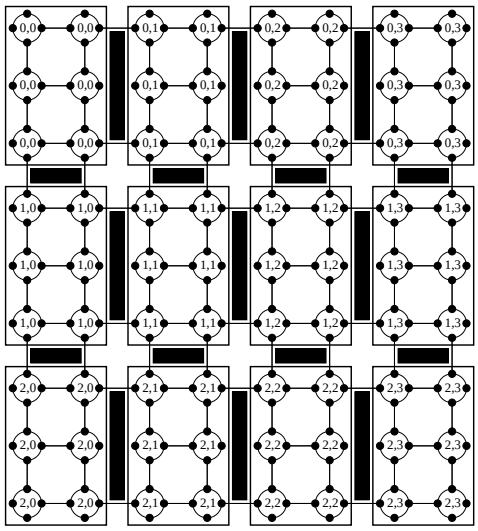

Figure 6.1: Contraction mapping of $\mathrm{RM}_{9\times 8}$ onto $\mathrm{RM}_{3\times 4}$ where number-pairs indicate processors of the simulating mesh.

6.4.1 Contraction Mapping

The most obvious approach to mapping processors of $\mathrm{RM}_{M\times N}$ to those of $\mathrm{RM}_{P\times Q}$ is to let processor $S(x,y)$ simulate the submesh of processors $R[x\frac{M}{P} : (x+1)\frac{M}{P}-1, y\frac{N}{Q} : (y+1)\frac{N}{Q}-1]$ for $0 \le x < P$ and $0 \le y < Q$. Ben-Asher *et al.* [4] have named this the *contraction mapping* and an example of this mapping is given in Figure 6.1.

Now each processor of the simulating mesh simulates $\frac{M}{P}\frac{N}{Q}$ processors of the simulated mesh. Let the processors of the simulated mesh have r registers each. Suppose an extra ε, independent of r, M, N, P, Q, registers are required per simulated processor for simulation purpose. An obvious mapping of registers under optimal space is that the k-th register of the simulated processor $R(x,y)$ is mapped onto the $\left(\left(\left(x \bmod \frac{M}{P}\right)\frac{N}{Q} + \left(y \bmod \frac{N}{Q}\right)\right)(r+\varepsilon) + k\right)$-th register of the corresponding simulating processor $S\left(\left\lfloor x/\frac{M}{P}\right\rfloor, \left\lfloor y/\frac{N}{Q}\right\rfloor\right)$ where $0 \le x < M$, $0 \le y < N$, $0 \le k < r$.

6.4.2 Windows Mapping

Let the simulated mesh $\mathrm{RM}_{M\times N}$ be divided into $\frac{M}{P}\frac{N}{Q}$ non-overlapping submeshes (windows) $R_{i,j}$ of size $P\times Q$ containing the processors $R[iP : (i+1)P-1, jQ : (j+1)Q-1]$ for $0 \le i < \frac{M}{P}$ and $0 \le j < \frac{N}{Q}$. In *windows mapping* [4] the simulated mesh is mapped into the

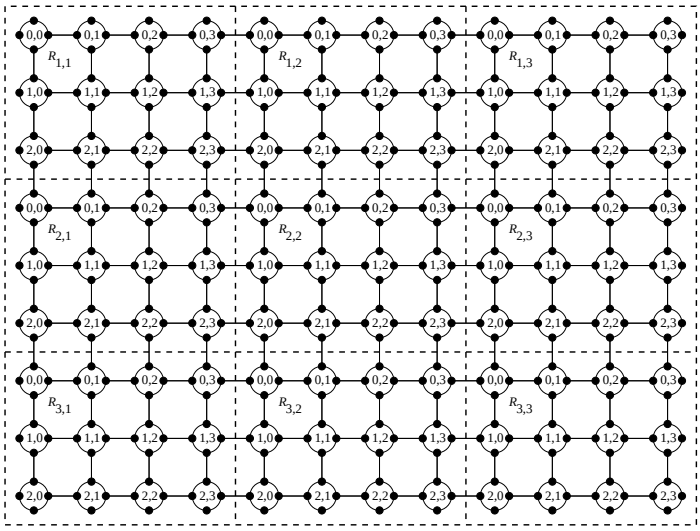

Figure 6.2: Windows mapping of $\mathrm{RM}_{9\times 8}$ onto $\mathrm{RM}_{3\times 4}$ where number-pairs indicate processors of the simulating mesh.

simulating mesh $\mathrm{RM}_{P\times Q}$ in such a way that $R(x,y)$ is simulated by $S(x \bmod P, y \bmod Q)$ for $0 \le x < M$ and $0 \le y < N$. This ensures one-to-one processor mapping of each simulated submesh $R_{i,j}$ onto the simulating mesh and whenever the simulating mesh simulates the submesh $R_{i,j}$ the processor $S(x,y)$ simulates the processors $R(iP+x, jQ+y)$ for $0 \le i < \frac{M}{P}$, $0 \le j < \frac{N}{Q}$, $0 \le x < M$ and $0 \le y < N$. An example of windows mapping is given in Figure 6.2.

As in the contraction mapping, each processor of the simulating mesh simulates $\frac{M}{P}\frac{N}{Q}$ processors of the simulated mesh. Let the processors of the simulated mesh have r registers each. Suppose an extra ε, independent of r, M, N, P, Q, registers are required per simulated processor for simulation purpose. An obvious mapping of registers under optimal space is that the k-th register of the simulated processor $R(x,y)$ is mapped onto the $\left(\left(\lfloor x/P \rfloor \frac{N}{Q} + \lfloor y/Q \rfloor\right)(r+\varepsilon) + k\right)$-th register of the corresponding simulating processor $S(x \bmod P, y \bmod Q)$ where $0 \le x < M$, $0 \le y < N$, $0 \le k < r$.

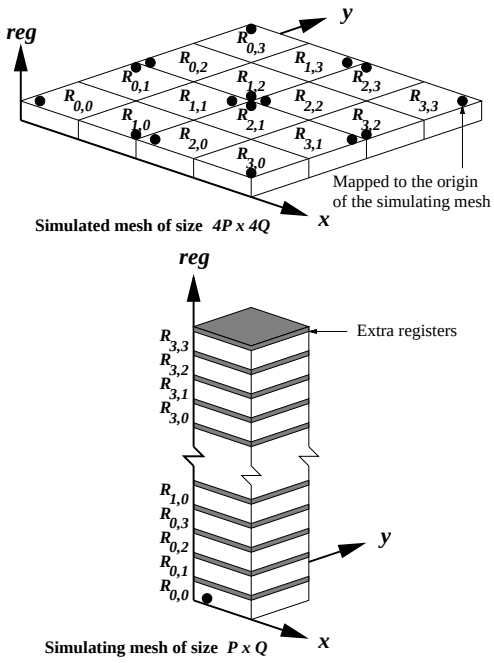

Figure 6.3: Folded-windows mapping of $\mathrm{RM}_{4P\times 4Q}$ onto $\mathrm{RM}_{P\times Q}$

6.4.3 Folded-Windows Mapping

The following two functions [4] play an important role in *folded-windows mapping* of the simulated mesh onto the simulating meshes:

$$\begin{aligned} \mathrm{FOLD}(a,b) &= \left\{ \begin{array}{ll} a \bmod b & \text{if } a \text{ div } b \text{ is even} \\ b-1-(a \bmod b) & \text{otherwise} \end{array} \right\} \\ \mathrm{UNFOLD}(a,b,c) &= \left\{ \begin{array}{ll} bc+a & \text{if } c \text{ is even} \\ b(c+1)-1-a & \text{otherwise} \end{array} \right\} \end{aligned} \tag{6.1}$$

As in the windows mapping, the simulated mesh $\mathrm{RM}_{M\times N}$ is divided into $\frac{M}{P}\frac{N}{Q}$ non-overlapping submeshes (windows) $R_{i,j}$ of size $P\times Q$ containing the processors $R[iP:(i+1)P-1, jQ:(j+1)Q-1]$ for $0\le i<\frac{M}{P}$ and $0\le j<\frac{N}{Q}$. Now, in *folded-windows mapping* [4] the simulated mesh is mapped into the simulating mesh $\mathrm{RM}_{P\times Q}$ in such a way that $R(x,y)$ is simulated by $S(\mathrm{FOLD}(x,P),\mathrm{FOLD}(y,Q))$ for $0\le x<M$ and $0\le y<N$. Like the windows mapping, folded-windows mapping also ensures one-to-one processor mapping of each simulated submesh $R_{i,j}$ onto the simulating mesh and whenever the simulating mesh simulates the submesh $R_{i,j}$ the processor $S(x,y)$ simulates the

processors $R(\text{UNFOLD}(x,P,i),\text{UNFOLD}(y,Q,j))$ for $0 \le i < \frac{M}{P}$, $0 \le j < \frac{N}{Q}$, $0 \le x < M$ and $0 \le y < N$. An example of folded-windows mapping is given in Figure 6.3.

The folded-windows mapping differs from the windows mapping for its unique characteristics that the external neighbours of a boundary processor, p of the submesh $R_{i,j}$ are mapped in the same simulating processor where p is also mapped, $0 \le i < \frac{M}{P}$ and $0 \le j < \frac{N}{Q}$. This keeps the broadcasts of simulation data low at the expense of the introduction of a mapping of the ports due to the change in the direction of x- and/or y-axes in some of the mapped submeshes. For the submesh $R_{i,j}$, $0 \le i < \frac{M}{P}$ and $0 \le j < \frac{N}{Q}$, the ports are mapped as follows:

$$MAP(\pi,i,j) = \begin{cases} \pi & \text{if } (\,\pi \in \{\mathbf{E},\mathbf{W}\} \wedge i \text{ even }) \vee (\,\pi \in \{\mathbf{N},\mathbf{S}\} \wedge j \text{ even }) \\ \tilde{\pi} & \text{otherwise} \end{cases}$$

where $\tilde{\pi}$ denotes the opposite port of π, e.g. $\tilde{\mathbf{E}}$ is $\mathbf{W}$.

The register mapping in the folded-windows mapping, assuming optimal space self-simulation, can be assumed identical. If *register* is considered as the third axis then the register mapping stacks the submeshes $R_{i,j}$ over the simulating mesh in column-major order as shown in Figure 6.3.

6.5 Simulation by Contraction Mapping

Ben-Asher *et al.* [4] have presented an optimal self-simulation algorithm for the HV-RM model based on the contraction mapping. A similar approach has also been proposed independently by Maresca [58] in self-simulating the PPA model which is identical to the HV-RM model from a self-simulation point of view.

Now simulation of $\text{RM}_{M\times N}$ by $\text{RM}_{P\times Q}$ can be done by the following algorithm:

Algorithm 6.1 Optimal Self-Simulation of the HV-RM Model [4]

1 *Every simulating processor simulates the simulated submesh it contains;*

2 *Data for the bus segments, which are contained in two or more simulating processors, are transmitted to all the corresponding simulating processors;*

3 A single pass for each row and column of the simulated submesh in individual simulating processors completes the simulation;

Lemma 6.7 *In the HV-RM model,* $\mathrm{RM}_{M\times N}$ *can be optimally simulated by* $\mathrm{RM}_{P\times Q}$ *with slowdown* $6\frac{M}{P}\frac{N}{Q}+O\left(\frac{M}{P}+\frac{N}{Q}\right)$.[1]

Proof. A single processor can simulate a reconfigurable linear array of l processors in exactly $2l$ steps. In the first pass of l steps, processors are simulated from left-to-right direction and the data for each bus segment is stored in the rightmost simulated processor connected to that bus segment. In the second pass, processors are simulated from right-to-left direction and the data stored in the previous pass is dispersed to the rest of the processors.

A single processor simulating a $\frac{M}{P}\times\frac{N}{Q}$ submesh of HV-RM model can equivalently be considered as simulating independently $\frac{M}{P}$ reconfigurable row arrays of $\frac{N}{Q}$ processors each and $\frac{N}{Q}$ reconfigurable column arrays of $\frac{M}{P}$ processors each. Therefore, phase 1 of the simulation can be done in $4\frac{M}{P}\frac{N}{Q}$ steps. Now phase 2 can be achieved by $O(\frac{M}{P})$ steps for row buses and $O(\frac{N}{Q})$ steps for column buses. Finally phase 3 takes $2\frac{M}{P}\frac{N}{Q}$ steps. ■

A detailed description of Algorithm 6.1 and the proof of Lemma 6.7 can be found in [4].

A careful observation reveals that in phase 1 of Algorithm 6.1, the second pass of simulating linear arrays as described in the proof of Lemma 6.7 can be discarded if the data for each bus segment is stored in both the leftmost and the rightmost simulated processors connected to that bus segment.

Theorem 6.8 *In the HV-RM model,* $\mathrm{RM}_{M\times N}$ *can be optimally simulated by* $\mathrm{RM}_{P\times Q}$ *with slowdown* $4\frac{M}{P}\frac{N}{Q}+O\left(\frac{M}{P}+\frac{N}{Q}\right)$. ■

[1] *Ben-Asher* et al. *[4, pp. 5] have mistakenly computed the slowdown to be* $5\frac{M}{P}\frac{N}{Q}+O\left(\frac{M}{P}+\frac{N}{Q}\right)$.

It can easily be shown that the contraction mapping method fails to provide any strong self-simulation in stronger models e.g., the LRM and the FR models; let alone any optimal self-simulation [4, 31].

6.6 Simulation by Computing Connected Components

In this section we discus some self-simulation algorithms where the simulation is transformed into the problem of labelling connected components of a graph. An optimal self-simulation algorithm for the LRM model is discussed in Section 6.6.1. Section 6.6.2 presents a strong self-simulation algorithm for the FR model. Two weak simulations of the general reconfigurable mesh model are discussed in Section 6.6.3.

6.6.1 Self-Simulation of the LRM Model

Ben-Asher *et al.* [4] has developed an optimal self-simulation algorithm for the LRM model using folded-windows mapping of processors. Their algorithm is very complex to describe and, therefore, a very rough sketch is given below:

Algorithm 6.2 Optimal Self-Simulation of the LRM Model [4]

---

1 *(Component determination phase) Traverse the simulated mesh with the simulating mesh in snake-like order. The simulating mesh moves from one window* $R_{i,j}$ *to the next one, keeping track of all necessary bus information. At every window position, the following phases occur:*

  1.1 *Every bus segment that is encountered is given some unique* id*;*

  1.2 *When bus segments join in some previously visited windows, the combined segment is given a single* id*;*

2 *(Data delivery phase) At this point, all the separate buses and the broadcast information on each bus have been detected. A traversal in opposite order thus completes the simulation;*

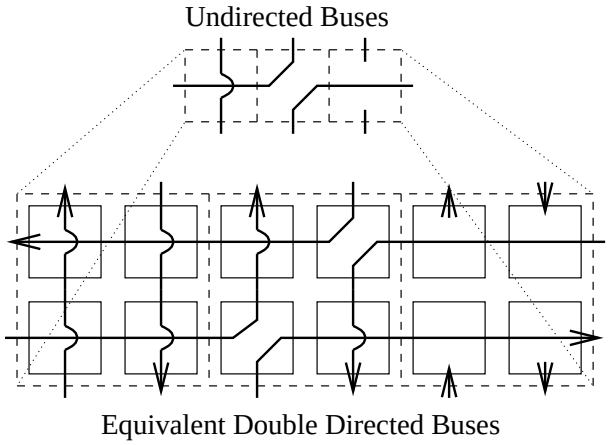

Figure 6.4: Representation of undirected buses by the double directed buses

Phase 1.1 of Algorithm 6.2 can be done through *leader election* by taking the maximum of the addresses of the two processors, where a bus segment terminates within a window, as the *id* of that bus segment. If each processor of the simulating mesh is replaced by a 2×2 square of processors and every bus is thus simulated by a double directed bus as shown in Figure 6.4, leader election can be done in $O(1)$ time [4].

Phase 1.2 of Algorithm 6.2 can also be solved in $O(1)$ time by a *linear connected component* algorithm [4] provided that a submesh of size $(P+Q) \times (P+Q)$ is provided for each window of size $P \times Q$.

Thus, to implement phases 1.1 and 1.2 for a window of size $P \times Q$, a larger mesh of size $2(P+Q) \times 2(P+Q)$ is required. But we are limited by the physical size $P \times Q$ of the simulating mesh. Now for the sake of simplicity, if we assume $P = Q = 4\delta$, where δ is an integer, then the effective window size becomes $\frac{P}{4} \times \frac{Q}{4}$. Considering two complete traversals in phases 1 and 2, the following can be concluded:

Theorem 6.9 *The slowdown of the optimal self-simulation Algorithm 6.2 for the LRM model is at least* $32\frac{M}{P}\frac{N}{Q}$. ■

6.6.2 Self-Simulation of the FR Model

Fernández-Zepeda *et al.* [31] have recently published a strong self-simulation algorithm for the FR model using windows mapping of processors. Their algorithm follows the same structure of Algorithm 6.2 where the component determination phase

uses a different technique of *prefix assimilation* while the data delivery phase is identical.

Theorem 6.10 *In the FR model,* $\mathrm{RM}_{M\times N}$ *can be strongly simulated by* $\mathrm{RM}_{P\times Q}$ *with slowdown* $O\left(\frac{M}{P}\frac{N}{Q}\log(P+Q)\right)$.

Proof. See [31]. ■

6.6.3 Self-Simulation of the General Model

Ben-Asher *et al.* [4] has presented a weak self-simulation of the general reconfigurable mesh using folded-windows mapping of processors. A rough sketch of this extremely complex algorithm is given below:

Algorithm 6.3 Optimal Self-Simulation of the General Model [4]

---

1 *(Component determination phase) For* $i = 0, 1, \ldots, \left(\log \frac{M}{P}\frac{N}{Q}\right)$ *do the following:*

  1.1 *Collect information on bus segments that are contained in windows of size* $2^iP \times 2^iQ$*;*

2 *(Data delivery phase) At this point, all the separate buses and the broadcast information on each bus have been detected. A traversal in opposite order thus completes the simulation;*

Labelling the connected components of a graph is among the problems for which constant time algorithms on reconfigurable mesh are not yet available. The best known algorithm [12, 66] has the running time $O(\log n)$ for an n node graph. As non-linear buses are allowed in the general model, joining connected bus segments during simulation requires logarithmic time instead of constant time.

Theorem 6.11 *The slowdown of the self-simulation Algorithm 6.3 for the general reconfigurable mesh is* $O\left(\frac{M}{P}\frac{N}{Q}\log(M+N)\log\left(\frac{M}{P}+\frac{N}{Q}\right)\right)$.

Proof. See [4]. ■

Fernández-Zepeda *et al.* [31] have improved the slowdown of self-simulation of the general model to $O\left(\frac{M}{P}\frac{N}{Q}\log(P+Q)\log\left(\frac{M}{P}+\frac{N}{Q}\right)\right)$ by adapting their strong self-simulation algorithm for the FR model as discussed in Section 6.6.2.

Recently, Matsumae and Tokura [62] have developed a new weak self-simulation algorithm for the general model with slowdown $O\left(\frac{M}{P}\frac{N}{Q}\log(P+Q)\log(M+N)\right)$ which is equivalent to the slowdown achieved by Fernández-Zepeda *et al.* in [31], if $P << M$ and $Q << N$, which is a desirable property in self-simulation.

Very recently, Fernández-Zepeda *et al.* [32] have further improved the slowdown of self-simulation of the general model to $O\left(\frac{M}{P}\frac{N}{Q}\log(M+N)\right)$.

Theorem 6.12 *In the general model,* $\mathrm{RM}_{M\times N}$ *can be simulated by* $\mathrm{RM}_{P\times Q}$ *with slowdown* $O\left(\frac{M}{P}\frac{N}{Q}\log(M+N)\right)$.

Proof. See [32]. ■

It is still an open problem whether self-simulation of the general reconfigurable mesh is possible with optimal slowdown unless the reconfiguring capability of the processors is severely restricted. It is now widely accepted that an additional polylogarithmic factor is inherent in the slowdown of self-simulating the unrestricted reconfigurable mesh [4].

6.7 Simulation by Simple Window Traversal

In Section 6.6.1 an optimal self-simulation algorithm for LRM model is presented by computing the connected components of graphs. Theorem 6.9 reveals that fairly large constant is associated with the optimal slowdown factor.

Is it possible to obtain optimal self-simulation algorithms by a sequence of simple windows traversals? In this section we answer to this question affirmatively for two restricted models within the LRM model. A generic self-simulation algorithm SIMPLE based only on windows traversal is presented in Section 6.7.1. In Section 6.7.2 we propose two restricted models within the LRM model and show that the algorithm SIMPLE is asymptotically optimal.

6.7.1 SIMPLE: a Self-Simulation Algorithm

We assume the common-write model where each processor on a bus can also detect whether one or more processors have transmitted or not. We also assume the folded-windows mapping of processors during the simulation.

Let B denote the set of all the boundary processors of the simulating mesh, i.e., $B = \{S(x,y) \mid x = 0 \vee x = P-1 \vee y = 0 \vee y = Q-1\}$. Let a port, t of a boundary processor, p be called **port* if t is not connected to any port external to p. Every boundary processor has exactly one *port except $S(0,0)$, $S(0,Q-1)$, $S(P-1,0)$, and $S(P-1,Q-1)$ which have two *ports each. Whenever the submesh $R_{i,j}$ is simulated, for each boundary processor, two registers from the ε extra registers are allocated for each *port. Let these special registers be called **reg1* and **reg2*.

For each step s of any reconfigurable-mesh-algorithm $\mathcal{A}$, let b(s), r(s), w(s), and c(s) denote the BUS, READ, WRITE, and COMPUTE substeps respectively. In the remainder whenever we mention that some steps or substeps are executed in the simulating mesh while simulating a specific submesh, it is assumed that the references to any register, to any port and to the coordinates of any processor are mapped accordingly as discussed in Section 6.4.3.

We now present a generic self-simulation algorithm without considering any specific model in mind. In Section 6.7.2 we show that this algorithm can optimally self-simulate some reconfigurable mesh models where restrictions are imposed over the global characteristics of bus reconfigurations.

Algorithm 6.4 SIMPLE(reconfigurable-mesh-algorithm: $\mathcal{A}$)

1 For each step s ∈ $\mathcal{A}$ do the following

1.1 For each boundary processor ∈ B do the following in parallel

*For each *port t do the following*

*For each mapped submesh $R_{i,j}$, $0 \le i < \frac{M}{P}$ and $0 \le j < \frac{N}{Q}$, set *reg1 to 0;*

1.2 Generate a finite sequence W_s *of pairs* $(i_1, j_1), (i_2, j_2), \ldots, (i_{L_s}, j_{L_s})$ *of length* L_s *where* $\forall k : 0 \leq i_k < P$ *and* $0 \leq j_k < Q$*;*

1.3 For each pair (i_k, j_k) *do the following on the mapped submesh* R_{i_k, j_k}

1.3.1 Execute b(s);

1.3.2a Execute w(s);

1.3.2b For each boundary processor $\in B$ *do the following in parallel*

*For each *port, t do the following*

*if *reg1 = 1 then write *reg2 to port t;*

1.3.3a Execute r(s);

1.3.3b For each boundary processor $\in B$ *do the following in parallel*

*For each *port, t do the following*

*if t senses signal then set *reg1 to 1 else set *reg1 to 0;*

*if *reg1 = 1 then read port t into *reg2;*

1.3.4a Execute c(s);

1.3.4b For each boundary processor $\in B$ *do the following in parallel*

*For each *port, t do the following*

*Copy *reg1 and *reg2 into the similar registers, allocated for t, of the neighbouring mapped submeshes;*

Phase 1.2 is a crucial part of the above algorithm. Generating the sequence W of length L which leads to correct self-simulation depends on many factors which are discussed in the next section.

Phase 1.1 of algorithm SIMPLE can be done in $\frac{M}{P}\frac{N}{Q}$ steps. Let the cost of generating the sequence W_s of L_s pairs in simulating step s in phase 1.2 is G_s. Phases 1.3.2a and 1.3.2b can be done in a single WRITE substep. Similarly phases 1.3.3a and 1.3.3b can be done in a single READ substep while phases 1.3.4a and 1.3.4b can be done in a single COMPUTE substep. So all the substeps of phase 1.3 can be executed in order $O(1)$. Hence the order of phase 1.3, in simulating step s, is $O(L_s)$.

Lemma 6.13 *Slowdown of algorithm SIMPLE is* $\max\left(\frac{M}{P}\frac{N}{Q}, \max_{\forall s}(G_s, O(L_s))\right)$. ■

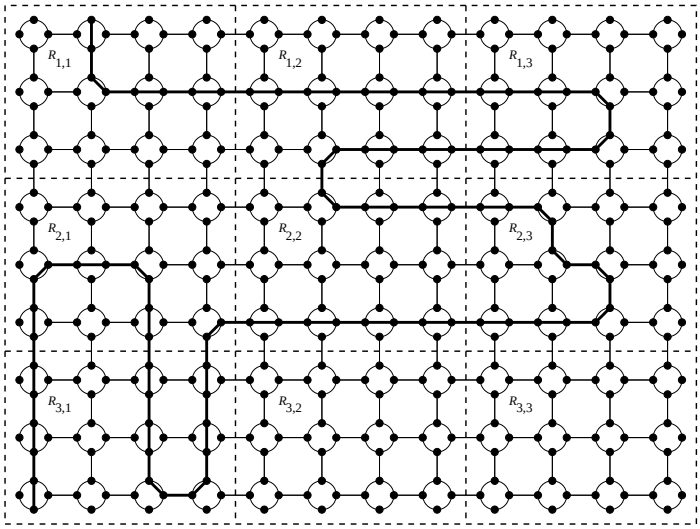

Figure 6.5: Trajectory of the only linear bus is (1,1), (1,2), (1,3), (1,2), (2,2), (2,3), (2,2), (2,1), (3,1), (2,1), (3,1).

Definition 6.5 *Let the* trajectory *of a linear bus in the simulated mesh be the sequence of pairs* (i_1, j_1), (i_2, j_2), $\ldots$, (i_t, j_t) *such that the bus starts and finishes in the windows* R_{i_1,j_1} *and* R_{i_t,j_t} *respectively and for* $k = 1, 2, \ldots, t-1$, *a portion of the bus contained in the window* R_{i_k,j_k} *is directly connected to the portion of bus contained in the window* $R_{i_{k+1},j_{k+1}}$. *Note that a window may contain one or more portions of the same linear bus as shown in Figure 6.5.*

Definition 6.6 *Let* $\overline{S}$ *denote the sequence* S *in reverse order,* $S_1 + S_2 + \cdots + S_n$ *denote the concatenation of sequences* $S_1, S_2, \ldots, S_n$ *in order and* S^k *denote the sequence* $\underbrace{S + S + \cdots + S}_{k \text{ times}}$.

Theorem 6.14 *Let* T *be the trajectory of a linear bus* V *in the simulated mesh during simulating step* s. *If the sequence* W_s *of pairs in algorithm SIMPLE contains the sequence* $T + \overline{T}$, *preserving the order but not necessarily in consecutive positions, then algorithm SIMPLE can self-simulate the bus* V *in step* s.

Proof. We use the following lemma which follows from substeps 1.3.2b and 1.3.3b of algorithm SIMPLE:

Lemma 6.15 *Algorithm SIMPLE ensures that whenever a portion of the bus V in some window $R_{i,j}$, where $(i,j) \in T$, is simulated, any message that has been transmitted through this portion of the bus is carried to both the neighbouring windows $R_{a,b}$ and $R_{c,d}$ w.r.t. the trajectory of the bus V, i.e., the sequence of pairs (a,b), (i,j), (c,d) is contained in T in consecutive positions.* ■

Now consider the pair $(p,q) \in T$ such that one or more processors of window $R_{p,q}$ on the bus V write a message on a portion B_u of bus V. As T is contained in W_s, by using Lemma 6.15 iteratively, it can be shown that the entire segment of bus B on the right of the bus portion V_u is simulated when phase 1.3 of algorithm SIMPLE completes the portion of sequence W_s which contains T. Similarly, it can also be shown that the rest of the bus is simulated when phase 1.3 of algorithm SIMPLE completes the rest of sequence W_s which contains $\overline{T}$. This completes the proof of Theorem 6.14. ■

6.7.2 Optimal Self-Simulation of Some Restricted Models

All of the models of reconfigurable mesh, discussed in Section 2.2.3, depend solely on local configurations of ports. Here we present two additional models where restrictions are imposed on the global characteristics of the buses.

Definition 6.7 *A function $f(x)$ is called* positive monotonic *w.r.t. x if $f(x_1) \geq f(x_2)$ whenever $x_1 \geq x_2$. Similarly a function $f(x)$ is called* negative monotonic *w.r.t. x if $f(x_1) \leq f(x_2)$ whenever $x_1 \geq x_2$. A function $f(x)$ is* monotonic *w.r.t. x if it is either positive or negative monotonic w.r.t. x.*

Definition 6.8 *A function $f(x)$ is called* piecewise-monotonic *w.r.t. x if axis-x can be divided into successive ranges such that $f(x)$ is positive monotonic in alternate ranges and negative monotonic in the other ranges.*

The Monotonic-Bus (MB) Model: Each bus represents a monotonic function w.r.t. either row and/or column index within a range. For example, all the buses in Figure 6.6 are monotonic while the only bus in Figure 6.5 is not monotonic. Note that Ben-Asher *et al.* [6] has defined a *non-monotonic* model where processors can

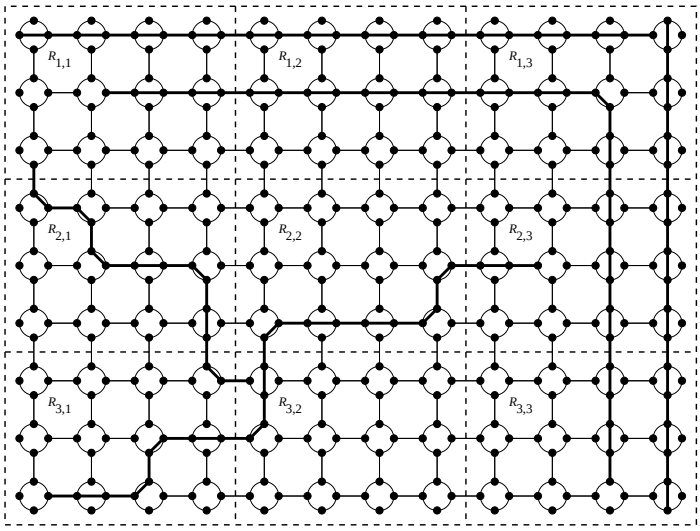

Figure 6.6: Monotonic buses.

invert signals at the ports. This non-monotonic model should not be confused with the MB model presented here.

The Piecewise-Monotonic-Bus (PMB) Model: Every bus represents a piecewise monotonic function w.r.t. either row or column index within a range. Moreover in any step all buses represent functions w.r.t. same index. For example, all the buses in Figure 6.7 are piecewise monotonic.

Observe that both the models are included in the LRM model. Also observe that the HV-RM model is included in the MB model which is again included in the PMB model.

We believe that the MB and PMB models are defined here for the first time (except for the author's paper [73]) but many published algorithms for the LRM model can readily be used in these models without any modifications or with very small modifications. Among them PARITY algorithms [56, 86], conversion between number representations algorithms [40], prefix-sums algorithm [79], sorting algorithms [83, 85] etc. can be adapted into the MB as well as the PMB models and prefix-remainders algorithm [79], integer summing algorithms [40, 79], integer multiplication algorithm [39], sorting algorithm [40] etc. can be applied into the PMB model only. Moreover it is quite obvious that all the algorithms suitable for the HV-RM model are applicable to

both the models.

Let $CSEQ(j)$ denote the sequence of pairs $(0,j), (1,j), \ldots, (\frac{M}{P}-1, j)$.

Theorem 6.16 *Algorithm SIMPLE can self-simulate the MB model optimally.*

Proof. Let us consider the sequences of pairs, $S_+ = CSEQ(0) + CSEQ(1) + \cdots + CSEQ(\frac{N}{Q}-1)$ and $S_- = \overline{CSEQ(0)} + \overline{CSEQ(1)} + \cdots + \overline{CSEQ(\frac{N}{Q}-1)}$. Let $S = S_+ + S_-$. Let $\mathcal{A}$ be an arbitrary algorithm on the simulated MB reconfigurable mesh. We now show that the sequence $S + \overline{S} = S_+ + S_- + \overline{S_-} + \overline{S_+}$ of length $4\frac{M}{P}\frac{N}{Q}$ can be used in the phase 1.2 of algorithm SIMPLE for each step of $\mathcal{A}$ to achieve a correct self-simulation.

First consider only the positive monotonic buses. Let any arbitrary positive monotonic bus u terminate in the submeshes $R_{a,b}$ and $R_{c,d}$. Now assume $a \leq c$ and this implies $b \leq d$ as u is positive monotonic. Based on the characteristics of positive monotonic bus we can say that the trajectory of the bus through various submeshes $R_{i,j}$ follows the sequence of pairs $S_u = (a,b), (a+1,b), \ldots, (k_b,b), (k_b,b+1), (k_b+1,b+1), \ldots, (k_{b+1},b+1), \ldots, (k_{d-1},d), (k_{d-1}+1,d), \ldots, (c,d)$ where $0 \leq k_b \leq c$ and $k_{l-1} \leq k_l \leq c$, $b < l < d$.

It is very easy to show that S_u and $\overline{S_u}$ are contained in S_+ and $\overline{S_+}$ respectively preserving the order but not necessarily as contiguous pairs. Then by Theorem 6.14, every monotonic bus is simulated by algorithm SIMPLE.

The sequences of pairs S_- and $\overline{S_-}$ play similar role in simulating negative monotonic buses correctly.

Now the generation of the sequence of pairs $S + \overline{S}$ is independent of any step of the arbitrary algorithm $\mathcal{A}$. Hence, for each step s of $\mathcal{A}$, G_s can be considered as $O(1)$ and thus by Lemma 6.13 the slowdown of Algorithm SIMPLE in self-simulating MB model is $4\frac{M}{P}\frac{N}{Q} + O\left(\frac{M}{P} + \frac{N}{Q}\right)$ which is optimal. ■

Note that the prefix-sum computation in Section 2.2.4.1 configures only linear buses and therefore, can be executed on the LRM model without any modification. As claimed in the beginning of this section, this computation can also be carried out, without any modification, on the MB model as each of the linear buses configured in the computation can be viewed as a representation of a monotonic function w.r.t.

either row and/or column index. Even if the computation is performed in the general "unrestricted" model, efficiency in self-simulation of this computation depends on the virtual restrictions imposed while selecting suitable self-simulation algorithms. If we consider the above computation is done on the LRM model then the constant associated with the highest order term in the slowdown of the optimal self-simulation Algorithm 6.2 is 32 while the same of our algorithm SIMPLE is only 4.

As a general solution to handle all linear bus configurations, including the spirals, Algorithm 6.2 becomes inefficient in handling some specific linear bus configurations e.g., the configurations covered by the MB and the PMB models.

Theorem 6.17 *Algorithm SIMPLE can self-simulate the PMB model correctly.*

Proof. Let $\mathcal{A}$ be an arbitrary algorithm on the simulated PMB reconfigurable mesh. Without any loss of generality we assume that the buses of any particular step of $\mathcal{A}$ be piecewise-monotonic w.r.t. column index.

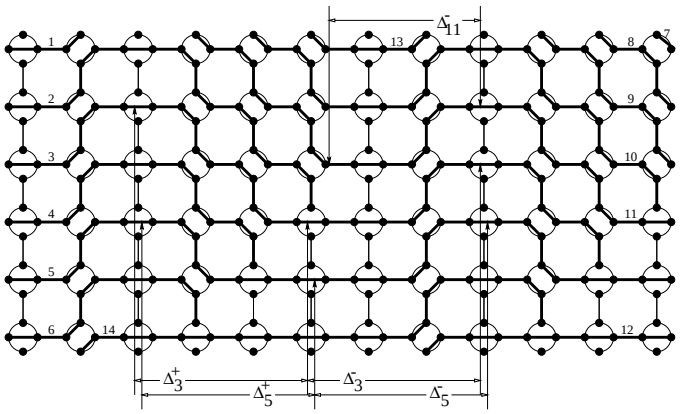

Figure 6.7: Configuration of buses in a step of an algorithm [40] for adding two integers of 3 bits each on a reconfigurable mesh of size 6×12. Among the 14 buses, only Δ_3^+, Δ_1^-, Δ_5^+, Δ_5^-, and Δ_{11}^- exist. Obviously $\Delta = 3$.

Let Δ_u^+ denote the minimum of the minimum processor-distance along the row axis of any two successive positive monotonic segments of bus u. Similarly let Δ_u^- denote the minimum of the minimum processor-distance along the row axis of any two successive negative monotonic segments of bus u. As a horizontal bus segment can be considered as both positive and negative monotonic, for the same bus u, Δ_u^+

and Δ_u^- can be varied depending on the point where a horizontal segment is assumed divided into positive and negative monotonic segments. In such case, the division point is considered in the position where $\Delta_u^+ = \Delta_u^-$. Let $\Delta = \min_{\forall u}(\min(\Delta_u^+, \Delta_u^-))$ and $K = \left\lceil \frac{Q}{2\Delta} \right\rceil$.

Consider the sequence of pairs $S = \sum_{j=0}^{\frac{N}{Q}} \left(CSEQ(j) + \overline{CSEQ(j)} \right)^K$. We now show that the sequence $S + \overline{S}$ of length $4K\frac{M}{P}\frac{N}{Q}$ can be used in the step 1.2 of Algorithm SIMPLE for each step of $\mathcal{A}$ to achieve a correct self-simulation.

Let any arbitrary piecewise-monotonic bus u terminates in the submeshes $R_{a,b}$ and $R_{c,d}$. Now assume $a \leq c$. Let the trajectory of the bus through various submeshes $R_{i,j}$ follow the sequence of pairs S_u. As this trajectory S_u passes through any submesh $R_{i,j}$ at most K times as positive monotonic segment and at most K times as negative monotonic segment, it is easy to show that S_u and $\overline{S_u}$ are contained in S and $\overline{S}$ respectively preserving the order but not necessarily consecutively.

By Theorem 6.14, every bus is thus simulated by algorithm SIMPLE. ■

In Theorem 6.17 nothing is stated about the slowdown of the self-simulation. In general cases the slowdown will not be optimal. However we can achieve optimal slowdown for instances where $\frac{Q}{\Delta} = O(1)$ which is possible if $\frac{N}{Q}$ is very large, a desirable property for self-simulation and Δ is a function of N.

6.8 AT^2 Optimality Issues in Self-Simulation

Is the resultant algorithm in self-simulation of a reconfigurable mesh with optimal slowdown AT^2 optimal?

To our knowledge, this question is raised here for the first time (except for author's paper [72]) and we have a negative answer. Consider any problem of size N with $AT^2 = \Omega(N^2)$ e.g., sorting of N items of size $\log N$ bits each. Now, in Sections 4.4.1–4.4.3 we have discussed a number of sorting algorithms which can sort N items on a reconfigurable mesh of size $N \times N$ in constant time (Theorem 4.8). Obviously the AT^2 measures of these algorithms are $\Theta(N^2)$ and thus these algorithms are AT^2 optimal.

Suppose one of this AT^2 optimal sorting algorithm is self-simulated, with optimal

Mesh Type	**Method**	**Size**	**Time**	AT^2	AT^2 **Optimal?**
reconfigurable	direct	$N \times N$	$O(1)$	$O(N^2)$	yes
reconfigurable	self-simulation	$M \times M$	$O\left(\frac{N^2}{M^2}\right)$	$O\left(\frac{N^4}{M^2}\right)$	NO
reconfigurable	self-simulation	$N^{1/2} \times N^{1/2}$	$O(N)$	$O(N^3)$	
ordinary	direct	$N^{1/2} \times N^{1/2}$	$O(N^{1/2})$	$O(N^2)$	yes

Table 6.1: Sorting of N items on various types of meshes—directly or by self-simulation.

slowdown $\Theta\left(\frac{N^2}{M^2}\right)$, on a reconfigurable mesh of size $M \times M$ where $M < N$. The AT^2 measure of the resultant sorting algorithm then becomes $\Theta\left(\frac{N^4}{M^2}\right)$ which is not optimal for $M = o(N)$.

Consider $M = N^{1/2}$. Then we cannot achieve an optimal sorting algorithm to sort N items on a reconfigurable mesh of size $N^{1/2} \times N^{1/2}$ by self-simulating one of the AT^2 optimal constant time sorting algorithms on a reconfigurable mesh of size $N \times N$. On the other hand, in Section 4.3 we have presented a large number of AT^2 optimal sorting algorithms to sort N elements on an ordinary mesh of size $N^{1/2} \times N^{1/2}$ in $O(N^{1/2})$ time.

Self-simulation makes the powerful reconfigurable mesh weaker than even the ordinary mesh. The strength of configurable computing is seriously weakened by the introduction of self-simulation as summarised in Table 6.1.

Theorem 6.18 *The resultant algorithms are not necessarily AT^2 optimal when AT^2 optimal algorithms are self-simulated on reconfigurable meshes even with optimal slowdown.* ■

Does Theorem 6.18 imply that scaling down of algorithms on large reconfigurable meshes is inefficient?

Fortunately we also have a negative answer to the above question. Self-simulation is not the only way to scale down algorithms on reconfigurable meshes. In fact, intent of developing self-simulation algorithms is precisely to preserve AT product, and certainly not AT^2, by design. Self-simulation algorithms preserve efficiency, which is all they can do for arbitrary algorithms because the simulating machine must perform the same work as the simulated machine. Many reconfigurable mesh algorithms sac-

rifice efficiency to achieve constant time. Self-simulation scaling of these algorithms naturally retains the burden of their inefficiency. In the next chapter we introduce the idea of developing smart self-scalable algorithms which not only run on reconfigurable meshes of various sizes and aspect ratios but also retain AT^2 optimality.

6.9 Conclusions

In this chapter we have proposed two new reconfigurable mesh models MB and PMB, within the LRM model, where restrictions are imposed over the global characteristics of bus reconfigurations. A simple generic self-simulation algorithm has been developed which remains optimal and asymptotically optimal while self-simulating the MB and the PMB models respectively. It can easily be shown that for self-simulating a large number of algorithms on the MB and the PMB models, our self-simulation algorithm SIMPLE requires fewer steps than the optimal self-simulation Algorithm 6.2 for the LRM model by Ben-Asher *et al.* [4].

In future we are interested in applying our algorithm SIMPLE on some special instances of the *tree-RM* mesh model [4]. The tree-RM mesh model is similar to the general model, except that no cycles are allowed in the configuration. Note that this is a global restriction and hence tree-RM mesh model seems to have similarity with the MB and the PMB models for which algorithm SIMPLE has been a success.

What is the relationship between the term *optimality* in self-simulation and area-time tradeoff of reconfigurable meshes? In Section 6.8 we have shown that self-simulation, even with optimal slowdown, compromises the AT^2 optimality of the resultant algorithms and in the course of finding a remedy, in the next chapter we introduce the idea of adaptive algorithms which remain AT^2 optimal regardless of the size and aspect ratio of the reconfigurable mesh.

Chapter 7

Adaptive Algorithms

Self-simulation is the obvious but not necessarily the only way to solve the problem of scaling down algorithms on the reconfigurable mesh as defined in Chapter 1. An alternative solution lies in developing self-scalable algorithm which can adapt to reconfigurable meshes of various sizes and aspect ratios. In Section 6.8 we have successfully argued that self-simulation methods compromise with AT^2 optimality of the resultant algorithms. Writing self-scalable algorithms is itself a challenging task but it would be far more appealing if we can develop self-scalable algorithms which not only adapts to a reconfigurable mesh of arbitrary size and aspect ratio but also retains AT^2 optimality during this adaptation process. Let these smart self-scalable algorithms be called *adaptive* algorithms.

The aim of this chapter is to develop a systematic approach to design efficient *adaptive* algorithms. In Section 7.1 the characteristics of adaptive algorithms are formally defined and a generic adaptive algorithm to solve an arbitrary problem is presented. An adaptive sorting algorithm is developed in Section 7.2 based on rotatesort Algorithm 4.4. In Section 7.3 we present another adaptive sorting algorithm based on the sorting Algorithm 4.1 of Schnorr and Shamir on ordinary meshes. An adaptive $\mathcal{M}$-contour algorithm is developed in Section 7.4. In Section 7.5 we propose a conjecture stating that it is sufficient to configure buses of length $O(k)$ in an arbitrary adaptive algorithm where k represents how much of the mesh is filled with data initially, and the conjecture is then supported by transforming our adaptive algorithms on k-constrained reconfigurable meshes.

7.1 Design of Adaptive Algorithms

Let a problem $\mathcal{P}$ of size n have $I(n)$ *information content* [113, pp. 51–54]. If this problem $\mathcal{P}$ is realized in a VLSI circuit with aspect ratio $\alpha \geq 1$ then, by Ullman [113, pp. 57], the AT^2 lower bound of $\mathcal{P}$ will be

$$\Omega(\alpha I^2(n)) \ . \tag{7.1}$$

Now, consider a reconfigurable mesh of size $p \times q$ where $pq = kI(n)$, $1 < p \leq q \leq I^2(n)$, and $k \geq 1$.

Throughout this chapter we assume that initially each item of $I(n)$ information content is contained in a distinct processor. In such case $k = \frac{pq}{I(n)}$ has a physical interpretation too. It represents how much of the mesh is filled with data.

Let $\mathcal{P}$ be solved, AT^2 optimally, on a reconfigurable mesh of size $p \times q$ in $O(T)$ time. Then

$$pqT^2 = \frac{q}{p} I^2(n) \ .$$

This implies

$$T = \frac{I(n)}{p} = \frac{q}{k} \ . \tag{7.2}$$

Observe that T is independent of q, the length of the larger side of the VLSI circuit. Now,

$$T = 1 \Leftrightarrow p = I(n) \ .$$

Thus, development of a constant time algorithm is feasible whenever $p = I(n)$ for any $q \geq p$. As we are interested in keeping the area at minimum, the minimum possible value of q should be considered. So, with the minimum area constraint,

$$T = 1 \Leftrightarrow p = q = k = I(n) \ . \tag{7.3}$$

Lemma 7.1 *To solve a problem $\mathcal{P}$ of size n with $I(n)$ information content AT^2 optimally in constant time, a reconfigurable mesh of size at least $I(n) \times I(n)$ is required.* ■

From equation (7.2),

$$k = 1 \Leftrightarrow T = q \ .$$

This implies that whenever the area of the VLSI circuit equals the information content

of the problem to be solved, the time of solution depends only on q, the length of the larger side of the VLSI circuit. As we are interested in keeping the time at minimum, the minimum possible value of q should be considered. So, $pq = I(n)$ and $p \leq q$ imply the following with the minimum time constraint,

$$k = 1 \Leftrightarrow p = q = T = \sqrt{I(n)}\,. \tag{7.4}$$

Lemma 7.2 *To solve a problem $\mathcal{P}$ of size n with $I(n)$ information content AT^2 optimally on a reconfigurable mesh of size $\sqrt{I(n)} \times \sqrt{I(n)}$ requires $\Omega\left(\sqrt{I(n)}\right)$ time.* ■

Lemma 7.2 has extra significance. The communication diameter of an ordinary mesh of size $\sqrt{I(n)} \times \sqrt{I(n)}$ is $\Theta\left(\sqrt{I(n)}\right)$. Thus, reconfigurability is of no assistance in solving problem $\mathcal{P}$ AT^2-optimally on a reconfigurable mesh of size $\sqrt{I(n)} \times \sqrt{I(n)}$.

Definition 7.1 *Consider an arbitrary problem $\mathcal{P}$ of size n with information content $I(n)$. An algorithm $\mathcal{A}$ is called* adaptive *if $\mathcal{A}$ takes $O\left(\frac{I(n)}{p}\right) \equiv O\left(\frac{q}{k}\right)$ time to solve $\mathcal{P}$ on a reconfigurable mesh of size $p \times q$, where $\sqrt{I(n)} \leq p \leq q \leq I(n)$ and $k = \frac{pq}{I(n)}$.*

Now, from equation (7.2) we find that $\frac{p}{k} = \frac{I(n)}{q}$. From Definition 7.1 we also find that $q \leq I(n)$. We thus can conclude that

$$k \leq p \tag{7.5}$$

which is an important relation assumed in Algorithm 7.1 below.

The study of adaptive algorithms on reconfigurable meshes reveals that the discussion of optimality issues in mesh-connected networks should not be limited to any specific class or model. It is obvious that existing AT^2 optimal algorithms on reconfigurable mesh will play significant role in the development of future adaptive algorithms but we must consider optimal algorithms on linear arrays and ordinary meshes as well.

Adaptive algorithms can be developed from scratch. But we are interested in designing adaptive algorithms mainly by threading optimal algorithms on linear arrays, ordinary meshes, and reconfigurable meshes. We use the following algorithmic structure in developing adaptive algorithms for specific problems:

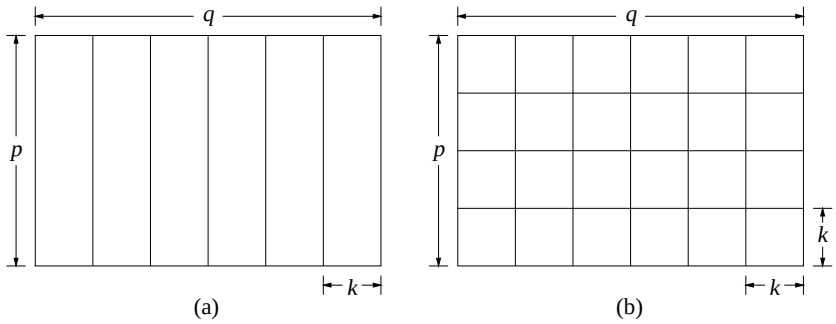

Figure 7.1: Physical division of a reconfigurable mesh in the principal module (a) and the supporting module (b).

Algorithm 7.1 Generic Adaptive Algorithm

Principal Module: *Here $\mathcal{P}$ of size n is solved on a reconfigurable mesh of size $p \times q$ where $p \leq q$ and $pq = kI(n)$. It is assumed that $I(n)$ items of information are contained in the first $\frac{I(n)}{p}$ columns where each processor $PE_{i,j}$, $0 \leq i < p$ and $0 \leq j < \frac{I(n)}{p}$, contains exactly one item of information.*

1p Divide the mesh of size $p \times q$ into $\frac{q}{k}$ submeshes of size $p \times k$ each;

2p Distribute the $I(n)$ information content equally among the submeshes in such a way that each processor $PE_{i,jk}$ of the main mesh, $0 \leq i < p$ and $0 \leq j < \frac{q}{k}$, receives one item of information. This ensures that each submesh of size $p \times k$ now contains exactly p items of information in its first column;

3p Solve the subproblem with p items of information in each submesh of size $p \times k$ using the algorithm of the supporting layer in parallel;

4p Merge the solutions of the $\frac{q}{k}$ subproblems using the entire mesh of size $p \times q$;

Supporting Module: *Here $\mathcal{P}$ of size at most p is solved on a reconfigurable mesh of size $p \times k$ where $k \leq p$. It is assumed that p items of information are contained in the first column where each processor contains exactly one item of information.*

1s Divide the mesh of size $p \times k$ into $\frac{p}{k}$ submeshes of size $k \times k$ each;

2s Distribute the p items of information equally among the submeshes in such a way that each submesh of size $k \times k$ contains exactly k items of information

in its top row;

3s Solve the subproblem with k items of information in each submesh of size $k \times k$ in parallel;

4s Merge the solutions of the $\frac{p}{k}$ subproblems using the entire mesh of size $p \times k$;

The above generic Algorithm 7.1 assumes $\frac{p}{k}$ and $\frac{q}{k}$ to be integers for the sake of simplicity.

As mentioned in Definition 7.1, the main goal of Algorithm 7.1 is to achieve $O\left(\frac{q}{k}\right)$ run time. Phases 1p and 1s of Algorithm 7.1 need no inter-processor communication and thus these can be done in constant time.

As $\frac{I(n)}{p} = \frac{q}{k}$ by equation (7.2), phase 2p can be considered as a problem of transferring column j to column kj for all $j : 0 \leq j < \frac{q}{k}$. Now each of the column transfer can be done in constant time by p row broadcasts in parallel. So, phase 2p can be done in $O\left(\frac{q}{k}\right)$ time.

Phase 1s not only divides the mesh of size $p \times k$ into $\frac{p}{k}$ submeshes of size $k \times k$ but also distributes p items of information equally among these submeshes such that every submesh contains k items of information in its first column. Hence, phase 2s can be done in two steps. In the first step, the item of information in processor $PE_{i,0}$ of each submesh of size $k \times k$ is transferred to processor $PE_{i,i}$ in parallel by row broadcasts for all $i : 0 \leq i < k$. In the second step, the item of information in each processor $PE_{j,j}$ is transferred to processor $PE_{0,j}$ in parallel by column broadcasts for all $j : 0 \leq j < k$. So, it can be concluded that phase 2s takes $O(1)$ time.

If we can fairly assume by equation (7.3) that a constant time algorithm exists to solve phase 3s then the only challenge that remains in designing an adaptive algorithm for any specific problem is to solve phases 4p and 4s in $O\left(\frac{q}{k}\right)$ time as phase 3s degenerates to phases 1s–4s.

The following interesting observations can be made from Algorithm 7.1:

- When $p = q = k = I(n)$, the only nontrivial phase is phase 3s, which degenerates into solving $\mathcal{P}$ AT^2-optimally in constant time on a reconfigurable mesh.

- When $p = q = \sqrt{I(n)}$ and $k = 1$, phases 2p, 1s, 2s, and 3s disappear and phases 3p and 4p degenerates into solving $\mathcal{P}$ AT^2-optimally on an ordinary mesh while phase 4s degenerates into solving $\mathcal{P}$ AT^2-optimally on a linear array of processors.

Phases 3p and 4p and phases 3s and 4s should not necessarily be simple implementations of the many-way divide-and-conquer strategy as mentioned in Algorithm 7.1. In many cases, it is more efficient to replace these phases by complex iteration if the motivation is to solve a large problem instance with the solution of smaller problem instances (Sections 7.2 and 7.3).

To our knowledge, the idea of developing adaptive algorithms, as opposed to inefficient method of scaling down AT^2 optimal algorithms by optimal self-simulation, is presented here for the first time (except for the author's paper [72]). In fact very little work has so far been done in the direction of adapting algorithms on meshes whose size and aspect ratio can vary according to the availability of a physical implementation. Jang and Prasanna [40] have developed a sorting algorithm which can sort N numbers in $O(T)$ time on a reconfigurable mesh of size $\frac{N}{T} \times \frac{N}{T}$, for $1 \leq T \leq N^{1/2}$. Preserving AT^2 values while scaling is also demonstrated by Park *et al.* [93] in multiplying matrices and Trahan *et al.* [110] in multiplying matrices with vectors.

7.2 An Adaptive Sorting Algorithm Based on *Rotatesort*

In this section we develop an adaptive sorting algorithm, using the framework of generic adaptive Algorithm 7.1, to sort n items AT^2 optimally on a reconfigurable mesh of size $p \times q$, $p \leq q$ and $pq = kn$, by connecting the AT^2 optimal rotatesort algorithm of Marberg and Gafni (Section 4.3.4) on ordinary meshes to any constant time sorting algorithm discussed in Sections 4.4.1–4.4.3 on reconfigurable meshes. The algorithm presented in this section was developed in [10, 11] for k-constrained reconfigurable meshes (Section 2.2.3.10) with a different objective.

The links of a reconfigurable linear array can be considered as unidirectional. In this section we use the following corollary of Theorem 4.2 on reconfigurable linear arrays:

Corollary 7.3 *Given s items in some s processors of a reconfigurable linear array of $m \geq s$ processors, these items can be sorted in $O(s)$ time.*

Proof. By connecting port **N** with **S** of all the processors which do not contain any of the given s items, the reconfigurable linear array of m processors can simulate an ordinary linear array of s processors. ■

As mentioned in Section 4.3.4, Algorithm 4.4 of Marberg and Gafni uses a constant number of linear transformations (sorting and cyclic rotation), made alternately to rows and columns, to sort MN items in $O(M+N)$ time on an ordinary mesh of size $M \times N$ where $M \geq N^{1/2}$. If $M \not\geq N^{1/2}$ then $N > M^2 \geq M^{1/2}$ and thus sorting can be done simply by transposing all row(column) operations into column(row) operations in Algorithm 4.4.

In this section we further assume $k = r^2$, $p = ks^2$, and $q = kt^2$ where r, s, and t are integers and $s \leq t$ for the sake of simplicity.

The principal module and the supporting module of this adaptive algorithm are presented in Sections 7.2.2 and 7.2.1 respectively.

7.2.1 The Supporting Module

In this section we develop an adaptive algorithm to sort p items on a reconfigurable mesh of size $p \times k$, $k \leq p$, following the supporting module of Algorithm 7.1. We assume that phases 1s and 2s of Algorithm 7.1 are already completed, i.e., the mesh is divided into $\frac{p}{k}$ submeshes of size $k \times k$ each and the given p items in the first column are distributed in such a way that each processor $PE_{ik,j}$, $0 \leq i < \frac{p}{k}$ and $0 \leq j < k$, receives an item. This distribution of items transformed the column of p items into an array of $\frac{p}{k} \times k$ items where each row is separated by k rows. Now, phases 3s and 4s of Algorithm 7.1 emulate Algorithm 4.4 of Marberg and Gafni. This emulation requires only the following basic operations in various phases of Algorithm 4.4:

If $k \geq \left(\frac{p}{k}\right)^{1/2}$

1. Sort k items in a row using a submesh of size $k \times k$.
2. Rotate $\left(\frac{p}{k}\right)^{1/2}$ items using a submesh of size $k\left(\frac{p}{k}\right)^{1/2} \times 1$.

3. Sorting $\frac{p}{k}$ items using a submesh of size $p \times 1$.
4. Rotate $\frac{p}{k}$ items using a submesh of size $p \times 1$.
5. Sort $\left(\frac{p}{k}\right)^{1/2}$ items in a row using a submesh of size $k \times \left(\frac{p}{k}\right)^{1/2}$.

Else $(\Rightarrow \frac{p}{k} > k^{1/2})$

6. Sort $\frac{p}{k}$ items using a submesh of size $p \times 1$.
7. Rotate $k^{1/2}$ items in a row using a submesh of size $k \times k^{1/2}$.
8. Sort k items in a row using a submesh of size $k \times k$.
9. Rotate k items in a row using a submesh of size $k \times k$.
10. Sort $k^{1/2}$ items using a submesh of size $kk^{1/2} \times 1$.

The problem of rotation can always be transformed into a sorting problem without any slowdown. A rotation, therefore, takes no more time than it does to sort.

Now, operations 1, 5, 7, 8, and 9 can be done in $O(1)$ time by Theorem 4.8. Using Corollary 7.3 it can be shown that operation 2 can be done in $O\left(\left(\frac{p}{k}\right)^{1/2}\right)$ time, operations 3, 4, and 6 can be done in $O\left(\frac{p}{k}\right)$ time, and operation 10 can be done in $O\left(k^{1/2}\right)$ time.

Theorem 7.4 *Given p items in the first column of a reconfigurable mesh of size $p \times k$, $k \leq p$, these items can be sorted in $O\left(\frac{p}{k}\right)$ time, which is AT^2 optimal.* ■

7.2.2 The Principal Module

In this section we develop an adaptive algorithm to sort n items on a reconfigurable mesh of size $p \times q$, $p \leq q$ and $pq = kn$, following the principal module of Algorithm 7.1. We assume that phases 1p and 2p of Algorithm 7.1 are already completed, i.e., the mesh is divided into $\frac{q}{k}$ submeshes of size $p \times k$ each and the given n items in the first $\frac{n}{p}$ columns are distributed in such a way that each processor $PE_{i,jk}$, $0 \leq i < p$ and $0 \leq j < \frac{q}{k}$, receives an item. This distribution of items transformed the given n items into an array of $p \times \frac{q}{k}$ items where each column is separated by k columns. Now, phases 3p and 4p of Algorithm 7.1 emulate Algorithm 4.4 of Marberg and Gafni as

done in Section 7.2.1. This emulation requires only the following basic operations in various phases of Algorithm 4.4:

If $p \geq \left(\frac{q}{k}\right)^{1/2}$

1. Sort p items in a column using a submesh of size $p \times k$.
2. Rotate $\left(\frac{q}{k}\right)^{1/2}$ items using a submesh of size $1 \times k\left(\frac{q}{k}\right)^{1/2}$.
3. Sort $\frac{q}{k}$ items using a submesh of size $1 \times q$.
4. Rotate $\frac{q}{k}$ items using a submesh of size $1 \times q$.
5. Sort $\left(\frac{q}{k}\right)^{1/2}$ items in a column using a submesh of size $\left(\frac{q}{k}\right)^{1/2} \times k$.

Else $\Rightarrow \frac{q}{k} > p^{1/2}$

6. Sort $\frac{q}{k}$ items using a submesh of size $1 \times q$.
7. Rotate $p^{1/2}$ items in a column using a submesh of size $p^{1/2} \times k$.
8. Sort p items in a column using a submesh of size $p \times k$.
9. Rotate p items in a column using a submesh of size $p \times k$.
10. Sort $p^{1/2}$ items using a submesh of size $1 \times kp^{1/2}$.

From equation (7.5) we get $k \leq p$. Hence, operations 1, 8, and 9 can be done in $O\left(\frac{p}{k}\right)$ time by Theorem 7.4.

If $k \geq \left(\frac{q}{k}\right)^{1/2}$ then operation 5 takes $O(1)$ time by Theorem 4.8 else it takes $O\left(q^{1/2}/k^{3/2}\right)$ time by Theorem 7.4. Similarly, if $k \geq p^{1/2}$ then operation 7 takes $O(1)$ time else it takes $O\left(p^{1/2}/k\right)$ time.

Using Corollary 7.3, operation 2 can be done in $O\left(\left(\frac{q}{k}\right)^{1/2}\right)$ time, operations 3, 4, and 6 can be done in $O\left(\frac{q}{k}\right)$ time and operation 10 can be done in $O\left(p^{1/2}\right)$ time.

Since $p \leq q$, it follows from the above argument that:

Theorem 7.5 *Given n items in the first $\frac{n}{p}$ columns of a reconfigurable mesh of size $p \times q$, $p \leq q$ and $pq = kn$, these items can be sorted in $O\left(\frac{q}{k}\right)$ time, which is AT^2 optimal.* ■

7.3 An Adaptive Sorting Algorithm Based on Algorithm 4.1

In this section we develop another adaptive sorting algorithm, using the framework of generic adaptive Algorithm 7.1, to sort n items AT^2 optimally on a reconfigurable mesh of size $p \times q$, $p \leq q$ and $pq = kn$, based on the efficient AT^2 optimal sorting Algorithm 4.1 of Schnorr and Shamir on ordinary meshes.

As mentioned in Section 4.3.2, Algorithm 4.1 of Schnorr and Shamir uses a constant number of linear transformations in the form of sorting and cyclic rotation along rows or columns. Besides these, Algorithm 4.1 also recursively degenerates into sorting problems on smaller meshes.

Algorithm 4.1 sorts MN items in $O(M+N)$ time on an ordinary mesh of size $M \times N$ as long as $M^2 \geq N$. As mentioned in Section 4.3.2 we further assume $M \geq N^{3/4}$ for the sake of simplicity. On the contrary, if $M \ngeq N^{3/4}$ then $N > M^{4/3} \geq M^{3/4}$ then Algorithm 4.1 can easily be adapted by considering blocks (Figure 4.3(a)) of size $M^{3/4} \times M^{3/4}$ instead and transposing all row(column) operations into column(row) operations.

In this section we further assume $k = 2^{4r}$, $p = 2^{4s}$, and $q = 2^{4t}$ where r, s, and t are integers and $r \leq s \leq t$ for the sake of simplicity.

The principal module and the supporting module of this adaptive algorithm are presented in Sections 7.3.2 and 7.3.1 respectively.

7.3.1 The Supporting Module

In this section we develop an adaptive algorithm to sort p items on a reconfigurable mesh of size $p \times k$, $k \leq p$, according to the supporting module of Algorithm 7.1. We assume that phases 1s and 2s of Algorithm 7.1 are already completed, i.e., the mesh is divided into $\frac{p}{k}$ submeshes of size $k \times k$ each and the given p items in the first column are distributed in such a way that each processor $PE_{ik,j}$, $0 \leq i < \frac{p}{k}$ and $0 \leq j < k$, receives an item. This distribution of items transformed the column of p items into an array of $\frac{p}{k} \times k$ items where each row is separated by k rows. Now, phases 3s and 4s of Algorithm 7.1 emulate Algorithm 4.1 of Schnorr and Shamir. As phase 2 of Algorithm 4.1 can be obtained by cyclic rotation of each row in parallel, this emulation requires only

the following basic operations in various phases of Algorithm 4.1:

If $k \geq \left(\frac{p}{k}\right)^{3/4}$

1. Sort $\left(\frac{p}{k}\right)^{3/4} \times \left(\frac{p}{k}\right)^{3/4}$ items using a submesh of size $k\left(\frac{p}{k}\right)^{3/4} \times \left(\frac{p}{k}\right)^{3/4}$.
2. Rotate $\frac{p}{k}$ items using a submesh of size $p \times 1$.
3. Sort k items in a row using a submesh of size $k \times k$.
4. Sort $\left(\frac{p}{k}\right)^{3/4} \times 2\left(\frac{p}{k}\right)^{3/4}$ items using a submesh of size $k\left(\frac{p}{k}\right)^{3/4} \times 2\left(\frac{p}{k}\right)^{3/4}$.
5. Sort $\frac{p}{k}$ items using a submesh of size $p \times 1$.
6. Perform $2\left(\frac{p}{k}\right)^{3/4}$ steps of the odd-even transposition sort.

Else $(\Rightarrow \frac{p}{k} > k^{3/4})$

7. Sort $k^{3/4} \times k^{3/4}$ items using a submesh of size $kk^{3/4} \times k^{3/4}$.
8. Rotate k items in a row using a submesh of size $k \times k$.
9. Sort $\frac{p}{k}$ items using a submesh of size $p \times 1$.
10. Sort $2k^{3/4} \times k^{3/4}$ items using a submesh of size $2kk^{3/4} \times k^{3/4}$.
11. Sort k items in a row using a submesh of size $k \times k$.
12. Perform $2k^{3/4}$ steps of the odd-even transposition sort.

As mentioned in Section 7.2.1, the problem of rotation can always be transformed into a sorting problem without any slowdown. Now operations 3, 8, and 11 can be done in $O(1)$ time by Theorem 4.8. Using Corollary 7.3, operations 2, 5, and 9 can be done in $O\left(\frac{p}{k}\right)$ time.

Algorithm 4.1 of Schnorr and Shamir can be applied, by treating the reconfigurable mesh as an ordinary mesh, to solve operation 1 and 4 in $O\left(\left(\frac{p}{k}\right)^{3/4}\right)$ time, operation 7 and 10 in $O\left(k^{3/4}\right) = O\left(\left(\frac{p}{k}\right)^{3/4}\right)$ time.

So the above algorithm can be considered as another proof of Theorem 7.4. As Algorithm 4.1 of Schnorr and Shamir has lower constant with the highest order term in the complexity bound than rotatesort algorithm of Marberg and Gafni, the algorithm developed in this section is a more efficient implementation supporting Theorem 7.4 than the algorithm in Section 7.2.1.

7.3.2 The Principal Module

In this section we develop an adaptive algorithm to sort n items on a reconfigurable mesh of size $p \times q$, $p \leq q$ and $pq = kn$, according to the principal module of Algorithm 7.1. We assume that phases 1p and 2p of Algorithm 7.1 are already completed, i.e., the mesh is divided into $\frac{q}{k}$ submeshes of size $p \times k$ each and the given n items in the first $\frac{n}{p}$ columns are distributed in such a way that each processor $PE_{i,jk}$, $0 \leq i < p$ and $0 \leq j < \frac{q}{k}$, receives an item. This distribution of items transformed the given n items into an array of $p \times \frac{q}{k}$ items where each column is separated by k columns. Now, phases 3p and 4p of Algorithm 7.1 emulate Algorithm 4.1 of Schnorr and Shamir which requires only the following basic operations in various phases of Algorithm 4.1:

If $p \geq \left(\frac{q}{k}\right)^{3/4}$

1. Sort $\left(\frac{q}{k}\right)^{3/4} \times \left(\frac{q}{k}\right)^{3/4}$ items using a submesh of size $\left(\frac{q}{k}\right)^{3/4} \times k \left(\frac{q}{k}\right)^{3/4}$.
2. Rotate $\frac{q}{k}$ items using a submesh of size $1 \times q$.
3. Sort p items in a column using a submesh of size $p \times k$.
4. Sort $2\left(\frac{q}{k}\right)^{3/4} \times \left(\frac{q}{k}\right)^{3/4}$ items using a submesh of size $2\left(\frac{q}{k}\right)^{3/4} \times k \left(\frac{q}{k}\right)^{3/4}$.
5. Sort $\frac{q}{k}$ items using a submesh of size $1 \times q$.
6. Perform $2\left(\frac{q}{k}\right)^{3/4}$ steps of the odd-even transposition sort.

Else $(\Rightarrow \frac{q}{k} > p^{3/4})$

7. Sort $p^{3/4} \times p^{3/4}$ items using a submesh of size $p^{3/4} \times kp^{3/4}$.
8. Rotate p items in a column using a submesh of size $p \times k$.
9. Sort $\frac{q}{k}$ items using a submesh of size $1 \times q$.
10. Sort $p^{3/4} \times 2p^{3/4}$ items using a submesh of size $p^{3/4} \times 2kp^{3/4}$.
11. Sort p items in a column using a submesh of size $p \times k$.
12. Perform $2p^{3/4}$ steps of the odd-even transposition sort.

From equation (7.5) we get $k \leq p$. Hence, operations 3, 8, and 11 can be done in $O\left(\frac{p}{k}\right)$ time by Theorem 7.4. Using Corollary 7.3, operations 2, 5, and 9 can be done in $O\left(\frac{p}{k}\right)$ time.

Algorithm 4.1 of Schnorr and Shamir can be applied, by treating the reconfigurable mesh as an ordinary mesh, to solve operation 1 and 4 in $O\left(\left(\frac{q}{k}\right)^{3/4}\right)$ time, operation 7 and 10 in $O\left(p^{3/4}\right) = O\left(\left(\frac{q}{k}\right)^{3/4}\right)$ time.

So the above algorithm can be considered as a second proof of Theorem 7.5. As Algorithm 4.1 of Schnorr and Shamir is more efficient than rotatesort algorithm of Marberg and Gafni, the algorithm developed in this section is a more efficient implementation supporting Theorem 7.5 than the algorithm in Section 7.2.2.

7.4 An Adaptive $\mathcal{M}$ -Contour Algorithm

In this section we develop an adaptive algorithm, using the framework of generic adaptive Algorithm 7.1, to compute the $\mathcal{M}$ -contour of n planar points AT^2 optimally on a reconfigurable mesh of size $p \times q$, $p \leq q$ and $pq = kn$, based on our AT^2 optimal $\mathcal{M}$ -contour Algorithm 5.5 on 3-dimensional reconfigurable meshes.

As mentioned in Section 5.5.2, Algorithm 5.5 is primarily based on our observation on $\mathcal{M}$ -contour as stated in Lemmas 5.1 and 5.2.

In this section we further assume $k = r^2$, $p = k^2s^2$, and $q = k^2t^2$ where r, s, and t are integers and $s \leq t$ for the sake of simplicity.

The principal module and the supporting module of this adaptive algorithm are presented in Sections 7.4.2 and 7.4.1 respectively.

7.4.1 The Supporting Module

In this section we develop an adaptive algorithm to compute $\mathcal{M}$ -contour of p planar points on a reconfigurable mesh of size $p \times k$, $k \leq p$, according to the supporting module of Algorithm 7.1. We assume that phases 1s and 2s of Algorithm 7.1 are already completed, i.e., the mesh is divided into $\frac{p}{k}$ submeshes of size $k \times k$ each and the given p points in the first column are distributed in such a way that each processor $PE_{ki,j}$, $0 \leq i < \frac{p}{k}$ and $0 \leq j < k$, receives a point.

In addition to the distribution of given points in phase 2s, we also sort the points w.r.t. x-coordinate in row-major order by the sorting algorithm in Section 7.2.1, omit-

ting phases 1s and 2s of the sorting algorithm, in $O\left(\frac{p}{k}\right)$ time.

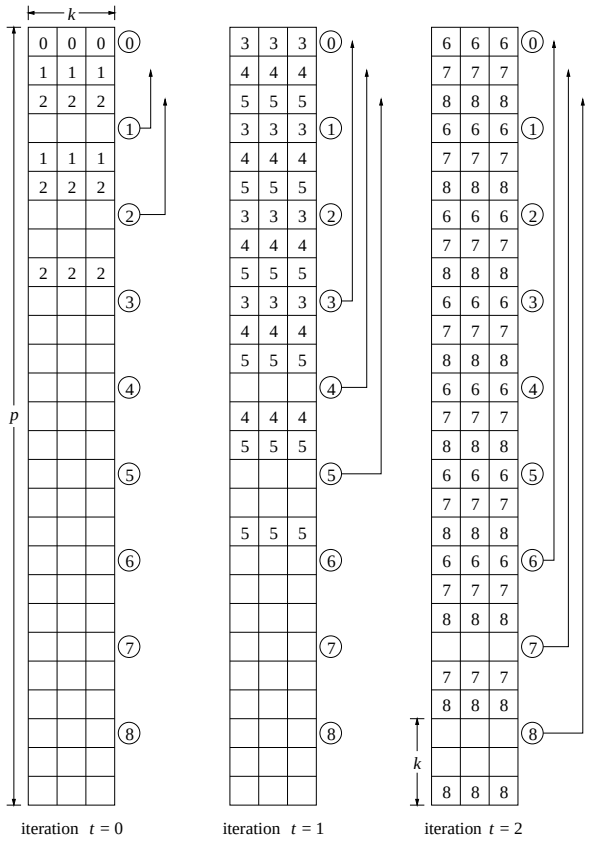

Figure 7.2: All the iterations to eliminate points based on Lemma 5.2. Here $p = 27$, $k = 3$, and the propagation of $max_y(m(S_i))$ is represented by integer i for $0 \leq i < 9$. Note that for simplification, k is not considered as r^2 for some integer r as assumed in this section.

Let the points residing in row ki be denoted by the set S_i, $0 \leq i < \frac{p}{k}$. Clearly these $\frac{p}{k}$ sets of planar points follow the condition of Lemma 5.2, i.e., for all $i : 0 \leq i < \frac{p}{k} - 1$, $max_x(S_i) \leq min_x(S_{i+1})$. The $\mathcal{M}$ -contour $m(S_i)$ is now computed in parallel using the i-th submesh of size $k \times k$ containing processors $PE_{r,*}$, $ki \leq r < k(i+1)$, for all $i : 0 \leq i < \frac{p}{k}$. By Theorem 5.8 this operation takes only $O(1)$ time. Using Lemma 5.1 we now transfer the $max_y(m(S_i))$ values to the first column of the mesh of size $p \times k$ by the following single step:

1. b: Any processor in row ki containing a point $\in m(S_i)$ disconnects all port interconnections while the rest of the processors connect port **E** with **W** for all $0 \leq i < \frac{p}{k}$;

w: Any processor in row ki containing a point $\in m(S_i)$ now writes the y-coordinate of the point to port **W** for all $0 \leq i < \frac{p}{k}$;

r: Every processor $PE_{ki,0}$ in the first column reads port **W** in for all $0 \leq i < \frac{p}{k}$;

The $\mathcal{M}$ -contour of the all p points can now be computed in the following phases using Lemma 5.2:

1. Iterate the following for $t = 0, 1, \ldots, \frac{p}{k^2} - 1$:

   1.1 Copy $max_y(m(S_{kt+u}))$ to processors $PE_{u+kr,j}$, $0 \leq j < k$, $r = 0, 1, \ldots, kt + u$, using a row broadcast then a column broadcast and finally a row broadcast for all $u : 0 \leq u < k$; An example is shown in Figure 7.2.

   1.2 Copy the y-coordinate of the point residing in processor $PE_{ki,j}$ to the processors $PE_{ki+r,j}$, $0 \leq r < k$, using a column broadcast for all $0 \leq i < k(t+1)$, $0 \leq j < k$;

   1.3 Now in the i-th submesh of size $k \times k$, the j-th column contains at most k max_y-values paired with the y-coordinate of a particular point, say d. Now, apply Lemma 5.2 to eliminate d by computing the *and* over the comparison values of at most k pairs using a single step similar to the only step in phase 8 of Algorithm 5.5 in constant time by Lemma 5.7 for all $i : 0 \leq i < k(t+1)$;

It is easy to show that the above iteration takes $O\left(\frac{p}{k^2}\right)$ time and thus it can be concluded that:

Theorem 7.6 *Given p planar points in the first column of a reconfigurable mesh of size $p \times k$, $k \leq p$, the $\mathcal{M}$ -contour of these points can be computed in $O\left(\frac{p}{k}\right)$ time, which is AT^2 optimal.* ■

7.4.2 The Principal Module

In this section we develop an adaptive algorithm to compute $\mathcal{M}$ -contour of n planar points on a reconfigurable mesh of size $p \times q$, $p \leq q$ and $pq = kn$, according to the principal module of Algorithm 7.1. We assume that phases 1p and 2p of Algorithm 7.1 are already completed, i.e., the mesh is divided into $\frac{q}{k}$ submeshes of size $p \times k$ each

and the given n points in the first $\frac{n}{p}$ columns are distributed in such a way that each processor $PE_{i,kj}$, $0 \leq i < p$ and $0 \leq j < \frac{q}{k}$, receives a point.

In addition to the distribution of given points in phase 2s, we also sort the points w.r.t. x-coordinate in column-major order by the sorting algorithm in Section 7.2.2, omitting phases 1p and 2p of the sorting algorithm, in $O\left(\frac{q}{k}\right)$ time.

Let the points residing in column kj be denoted by the set S_j, $0 \leq j < \frac{q}{k}$. Clearly these $\frac{q}{k}$ sets of planar points follow the condition of Lemma 5.2, i.e., $\forall j : 0 \leq j < \frac{q}{k} - 1$, $max_x(S_j) \leq min_x(S_{j+1})$. The $\mathcal{M}$ -contour $m(S_j)$ is now computed in parallel using the j-th submesh of size $p \times k$ containing processors $PE_{*,r}$, $kj \leq r < k(j+1)$, for all $j : 0 \leq j < \frac{q}{k}$. By Theorem 7.6 this operation takes only $O\left(\frac{p}{k}\right)$ time. Using Lemma 5.1 we now transfer the $max_y(m(S_j))$ values to the first row of the mesh of size $p \times q$ by the following single step:

1. b: Any processor in column kj containing a point $\in m(S_j)$ disconnects all port interconnections while the rest of the processors connect port **N** with **S** for all $0 \leq j < \frac{q}{k}$;

   w: Any processor in column kj containing a point $\in m(S_j)$ now writes the y-coordinate of the point to port **S** for all $0 \leq j < \frac{q}{k}$;

   r: Every processor $PE_{0,kj}$ in the first row reads port **S** in for all $0 \leq j < \frac{q}{k}$;

The $\mathcal{M}$ -contour of the all n points can now be computed in the following phases using Lemma 5.2 as done in Section 7.4.1:

1. Iterate the following for $t = 0, 1, \ldots, \frac{q}{k^2} - 1$:

   1.1 Copy $max_y(m(S_{kt+v}))$ to processors $PE_{i,v+kr}$, $0 \leq i < p$, $r = 0, 1, \ldots, kt+v$, using a column broadcast then a row broadcast and finally a column broadcast for all $v : 0 \leq v < k$;

   1.2 Copy the y-coordinate of the point residing in processor $PE_{i,kj}$ to the processors $PE_{i,kj+r}$, $0 \leq r < k$, using a row broadcast for all $0 \leq i < p$, $0 \leq j < k(t+1)$;

   1.3 Now in the j-th submesh of size $p \times k$, the i-th row contains k max_y-values paired with the y-coordinate of a particular point, say d. Now, apply

Lemma 5.2 to eliminate d by computing the *and* over the comparison values of at most k pairs using a single step similar to the only step in phase 8 of Algorithm 5.5 in constant time by Lemma 5.7 for all $j : 0 \leq j < k(t+1)$;

It is easy to show that the above iteration takes $O\left(\frac{q}{k^2}\right)$ time and thus it can be concluded that:

Theorem 7.7 *Given n planar points in the first $\frac{n}{p}$ columns of a reconfigurable mesh of size $p \times q$, $p \leq q$ and $pq = kn$, the $\mathcal{M}$ -contour of these points can be computed in $O\left(\frac{q}{k}\right)$ time, which is AT^2 optimal.* ■

7.5 A Conjecture

Is it possible that the parameter $k = \frac{pq}{I(n)}$ in Definition 7.1 has extra significance?

In Lemma 7.1, $k = \frac{I(n)I(n)}{I(n)} = I(n)$ and the reconfigurable mesh of size $I(n) \times I(n)$ must form some buses of length $\Omega(I(n))$, i.e., $\Omega(k)$ so that unit virtual communication diameter is achieved which is vital in getting constant time solution. In Lemma 7.2, $k = \frac{\sqrt{I(n)}\sqrt{I(n)}}{I(n)} = 1$. We already pointed out in Section 7.1 that the power of reconfigurability becomes absolutely useless as buses of length $O(1)$, i.e., $O(k)$ is enough, when problem $\mathcal{P}$ is tried to be solved AT^2 optimally on a reconfigurable mesh of size $\sqrt{I(n)} \times \sqrt{I(n)}$.

The following conjecture is plausible:

Conjecture 7.1 *To obtain an AT^2 optimal algorithm to solve a problem of size n with $I(n)$ information contents on a reconfigurable mesh of size $p \times q$ where $\sqrt{I(n)} \leq p \leq q \leq I(n)$ and $k = \frac{pq}{I(n)}$, it is sufficient to form buses of length $O(k)$.*

Does Conjecture 7.1 imply that the concept of adaptive algorithm on a reconfigurable mesh compromises with the power of reconfigurability as the unconstrained reconfigurable mesh can be considered as a k-constrained reconfigurable mesh? The answer is no. It is true that a k-constrained reconfigurable mesh is asymptotically no faster than an ordinary mesh as mentioned in Section 2.2.3.10 if and only if k is a con-

stant. Whereas in Conjecture 7.1, k is a function of $I(n)$ and k is a constant only in the extreme case when $q = O\left(\sqrt{I(n)}\right)$.

To support Conjecture 7.1, we show in the next section that the adaptive algorithms developed so far in this chapter can also be transformed on k-constrained reconfigurable meshes.

7.5.1 Adaptive Algorithms on Constrained Reconfigurable Meshes

Note that by the definition of k-constrained reconfigurable meshes in Section 2.2.3.10, a message on a bus of length l can propagate at most k length of the bus in unit time and, therefore, it requires $\Omega\left(\frac{l}{k}\right)$ time to transmit a message over the entire length of the bus.

Lemma 7.8 *Phase 2p of Algorithm 7.1 can be done in $O\left(\frac{q}{k}\right)$ time on a k-constrained reconfigurable mesh.*

Proof. For $0 \leq j < \frac{q}{k}$, let C_j denote the contents of column j in the initial condition. As mentioned in page 107 of Section 7.1, phase 2p can be considered as a problem of transferring C_j from column j to column kj using j steps while each step can be done in constant time by p broadcasts in parallel for all $j : 0 \leq j < \frac{q}{k}$. If the columns are transferred one at a time, phase 2p takes $O\left(\left(\frac{q}{k}\right)^2\right)$ time which is unacceptable.

However, as all the C_js are to be transferred in one direction, we can easily start to transfer these simultaneously using ideas of *pipelining* in the following way:

As we assume $k = r^2$ and $q = k^2t^2$ where r and t are integers in Section 7.4, we may conclude that $\frac{q}{k} = kt^2$. In the first pass, every C_j moves from column j to column $j+1$, for $1 \leq j < \frac{q}{k} - 1$ and $C_{\frac{q}{k}-1}$ moves from column $\frac{q}{k} - 1$ to column kt^2 as shown in Figure 7.3(pass 1). In the second pass, every C_j moves from column $j+1$ to column $j+2$, for $1 \leq j < \frac{q}{k} - 2$, $C_{\frac{q}{k}-2}$ moves from column $\frac{q}{k} - 1$ to column kt^2 and $C_{\frac{q}{k}-1}$ moves from column kt^2 to column $k(t^2+1)$ as shown in Figure 7.3(pass 2) and so on. There should be at most $\frac{q}{k} - 1$ passes and whenever any of C_js reaches its destination column in some pass, it will not take part in any of the remaining passes.

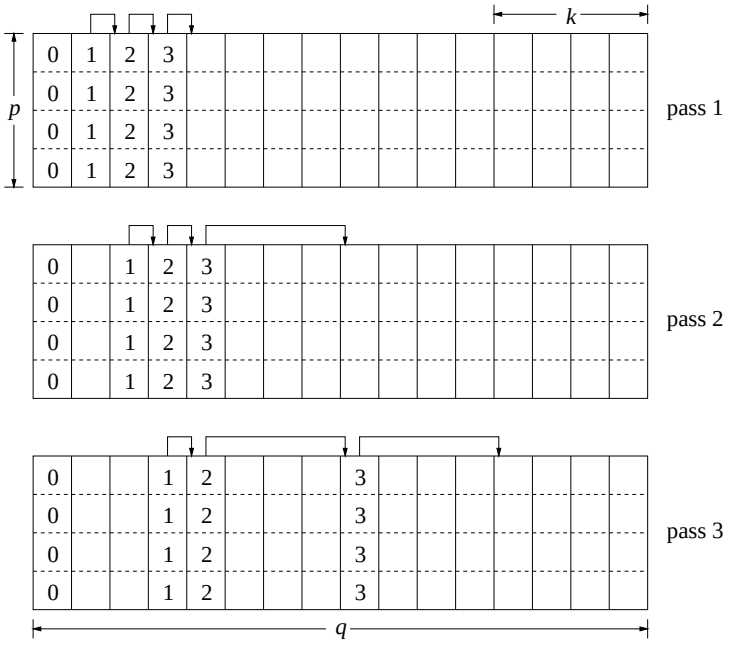

Figure 7.3: All the passes of the pipelining in the proof of Lemma 7.8. Here $p = 4$, $q = 16$, and $k = 4$ and C_j is represented by a column of integer j for $0 \leq j < 4$.

As none of the passes configures buses of length more than k, the above pipelining can be done in $O\left(\frac{q}{k}\right)$ time. ■

Lemma 7.9 *Phase 2s of Algorithm 7.1 can be done in $O(1)$ time on a k-constrained reconfigurable mesh.*

Proof. As mentioned in page 107 of Section 7.1, phase 2p can be done in two steps where buses of length at most k are constructed. ■

Lemma 7.10 *Given s items in processors PE_{kj}, $0 \leq j < s$, of a k-constrained reconfigurable array of ks processors, these item can be sorted in $O(s)$ time.*

Proof. As successive items are separated by exactly k processors, Corollary 7.3 on unconstrained reconfigurable arrays is still applicable without any modification. ■

Lemma 7.11 *Sorting of s items in a row of a k-constrained reconfigurable mesh of size $t \times s$ can be done in $O(1)$ time as long as $t \geq s$ and $s = O(k)$.*

Proof. It can be easily shown that all the algorithms discussed in Sections 4.4.1–4.4.3 configure buses of length at most $O(s) = O(k)$ to sort s items on a reconfigurable mesh of size $s \times s$. ■

The following two theorems provide adaptive sorting algorithms on k-constrained reconfigurable meshes:

Theorem 7.12 *Given p items in the first column of a k-constrained reconfigurable mesh of size $p \times k$, $k \leq p$, these items can be sorted in $O\left(\frac{p}{k}\right)$ time, which is AT^2 optimal.*

Proof. Consider the supporting module in Section 7.2.1 on a k-constrained reconfigurable mesh of size $p \times k$ where $k \leq p$. Operations 1, 5, 7, 8, and 9 can be done in $O(1)$ time by Lemma 7.11. Using Lemma 7.10 it can be shown that operation 2 can be done in $O\left(\left(\frac{p}{k}\right)^{3/4}\right)$ time, operations 3, 4, and 6 can be done in $O\left(\frac{p}{k}\right)$ time, and operation 10 can be done in $O\left(k^{3/4}\right)$ time as in each case items are separated by exactly k processors. Then Lemma 7.9 completes the proof. ■

Theorem 7.13 *Given n items in the first $\frac{n}{p}$ columns of a k-constrained reconfigurable mesh of size $p \times q$, $p \leq q$ and $pq = kn$, these items can be sorted in $O\left(\frac{q}{k}\right)$ time, which is AT^2 optimal.*

Proof. Consider the principal module in Section 7.2.2 on a k-constrained reconfigurable mesh of size $p \times q$ where $p \leq q$. Operations 1, 8, and 9 can be done in $O\left(\frac{p}{k}\right)$ time by Theorem 7.12.

If $k \geq \left(\frac{q}{k}\right)^{3/4}$ then operation 5 takes $O(1)$ time by Lemma 7.11 else it takes $O\left(q^{1/2}/k^{3/2}\right)$ time by Theorem 7.12. Similarly, if $k \geq p^{3/4}$ then operation 7 takes $O(1)$ time else it takes $O\left(p^{1/2}/k\right)$ time.

Using Lemma 7.10, operation 2 can be done in $O\left(\left(\frac{q}{k}\right)^{3/4}\right)$ time, operations 3, 4, and 6 can be done in $O\left(\frac{q}{k}\right)$ time and operation 10 can be done in $O\left(p^{3/4}\right)$ time. Lemma 7.8 completes the proof as $p \leq q$. ■

Theorem 7.12 first appeared in [10, 11]. Theorem 7.13 also first appeared in [10] but the proof presented in [10] is incomplete as Lemma 7.8 is not established which is an important part of our proof. In fact Theorem 7.13 has been removed from [11] which is an refined version of [10].

Lemma 7.14 *Computing the $\mathcal{M}$ -contour of k planar points in a row of a k-constrained reconfigurable mesh of size $k \times k$ can be done in $O(1)$ time.*

Proof. It can be easily shown that Algorithm 5.4 configures buses of length at most k. ■

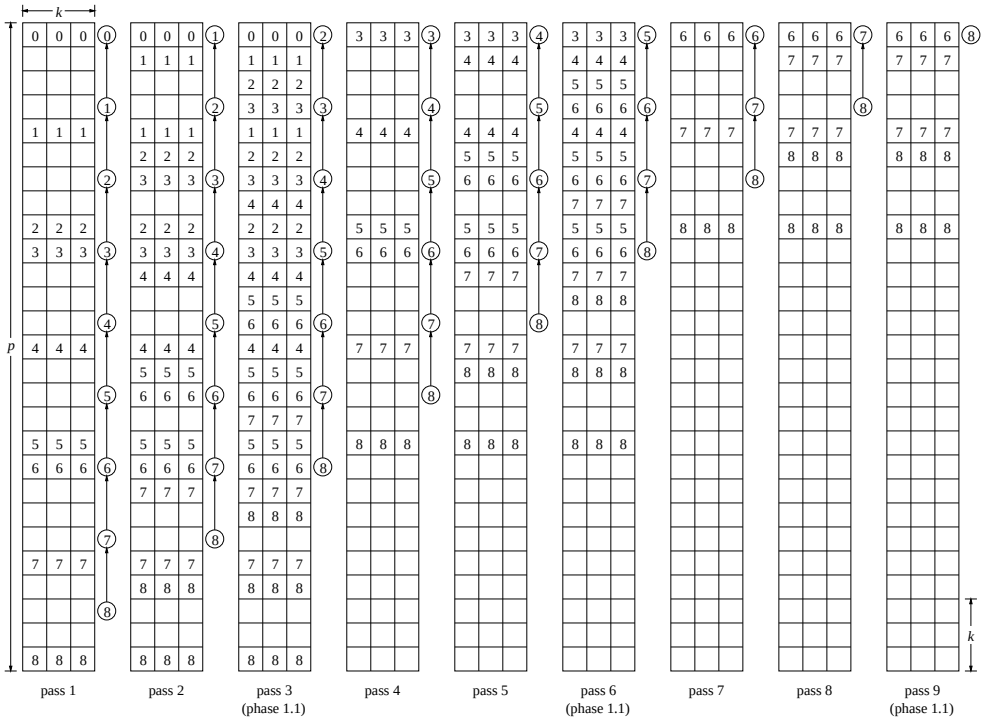

Figure 7.4: All the passes of the pipelining in the proof of Theorem 7.15. Here $p = 27$, $k = 3$, and the propagation of $max_y(m(S_i))$ is represented by integer i for $0 \leq i < 9$. Note that for simplification, k is not considered as r^2 for some integer r as assumed in Section 7.4.

The next two theorems provide adaptive $\mathcal{M}$ -contour algorithms on k-constrained reconfigurable meshes:

Theorem 7.15 *Given p planar points in the first column of a k-constrained reconfigurable mesh of size $p \times k$, $k \leq p$, the $\mathcal{M}$ -contour of these points can be computed in $O\left(\frac{p}{k}\right)$ time, which is AT^2 optimal.*

Proof. Consider the supporting module in Section 7.4.1 on a k-constrained reconfigurable mesh of size $p \times k$ where $k \leq p$. Additional sorting in phase 2s can be done in $O\left(\frac{p}{k}\right)$ time by Theorem 7.12. The $\mathcal{M}$ -contours $m(S_i)$, $0 \leq i < \frac{p}{k}$, can be computed in

parallel using a submesh of size $k \times k$ for each computation; this takes constant time by Lemma 7.14. The single step to transfer the $max_y(m(S_i))$ values to the first column of the mesh of size $p \times k$ configures buses of length at most k.

Now consider the $\frac{p}{k^2}$ iterations to eliminate points by comparing them with the $max_y(m(S_i))$ values. The length of the buses in the row broadcasts of phase 1.1 is k while the same in the column broadcasts is $k^2(t+1)$ in the t-th iteration. Thus, phase 1.1 takes $O(k(t+1))$ steps. Phases 1.2 and 1.3 configure buses of length at most k. So the entire iteration takes $O\left(\frac{p}{k^2}\frac{p}{k}\right)$ steps which is unacceptable.

Again, the ideas of *pipelining* as used in the proof of Lemma 7.8 can be applied to the above iteration to achieve optimal time. Observe that in the t-th iteration, data are moved from the u-th submeshes of size $k \times k$ each, $kt \leq u < k(t+1)$, to all the v-th submeshes of size $k \times k$ each, $0 \leq v < k(t+1)$. So, we can start all the iterations simultaneously without making any bus-access conflict by pipelining data as shown in Figure 7.4. Obviously such pipeline emulation of the above iteration takes only $O\left(\frac{p}{k}\right)$ steps as buses of length at most k are configured.

Lemma 7.9 completes the proof. ■

Theorem 7.16 *Given n planar points in the first $\frac{n}{p}$ columns of a k-constrained reconfigurable mesh of size $p \times q$, $p \leq q$ and $pq = kn$, the $\mathcal{M}$ -contour of these points can be computed in $O\left(\frac{q}{k}\right)$ time, which is AT^2 optimal.*

Proof. Consider the principal module in Section 7.4.2 on a k-constrained reconfigurable mesh of size $p \times q$ where $p \leq q$. Additional sorting in phase 2p can be done in $O\left(\frac{q}{k}\right)$ time by Theorem 7.13. The $\mathcal{M}$ -contours $m(S_j)$, $0 \leq j < \frac{q}{k}$, can be computed in parallel using a submesh of size $p \times k$ for each computation in $O\left(\frac{p}{k}\right)$ time by Theorem 7.15. The single step to transfer the $max_y(m(S_j))$ values to the first row of the mesh of size $p \times q$ configures buses of length at most k.

Now consider the $\frac{q}{k^2}$ iterations to eliminate points by comparing them with the $max_y(m(S_j))$ values. The length of the buses in the column broadcasts of phase 1.1 is k while the same in the row broadcasts is $k^2(t+1)$ in the t-th iteration. Thus, phase 1.1 takes $O(k(t+1))$ steps. Phases 1.2 and 1.3 configure buses of length at most k. So the entire iteration takes $O\left(\frac{q}{k^2}\frac{q}{k}\right)$ steps which is unacceptable.

Again, the ideas of *pipelining* can be applied to the above iteration to achieve optimal time. Observe that in the t-th iteration, data are moved from the u-th submeshes of size $p \times k$ each, $kt \leq u < k(t+1)$, to all the v-th submeshes of size $k \times k$ each, $0 \leq v < k(t+1)$. So, we can start all the iterations simultaneously without making any bus-access conflict by pipelining data in a similar way as done in the proof of Theorem 7.15. Obviously such pipeline emulation of the above iteration takes only $O\left(\frac{q}{k}\right)$ steps as buses of length at most k are configured.

Lemma 7.8 completes the proof. ■

7.6 Conclusions

In Chapter 6 we have shown that even with optimal slowdown, the resultant algorithms fail to remain AT^2 optimal when a large reconfigurable mesh is self-simulated on a smaller mesh for which AT^2 optimal algorithms exist. Although this observation devalues, if not rejects, the concept of self-simulation of reconfigurable meshes, fortunately the problem of efficient scaling down algorithms on reconfigurable meshes escapes the blow as we have introduced in this chapter the idea of developing adaptive algorithms which can run on reconfigurable meshes of variable sizes and aspect ratios without compromising AT^2 optimality. We have supported this idea by developing adaptive algorithms for sorting items and computing the $\mathcal{M}$ -contour of a set of planar points on reconfigurable mesh.

We have also proposed a conjecture which relates normal unrestricted reconfigurable meshes to constrained reconfigurable meshes in the process of designing adaptive algorithms. To substantiate this conjecture we have successfully transformed our adaptive algorithms on constrained reconfigurable meshes.

Adaptive algorithms make efficient use of optimal algorithms not merely on reconfigurable meshes but on ordinary meshes and linear arrays as well. We believe that the study of adaptive algorithms on reconfigurable meshes for solving any specific problem will enable researchers to develop better understanding of the problem across the a ranges of mesh-connected networks and ultimately this may lead to developing new efficient algorithms which are better than the existing ones.

In particular, we have been able to develop, in Chapter 8, a new AT^2 optimal $\mathcal{M}$-contour algorithm on ordinary meshes, by putting $k = 1$ in Theorem 7.16, with lower constant associated with the highest order term in the complexity order. This algorithm is then compared with the existing AT^2 optimal $\mathcal{M}$-contour Algorithm 5.3 of Dehne in Chapter 8.

Chapter 8

A New Asymptotically Optimal $\mathcal{M}$ -Contour Algorithm on Ordinary Meshes

We believe that the study of adaptive algorithms on reconfigurable meshes for solving any specific problem will lead to developing new efficient algorithms on mesh-connected networks—reconfigurable or ordinary ones. The aim of this chapter is to illustrate our belief by developing a new AT^2 optimal $\mathcal{M}$ -contour algorithm on ordinary meshes, based on the adaptive algorithm reflected in Theorem 7.16, with lower constant associated with the highest order term in the complexity order.

This chapter is organised as follows. The motivation of this chapter is given in the next introductory section. In Section 8.2 we present some relevant results and algorithms on transposing, broadcasting, and finding maxima on ordinary meshes. The minimum achievable constant factor in the highest order term in the complexity of Algorithm 5.3 is worked out in Section 8.3. In Section 8.4 we present a new optimal $\mathcal{M}$ -contour algorithm and it is shown that the constant factor of the highest order term in the complexity of this algorithm is much lower than that of Dehne's Algorithm 5.3.

8.1 Introduction

Dehne [24] has presented an optimal Algorithm 5.3 to compute the $\mathcal{M}$ -contour of N planar points on an ordinary mesh of size $N^{1/2} \times N^{1/2}$. The exact constant factor of the highest order term in the complexity of Algorithm 5.3, which $O(N^{1/2})$ expression hides, is not given in [24]. In this chapter we estimate that the straightforward imple-

mentation of the recursive binary divide-and-conquer Algorithm 5.3 requires $23N^{1/2}$ steps. We then reduce the required steps to $19N^{1/2}$ through pre-sorting and using an efficient strategy in dividing the mesh into halves. A lower bound is also established: any implementation of Dehne's Algorithm 5.3 requires at least $14N^{1/2}$ steps. In order to further reduce the constant factor in the complexity, we develop a new optimal non-recursive $N^{1/2}$-ary divide-and-conquer algorithm which requires only $7N^{1/2}$ steps.

8.2 Preliminaries

In most of the cases, throughout this chapter, we consider the constant factor of the highest order term but ignore the lower order additive terms in the complexity functions.

As stated in Section 5.4 we also assume that the output of an $\mathcal{M}$ -contour computation preserves the partial ordering of the maximal elements w.r.t. x-coordinates but these maximal elements are not necessarily contained in consecutive processors.

For simplified exposition, we further assume $N = 2^{2r}$ for some integer r.

This section is organised as follows. In Section 8.2.1 we discuss optimal transposing of $\frac{N}{2}$ items on a mesh of size $N^{1/2} \times \frac{1}{2}N^{1/2}$. Shuffled order, an interesting order of the processors for sorting, is discussed in Section 8.2.2. In Section 8.2.3 some relevant results on broadcasting and finding the maximum among a set of data items on ordinary meshes are presented.

8.2.1 Transposing on Specific Rectangular Meshes

Consider a mesh of size $N^{1/2} \times \frac{1}{2}N^{1/2}$ where $\frac{N}{2}$ items are stored one item per processor.

Definition 8.1 *Let a specific problem of permutation be called* transposing *a mesh if and only if the items of the mesh are rearranged in such a way that if the items are initially assumed in snake-like-column-major order, after the permutation these are now arranged in snake-like-row-major order as shown in Figure 8.1. The rearrangement sequence in transposing can start either from the top-left processor or from the top-right processor, Let these be denoted as* normal *and* inverted *transposing respectively (Figure 8.1).*

Throughout this chapter, if not stated otherwise, transposing will always be assumed to be normal on a rectangular mesh where the height is twice the width.

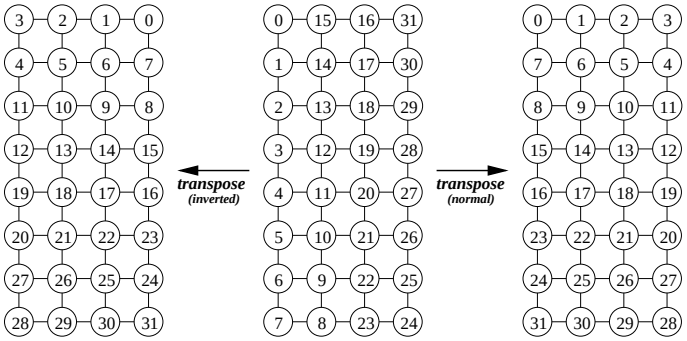

Figure 8.1: Transposing of an ordinary mesh.

In normal transposing the item in the top-right processor is moved to the bottom-left processor and in inverse transposing the item in the bottom-left processor is moved to the processor just below the top-right processor. Hence we may conclude the following lemma:

Lemma 8.1 *Transposing of* $\frac{N}{2}$ *items on an ordinary mesh of size* $N^{1/2} \times \frac{1}{2}N^{1/2}$ *takes at least* $\frac{3}{2}N^{1/2} - 2$ *and* $\frac{3}{2}N^{1/2} - 3$ *steps for normal and inverted transposing respectively.* ■

The general problem of permutation, which includes transposing as a simple case, can be solved in $\frac{3}{2}N^{1/2}$ steps (the fewest possible) using a greedy algorithm where at each step, each item that still needs to move does so first along the column then along the row provided there is no contention for the same edge. Contention is resolved by the *farthest-first* protocol [51, Section 1.7.1]. The only problem with such a greedy algorithm is the development of queues in the processors while resolving edge contentions. For solving the general problem of permutation, the queue size can become as large as $\frac{1}{3}N^{1/2} - 3$ [51, Section 1.7.1]. The maximum queue size in transposing can be computed as follows:

Consider the column i where $N^{1/2}$ items are stored from top to bottom order and also consider that after transposing the first $\frac{1}{2}N^{1/2}$ items are stored from left to right in the row $2i$ and the rest of the items are stored in the row $2i+1$ from right to left. Now, the items in processors $PE_{i,i}$, $PE_{i+1,i}$, ..., $PE_{\frac{1}{2}N^{1/2}-1,i}$ will all be contending for

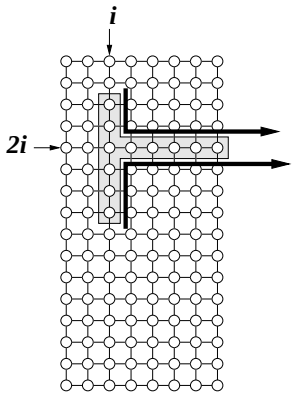

Figure 8.2: Development of queues in greedy transposing.

the east-bound link of the processor $PE_{2i,i}$ as shown in Figure 8.2 and this contention will form a queue of size $\min\left(\frac{1}{2}N^{1/2}-2i, i-1\right)$ in the processor $PE_{2i,i}$. The maximum queue size in transposing then will be

$$\max_{i=0}^{\frac{1}{2}N^{1/2}-1} \min\left(\frac{1}{2}N^{1/2}-2i, i-1\right) = \frac{1}{6}N^{1/2}-1\,.$$

Lemma 8.2 *$\frac{N}{2}$ items on an ordinary mesh of size $N^{1/2}\times\frac{1}{2}N^{1/2}$, with no penalty for excessive storage per processor, can be transposed in $\frac{3}{2}N^{1/2}$ steps while forming queues of maximum $\frac{1}{6}N^{1/2}-1$ items in the processors during the process.* ■

The general problem of permutation can be easily transformed into the problem of sorting where keys will be the rank of the destination processor in some specific ordering of the processors. The above Lemma 8.2 thus can also be realised by putting $k=\frac{1}{6}N^{1/2}-1$ in Lemma 4.5.

8.2.1.1 Transposing Without Forming Queues

As mentioned in Section 4.3.2, Lemma 4.5 is not applicable to a mesh where computing and storage capability of each processor is very limited. The general problem of permutation can certainly be realized on an ordinary mesh of size $M\times N$, with limited storage per processor, in $\max(M,N)+2\min(M,N)$ routing steps, by Lemma 4.4. We may conclude the that an ordinary mesh of size $N^{1/2}\times\frac{1}{2}N^{1/2}$ with $\frac{N}{2}$ items can be transposed in $2N^{1/2}$ steps, without forming any queues, by transforming the problem of transposing to a problem of sorting.

Transposing a mesh by means of sorting is not likely to be very efficient in practice, since for moderate values of N, the lower order term in the complexity of sorting is significant. Moreover the above technique also fails to achieve the lower bound in Lemma 8.1.

In this section we develop a queue-free transposing algorithm on ordinary meshes, the complexity of which is extremely close to the lower bound in Lemma 8.1.

Algorithm 8.1 Transposing Without Forming Queues

1 *Transpose* $\frac{N}{4}$ *items normally on the upper square submesh of size* $\frac{1}{2}N^{1/2} \times \frac{1}{2}N^{1/2}$ *and* $\frac{N}{4}$ *items invertedly on the lower square submesh of size* $\frac{1}{2}N^{1/2} \times \frac{1}{2}N^{1/2}$ *in parallel;*

2 *Perform* $\frac{1}{2}N^{1/2}$ *steps of the odd-even transposition sort (Section 4.2.2) along each column in parallel;*

Correctness of this algorithm can be established easily using the fact that every column is transposed into two successive rows which are arranged in mutually inverted order and after phase 1, each of the rows needs to be moved at most $\frac{1}{2}N^{1/2}$ positions in the upward or downward directions.

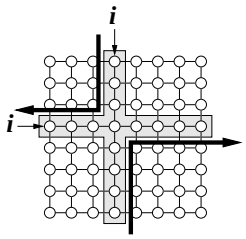

Figure 8.3: Greedy transposing of square meshes is queue free.

To show that Algorithm 8.1 is queue-free, it is sufficient to prove that the greedy algorithm for transposing square meshes never forms any queue. Consider column i which will be transposed into row i. Now every item in column i will be forwarded to the processor $PE_{i,i}$ but these will never contend for the same edge as the items

above the processor $PE_{i,i}$ will be taking the west-bound link and the items below the processor $PE_{i,i}$ will be taking the east-bound link as shown in Figure 8.3.

Lemma 8.3 *Algorithm 8.1 transposes* $\frac{N}{2}$ *items on an ordinary mesh of size* $N^{1/2} \times \frac{1}{2}N^{1/2}$ *in exactly* $\frac{3}{2}N^{1/2}$ *steps without forming any queue.*

Proof. It is obvious that the greedy algorithm, where at each step every item that still needs to move does so first along the column then along the row, takes at most $N^{1/2}$ steps to transpose $\frac{N}{4}$ items, normally or invertedly, on an ordinary square mesh of size $\frac{1}{2}N^{1/2} \times \frac{1}{2}N^{1/2}$. Hence, phase 1 of Algorithm 8.1 can be done in exactly $N^{1/2}$ steps. Clearly phase 2 takes $\frac{1}{2}N^{1/2}$ steps. ■

0	1	4	5	16	17	20	21
2	3	6	7	18	19	22	23
8	9	12	13	24	25	28	29
10	11	14	15	26	27	30	31
32	33	36	37	48	49	52	53
34	35	38	39	50	51	54	55
40	41	44	45	56	57	60	61
42	43	46	47	58	59	62	63

(a)

0	2	8	10	32	34	40	42
1	3	9	11	33	35	41	43
4	6	12	14	36	38	44	46
5	7	13	15	37	39	45	47
16	18	24	26	48	50	56	58
17	19	25	27	49	51	57	59
20	22	28	30	52	54	60	62
21	23	29	31	53	55	61	63

(b)

Figure 8.4: Shuffled row-major (a) and shuffled column-major (b) ordering of the processors of mesh for sorting.

8.2.2 Shuffled Sorting Orders

As mentioned in Section 4.3, there is no single natural ordering of the processors of a mesh for sorting. An unnatural ordering, named *shuffled ordering* [67], has been found to be very useful in many applications of divide-and-conquer approach. Shuffled ordering has the property that the first quarter of the processors form one quadrant, the next quarter form another quadrant, etc., with this property holding recursively within each quadrant. Based on the ordering of the quadrants, there can be many types of shuffled ordering; of these shuffled row-major (Figure 8.4(a)) and shuffled column-major (Figure 8.4(b)) orderings are most common.

Lemma 8.4 *Sorting of N items in shuffled row(column)-major order on an ordinary mesh of size* $N^{1/2} \times N^{1/2}$ *can be done in* $6N^{1/2} + O(\log N)$ *steps.*

Proof. First the items are sorted in snake-like order in $3N^{1/2}$ steps by Lemma 4.4 and then each processor computes the rank of the item it contains from the address of the processor. Now, in each processor, the destination processor, in shuffled order, for each item is computed from it's rank in $O(\log N)$ time by the binary search technique. The snake-like order rank of the destination processor of each item is then computed in constant time and the items are then sorted again in snake-like order using this rank as the key in $3N^{1/2}$ steps by Lemma 4.4. ■

An algorithm that directly sorts items in shuffled row(column)-major order may take fewer steps and to our knowledge, it still remains an open problem.

8.2.3 Broadcasting and Finding Maximum Item on Ordinary Meshes

The following lower bounds have been used in this chapter to analyse the complexity of the $\mathcal{M}$ -contour algorithms:

Lemma 8.5 *Broadcasting some data contained in some processor* $PE_{i,j}$ *on an ordinary mesh of size* $M \times N$ *can be done in* $M + N - 2$ *steps.*

Proof. First broadcast along row *i* and then broadcast along all the columns in parallel. ■

Lemma 8.6 *The maximum item on an ordinary mesh of size* $M \times N$ *can be accumulated into the processor* $PE_{0,0}$ *in* $M + N$ *steps.*

Proof. Accumulate the maximum of each column into the processors at row 0 by sorting each column in parallel in M steps by Theorem 4.1. Then accumulate the maximum of these maximums into the processor $PE_{0,0}$ by sorting row 0 in N steps by Theorem 4.1. ■

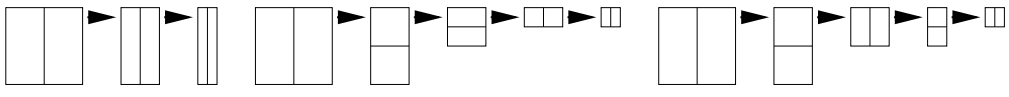

Figure 8.5: Different ways of dividing a square mesh into halves recursively.

8.3 Complexity Analysis of Dehne's Algorithm 5.3

In [24] Dehne develops an optimal Algorithm 5.3 to compute the $\mathcal{M}$-contour of N planar points on an ordinary mesh of size $N^{1/2} \times N^{1/2}$. The exact constant factor of the highest order term in the complexity of Algorithm 5.3, which $O(N^{1/2})$ expression hides, is not given in [24]. In this section we compute the lower bound of the constant factor of the highest order term in the complexity of Algorithm 5.3 to compare it with our new optimal $\mathcal{M}$-contour algorithm on ordinary meshes presented in Section 8.4.

If the recursive division of the mesh into halves, in Algorithm 5.3, is made only along columns as shown in Figure 8.5(a), phases 6 and 7 require $N^{1/2} + o(N^{1/2})$ steps (Lemmas 8.5 and 8.6) as long as a half contains a full column and the optimality will then be lost as Algorithm 5.3 will take $O(N^{1/2} \log N)$ time. Hence, the recursive division of the mesh into halves must be made as either Figure 8.5(b) or Figure 8.5(c). Now, if the sorting order in phase 1 is carefully chosen between snake-like-row-major and snake-like-column-major orders then phases 2 and 3 can be done in constant time. Phase 8 always takes constant time as no communication is needed. So, the following recurrence equation can be written for Algorithm 5.3 applying Lemmas 4.4:

$$\left.\begin{aligned} T(n) &= \overbrace{3N^{1/2}}^{\text{phase 9}} + T'(n) \\ T'(n) &= \underbrace{\overbrace{3N^{1/2}}^{\text{phase 1}} + 2\overbrace{\left(N^{1/2} + \frac{1}{2}N^{1/2}\right)}^{\text{phases 6 and 7}}}_{\text{pass 0}} + \underbrace{\overbrace{2N^{1/2}}^{\text{phase 1}} + 2\overbrace{\left(\frac{1}{2}N^{1/2} + \frac{1}{2}N^{1/2}\right)}^{\text{phases 6 and 7}}}_{\text{pass 1}} + T'(\tfrac{N}{4}) \end{aligned}\right\} \quad (8.1)$$

Solving equation (8.1) we get that Algorithm 5.3 requires $23N^{1/2}$ steps.

Now, a careful observation reveals that if the mesh is pre-sorted before applying

Algorithm 5.3 and the mesh is recursively divided as Figure 8.5(b) then phase 1 can be omitted in all the even numbered passes of the recursion. Hence,

$$\left.\begin{array}{rcl} T(n) & = & \overbrace{3N^{1/2}}^{\text{pre-sort}} + \overbrace{3N^{1/2}}^{\text{phase 9}} + T'(n) \\ T'(n) & = & \underbrace{2\left(N^{1/2}+\frac{1}{2}N^{1/2}\right)}_{\text{pass 0}} + \underbrace{\overbrace{2N^{1/2}}^{\text{phase 1}} + \overbrace{2\left(\frac{1}{2}N^{1/2}+\frac{1}{2}N^{1/2}\right)}^{\text{phases 6 and 7}}}_{\text{pass 1}} + T'(\tfrac{N}{4}) \end{array}\right\} \quad (8.2)$$

Solving equation (8.2) we get $20N^{1/2}$ steps.

Again, the purpose of phase 1 in all the odd passes the recursion in Algorithm 5.3 can also be achieved through transposing in fewer steps and thus we can further reduce the steps required by Algorithm 5.3 to $19N^{1/2}$ by using Lemma 8.3.

Interestingly, Algorithm 5.3 can also be executed in $19N^{1/2}$ steps if the mesh is pre-sorted in shuffled column-major order before applying Algorithm 5.3 and Figure 8.5(c) is followed in dividing the mesh. In such case phases 1–3 are not at all necessary. Hence,

$$\left.\begin{array}{rcl} T(n) & = & \overbrace{6N^{1/2}}^{\text{shuffled pre-sort}} + \overbrace{3N^{1/2}}^{\text{phase 9}} + T'(n) \\ T'(n) & = & \overbrace{\underbrace{2\left(N^{1/2}+\frac{1}{2}N^{1/2}\right)}_{\text{pass 0}} + \underbrace{2\left(\frac{1}{2}N^{1/2}+\frac{1}{2}N^{1/2}\right)}_{\text{pass 1}}}^{\text{phases 6 and 7}} + T'(\tfrac{N}{4}) \end{array}\right\} \quad (8.3)$$

Theorem 8.7 *Any implementation of Algorithm 5.3 requires at least* $14N^{1/2}$ *steps.*

Proof. Phases 1–3 can be discarded if the items are pre-sorted in shuffled column-major order. Lower bounds of phases 6 and 7 are stated in Lemmas 8.5 and 8.6. As mentioned in Section 8.2.2, sorting N items directly in shuffled row(column)-major order on a mesh of size $N^{1/2} \times N^{1/2}$ may take $< 6N^{1/2}$ steps. By Theorem 4.3, the lower bound of performing the shuffled pre-sort and phase 9 is $2N^{1/2}$ steps each. Hence, the

following recurrence equation holds:

$$\left.\begin{aligned} T(n) &= \overbrace{2N^{1/2}}^{\text{shuffled pre-sort}} + \overbrace{2N^{1/2}}^{\text{phase 9}} + T'(n) \\ T'(n) &= \overbrace{\underbrace{2\left(N^{1/2}+\frac{1}{2}N^{1/2}\right)}_{\text{pass 0}} + \underbrace{2\left(\frac{1}{2}N^{1/2}+\frac{1}{2}N^{1/2}\right)}_{\text{pass 1}}}^{\text{phases 6 and 7}} + T'(\tfrac{N}{4}) \end{aligned}\right\} \quad (8.4)$$

■

8.4 A New Optimal $\mathcal{M}$ -Contour Algorithm

Dehne's Algorithm 5.3 is a binary divide-and-conquer algorithm based on Lemma 5.2 where K is assumed to be 2. Lemma 5.2 also suggests that it is possible to develop a multi-ary divide-and-conquer $\mathcal{M}$ -contour algorithm where the problem will be divided recursively into more than two halves and the optimality of such an algorithm depends entirely on the complexity of merging the solutions of the divided subproblems. Moreover, recursion disappears when the problem is divided into $N^{1/2}$ subproblems. By putting $k = 1$ in Theorem 7.16 we get the following optimal $\mathcal{M}$ -contour algorithm on ordinary meshes:

Algorithm 8.2 A New Optimal $\mathcal{M}$ -Contour Algorithm

1 *Sort S in snake-like column-major order w.r.t. the x-coordinate of the points;*

2 *Let the points in column c be denoted by the set S_c for all $0 \leq c < N^{1/2}$. Compute $m(S_c)$, $0 \leq c < N^{1/2}$, in parallel;*

3 *For all $0 \leq c < N^{1/2}$: compute $max_y(m(S_c))$ in parallel;*

4 *Broadcast $max_y(m(S_c))$, $1 \leq c < N^{1/2}$, to all columns i, $0 \leq i < c$;*

5 *For all $0 \leq c < N^{1/2}$: for all $p \in m(S_c)$: if $y(p) > max_y(m(S_i))$, for all $i > c$ then set $m(S) \leftarrow m(S) \cup \{p\}$;*

Theorem 8.8 *Algorithm 8.2 requires at most* $7N^{1/2}$ *steps.*

Proof. By Lemma 4.4 phase 1 takes $3N^{1/2}$ steps. Phase 2 can be done in $N^{1/2}-1$ steps in the following way:

After phase 1 each column is sorted in either ascending or descending order. Every point in a column, say c, is systolically shifted to the descending direction for $N^{1/2}-1$ times and each processor in column c then try to discard the point it contains, from the $m(S_c)$ by comparing it with the shifted points. The correctness of this procedure to compute $m(S_c)$ can easily be established using Lemma 5.2.

Phase 3 can be done in $N^{1/2}$ steps by Theorem 4.1. Phase 4 can be done in $2N^{1/2}-2$ steps as follows:

Every $max_y(m(S_c))$ is broadcast to all the processors in column c, for all $0 \leq c < N^{1/2}$, in parallel by $N^{1/2}-1$ steps. Now, these maximum values are systolically shifted to the left in parallel for $N^{1/2}-1$ times.

Phase 5 takes constant time as it does not involve any communication. ■

The upper time bound of Algorithm 8.2 is just half of the lower time bound of Algorithm 5.3.

8.5 Conclusions

We have estimated that the straightforward implementation of the recursive binary divide-and-conquer Algorithm 5.3 of Dehne requires $23N^{1/2}$ steps to compute $\mathcal{M}$-contour of N planar points on an ordinary mesh of size $N^{1/2} \times N^{1/2}$. We have been able to reduce the required steps to $19N^{1/2}$ through pre-sorting and using an efficient strategy in dividing the mesh into halves. A lower bound has also been established by showing that any implementation of Dehne's Algorithm 5.3 requires at least $14N^{1/2}$ steps. We have also developed a new optimal non-recursive $N^{1/2}$-ary divide-and-conquer algorithm which requires only $7N^{1/2}$ steps to compute $\mathcal{M}$-contour of N planar points on an ordinary mesh of size $N^{1/2} \times N^{1/2}$. This is just half of the lower bound of Algorithm 5.3.

Chapter 9

Conclusions

In the domain of parallel computation, the reconfigurable mesh has recently drawn much attention because of its superior interprocessor communication capability through reconfiguration of an underlying bus architecture. Yet a commercially viable parallel computer is only possible if we solve some key problems like deriving a high level programming models without compromising the power of the reconfigurable mesh and developing strategies and algorithms for efficient scaling down of algorithms written on larger meshes. Ironically the major resistance to overcome in solving these problems arises from the strength of reconfiguration.

The aim of this thesis has been to contribute to the acceptance of the reconfigurable mesh as the next generation architecture of massively parallel computers. In pursuing this aim we have developed a new programming model for the reconfigurable mesh and developed a new strategy of adaptation to address the problem of scaling down algorithms. We have also contributed to the existing strategy of self-simulation in solving the same scaling down problem by developing new self-simulation algorithms. Finally we have extended the application of the reconfigurable mesh by developing a number of efficient and optimal constant time algorithms for computing the contour of maximal elements of a set of planar points on reconfigurable meshes of various dimensions. To assist in the development of the above algorithms we have also made an extensive survey of optimal sorting algorithms on mesh-connected networks.

The main contributions of the thesis, with reference to possible future extensions of the results, are outlined below:

i) We have defined a new programming model for 3-dimensional reconfigurable meshes by means of a new programming language, RMPC (Reconfigurable Mesh Parallel C). A serial simulator, RMSIM (Reconfigurable Mesh SIMulator), has also been developed to assist in developing and executing RMPC programs on a simulated 3-dimensional reconfigurable mesh. The strength of RMPC over other existing programming models for the reconfigurable mesh lies in its unique ability to reuse programs, like subroutine calls, in different axis-orientations and/or within restricted regions. RMPC has also opened up some key issues, in defining programming models for multi-dimensional reconfigurable meshes, to be addressed in future development.

ii) We have introduced two unique properties of the maximal contour ($\mathcal{M}$ -contour) of a set of planar points which have aided immensely in developing efficient parallel algorithms. Exploiting these properties we have developed a number of AT^2 optimal parallel algorithms on different mesh-connected networks of various dimensions. In fact the $\mathcal{M}$ -contour problem can be regarded as the second theme of the thesis as we have used this problem in establishing most of our ideas, from developing adaptive algorithms on restricted as well as unrestricted reconfigurable meshes to extending the application of adaptive algorithms by extracting new algorithms on the ordinary mesh. We have also made an extensive survey of optimal parallel $\mathcal{M}$ -contour algorithms on mesh-connected networks. Our algorithm on 2-dimensional reconfigurable meshes can also be easily adapted to compute maximal elements of a set of points in *multi-dimensional space* AT^2 optimally. It is still an open problem whether an AT^2 optimal algorithm exists for solving the above problem on higher dimensional reconfigurable meshes.

iii) We have surveyed a number of self-simulation algorithms as self-simulation is the widely-accepted strategy for solving the problem of scaling down algorithms written on larger meshes. We have argued that the existing self-simulation algorithms, where the simulation involves the problem of computing the connected components of graphs, are unnecessarily complex and inefficient for self-simulating a large class of problems. To support this argument we have pre-

sented two restricted models of the reconfigurable mesh, the Monotonic-Bus (MB) model and the Piecewise-Monotonic-Bus (PMB) model, and then we have developed a new generic self-simulation algorithm SIMPLE, avoiding any computation of connected components, which can self-simulate the new models with asymptotically optimal slowdown, and for which the constant factor associated with the optimal slowdown is much less than that of the algorithms which exploit computations of connected components. It would be interesting to see whether our algorithm SIMPLE can also self-simulate some special instances of the tree-RM model which resembles our new models in making global restrictions of bus configuration.

iv) An important contribution of the thesis is in devaluing, if not rejecting, self-simulation as an efficient strategy for solving the problem of scaling down algorithms with the finding that, even with optimal slowdown, the resultant algorithms fail to remain AT^2 optimal when a large reconfigurable mesh is self-simulated on a smaller mesh for which AT^2 optimal algorithms exist. As an alternative strategy for solving the problem of scaling down algorithms, we have introduced the idea of developing adaptive algorithms which can run on reconfigurable meshes of variable sizes and aspect ratios without compromising AT^2 optimality. We have supported this idea by developing adaptive algorithms for sorting items and computing the $\mathcal{M}$ -contour of a set of planar points on reconfigurable meshes.

v) We have conjectured that in developing adaptive algorithms, it is sufficient to configure buses whose lengths are bounded solely by the parameter which represents how much the mesh is filled with data initially. To substantiate our conjecture we have successfully transformed our adaptive algorithms on the *constrained* reconfigurable mesh where buses of at most a fixed length are allowed to be formed.

vi) We have argued that the study of adaptive algorithms on reconfigurable meshes for solving any specific problem will lead to developing new efficient algorithms on mesh-connected networks—reconfigurable or ordinary ones. This argument

has then been supported by extracting a new AT^2 optimal $\mathcal{M}$ -contour algorithm, on the ordinary mesh, from our adaptive $\mathcal{M}$ -contour algorithm. The new algorithm has lower constant associated with the highest order term in the complexity function than the existing optimal algorithm. This example has opened up a new frontier in the study of adaptive algorithms.

We have used only two problems in supporting our idea of developing adaptive algorithms. We are hopeful that the idea of adaptive algorithms will be well-established in future and researchers will come forward in developing adaptive algorithms to solve hundreds of problems available in the field of computational geometry, graphs, image processing, routing and ranking, arithmetic and vector computations etc. Exploring the relationship between the restricted and the unrestricted general reconfigurable mesh remains a challenging area of future research. We also believe that the process of developing adaptive algorithms on the reconfigurable mesh will lead to the development of many new efficient optimal algorithms on different mesh-connected networks. It is also interesting to see how the idea of adaptive algorithms can aid in scaling down algorithms on other reconfigurable architectures–existing and yet to appear.

The reconfigurable mesh is an exciting idea whose time has not yet come. We hope that the results and methods of this thesis, focusing on programming, algorithmic, scaling, and optimality issues, will make a significant contribution towards hastening that time.

Appendix A

The Source Code of RMSIM

The complete source code of the serial simulator RMSIM (Reconfigurable Mesh SIMulator), presented in Chapter 3, with a reasonable amount of technical details and example programs can be obtained by contacting the author and the same is also freely available by `ftp://cslab.anu.edu.au/pub/Manzur/RMSIM`.

Bibliography

[1] A. Aggarwal and S. Suri. Fast algorithms for computing the largest empty rectangle. In *Proceedings of the Third Annual ACM Symposium on Computational Geometry*, pp. 178–290, 1987.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers : Principles, Techniques, and Tools*. Addison-Wesley Pub. Co., 1985.

[3] A. Asano, M. Sato, and T. Ohtsuki. Computational geometry algorithms. In T. Ohtsuki, editor, *Layout Design and Verification*, pp. 295–347. Elsevier Science Publishers, North Holland, 1986.

[4] Y. B. Asher, D. Gordon, and A. Schuster. Efficient self-simulation algorithms for reconfigurable arrays. *Journal of Parallel and Distributed Computing*, 30:1–22, 1995.

[5] Y. Ben-Asher, K. J. Lange, D. Peleg, and A. Schuster. The complexity of reconfiguring network models. *Information and Computation*, 121:41–58, 1995.

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

[7] Y. Ben-Asher and A. Schuster. Ranking on reconfigurable networks. *Parallel Processing Letters*, 1:149–156, 1991.

[8] Y. Ben-Asher and A. Schuster. Time-size tradeoffs for reconfigurable meshes. *Parallel Processing Letters*, 6:231–245, 1996.

[9] Y. Ben-Asher and A. Schuster. Single step undirected reconfigurable networks. To appear in *VLSI Design*, Special issue on High Performance Bus-Based Architectures, 1998.

[10] B. Beresford-Smith, O. Diessel, and H. ElGindy. Optimal algorithms for constrained reconfigurable meshes. *Australian Computer Science Communications*, 17:32–41, 1995.

[11] B. Beresford-Smith, O. Diessel, and H. ElGindy. Optimal algorithms for constrained reconfigurable meshes. *Journal of Parallel and Distributed Computing*, 39:74–78, 1996.

[12] P. Bhattacharya. Connected component labeling for binary images on a reconfigurable mesh architecture. *Journal of Systems Architecture*, 42:309–313, 1996.

[13] V. Bokka, H. Gurla, S. Olariu, and J. L. Schwing. Constant-time convexity problems on reconfigurable meshes. *Journal of Parallel and Distributed Computing*, 27:86–99, 1995.

[14] K. Bondalapati and V. K. Prasanna. Reconfigurable meshes: theory and practice. In *Reconfigurable Architectures Workshop*. International Parallel Processing Symposium, April 1997.

[15] C.-C. Chao, W.-T. Chen, and G.-H. Chen. Multiple search problems on reconfigurable meshes. *Information Processing Letters*, 58:65–69, 1996.

[16] B. Chazelle, R. L. Drysdale, and D. T. Lee. Computing the largest empty rectangle. *SIAM Journal on Computing*, 15:300–315, 1986.

[17] G. H. Chen, S. Olariu, J. L. Schwing, B. F. Wang, and J. Zhang. Constant-time tree algorithms on reconfigurable meshes on size $n \times n$. *Journal of Parallel and Distributed Computing*, 26:137–150, 1995.

[18] G.-H. Chen and B.-F. Wang. Sorting and computing convex hulls on processor arrays with reconfigurable bus systems. *Information Sciences*, 72:191–206, 1993.

[19] G.-H. Chen, B.-F. Wang, and H. Li. Deriving algorithms on reconfigurable networks based on function decomposition. *Theoretical Computer Science*, 120:215–227, 1993.

[20] Y.-C. Chen and W.-T. Chen. Constant time sorting on reconfigurable meshes. *IEEE Transactions on Computers*, 43:749–751, 1994.

[21] K. L. Chung. Fast string matching algorithms for run-length coded string. *Computing*, 54:119–125, 1995.

[22] K.-L. Chung and H.-Y. Lin. Hough transform on reconfigurable meshes. *Computer Vision and Image Understanding*, 61:278–284, 1995.

[23] A. Datta and K. Krithivasan. Efficient algorithms for the maximum empty rectangle problem in shared memory and other architectures. In *Proceedings of the International Conference on Parallel Processing*, pp. 344–345, 1990.

[24] F. Dehne. $O(n^{1/2})$ algorithms for the maximal elements and ECDF searching problem on a mesh-connected parallel computer. *Information Processing Letters*, 22:303–306, 1986.

[25] F. Dehne. Computing the largest empty rectangle on one- and two-dimensional processor arrays. *Journal of Parallel and Distributed Computing*, 9:63–68, 1990.

[26] O. Diessel, H. ElGindy, and L. Wetherall. Efficient broadcasting procedures for constrained reconfigurable meshes. Technical Report 96-07, Department of Computer Science, The University of Newcastle, Australia, 1996.

[27] H. ElGindy. A sparse matrix multiplication algorithm for the reconfigurable mesh architecture. Technical Report 96-08, Department of Computer Science, The University of Newcastle, Australia, 1996.

[28] H. ElGindy and L. Wetherall. An L_1 Voronoi diagram algorithm for a reconfigurable mesh. In *Proceedings of the First International Conference on Algorithms and Architectures for Parallel Processing*, pp. 442–449, Brisbane, Australia, April 1995.

[29] H. ElGindy and L. Wetherall. A simple Voronoi diagram algorithm for a reconfigurable mesh. *IEEE Transactions on Parallel and Distributed Systems*, 8:1133–1142, 1997.

[30] F. Ercal and H. C. Lee. Time-efficient maze routing algorithms on reconfigurable mesh architectures. *Journal of Parallel and Distributed Computing*, 44:133–140, 1997.

[31] J. A. Fernández-Zepeda, R. Vidyanathan, and J. L. Trahan. Scaling simulation of the fusing-restricted reconfigurable mesh. *IEEE Transactions on Parallel and Distributed Systems*, 9:861–871, 1998.

[32] J. A. Fernández-Zepeda, R. Vidyanathan, and J. L. Trahan. Improved scaling simulation of the general reconfigurable mesh. In *Proceedings of the 6th Reconfigurable Architecture Workshop (Parallel and Distributed Processing; Lecture Notes Comp. Sci. #1586)*, pp. 616–624, 1999.

[33] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, U.S.A., 1992.

[34] J. Jang, H. Park, and V. K. Prasanna. A bit model of reconfigurable mesh. In *Proceedings of Reconfigurable Architecture Workshop: International Parallel Processing Symposium*, April 1994.

[35] J. Jang and V. K. Prasanna. A fast sorting algorithm on higher dimensional reconfigurable mesh. In *Proceedings of the 26th Conference on Information Sciences and Systems*, Princeton, 1992.

[36] J.-W. Jang and K. G. Lee. Fast parallel radix sort using a reconfigurable mesh. *International Journal of High Speed Computing*, 9:25–39, 1997.

[37] J.-W. Jang, M. Nigam, V. K. Prasanna, and S. Sahni. Constant time algorithms for computational geometry on the reconfigurable mesh. *IEEE Transactions on Parallel and Distributed Systems*, 8:1–12, 1997.

[38] J.-W. Jang, H. Park, and V. K. Prasanna. A fast algorithm for computing a histogram on reconfigurable mesh. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17:97–106, 1995.

[39] J.-W. Jang, H. Park, and V. K. Prasanna. An optimal multiplication algorithm on reconfigurable mesh. *IEEE Transactions on Parallel and Distributed Systems*, 8:521–532, 1997.

[40] J.-W. Jang and V. K. Prasanna. An optimal sorting algorithm on reconfigurable mesh. *Journal of Parallel and Distributed Computing*, 25:31–41, 1995.

[41] J.-F. Jeno and S. Sahni. Reconfigurable mesh algorithms for the hough transform. *Journal of Parallel and Distributed Computing*, 20:69–77, 1994.

[42] T.-W. Kao, S.-J. Horng, and Y.-H. Guo. Constant time algorithms for graph connectivity problems on reconfigurable meshes using fewer processors. *International Journal of High speed Computing*, 4:371–385, 1996.

[43] A. Kapoor, H. Schröder, and B. Beresford-Smith. Constant time sorting on a reconfigurable mesh. In *Australian Computer Science Communications*, volume 15, pp. 121–132, 1993.

[44] M. Kaufmann, H. Schröder, and J. F. Sibeyn. Routing and sorting on reconfigurable meshes. *Parallel Processing Letters*, 5:81–95, 1995.

[45] D. E. Knuth. *The Art of Computer Programming*, volume Vol. 3: Sorting and Searching. Addison-Wesley, Reading, MA, 1973.

[46] H. T. Kung. On the computational complexity of finding the maxima of a set of vectors. In *15th Annual IEEE Symp. on Switching and Automata Theory*, pp. 117–121, Oct. 1974.

[47] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *Journal of the ACM*, pp. 469–476, 1975.

[48] T.-H. Lai and M.-J. Sheng. Triangulation on reconfigurable meshes: a natural decomposition approach. *Journal of Parallel and Distributed Computing*, 30:38–51, 1995.

[49] T. H. Lai and M.-J. Sheng. Constructing Euclidean minimum spanning trees and all nearest neighbors on reconfigurable meshes. *IEEE Transactions on Parallel and Distributed Systems*, 7:806–817, 1996.

[50] F. T. Leighton. *Complexity Issues in VLSI*. MIT Press, Cambridge, MA, 1983.

[51] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays - Trees - Hypercubes*. Morgan Kaufmann Publishers, San Mateo, California, 1992.

[52] T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34:344–354, 1985.

[53] H. Li and M. Maresca. Polymorphic-torus network. *IEEE Transactions on Computers*, 38:1345–1351, 1989.

[54] H. Li and Q. F. Stout. Reconfigurable SIMD massively parallel computers. *Proc. IEEE*, 79:1345–1351, 1991.

[55] R. Lin, S. Olariu, J. L. Schwing, and J. Zhang. Sorting in $O(1)$ time on an $n \times n$ reconfigurable mesh. In *Parallel Computing: From Theory to Sound Practice, Proceedings of the European Workshop on Parallel Computing*, pp. 16–27, Amsterdam, 1992.

[56] P. D. Mackenzie. A separation between reconfigurable mesh models. *Parallel Processing Letters*, 5:15–22, 1995.

[57] J. M. Marberg and E. Gafni. Sorting in constant number of row and column phases on a mesh. *Algorithmica*, 3:561–572, 1988.

[58] M. Maresca. Polymorphic processor arrays. *IEEE Transactions on Parallel and Distributed Systems*, 4:490–506, 1993.

[59] M. Maresca and H. Li. Connection autonomy in SIMD computers: a VLSI implementation. *Journal of Parallel and Distributed Computing*, 7:302–320, 1989.

[60] M. Maresca and H. Li. Virtual parallelism support in reconfigurable processor arrays. Technical Report 91-041, UCB-ICSI Tech. Rep., July 1991.

[61] Y. Matias and A. Schuster. Fast, efficient mutual and self-simulations for shared memory and reconfigurable mesh. In *Proceedings of the Seventh Symposium on Parallel and Distributed Processing*, pp. 238–246, 1995.

[62] S. Matsumae and N. Tokura. Simulation algorithms among enhanced mesh models. Accepted in IEICE Transactions on Information and Systems.

[63] M. S. Merry and J. Baker. A constant time sorting algorithm for a three dimensional reconfigurable mesh and reconfigurable network. *Parallel Processing Letters*, 5:401–412, 1995.

[64] R. Miller, V. K. P. Kumar, D. I. Reisis, and Q. F. Stout. Data movement operations and applications on reconfigurable VLSI arrays. In *Proc. International Conference on Parallel Processing*, pp. 205–208, 1988.

[65] R. Miller, V. P. Kumar, D. I. Reisis, and Q. F. Stout. Meshes with reconfigurable buses. In *Proceedings of the 5th MIT Conference on Advanced Research in VLSI*, pp. 163–178, March 1988.

[66] R. Miller, V. K. Prasanna-Kumar, D. I. Reisis, and Q. F. Stout. Parallel computations on reconfigurable meshes. *IEEE Transactions on Computers*, 42:678–692, 1993.

[67] R. Miller and Q. F. Stout. Mesh computer algorithms for computational geometry. *IEEE Transactions on Computers*, 38:321–340, 1989.

[68] K. Miyashita, Y. Shimizu, and R. Hashimoto. A visualization system for algorithms on PARBS. In *Proceedings of the International Conference on Computer and Information Technology*, pp. 215–219, Dhaka, Bangladesh, December 1998.

[69] M. M. Murshed and R. Brent. Maximal contour algorithms on constrained reconfigurable meshes. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, U.S.A., 1999.

[70] M. M. Murshed and R. Brent. A new adaptive sorting algorithm on the reconfigurable mesh, an image understanding architecture. In *Proceedings of the International Symposium on Intelligent Multimedia and Distance Education*, Baden-Baden, Germany, 1999.

[71] M. M. Murshed and R. P. Brent. Constant time algorithms for computing the contour of maximal elements on the reconfigurable mesh. In *Proceedings of the 1997 International Conference on Parallel and Distributed Systems*, pp. 172–177, Seoul, Korea, November 1997. Korea University, IEEE Computer Society.

[72] M. M. Murshed and R. P. Brent. Adaptive AT^2 optimal algorithms on reconfigurable meshes. In *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Systems*, pp. 190–195, Las Vegas, Nevada, U.S.A., October 1998.

[73] M. M. Murshed and R. P. Brent. Algorithms for optimal self-simulation of some restricted reconfigurable meshes. In *Proceedings of the 2nd International Conference on Computational Intelligence and Multimedia Applications 1998*, pp. 734–744, Gippsland, Australia, February 1998. Monash University, World Scientific.

[74] M. M. Murshed and R. P. Brent. Constant time algorithms for computing the contour of maximal elements on a reconfigurable mesh. *Parallel Processing Letters*, 8:351–361, 1998.

[75] M. M. Murshed and R. P. Brent. Serial simulation of reconfigurable mesh, an image understanding architecture. In *Advances in Computer Cybernetics (Vol. V), Proceedings of the International Symposium on Audio, Video, Image Processing and Intelligent Applications*, pp. 92–97, Baden-Baden, Germany, August 1998. The International Institute for Advanced Studies in Systems Research and Cybernetics.

[76] M. M. Murshed and M. Hegland. Optimal computation of the contour of maximal elements on mesh-connected computers. In *Proceedings of the International Conference on Computer and Information Technology*, pp. 209–214, Dhaka, Bangladesh, December 1998.

[77] K. Nakano. A bibliography of published papers on dynamically reconfigurable architectures. *Parallel Processing Letters*, 5:111–124, 1995.

[78] K. Nakano. Optimal initializing algorithms for a reconfigurable mesh. *Journal of Parallel and Distributed Computing*, 24:218–223, 1995.

[79] K. Nakano. Prefix-sums algorithms on reconfigurable meshes. *Parallel Processing Letters*, 5:23–35, 1995.

[80] K. Nakano, T. Masuzawa, and N. Tokura. A sub-logarithmic time sorting algorithm on a reconfigurable mesh. *IEICE Transactions*, E-74:894–901, 1991.

[81] K. Nakano and S. Olariu. An optimal algorithm for the angle-restricted all nearest neighbor problem on the reconfigurable mesh, with applications. *IEEE Transactions on Parallel and Distributed Systems*, 8:983–990, 1997.

[82] D. Nassimi and S. Sahni. Bitonic sort on a mesh-connected parallel computer. *IEEE Transactions on Computers*, 27:2–7, 1979.

[83] M. Nigam and S. Sahni. Sorting n numbers on $n \times n$ reconfigurable meshes with buses. *Journal of Parallel and Distributed Computing*, 23:37–48, 1994.

[84] M. Nigam and S. Sahni. Sorting N^2 numbers on $N \times N$ meshes. *IEEE Transactions on Parallel and Distributed Systems*, 6:1221–1225, 1995.

[85] S. Olariu and J. L. Schwing. A novel deterministic sampling scheme with applications to broadcast-efficient sorting on the reconfigurable mesh. *Journal of Parallel and Distributed Computing*, 32:215–222, 1996.

[86] S. Olariu, J. L. Schwing, and J. Zhang. On the power of two-dimensional processor arrays with reconfigurable bus systems. *Parallel Processing Letters*, 1:29–34, 1991.

[87] S. Olariu, J. L. Schwing, and J. Zhang. Applications of reconfigurable meshes to constant-time computations. *Parallel Computing*, 19:229–237, 1993.

[88] S. Olariu, J. L. Schwing, and J. Zhang. Integer problems on reconfigurable meshes, with applications. *J. Comput. Software Engrg.*, 1:33–45, 1993.

[89] S. Olariu, J. L. Schwing, and J. Zhang. Optimal convex hull algorithms on enhanced meshes. *BIT*, 33:396–410, 1993.

[90] S. Olariu, W. Shen, and L. Wilson. Sub-logarithmic algorithms for the largest empty rectangle problem. *Parallel Processing Letters*, 3:79–85, 1993.

[91] M. Orlowski. A new algorithm for the largest empty rectangle problem. *Algorithmica*, 5:65–73, 1990.

[92] A. Park and K. Balasubramanian. Reducing communication costs for sorting on mesh-connected and linearly connected parallel computers. *Journal of Parallel and Distributed Computing*, 9:318–322, 1990.

[93] H. Park, H. J. Kim, and V. K. Prasanna. An $O(1)$ time optimal algorithm for multiplying matrices on reconfigurable mesh. *Information Processing Letters*, 47:109–113, 1993.

[94] B. Pradeep and S. R. Murthy. Parallel arithmetic expression evaluation on reconfigurable meshes. *Computer Language*, 20:267–277, 1994.

[95] T. H. K. Prasad and C. Pandurangan. New parallel algorithms for the maximum empty rectangle problem. In *Proceedings of the International Conference on Parallel Processing*, pp. 286–288, 1987.

[96] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, U.S.A., 1985.

[97] S. Rajasekaran. Mesh connected computers with fixed and reconfigurable buses: packet routing and sorting. *IEEE Transactions on Computers*, 45:529–539, 1996.

[98] K. Sado and Y. Igarashi. Some parallel sorts on a mesh-connected array and their time efficiency. *Journal of Parallel and Distributed Computing*, 3:398–410, 1986.

[99] R. Sasada. Production of processor array with reconfigurable bus systems simulator. Technical report, Department of Information and Computer Sciences, Osaka University, Japan, 1996. in Japanese.

[100] I. D. Scherson, S. Sen, and A. Shamir. Shear sort: A true two-dimensional sorting technique for VLSI networs. In *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 903–908, 1986.

[101] C. P. Schnorr and A. Shamir. An optimal sorting algorithm for mesh connected computers. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pp. 255–263, May 1986.

[102] A. Schuster. *Dynamic reconfiguring networks for parallel computers: algorithms and complexity bounds*. PhD thesis, Dept. of Computer Science, Hebrew University, Israel, 1991.

[103] H. Shi, G. X. Ritter, and J. N. Wilson. Simulations between two reconfigurable mesh models. *Information Processing Letters*, 55:137–142, 1995.

[104] D. B. Shu and J. G. Nash. The gated interconnection network for dynamic programming. In S. K. Tewsburg et al., editors, *Concurrent Computations*, pp. 645–658. Plenum, New York, 1988.

[105] T. Suel. Permutation routing and sorting on meshes with row and column buses. *Parallel Processing Letters*, 5:63–80, 1995.

[106] T. G. Szymanski and C. J. V. Wyk. Layout analysis and verification. In *Physical Design Automation of VLSI Systems*. Benjamin/Cummings, Menlo Park, California, 1988.

[107] C. D. Thompson and H. T. Kung. Sorting on a mesh-connected parallel computer. *Communications of ACM*, 20:263–271, 1977.

[108] G. Toussaint, editor. *Computational Geometry: Machine Intelligence and Pattern Recognition*, volume 2. Elsevier Science Publishers, North Holland, 1985.

[109] J. L. Trahan, A. G. Bourgeois, and R. Vaidyanathan. Tighter and broader complexity results for reconfigurable models. *Parallel Processing letters*, 8:271–282, 1998.

[110] J. L. Trahan, C.-m. Lu, and R. Vaidyanathan. Integer and floating point matrix-vector multiplication on the reconfigurable mesh. In *Proceedings of the 10th International Parallel Processing Symposium*, pp. 702–706, 1996.

[111] J. L. Trahan, Y. Pan, R. Vaidyanathan, and A. G. Bourgeois. Scalable basic algorithms on a linear array with a reconfigurable pipelined bus system. In *Proceedings of the 10th ISCA International Conference on Parallel and Distributed Computing Systems*, pp. 564–569, New Orleans, L.A., U.S.A, October 1997.

[112] J. L. Trahan, R. Vaidyanathan, and R. K. Thiruchelvan. On the power of segmenting and fusing buses. *Journal of Parallel and Distributed Computing*, 34:82–94, 1996.

[113] J. D. Ullman. *Computational Aspects of VLSI*. Computer Science Press, Rockville, Maryland, 1984.

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

[115] B. F. Wang. *Configurational computation: A new algorithm design strategy on processor arrays with reconfigurable bus systems*. PhD thesis, National Taiwan University, 1991.

[116] 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 Transactions on Parallel and Distributed Systems*, 1:500–507, 1990.

[117] 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*, 36:31–36, 1990.

[118] B.-F. Wang, G.-H. Chen, and F.-C. Lin. Constant time sorting on a processor array with a reconfigurable bus system. *Information Processing Letters*, 34:187–192, 1990.

[119] T. Watanabe, K. Nakano, and T. Hayashi. A visualizing tool for parallel algorithms on the reconfigurable mesh. *IPSJ Algorithm*, 61:31–38, 1998. in Japanese.

[120] C. C. Weems et al. The image understanding architecture. *International Journal of Computer Vision*, 2:251–282, 1989.

[121] D. Wood. An isothetic view of computational geometry. In G. Toussaint, editor, *Computational Geometry: Machine Intelligence and Pattern Recognition*, volume 2. Elsevier Science Publishers, North Holland, 1985.

[122] F. F. Yao. On finding the maximal elements in a set of plane vectors. Technical report, Comput. Sci. Dep. Rep., U. of Illinois at Urbana-Champaign, 1974.