The Relationships Between Several Parallel Computation Models

by

Yuanqiao Zhang

A thesis submitted to the

Department of Computer Science

in conformity with the requirements for

the degree of Master of Science

Bishop's University

Sherbrooke, Quebec, Canada

December 2008

Copyright © Yuanqiao Zhang, 2008

Abstract

The Parallel Random Access Machine (PRAM for short) is the most convenient and widely used model of parallel computation. Other, more complex models have also been studied. Examples include the Broadcast with Selective Reduction (or BSR) and the Reconfigurable Multiple Bus Machine (or RMBM). While the PRAM and the BSR are shared memory models, the RMBM accomplishes the communication between processors using buses. In this thesis we identify surprising relationships between these models: We show that several variants are equivalent with each other in a strong sense, but we also establish strict distinctions between the other variants. Some models are folklorically considered feasible and some others are not considered so. We find a delimitation that matches the folklore, but we also find important (and intriguing) equivalencies.

Acknowledgments

This thesis could not have been finished without the help and support of people who are gratefully acknowledged here.

At the first, I'm honored to express my deepest thanks to my supervisor Stefan D. Bruda; without his able step-by-step guidance I could not have worked out this thesis. He has offered me valuable ideas, suggestions and criticisms with his profound knowledge in forensic linguistics and rich research experience. He guided me into this field. His patience and kindness are greatly appreciated.

Beside, he is also my English teacher, teaching me how to express myself in both oral and written English. He also gave me much help besides study. I am very much obliged to his efforts of helping me complete the dissertation.

Also, I wish to extend my thanks to every member of the PART research group at Bishop's for their help on this study. This manuscript has been improved as a result of the comments of the members of my examination committee Layachi Bentabet, Ke Qiu, and Dimitri Vouliouris.

Last but not least, I would like to thank my wife Lili Liu for her support all the way from the very beginning of my postgraduate studies. I am thankful to all my family members for their thoughtfulness and encouragement.

Contents

List of Figures

Chapter 1

Introduction

Technology is developing so fast that pursuing faster and faster computations is instinctual. First, we make the processors fast. Then we use multiple processor to do a task in parallel.

More and more evidence shows that some limit on the CPU frequency is approaching. This justifies an earlier trend (which is becoming mainstream) to perform computations in parallel by multiple processors or equivalently multiple cores in a singe CPU.

What is parallel computation? At the deeper theoretical level it includes two main concepts: computational models and algorithms. The latter depends on the former, as different models implement algorithms differently. A model of computation is the definition of the set of allowable operations used in computation and their respective costs. Only assuming a certain model of computation it is possible to analyze the computational resources required, such as the execution time or memory space, or to discuss the limitations of algorithms or computers. An algorithm is a sequence of instructions, often used for calculations and data processing. It is formally a type of effective

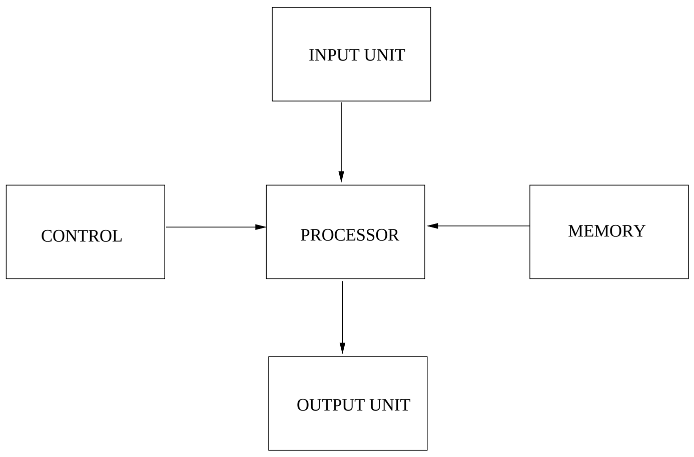

Figure 1.1: Sequential computation

method in which a list of well-defined instructions for completing a task will, when given an initial state, proceed through a well-defined series of successive states, eventually terminating in an end-state.

People at Princeton University proposed in the 1940s a design that ushered in the modern computer era [1]. This architecture continues to work and is the earliest model for sequential computation Figure 1.1 shows the architecture of sequential computation. The computation unit (or processor) is the core, and other devices (I/O device,memory and other control units) communicate with the processor to do a job. The general structure of a parallel computation model is shown in Figure 1.2. This model features more computation units (or processors). The processors on such a model (or machine) can communicate with each other by shared memory or buses.

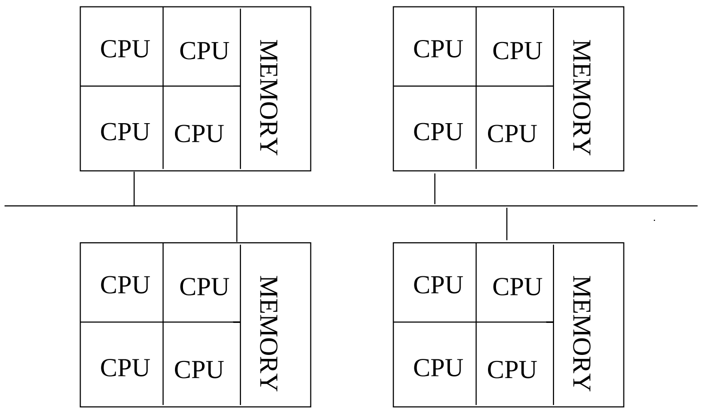

Figure 1.2: Parallel computation

1.1 History

In 1964, Daniel Slotnick proposes building a massively-parallel machine for the Lawrence Livermore National Laboratory [25, 27]. Slotnick's design evolves into the ILLIAC-IV. The machine is built at the University of Illinois, with Burroughs and Texas Instruments as primary subcontractors. In 1969, Honeywell delivers the first Multics system (symmetric multiprocessor with up to 8 processors). In 1977, the C.mmp multiprocessor is completed at Carnegie-Mellon University. The machine contains 16 PDP-11 minicomputers connected by a crossbar to shared memories, and supports most of the early work on languages and operating systems for parallel machines.

In 1982, Steve Chen's group at Cray Research produces the first X-MP, containing two pipelined processors (compatible with the CRAY-I) and shared memory.

In 1991, Sun begins shipping the SPARC server 600 (also called Sun-4/600) series machines (shared-memory multiprocessors containing up to 4 SPARC CPUs each).

In 2000, Blue Horizon, built by IBM, is located at the San Diego Supercomputer Center and first come into full production operation March 1st, 2000. The hardware consists of 1152 processors.

Each processor runs at 222 MHz and they are grouped into nodes of 8 processors per node. Each node is able to access 4 GB of RAM, this gives a total memory of 576Gb.

In 2003, TeraGrid is the world's largest, fastest, distributed infrastructure for open scientific research. It includes 20 tera flops of computing power distributed at five sites, facilities capable of managing and storing nearly 1 peta byte of data, high-resolution visualization environments, and toolkits for grid computing. These components will be tightly integrated and connected through a network that will operate at 40 gigabits per second—the fastest research network on the planet.

Now [26] the world's first hybrid supercomputer has broken through the ”petaflop barrier” of 1,000 trillion operations per second, according to the U.S. Department of Energy. Codenamed ”Roadrunner,” the machine is designed by IBM and uses Cell Broadband Engine chips originally developed for video game platforms.

1.2 Applications

The main shortcoming of single-CPU systems comes from the inherent lack of memory and computational resources; then parallel computation provides a solution for this problem. Parallel computation provides higher performance than single-CPU, as parallel computation can solve same sized problems faster or can treat larger problems that require more processing power and/or more memory. Moreover, it is possible to easily upgrade a parallel computer by adding more CPUs and memory.

Parallel computing is used for many applications, and it would be futile to try to list them all. Some of the major market sectors are:

- Numerically intensive simulations, including: computational fluid dynamics (CFD), computational electromagnetics (CEM), oil reservoir simulations, combustion modelling, molecular dynamics modelling, quantum chromodynamics, quantum chemistry, etc;
- Graphical visualization including image processing;
- Database operations and information systems including: client and inventory database management, data mining, online transaction processing, management information systems, geographic information systems, seismic data processing, etc;
- Real-time systems and control applications including: hardware and robotics control, speech processing, pattern recognition, etc.

1.3 Future

According to Moore's Law, the number of transistors on a microprocessor would double approximately every eighteen months, which is to say that for the next two decades computer chips would double in speed every eighteen months. This makes it possible for parallel computers to increase performance at this rate as well. One can guess at the capabilities and possibilities of parallel computers in the future. The following are some of what is emerging in the parallel computing field: computers similar to the TeraGrid (the world's fastest network of parallel computers, mentioned earier), and grid support software that will provide a relatively "seamless" interface among geographically separated computers sharing data and computations.

Other developments include:

- Return of the vector machines—one of the first technology for parallel systems is enjoying a "renaissance" in the US.
- Computers that will take advantage of the inherent parallelism to higher levels:
    1. Processor-In-Memory (PIM) designed to compensate for the disparity between memory access times and computation times.
    2. Quantum Computers—will take advantage of parallelism inherent in quantum mechanical systems.

1.4 The Issue

The notion of one model being more powerful than another is rather intuitive. However, although not always explicit, we always have in mind real-time computations, so in this thesis we are using a strong notion of "more powerful": We say that model A is (not necessarily strictly) more powerful than model B only if $t(n)$ computational steps of model B using polynomial resources can be simulated in $O(t(n))$ steps of model A using polynomial resources.

The shared memory and bus models are the major parallel computation models (a third model, the interconnection network is a particular case of the bus model). They have been researched by the industry and academia alike. Based on such models, many relevant products and demos showed up. They have been used widely and with significant success. In shared memory models (such as the PRAM), any computation unit can access any memory location. Similarly, in bus models any computation unit can access any bus, which thus replaces the memory for inter-processor communication. Shared memory models have emerged as a good theoretical model, while bus models are considered closer to practice. A natural question that has spawned the research on parallel computation is whether these two models are similar in power. To make matters more complicated, some shared memory models support concurrent write, but only produce a flag if collision happens; some other models support priority concurrent write, and some others can combine concurrent writes. Combining concurrent write shared memory model appear intuitively to be the most powerful, though this has not been proven to date.

More specifically, the Priority concurrent-read concurrent-write parallel random access machine[1] (CRCW PRAM) is the most convenient and powerful model of parallel computation and so it is used extensively in analyzing parallel solutions to various problems. Lower bounds on the PRAM are in particular very strong. The Priority CRCW PRAM is sometimes considered [19] to be at the upper level of feasible parallel models. The broadcast with selective reduction (BSR) on the other hand is at present the most powerful model of parallel computation, with the Combining CRCW PRAM falling somewhere in between. By logical extension of the Priority CRCW PRAM being at the upper end of the feasibility chain [19], the Combining CRCW PRAM and the BSR should not be considered feasible; however, efficient implementations for them have been proposed [1, 2]. Finally, models with directed reconfigurable buses (namely the directed reconfigurable multiple bus

[1]Reading from shared resources can happen concurrently or not, so exclussive-read (ER) and the concurrent-read (CR) variants exist for most models. Similarly, a model can or cannot write concurrently into shared resources, so the EW and CW variants exist. Combining these features, we distinguish between EREW, CREW , CRCW variants of a model. Writing concurrently into a shared resource can lead to conflicts which are resolved using a conflict resolution rule such as Common (concurrent writing is allowed only when all the processors write the same value), Collision (a special collision marker ends up written instead of any processor provided data), Priority (only the writing of the highest priority processors succeeds), and Combining (a combination is written in the shared location). These notions presented in mode details in Section 2.3.

machine or DRMBM and the directed reconfigurable network of DRN) are not only feasible, but they have also been implemented in VLSI circuits [22]; still, they are considered complex and of restricted feasibility as a general model of parallel computation.

It is widely believed (though to our knowledge not proven) that Priority CRCW PRAM is strictly less powerful than the Combining CRCW PRAM, which is in turn strictly less powerful then the BSR. The models with directed reconfigurable buses have been shown to be at least as powerful as the Priority CRCW PRAM [23] but are otherwise not placed anywhere in this hierarchy.

In all, mirroring the beliefs and formal results summarized above, one can identify two "categories" of models of parallel computation: we thus call the Priority CRCW PRAM and the models below it in terms of computational power *lightweight* models, with the *heavyweight* models represented by the Combining CRCW PRAM, the BSR, and the models with directed reconfigurable buses.

In this thesis we attempt to clarify the relationship between all these various models, with particular focus on the heavyweight class. We find that all the models in the heavyweight class are actually equivalent with each other, and that the choice of two classes (heavyweight and lightweight) is justified. We also provide results regarding the bus models, bringing them yet closer to the practical realm.

1.5 The Hierarchy of Parallel Models

The work presented in this thesis can be grouped along three main ideas:

- The relationship between PRAM varieties (Collision, Priority, Combining).

- The relationship between PRAM and the BSR(a Combining PRAM with a Broadcast instruction added on top).
- The relationship between directed reconfigurable buses and the BSR.

Recall that we called the computational models below Priority CRCW PRAM lightweight, while the BSR, Combining CRCW PRAM and directed reconfigurable buses get to be heavyweight models.

Previous work on parallel real-time computation [9] has produced a number of incidental results regarding these models. Specifically, a tight characterization of constant time computations on directed reconfigurable multiple bus machines (DRMBM) was offered: DRMBM and directed reconfigurable networks (DRN) with constant running time have been found to have the same computational power, which in turn is the same power as nondeterministic logarithmic space-bounded Turing machines. In addition, it was shown that in the case of constant time DRMBM computations there is no need for such powerful write conflict resolution rules as Priority or Combining as they do not add computational power over the easily implementable Collision rule, that an unitary bus width is enough (i.e., a simple wire as bus will do for all constant time DRMBM computations), and that segmenting buses does not add computational power over fusing buses.

Such properties (Collision being the most powerful resolution rule and unitary bus width being sufficient) turn out to hold for the other model with directed reconfigurable buses, namely the DRN [7]. Finally, whether the Conflict resolution rule is generally (i.e., not only for constant time computations) universal on DRMBM and DRN was also shown true [7].

We use the results mentioned in the above paragraph as both tool and motivation to offer an analysis of computational power for various models of parallel computation. As expected, we find that the class of heavyweight models is strictly more powerful than the class of lightweight models. Surprisingly, we also find that all the heavyweight models are however equivalent with each other, despite the perceived high computational power of the BSR.

Since one class of models we consider extensively is models with reconfigurable buses, we also linger a bit more on the matter. We clarify the notion of write conflict on these models, and we show that either exclusive-write or Combining concurrent-write is universal on all these models (be they directed or undirected). We also note that all the reconfigurable models can be laid out as meshes. Both these results are practically significant in the domain of VLSI circuits.

1.6 Thesis Summary

The remainder of this thesis is organized as follows: Chapter 2 introduces the concepts used throughout the thesis. We first review briefly (Section 2.2) the necessary notions from the complexity theoretical realm, including the notion of complexity classes, Turing machines, and the Graph Accessibility Problem (which is central to our results). We then introduce the parallel computation models that we will use, namely the PRAM (with all its variants including the BSR, in Section 2.3), the models with reconfigurable buses (Section 2.4), and the Boolean circuit (Section 2.5).

We mentioned earlier that our work is based on and is also motivated bt previous results about models with reconfigurable buses. Chapter 3 presents these results, namely the characterization of constant time computations on the RMBM (Section 3.1) and the RMBM (Section 3.2), as well as the universality of Collision on directed reconfigurable buses (Section 3.3).

Chapter 4 presents the main contribution of this thesis. The equivalence of all the heavyweight models and the difference between the heavyweight and the lightweight classes are the subject of Section 4.1, while consequences (including real-time considerations) are presented in Section 4.2.

Further consequences of our results are addressed in Chapter 5: we address here the different definitions of Collision on reconfigurable buses and we show how our previous results adjust depending on the definition used (Section 5.1), and we also focus on the simulation of reconfigurable buses on their simplest variant, the mesh (Section 6.2).

Chapter 2

Models And Computational Complexity

We introduce here the parallel computation models of interest in our work. We give the main features of these models, such as the type of resources used (processors, memory, buses, switches), the size of these resources, and the way they can talk to each other effectively.

We do not consider here some models, as they are too weak to simulate the others: Some models with limited interconnection schemes can be too weak to simulate other models [11]. For example, in the tree connected parallel machine, although any two processors can communicate via short paths, there is a bottleneck at the root that limits the bandwidth of the communication between processors. The mesh connected parallel machine can only communicate directly with its neighbors, and this results in an average path of length $\sqrt{n}$ for n processors.

Some other models of parallel computation are too powerful to be simulated by the more common models. This includes machines that can generate exponentially long values or activate exponential numbers of processors in polylogarithmic time.

Before we start presenting the models as promised, we present briefly some complexity theoretical notions used throughout the thesis.

Results proved elsewhere are introduced henceforth as Propositions, whereas results proved in this work are introduced as Theorems. Intermediate results are all Lemmata.

2.1 GAP and NL

GAP$_{i,j}$ denotes the following problem: Given a directed graph $G = (V, E)$, $V = \{1, 2, ..., n\}$ (expressed e.g., by the (Boolean) adjacency matrix I), determine whether vertex j is accessible from vertex i. PARITY$_n$ denotes the following problem: given n integer data $x_1, x_2, \ldots, x_n$, compute the function PARITY$_n(x_1, x_2, \ldots, x_n) = (\sum_{i=1}^{n} x_i) \bmod 2$.

We denote by L [NL] the set of languages that are accepted by deterministic [nondeterministic] Turing machines that use at most $O(\log n)$ space (not counting the input tape) on any input of length n [20].

For some language $L \in \mathsf{NL}$ there exists a nondeterministic Turing machine $M = (K, \Sigma, \delta, s_0)$ that accepts L and uses $O(\log n)$ working space. K is the set of states, Σ is the tape alphabet (we consider without loss of generality that $\Sigma = \{0, 1\}$), δ is the transition relation, and s_0 is the initial state. M accepts an input string x iff M halts on x. A configuration of M working on input x is defined as a tuple (s, i, w, j), where s is the state, i and j are the positions of the heads on input and working tape, respectively, and w is the content of the working tape. There are $poly(n)$ possible configurations of M. For two configurations v_1 and v_2, we write $v_1 \vdash v_2$ iff v_2 can be obtained by applying δ exactly once on v_1 [20].

The set of possible configurations of M working on x forms a directed graph $G(M, x) = (V, E)$ as follows: V contains one vertex for each and every possible configuration of M working on x, and $(v_1, v_2) \in E$ iff $v_1 \vdash v_2$. It is clear that $x \in L$ iff some configuration (h, i_h, w_h, j_h) is accessible in $G(M, x)$ from the initial configuration (s_0, i_0, w_0, j_0). For any language $L \in \mathsf{NL}$ and for any x, determining whether $x \in L$ can be reduced to the problem of computing GAP$_{1,|V|}$ for $G(M, x) = (V, E)$, where M is some NL Turing machine deciding L.

The class of problems in NL with the addition of (any kind of) real-time constraints is denoted by NL/rt [9]. We denote by rt-PROC$^M(f)$ the class of those problems solvable in real time by the parallel model of computation M that uses $f(n)$ processors (and also $f(n)$ buses if applicable) for any input of size n [8]. The following strongly supported conjecture is then established.

Claim 1 [9] rt-PROC$^{\text{CRCW F-DRMBM}}(poly(n)) = \mathsf{NL}/rt$.

2.2 Other Complexity Classes

Typically, a complexity class is defined by

- A model of computation.
- A resource (or collection of resources).
- A function known as the complexity bound for each resource.

The models used to define complexity classes fall into two main categories:

- Machine-based models.

- Circuit-based models.

Turing machines (TMs) and random-access machines (RAMs) are the two principal families of machine models. We describe circuit-based models later, in Section 2.5. Other kinds of (Turing) machines includes deterministic, nondeterministic, alternating, and oracle machines. We emphasize the fundamental resources of time and space for deterministic and non-deterministic Turing machines. We concentrate on resource bounds between logarithmic and exponential, because those bounds have proved to be the most useful for understanding problems that arise in practice.

Given functions $t(n)$ and $s(n)$: DTIME$[t(n)]$ is the class of languages decided by deterministic Turing machines with $t(n)$ running time. NTIME$[t(n)]$ is the class of languages decided by nondeterministic Turing machines with $t(n)$ running time. DSPACE$[s(n)]$ is the class of languages decided by deterministic Turing machines using $s(n)$ tape space. NSPACE$[s(n)]$ is the class of languages decided by nondeterministic Turing machines using $s(n)$ tape space. We sometimes abbreviate DTIME$[t(n)]$ to DTIME$[t]$ (and so on) when t is understood to be a function, and when no reference is made to the input length n.

The following are the canonical complexity classes [4]:

- $\mathsf{L} =$ DSPACE$[logn]$ (deterministic log space)
- $\mathsf{NL} =$ NSPACE$[logn]$ (nondeterministic log space)

  GAP $\in$ NL-Complete [11]
- $\mathsf{P} =$ DTIME$[n^{O(1)}]$ (polynomial time)
- $\mathsf{NP} =$ NTIME$[n^{O(1)}]$ (nondeterministic polynomial time)

- PSPACE $=$ DSPACE$[n^{O(1)}]$ (polynomial space)
- EXP $=$ DTIME$[2^{O(n)}]$ (deterministic exponential time)
- NEXP $=$ NTIME$[2^{O(n)}]$ (nondeterministic exponential time)
- EXPSPACE $=$ DSPACE$[2^{n^{O(1)}}]$ (exponential space)

The space classes PSPACE and EXPSPACE are defined in terms of the DSPACE complexity measure. By Savitch's Theorem, the NSPACE measure with polynomial bounds also yields PSPACE, and with $2^{n^{O(1)}}$ bounds yields EXPSPACE.

2.3 The PRAM and the BSR

The PRAM [1, 19] is the most convenient and thus most popular model of parallel computation. A PRAM consists of a number of processors that share a common random-access memory. The processors execute the instructions of a parallel algorithm synchronously. as shown in Figure 2.1. The shared memory stores intermediate data and results, and also serves as communication medium for the processors. The model is further specified by defining the memory access mode; we thus obtain exclusive-read exclusive-write (EREW), concurrent-read exclusive-write (CREW) and concurrent-read concurrent-write (CRCW) PRAM. While reading concurrently from the shared memory is defined straightforwardly, writing concurrently into the shared memory requires the introduction of a conflict resolution rule (for the case in which two or more processors write into the same memory location). Four such conflict resolution rules are in use: Common (the processors writing simultaneously in the same memory location must write the same value or else the algorithm fails),

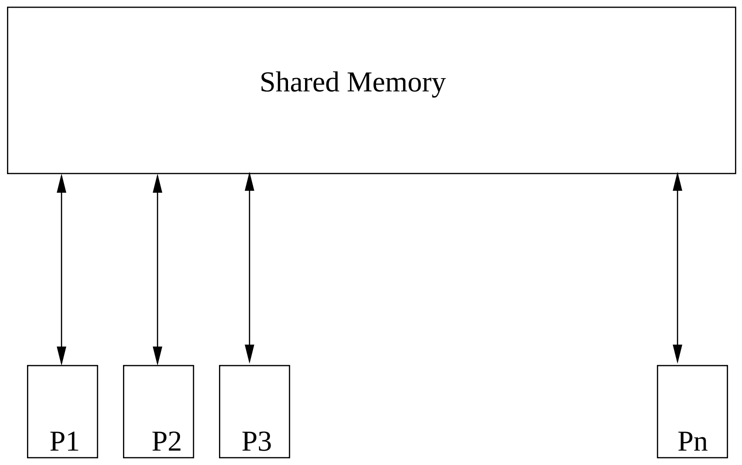

Figure 2.1: The PRAM architecture

Collision (multiple processors writing into the same memory location garble the result so that a special "collision" marker ends up written at that location instead of any processor-provided data), Priority (processors are statically numbered and the memory location receives the value written by the lowest numbered processor), and Combining (where a binary, associative reduction operation is performed on all the values sent by all the processors to the same memory location and the result is stored in that memory location).

Given the obvious decreased computational power as well as the straightforward implementation of concurrent-read machines, we will not consider exclusive-read variants. For similar reasons, exclusive-write machines will receive a spotty consideration, if any.

The BSR model [1, 2, 3] is an extension of the Combining CRCW PRAM. All the read and write operations of the CRCW PRAM can also be performed by the BSR. In addition, all the BSR processors can write simultaneously into all the memory locations (the Broadcast instruction, illustrated graphically in Figure 2.2). Every Broadcast instruction consists of three steps: In the *broadcasting step*, all the n participating processors produce a datum d_i and a tag g_i, $1 \leq i \leq n$, destined to

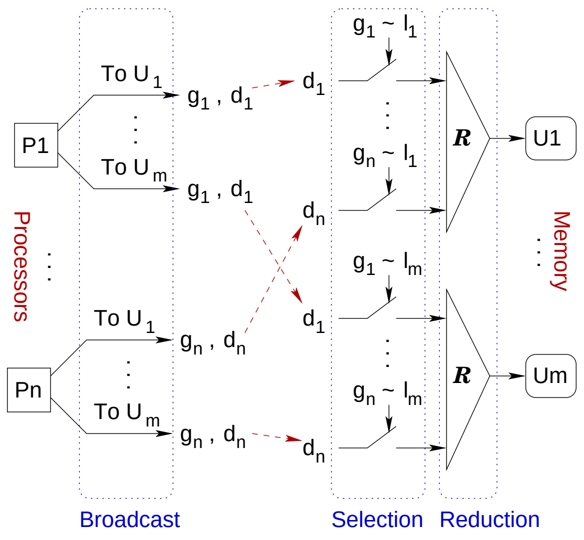

Figure 2.2: The BSR model

all the m memory locations. In the *selection step* each of the m memory locations uses a limit l_j, $1 \leq j \leq m$ and a selection rule $\sigma \in \{<, \leq, =, \geq, >, \neq\}$ to test the received data; the datum d_i is selected for the next step iff $g_i \, \sigma \, l_j$. Finally, the *reduction step* combines all the data d_i destined for memory location j, $1 \leq j \leq m$ and selected in the previous step using a binary, associative operator $\mathcal{R}$, and then writes the result into memory location j. The Broadcast instruction is performed simultaneously for all the processors and all the memory locations.

Typically, the reduction operator $\mathcal{R}$ of the BSR as well as the Combining operator of the Combining CRCW PRAM can be any of the following operations: Σ (sum), Π (product), $\bigwedge$ (logical conjunction) $\bigvee$ (logical disjunction), $\bigoplus$ (logical exclusive disjunction), $\max$ (maximum), and $\min$ (minimum).

We denote by X CRCW PRAM$(p(n), t(n))$ the class of problems of size n that are solvable in time $t(n)$ on a CRCW PRAM that uses $p(n)$ processors and X as collision resolution rule, $X \in \{\text{Common}, \text{Collision}, \text{Priority}, \text{Combining}\}$. Similarly, we denote by BSR$(p(n),\ t(n))$ the class of problems of size n that are solvable in time $t(n)$ on a BSR that uses $p(n)$ processors.

The notion of uniform families of PRAM or BSR machines exists [14] and is similar to the notion of uniformity for models with reconfigurable buses (to be introduced below). However, we do not need such a notion in this thesis. Still, we assume as customary that one location in the shared memory has $\log n$ size for an input of size n, and that the number of memory locations are upper bounded by a polynomial in the number of processors used. Again as usual, we assume that every BSR or PRAM processor has a constant number of internal registers, each of size $\log n$. Finally, we assume that every PRAM processor knows its number as well as the total number of processors in the system.

The following facts will be used later:

Proposition 2.3.1 [10] PARITY$_n \notin$ Priority CRCW PRAM$(poly(n), O(1))$.

Proposition 2.3.2 [12] *The reflexive and transitive closure of a graph with n vertices (and thus* GAP$_{i,j}$ *on the given graph) are in* BSR$(O(n^2), O(1))$.

2.4 Models with Reconfigurable Buses

2.4.1 The Reconfigurable Multiple Bus Machine (RMBM)

An RMBM [21] consists of a set of p *processors* and b *buses*. Figure 2.3 shows the structure of an RMBM. For each processor i and bus b there exists a *switch* controlled by processor i. The details of the switches are shown in Figure 2.4. Using these switches, a processor has access to the buses by being able to read or write from/to any bus. A processor may be able to *segment* a bus (open the

$S_{i,j,1}$ or $S_{i,j,0}$), obtaining thus two independent, shorter buses, and it is allowed to *fuse* any number of buses together by using a *fuse line* perpendicular to and intersecting all the buses. DRMBM, the *directed* variant of RMBM, is identical to the undirected model, except for the definition of fuse lines: Each processor features two fuse lines (*down* and *up*). Each of these fuse lines can be electrically connected to any bus. Assume that, at some given moment, buses i_1, i_2, ..., i_k are all connected to the down [up] fuse line of some processor. Then, a signal placed on bus i_j is transmitted in one time unit to all the buses i_l such that $i_l \geq i_j$ [$i_l \leq i_j$]. If some RMBM [DRMBM] is not allowed to segment buses, then this restricted variant is denoted by F-RMBM [F-DRMBM] (for "fusing" RMBM/DRMBM). The *bus width* of some RMBM (DRMBM, etc.) denotes the maximum size of a word that may be placed (and read) on (from) any bus in one computational step.

For CRCW RMBM, the most realistic conflict resolution rule is Collision , where two values simultaneously written on a bus result in the placement of a special, "collision" value on that bus. We consider for completeness other conflict resolution rules such as Common, Priority, and Combining. However, we find that all of these rules are in fact equivalent to the seemingly less powerful Collision rule (see Proposition 3.1.1(3)). We restrict only the Combining mode, requiring that the combining operation be associative and computable in nondeterministic linear space.

An RMBM (DRMBM, F-DRMBM, etc.) *family* $R = (R_n)_{n\geq 1}$ is a set containing one RMBM (DRMBM, F-DRMBM, etc.) construction for each $n > 0$. A family R solves a problem P if, for any n, R_n solves all inputs for P of size n. We say that some RMBM family R is a *uniform* RMBM *family* if there exists a Turing machine M that, given n, produces the description of R_n using $O(\log(p(n)b(n)))$ cells on its working tape. We henceforth drop the "uniform" qualifier, with the understanding that any RMBM family described in this paper is uniform. Assume that

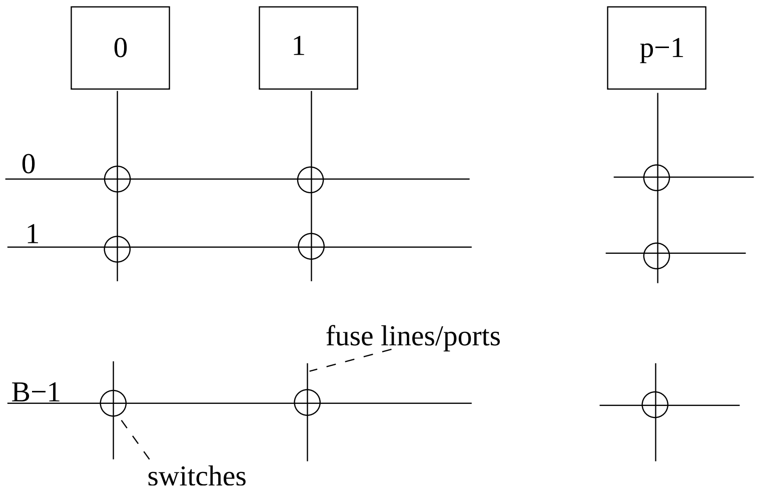

Figure 2.3: The RMBM structure

some family $R = (R_n)$ solves a problem P, and that each R_n, $n > 0$, uses $p(n)$ processors, $b(n)$ buses, X as write conflict resolution rule, and has a running time $t(n)$. We say then that $P \in X,$RMBM$(p(n), b(n), t(n))$ (or $P \in X$ F-DRMBM$(p(n), b(n), t(n))$, etc.), and that R has *size complexity* $p(n)b(n)$ and *time complexity* $t(n)$. Whenever we state a property that holds for any conflict resolution rule we drop the X (thus writing $P \in$ RMBM$(p(n), b(n), t(n))$, etc.).

It should be noted that a directed RMBM can simulate a undirected RMBM by simply keeping all the up and down fuse lines synchronized with each other. That is, $X\,Y\,Z$ RMBM$(x(n), y(n),$ $z(n))$ $\subseteq X\,Y\,Z$DRMBM$(x(n), y(n), z(n))$ for any $x, y, z : \mathbb{N} \rightarrow \mathbb{N}$, $X \in \{$Common, Collision, Priority, Combining$\}$, $Y \in \{$CRCW, CREW$\}$, and $Z \in \{$F-$, \varepsilon\}$.

2.4.2 The Reconfigurable Network (RN)

An RN [5] is a network of processors that can be represented as a connected graph whose vertices are the processors and whose edges represent fixed connections between processors. Each edge incident

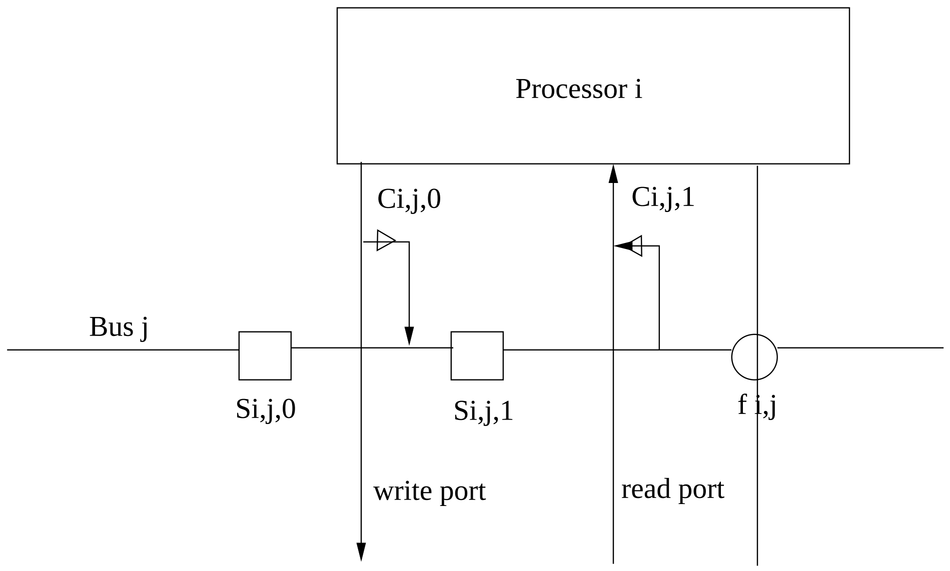

Figure 2.4: RMBM switches

to a processor corresponds to a (bidirectional) port of the processor. A processor can internally partition its ports such that all the ports in the same block of that partition are electrically connected (or fused) together. A sample RN structure is shown in Figure 2.5. Two or more edges that are connected together by a processor that fuses some of its ports form a bus which connects ports of various processors together. CREW, Common CRCW, Collision CRCW, etc. are defined as for the the RMBM model. The *directed* RN (DRN for short) is similar to the general RN, except that the edges are directed. The concept of (uniform) RN family is identical to the concept of RMBM family. For some write conflict resolution rule X, the class X RN$(p(n), t(n))$ [X DRN$(p(n), t(n))$] is the set of problems solvable by RN [DRN] uniform families with $p(n)$ processors ($p(n)$ is also called the *size complexity*) and $t(n)$ running time using the conflict resolution rule X. As in the RMBM case, we drop X when the stated property refers to any conflict resolution rule.

Once more similar to the RMBM case, we note that given a undirected RN, we can create an

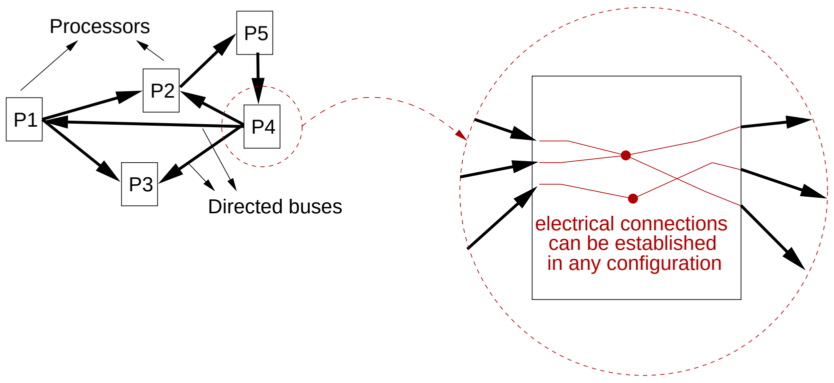
Processors
P1
P2
P3
P4
P5
Directed buses
electrical connections can be established in any configuration

Figure 2.5: The RN structure

equivalent DRN by simply replacing every undirected bus with a pair of directed buses. That is, $X\,Y\,\mathrm{RN}(x(n), y(n)) \subseteq X\,Y\,\mathrm{DRN}(x(n), y(n))$ for any $x, y : \mathbb{N} \rightarrow \mathbb{N}$, $X \in \{\mathrm{Common}, \mathrm{Collision}, \mathrm{Priority}, \mathrm{Combining}\}$, and $Y \in \{\mathrm{CRCW}, \mathrm{CREW}\}$.

2.4.3 The Reconfigurable Mesh

A two-dimensional reconfigurable mesh is a special case of RN. This special case is of particular significance for implementation reasons. An $R \times C$ 2-dimensional reconfigurable mesh (or simply an $R \times C$ R-Mesh) consists of $R \times C$ processors arranged in R rows and C columns as an $R \times C$ two-dimensional mesh. Figure 2.6 shows the structure of a 3×5 R-Mesh. We will usually number rows [columns] 0,1,..., $R-1$ [0, 1, ..., $C-1$] with row 0 at the top [columns 0 at left]. Each processor has four ports, N, S, E, and W, through which it connects to processors (if any) to its North, South, East, and West. Besides these external connections, each processor can also establish internal connections among its ports that correspond to partitions of the set N, S, E, W of ports. There are 15 possible port partitions along with the corresponding internal connections. A processor can change its internal connections at each step. The external and internal connections together connect processor ports by buses. The buses formed by a given set of port partitions of processors of the R-Mesh comprise a bus configuration of the R-Mesh. Figure 2.7 shows a bus configuration of a 3×5 R-Mesh with one of the buses shown in bold. We say that a port (or processor) through which a bus passes is incident on that bus or that the bus traverses the port (or processor). Any port that is incident on a bus can read from and write to that bus.

Similarly, an $R \times C$ 2-dimensional DR-MESH is an $R \times C$ reconfigurable mesh, except that the bus communication is directed: the N, S, E, and W ports of the processors are directed, and the inter-port connection in a processor is also directed.

In passing, note that the definition of a two-dimensional (directed or undirected) reconfigurable mesh extends naturally to higher dimension.

2.5 The Boolean Circuit

Although the PRAM model is a natural parallel extension of the RAM model, it is not obvious that the model is actually reasonable. Many experts had questions about the PRAM. For example: does the PRAM model correspond, in capability and cost, to a physically implementable device? Is it fair to allow unbounded numbers of processors and memory cells? Is it sufficient to simply have a unit charge for the basic operations? Is it possible to have unbounded numbers of processors accessing any portion of shared memory for only unit cost? Is synchronous execution of one instruction on each processor in unit time realistic? To expose issues like these, it is useful to have a more primitive model that, although being less convenient to program, is more closely related to

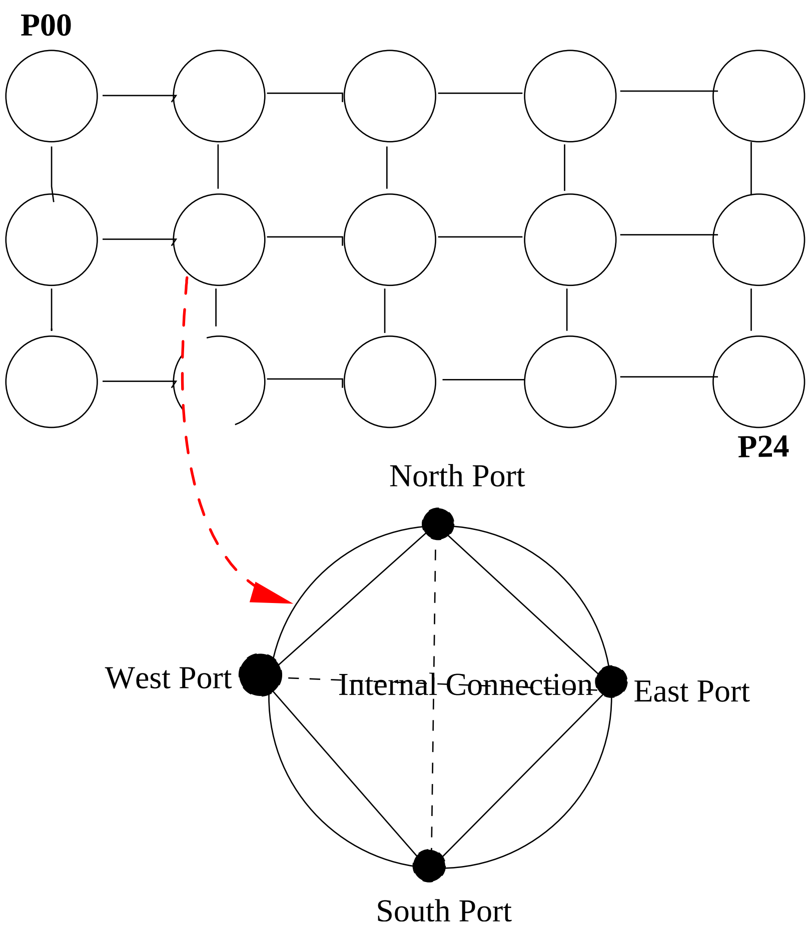

Figure 2.6: The MESH model

the realities of physical implementation. Such a model is the Boolean circuit. The model is simple to describe and mathematically easy to analyze. Circuits are basic technology, consisting of very simple logical gates connected by bit-carrying wires. They have no memory and no notion of state. Circuits avoid almost all issues of machine organization and instruction repertoire. Their computational components correspond directly with devices that we can actually manufacture. The circuit model is still an idealization of real electronic computing devices. It ignores a host of important practical considerations such as circuit area, volume, pin limitations, power dissipation, packaging,

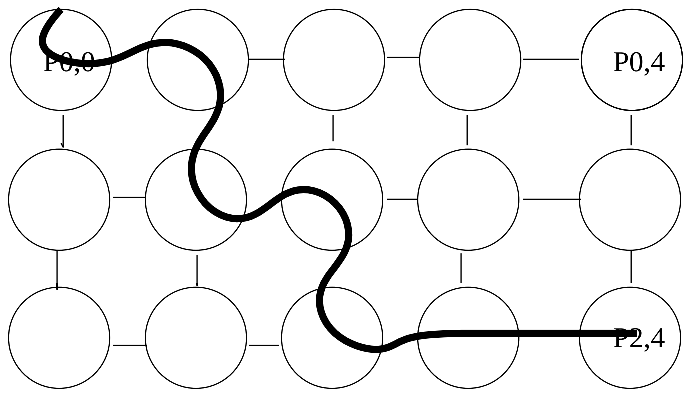

Figure 2.7: A sample reconfigurable MESH

and signal propagation delay. Such issues are addressed more accurately by more complex VLSI models, but for many purposes the Boolean circuit model seems to provide an excellent compromise between simplicity and realism. For example, one feature of PRAM models that has been widely criticized as unrealistic and unimplementable is the assumption of unit time access to shared memory. Consideration of (bounded fan-in) circuit models exposes this issue immediately, since a simple fan-in argument provides a lower bound of processors (log p) on the time to combine bits from p sources, say by logical or, a trivial problem on a unit cost CRCW PRAM. A circuit is simply a formal model of a combinational logic circuit. It is an acyclic directed graph in which the edges carry unidirectional logical signals and the vertices compute elementary logical functions. The entire graph computes a Boolean function from the inputs to the outputs in a natural way. Let $B_k = \{f \mid f : \{0,1\}^k \rightarrow \{0,1\}\}$ denote the set of all k-ary Boolean functions. We refer informally to such functions by strings “1,” “0,” “$\wedge$,” “$\vee$,” among others.

Since the 1970s, research on circuit complexity has focused on problems that can be solved quickly in parallel, with feasible amounts of hardware—circuit families of polynomial size and depth as small as possible. Note, however, that the meaning of the phrase ”as small as possible” depends on the technology used. With unbounded fan-in gates, depth O(1) is sufficient to carry out interesting computation, whereas with fan-in two gates, depth less than log(n) is impossible if the value at the output gate depends on all of the input bits. In any technology, however, a circuit with depth nearly logarithmic is considered to be very fast. This observation motivates the following definitions.

Boolean circuit α is a labeled finite directed acyclic graph. Each vertex v has a type $(v) \in I \cup \left(\bigcup_{i>0} B_i\right)$. A vertex v with $(v) = I$ has in-degree 0 and is called an input. The inputs of α are given by a tuple $(x_1, \ldots, x_n)$ of distinct vertices. A vertex v with out-degree 0 is called an output. The outputs of α are given by a tuple $(y_1, \ldots, y_m)$ of distinct vertices. A vertex v with $(v) \in B_i$ must have in-degree i and is called a gate [11].

A Boolean circuit α with inputs $(x_1, \ldots, x_n)$ and outputs $(y_1, \ldots, y_m)$ computes a function $f : \{0,1\}^n \to \{0,1\}^m$ in the following way: input x_i is assigned a value $\nu(x_i)$ from $\{0,1\}$ representing the i-th bit of the argument to the function. Every other vertex v is assigned the unique value $[v] \in \{0,1\}$ obtained by applying (v) to the value(s) of the vertices incoming to v. The value of the function is the tuple $(\nu(y_1), \ldots, \nu(y_m))$ with output y_j contributing the j-th bit of the output. When the logical function associated with a gate is not symmetric, the order of the incoming edges into the gate is important [11].

Like we can define sequential complexity classes using Turing machine, we can use Boolean Circuits to define parallel complexity classes. Parallel complexity measures (for a given circuit C) are:

- Depth = parallel time.

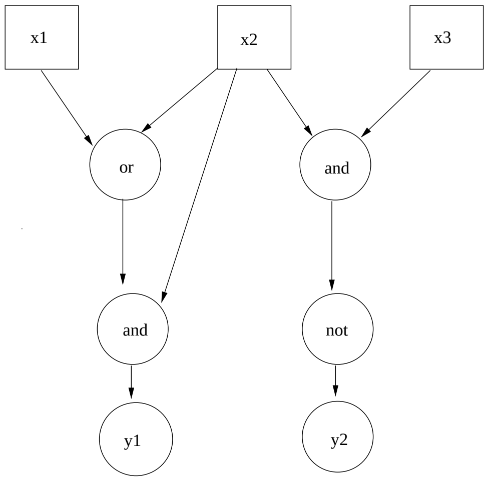

Figure 2.8: A sample Boolean circuit

- Size (number of gates) = parallel work.

Let $C = (C_i)_{i \geq 0}$ be a family of Boolean circuits, and let $f(n)$ and $g(n)$ be functions from integers to integers. We say that parallel time of C is at most $f(n)$ if for all n the depth of C_n is at most $f(n)$. Analogously, the size $g(n)$ is equivalent with parallel work.

We define $PT/WK(f(n), g(n))$ to be the class of all languages $L \subseteq \{0,1\}^*$ such that there is a family of circuits C deciding L with $O(f(n))$ parallel time and $O(g(n))$ parallel work. For example,

- MATRIX MULTIPLICATION $\in PT/WK(\log n, n^3/\log n)$ [11]

- REACHABILITY $\in PT/WK(\log^2 n, n^3 \log n)$ [11]

Proposition 2.5.1 [11] *If $L \in \{0,1\}^*$ is in $PT/WK(f(n), g(n))$, then there is a PRAM that computes the corresponding function FL mapping $\{0,1\}^*$ to $\{0,1\}$ in parallel time $O(f(n))$ using $O(g(n)/f(n))$ processors.*

That is, circuits can be efficiently simulated by the PRAM. One processor will simulate approximately $g(n)/f(n)$ gates. Simulating a gate is straightforward, we only need to make sure that the input of the gate is already computed, i.e., we need to compute the time for each gate when its input data will be ready for sure and then execute its program at that time.

The converse is also true: circuits can simulate the PRAM quite efficiently.

Proposition 2.5.2 [11] *If a function F can be computed by a PRAM family in time $f(n)$ with $g(n)$ processors, then there is a family C of circuits of depth $O(f(n)(f(n) + logn))$ and size $O(g(n)f(n)(n_k f(n) + g(n)))$ which computes the binary representation of F, where n_k is the time bound of the log-space Turing machine which outputs the n-th PRAM in the family.*

A configuration of the PRAM—including all program counters and all registers —is of size at most $f(n)g(n)$. The next configuration can be computed in time $O(\log l)$, where l is the largest number in a register. We have quite a number of tests performed in parallel, but all are conceptually straightforward.

The circuit shown in Figure 2.8 has size eight and depth three. This description can be thought of as a blueprint for that circuit, or alternatively as a parallel program executed by a universal circuit simulator.

Chapter 3

Preliminary And Known Results

3.1 DRMBM and Small Space Computations

The characterization of constant time DRMBM computations described in the introductory chapter can be formally summarized as follows:

Proposition 3.1.1 [9]

1. DRMBM$(poly(n), poly(n), O(1)) = \mathsf{NL}$ = Collision CRCW F-DRMBM$(poly(n), poly(n), O(1))$ *with bus width* 1.

2. DRMBM$(poly(n), poly(n), O(1)) =$ DRN$(poly(n), O(1))$.

3. *For any problem π solvable in constant time by some (directed or undirected)* RMBM *family using $poly(n)$ processors and $poly(n)$ buses, π in*Collision CRCW F-DRMBM$(poly(n), poly(n), O(1))$ *with bus width* 1.

The crux of the proof for Proposition 3.1.1 is given by the following Lemmata.

Lemma 3.1.2 [9] CRCW DRMBM$(poly(n), poly(n), O(1)) \subseteq \mathsf{NL}$*, for any write conflict resolution rule and any bus width.*

Proof. Consider some $R \in$ CRCW DRMBM$(poly(n), poly(n), O(1))$ performing step d of its computation $(d \leq O(1))$. We need to find an NL Turing machine M_d that generates the description of R after step d using $O(\log n)$ space, and thus [11] an NL Turing machine M'_d that receives n' (the number of processors in R) and some i, $1 \leq i \leq n'$, and outputs the ($O(\log n)$ long) description for processor i instead of the whole description. We establish the existence of M_d (and thus M'_d) by induction over d, and thus we complete the proof.

M_0 exists by the definition of a uniform RMBM family. We assume the existence of M_{d-1}, M'_{d-1} and show how M_d is constructed. For each processor p_i and each bus k read by p_i during step d, M_d performs (sequentially) the following computation: M_d maintains two words b and ρ, initially empty. For every p_j , $1 \leq j \leq poly(n)$, M_d determines whether p_j writes on bus k. This implies the computation of GAP$_{j,i}$ (clearly computable in nondeterministic $O(\log n)$ space since it is NL-complete [20]). The local configurations of fused and segmented buses at each processor (i.e., the edges of the graph for GAP$_{j,i}$) are obtained by calls to M'_{d-1}. The computation of GAP$_{j,i}$ is necessary to ensure that we take p_j into account even when p_j does not write directly to bus k but instead to another bus that reaches bus k through fused buses.

If p_j writes on bus k, then M_d uses M'_{d-1} to determine the value v written by p_j , and updates b and ρ as follows: (a) If b is empty, then it is set to v (p_j is currently the only processor that writes to bus k), and ρ is set to j. Otherwise: (b) If R uses the Collision rule, the collision signal is placed in b. (c) If the conflict resolution rule is Priority, ρ and j are compared; if the latter denotes a processor with a larger priority, then b is set to v and ρ is set to j, otherwise, neither b nor ρ are modified; the Common rule is handled similarly. (d) Finally, if R uses the Combining resolution rule with $\circ$ as combining operation, b is set to the result of $b \circ v$ (since the operation $\circ$ is associative, the final content of b is indeed the correct combination of all the values written on bus k).

Once the content of bus k has been determined, the configuration of p_i is updated accordingly, b and ρ are reset to the empty word, and the same computation is performed for the next bus read by p_i or for the next processor. The whole computation of M_d clearly takes $O(\log n)$ space. ■

Lemma 3.1.3 [9] *Let* $M = (K, \Sigma, \delta, s_0)$ *be an* NL *Turing machine that accepts* $L \in$ NL. *Then, given some word* x, $|x| = n$, *there exists a* CREW *or* CRCW F-DRMBM *algorithm that computes* $G(M, x)$ *(as an adjacency matrix* I*) in* $O(1)$ *time, and using* $poly(n)$ *processors and* $poly(n)$ *buses of width* 1.

Proof. Put $n' = |V|$ ($n' = poly(n)$). The RMBM algorithm uses $n + ({n'}^2 - n')$ processors: The first n processors p_i, $1 \leq i \leq n$, contain x, i.e., each p_i contains x_i, the i-th symbol of x; p_i does nothing but writes x_i on bus i. We shall refer to the remaining ${n'}^2 - n'$ processors as p_{ij}, $1 \leq i, j \leq n'$. Each p_{ij} assembles first the configurations corresponding to vertices v_i and v_j of $G(M, x)$ and then considers the potential edge (v_i, v_j) corresponding to I_{ij}. If such edge exists, then p_{ij} writes True to I_{ij}, and False otherwise. There is no inter-processor communication between processors p_{ij}, thus any RMBM model is able to carry on this computation.

Clearly, given a configuration v_i, p_{ij} can compute in constant time any configuration v_l accessible in one step from v_i, as this implies the computation of at most a constant number ($O(2^k)$) of configurations. The whole algorithm runs thus in constant time. ■

3.2 DRN and Small Space Computations

The generality of the Collision resolution rule is not limited to DRMBM computations. Indeed, the same property holds for constant time computations on DRN as well. We also find that a DRN is able to carry out any constant time computation using only buses of width 1.

Proposition 3.2.1 **[7]** *For any problem π solvable in constant time on some variant of* RN*, it holds that $\pi \in$* CRCW DRN$(poly(n), O(1))$ *with* Collision *resolution rule and bus width* 1.

The proof of Theorem 3.2.1 is based on the following intermediate results.

Lemma 3.2.2 *For any $X \in \{\text{CRCW}, \text{CREW}\}$, $Y \in \{\text{D}, \varepsilon\}$, and for any write conflict resolution rule, it holds that $X\,Y$*RN$(poly(n), O(1)) \subseteq$ Collision CRCW DRN$(poly(n), O(1))$.

Proof. First, note that Collision CRCW DRN$(poly(n), O(1)) = \mathsf{NL}$ [5]. Thus, we complete the proof by showing that, for any conflict resolution rule, CRCW DRN$(poly(n), O(1)) \subseteq \mathsf{NL}$.

This result is however given by the proof of Lemma 3.1.2. Indeed, it is immediate that the Turing machines M_d and M'_d, $0 \leq d \leq c$ for some constant $c \geq 1$, provided in the mentioned proof work in the case of an RN R just as well as for the RMBM simulation. The only difference is that buses are not numbered in the RN case. So, we first assign arbitrary (but unambiguous) sequence numbers for the RN buses as follows: There exists an $O(\log n)$ space-bounded Turing machine that generates a description of R, since R belongs to a uniform RN family (in fact, such a Turing machine is M_0). Then, in order to find "bus k," M_d uses M_0 to generate the description of R until exactly k buses are generated. The description is discarded, except for the last generated bus, which is considered to be "bus k." Since M_0 is deterministic, it always generates the description in the same order. Thus, it is guaranteed that “bus k” is different from “bus j” if and only if $k \neq j$. The proof of Lemma 3.1.2 continues then unchanged.

The extra space used in the process of generating bus k consists of two counters over the set of buses (one to keep the value k and the other one to count how many buses have been already generated). The counters take $O(\log n)$ space each, since there are at most $poly(n)$ processors, and $(poly(n))^2 = poly(n)$. Thus, the overall space complexity remains $O(\log n)$, as desired. ■

Lemma 3.2.3 GAP$_{1,n}$ $\in$ Collision CRCW DRN$(n^2, O(1))$ *with bus width* 1.

Proof. Let R be the DRN solving GAP$_{1,n}$ instances of size n. Then, R uses n^2 processors (referred to as p_{ij}, $1 \leq i, j \leq n$), connected in a mesh. That is, there exists a (directional) bus from p_{ij} only to $p_{(i+1)j}$ if and only if $i + 1 \leq n$, and to $p_{i(j+1)}$ if and only if $j + 1 \leq n$, as shown in Figure 3.1(a). As shown in the figure, we also denote by E, S, N, and W the ports of p_{ij} to the buses going to $p_{i(j+1)}$, going to $p_{(i+1)j}$, coming from $p_{i(j-1)}$, and coming from $p_{(i-1)j}$, respectively.

We assume that the input graph $G = (V, E)$, $|V| = n$, is given by its adjacency matrix I, and that each processor p_{ij} knows the value of I_{ij}.

The DRN R works as follows: Each processor p_{ij}, $i < j$ fuses its W and S ports if and only if $I_{ij} =$ True. Analogously, each processor p_{ij}, $i > j$ fuses its N and E ports if and only if $I_{ij} =$ True. Finally, each processor p_{ii} fuses all of its ports.

Then, a signal is placed by p_{11} on both its outgoing buses. If p_{nn} receives some signal (either the original one emitted by p_{11} or the signal corresponding to a collision) the input is accepted; otherwise, the input is rejected.

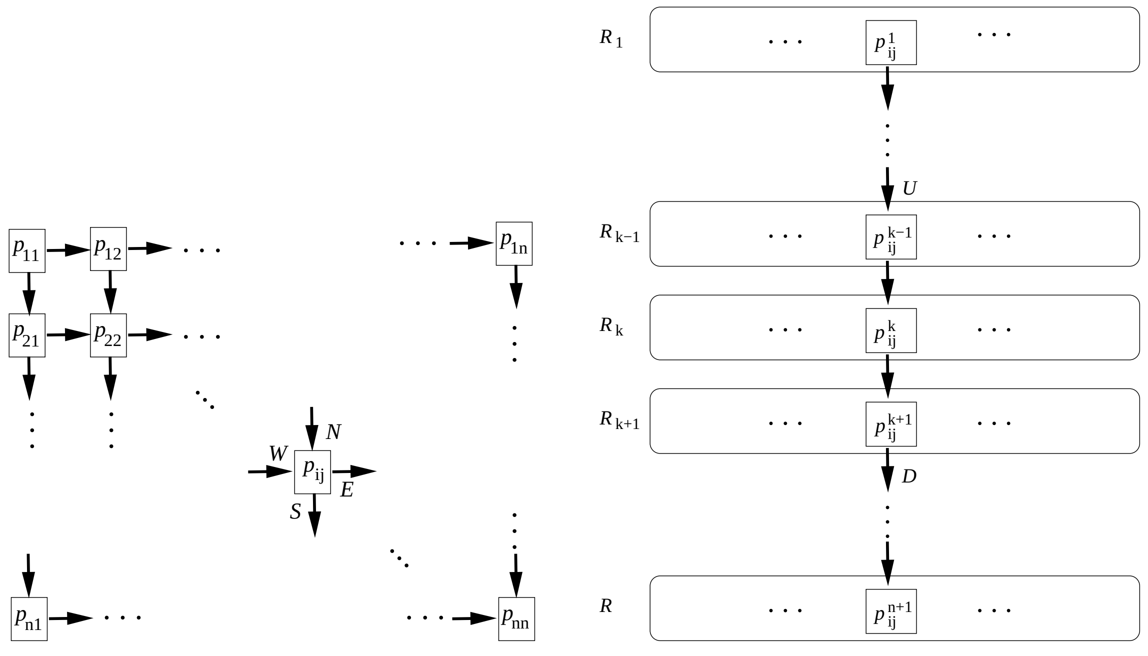

Figure 3.1: (a) A mesh of $n \times n$ processors. (b) A collection of n meshes connected together

It is immediate that R solves GAP$_{1,n}$, by an argument similar to the one for RMBM [9] (also note that a similar construction is presented and proved correct elsewhere [22]). In addition, the content of the signal received by p_{nn} is clearly immaterial, so a bus of width 1 suffices. ■

Recall now that the graph $G(M,x)$ is the graph of configurations of the Turing machine M working on input x.

Lemma 3.2.4 *For any language $L \in \mathsf{NL}$ (with the associated NL Turing machine M accepting L), and given some word x, $|x| = n$, there exists a constant time* CREW *(and thus* CRCW*)* DRN *algorithm using $poly(n)$ processors and buses of width 1 that computes $G(M,x)$ (as an adjacency matrix I).*

Proof. This fact is obtained by the same argument as the one presented in the proof of Lemma 3.1.3. Indeed, except for the distribution of input x to processors, there is no inter-processor communication; as such, any parallel machine will do.

Thus, the computation of $G(M, x) = (V, E)$ will be performed by the same mesh of processors R depicted in Figure 3.1(a), this time of size $n' \times n'$ (where $n' = |V|$). In addition, the desired input distribution will be accomplished by n additional meshes identical to R. We will denote these meshes by R_i, $1 \leq i \leq n$. For any $1 \leq i, j \leq n'$ and $1 \leq k \leq n$, the processor at row i, column j in mesh R_k [R], will be denoted by p_{ij}^k [p_{ij}^{n+1}]. Each processor p_{ij}^k has two new ports U and D. There exists a bus connecting port D of p_{ij}^k to port U of p_{ij}^{k+1} for any $1 \leq k \leq n$. The $n+1$ meshes and their interconnection are shown in Figure 3.1(b).

At the beginning of the computation, x_k, the k-th symbol of input x, is stored in a register of processor p_{11}^k, $1 \leq k \leq n$.

We note from the proof of Lemma 3.1.3 that each processor p_{ij}^{n+1} of R is responsible for checking the existence of a single edge (i, j) of $G(M, x)$. In order to accomplish this, it needs only *one* symbol $x_{h_{ij}}$ from x, namely the symbol scanned by the head of the input tape in configuration i. We assume that all the processors p_{ij}^k, $1 \leq k \leq n$, know the configuration i (and thus the value of h_{ij}).

It remains therefore to show now how $x_{h_{ij}}$ reaches processor p_{ij}^{n+1} in constant time, for indeed, after this distribution is achieved, R is able to compute the adjacency matrix I exactly as shown in the proof of Lemma 3.1.3. The set of $n + 1$ meshes performs the following computation: For all $1 \leq k \leq n$ and $1 \leq i, j \leq n'$,

1. Each p_{11}^k broadcasts x_k to all the processors in R_k. To do this, all processors p_{ij}^k fuse together

their N, S, E, and W ports, and then p_{11}^k places x_k on its outgoing buses.

2. Each p_{ij}^k compares k and h_{ij}, and writes True in one of its registers d if they are equal and False otherwise.

3. Each p_{ij}^k fuses its U and D ports, thus forming $i \times j$ "vertical" buses.

4. Each p_{ij}^k for which $d =$ True places x_k on its port D.

5. Finally, each p_{ij}^{n+1} stores the value it receives on its U port. This is the value of $x_{h_{ij}}$ it needs in order to compute the element I_{ij} of the adjacency matrix.

It is immediate that the above processing takes constant time. In addition, it is also immediate that exactly one processor writes on each "vertical" bus, and thus no concurrent write takes place. Indeed, there exists exactly one processor p_{ij}^k, $1 \leq k \leq n$, such that $k = h_{ij}$. Therefore, we realized the input distribution.

I_{ij} is then computed by processor p_{ij}^{n+1} without further communication, as shown in the proof of Lemma 3.1.3. The construction of the DRN algorithm that computes I is therefore complete. Clearly, buses of width 1 are enough for the whole processing, since x is a word over an alphabet with 2 symbols. ■

Given Lemmata 3.2.3, 3.2.4, and 3.2.2 we can now prove our first main result.

Proof of Proposition 3.2.1. That the Collision resolution rule is the most powerful follows from Lemma 3.2.2. It remains to be shown only that a bus width 1 suffices.

Given some language $L \in \mathsf{NL}$, let M be the (NL) Turing machine accepting L. For any input x, the DRN algorithm that accepts L works as follows: Using Lemma 3.2.4, it obtains the graph

$G(M, x)$ of the configurations of M working on x. Then, it applies the algorithm from Lemma 3.2.3 in order to determine whether vertex n (halting/accepting state) is accessible from vertex 1 (initial state) in $G(M, x)$, and accepts or rejects x, accordingly. In addition, note that the values I_{ij} computed by (and stored at) p_{ij}^{n+1} in the algorithm from Lemma 3.2.4 are in the right place as input for p_{ij} in the algorithm from Lemma 3.2.3 (that uses only the mesh R). It is immediate given the aforementioned lemmata that the resulting algorithm accepts L and uses no more than $poly(n)$ processors, and unitary width for all the buses.

The proof is now complete, since all the problems solvable in constant time on DRN are included in NL. ■

3.3 Collision is Universal on Directed Reconfigurable Buses

The results regarding constant time computations are useful to extend the universality of the Collision resolution rule to any running time.

Proposition 3.3.1 [7] *The* Collision *resolution rule is universal on models with directed reconfigurable buses. That is:*

For any $X \in \{\text{CRCW}, \text{CREW}\}$, $Y \in \{\text{D}, \varepsilon\}$, $Z \in \{\text{RN}(poly(n), \cdot), \text{RMBM}(poly(n), poly(n), \cdot)\}$, *and* $t : \mathbb{N} \to \mathbb{N}$ *it holds that* $X\,Z(t(n)) \subseteq$ Collision $X\,\text{D}Z(O(t(n)))$.

Proof. The proof is immediate for CREW machines. Let now R be an RMBM family in CRCW D RMBM$(poly(n),\ poly(n), t(n))$, and recall the NL machines M'_d constructed in the proof of Lemma 3.1.2. We then take the original R, replace its conflict resolution rule with Collision, and then split every step i of the computation of R into a constant number of steps, as follows:

1. Each processor p of R reads the content of the original buses as required and then performs the prescribed computation for step i, except that whenever p wants to write to bus k it also writes the same value of a dedicated bus k_p (there is one such a bus for each processor).

2. A suitably modified machine M'_d from the proof of Lemma 3.1.2 (call this machine M) computes the content of all the original buses of the network, based on the configurations of all the processors and the content of the (new) k_p buses; the content of bus k thus computed is placed on a brand new bus k'. The meaning of "suitably modified" is that the machine does not need to determine the configuration of the processors of R; indeed, the configurations are already present in the processors themselves. So the machine starts directly with the computation of GAP problems to determine the content of the buses.

3. A designated processor p_k transfers the content of bus k' onto bus k and the algorithm continues with step $i + 1$.

Given that M is an NL Turing machine, it can be implemented by an polynomially bounded DRMBM R_M that runs in constant time, so the modified step i takes $O(1)$ time (and then the whole computation takes $O(t(n))$ time, as desired). This new RMBM needs to read the configurations of R; for this purpose a polynomial number of new buses can interconnect each processor of R with each processor of R_M. We end up with a polynomial number of buses; that we have a polynomial number of processors is immediate. The correctness of the transformation follows from Lemma 3.1.2 and Proposition 3.1.1.

The proof for RN represents a minor variation of the above proof in light of Lemma 3.2.2 (that replaces Lemma 3.1.2) and of the construction used in Lemma 3.2.4 to distribute values to processors. ■

Chapter 4

Relationships Between Parallel Models

We establish here our main result, the hierarchy of parallel models of computation and its collapse at the top. We then consider some consequence, including a generalization of previous results regarding real-time computations.

4.1 The Heavyweight and the Lightweight Classes of Parallel Models

Recall that we called Priority CRCW PRAM and all the models of less computational power lightweight, while the Combining CRCW PRAM, the BSR, and the models with directed reconfigurable buses were called heavyweight. We show in this section that all the heavyweight models have the same computational power, and that they are strictly more powerful than the lightweight models. We thus obtain our main result:

Theorem 4.1.1 *For any* $X \in \{\text{Collision}, \text{Common}\}$,

$$X \text{ CRCW PRAM}(poly(n), O(t(n))) \subseteq \text{Priority CRCW PRAM}(poly(n), O(t(n))) \subsetneq$$

$$\text{Combining CRCW PRAM}(poly(n), O(t(n))) = \text{BSR}(poly(n), O(t(n))) =$$

$$\text{DRMBM}(poly(n), poly(n), O(t(n))) = \text{DRN}(poly(n), O(t(n))).$$

Proof. Theorem 4.1.1 is a direct consequence of Lemmata 4.1.2 to 4.1.7 below. Specifically, all the inclusions shown in the theorem are proved in the mentioned Lemmata one by one. Additionally, it is already known [21] that DRMBM$(poly(n), poly(n), O(t(n))) =$ DRN$(poly(n), O(t(n)))$. ■

We complete all the proofs below by showing how the model on the right hand side of the inclusion simulates in constant time one computational step of the model on the left hand side. Once this is shown, the inclusion that needs to be proved becomes immediate.

Lemma 4.1.2 Collision CRCW PRAM$(poly(n), O(t(n))) \subseteq$ Priority CRCW PRAM$(poly(n), O$ $(t(n)))$.

Proof. A Collision CRCW PRAM with k processors p_i, $1 \leq i \leq k$ and m memory locations u_j, $1 \leq j \leq m$ is readily simulated by a Priority CRCW PRAM with $2k + m$ processors denoted by $p_i^{\uparrow}$ ($1 \leq i \leq k$), $p_i^{\downarrow}$ ($1 \leq i \leq k$), and p_j^m ($1 \leq j \leq m$). (Note however that the processor group p_j^m and the processor group $p_i^{\uparrow}$ plus $p_i^{\downarrow}$ take turns in the simulation, so the actual number of processors required is $\max(2k, m)$; however, the differentiation eases the presentation.)

In addition to the original memory locations u_j, we use two more "banks" of the same size $u_j^{\uparrow}$ and $u_j^{\downarrow}$, $1 \leq j \leq m$. A Collision CRCW PRAM step (read,compute, write) is then simulated as follows:

1. For every $1 \leq i \leq k$, both the processors $p_i^{\uparrow}$ and $p_{k+1-i}^{\downarrow}$ perform the same read, compute, and write cycle as the original p_i, with the following addition: Whenever processor p_i writes into

memory location u_j, it also writes its number into memory location $u_j^{\uparrow}$; similarly, whenever processor p_{n+1-i} writes into memory location u_j, it also writes $n+1-i$ into memory location $u_j^{\uparrow}$.

2. Every processor p_i^m writes the collision value into memory location u_j iff $u_j^{\uparrow} = u_j^{\downarrow}$.

That the above simulation takes constant time is immediate. Note further that after Step 1 of the simulation the location $u_j^{\uparrow}$ [$u_j^{\downarrow}$] contains the index of the lowest [highest] ranked processor that modified the memory location u_j (indeed, we operate on a Priority CRCW model, so only the lowest numbered processor succeeds in writing into a given memory location; we then chose the processors numbers in appropriate manner for this to happen). Then, whenever $u_j^{\uparrow} \neq u_j^{\downarrow}$, more than one processor wrote into the given memory location, so a collision occurred. Step 2 places collision markers accordingly. The simulation is complete. ■

Lemma 4.1.3 Common CRCW PRAM$(poly(n),\ O(t(n)))$ $\subseteq$ Priority CRCW PRAM$(poly(n),$ $O(t(n)))$.

Proof. We simulate now a computational step of a Common CRCW PRAM with k processors and m memory locations in constant time using a Priority CRCW PRAM. The simulation will use the same number k of processors (we denote them by p_i, $1 \leq i \leq k$) and the same memory space (denoted by u_j, $\leq j \leq m$). The simulation proceeds as follows:

1. All the processors p_i carry on the computational step prescribed by the Common CRCW PRAM algorithm, including the operation of writing into the shared memory (recall, however, that we are now using the Priority conflict resolution rule).

2. Each processor p_i that wrote a value v_i into memory location u_j in Step 1 also remembers v_i by storing it into an otherwise unused internal register ρ.

3. Every p_i that wrote a value into location u_j in Step 1 compares the content of u_j with the content its register ρ.

   (a) If the contents of u_j and ρ are the same then either (a) p_i is the sole processor which wrote into u_j, (b) p_i is the highest priority processor which wrote into u_j, or (c) p_i and the highest priority processor agree on the value written into u_j. None of these cases violate the Common resolution rule, so p_i does not do anything.

   (b) If on the other hand u_j and ρ contain different values, then the value written into u_j by p_i disagrees with the value written in the same location by some other processor, which in turn violates the Common resolution rule. So p_i aborts the algorithm and reports failure.

Note that in effect we chose *one* of the processors writing concurrently into a memory location as representative for all the others (given that we have a Priority machine at our disposal, that representative turned out to be the processor with the highest priority—however the way we chose a representative is immaterial). Every processor which wants to write a value in some memory location compares now its value with the value already written by its representative; if the value is different, then the Common conflict resolution rule is violated; otherwise all is good and the overall algorithm continues with the next step.

The above proof uses the usual definition of Common, as presented in Section 2.3. Still, we note that sometimes this definition is termed "Fail Common," case in which the "Error-safe Common" variant is also defined. In such a variant, any computational step that violates the Common resolution rule is discarded completely (that is, for all the processors in the system) and the algorithm continues with the next step (instead of aborting the computation). A proof for this modified variant of Common is readily possible. Indeed, all we need is “backup” copies of the memory locations used by the algorithm, plus one memory location used to signal any violation to everybody. The processors now use the backup memory to perform all the simulation described above, except that they set the violation flag instead of aborting the algorithm whenever a violation of the Common resolution rule occurs (since is is just a flag, using Priority as conflict resolution rule will do just as well as almost any other rule). In the end, all the processors inspect the flag and write again the values they wanted to write in the first place (this time in the main memory, not the backup) iff the flag is not set. ■

Lemma 4.1.4 Priority CRCW PRAM$(poly(n), O(t(n))) \subsetneq$ Combining CRCW PRAM$(poly(n),$ $O(t(n)))$.

Proof. A Priority CRCW PRAM with k processors p_i, $1 \leq i \leq k$ and m memory locations u_j, $1 \leq j \leq m$ is readily simulated by a Combining CRCW PRAM with the same number of processors (denoted by p'_i) and $2m$ memory locations (denoted by u_i and u'_i):

1. Each processor p'_i performs the same read and compute operations as p_i. Instead of writing (to memory location u_j), p'_i however performs a “dry run” by writing its number into memory location u'_j using a Combining CRCW write with $\min$ as combining operation.

2. Each processor p'_i performs now the real operation: It writes into the memory location u_j in which it wanted to write to begin with, but only iff its number matches the value stored in u'_j.

The $\min$ as combining operation performed over the locations u'_j in the previous step ensures that a matching occurs only for the lowest numbered processor, as desired.

That Combining CRCW PRAM$(poly(n), O(t(n))) \not\subseteq$ Priority CRCW PRAM$(poly(n), O(t(n)))$ is an immediate extension of Proposition 2.3.1. Indeed, PARITY$_n$ is trivially solvable in constant time by a Combining CRCW PRAM. Such a machine performs a Combining CRCW using Σ as combining operation, followed by a modulo operation on one memory location. ■

Lemma 4.1.5 Combining CRCW PRAM$(poly(n), t(n)) =$ BSR$(poly(n), O(t(n)))$.

Proof. That Combining CRCW PRAM$(poly(n), t(n)) \subseteq$ BSR$(poly(n), O(t(n)))$ is immediate from the definition of BSR. Surprisingly enough, the reverse inclusion is also true. We show this reverse inclusion by showing how one BSR computational step is simulated by a Combining CRCW PRAM in constant time.

Consider a BSR with k processors and m memory locations. Every BSR processor p_i is simulated by a set of m PRAM processors r_{ij}, $1 \leq j \leq m$. The PRAM memory is doubled, every BSR memory location u_j will be simulated by two PRAM memory locations u^d_j and u^l_j, $1 \leq j \leq m$. Finally, the PRAM uses extra processors p^u_j, $1 \leq j \leq m$. The PRAM simulation of a BSR Broadcast step (read, compute, Broadcast) proceeds as follows:

- *Read and Compute:* All the processors r_{ij}, $1 \leq j \leq m$ perform the reading and the computation prescribed for p_i. Every time some processor wants to read the value of u_j it will read the value of u^d_j instead. Processors r_{ij} will then *all* hold the values of the datum d_i and the tag g_i computed by p_i.

- *Selection limits:* Every processor p_j^u computes the limit l_j associated with u_j in the selection phase of the BSR step, and stores it in the memory location u_j^l.

- *Broadcast instruction:* r_{ij} will be responsible for the data written by the BSR processor p_i into memory location r_j (note that all the processors r_{ij}, $1 \leq j \leq m$ contain identical data, so p_i's replacement covers all the memory locations, thus realizing the desired broadcast):

  1. r_{ij} reads l_j from memory location u_j^l so that it holds d_i, g_i, and l_j;
  2. r_{ij} then computes the selection criterion $g_i \, \sigma \, l_j$ as prescribed by the BSR algorithm;
  3. r_{ij} writes d_j into memory location u_j^d iff $g_i \, \sigma \, l_j =$ True, using a Combining CRCW write with the combining operator prescribed by the BSR algorithm.

In effect, we use one PRAM processor for every pair processor–memory location in the BSR algorithm. This allows for an easy simulation of the broadcast phase of a Broadcast instruction: Instead of broadcasting, every PRAM processor is responsible for writing to one memory location; since we have as many processors as memory locations, we nonetheless write to all the memory locations at once, as desired. The rest of the simulation is immediate, as is the overall constant running time.

■

Lemma 4.1.6 BSR$(poly(n), t(n)) \subseteq$ DRMBM$(poly(n), poly(n), O(t(n)))$.

Proof. We are given a BSR with k processors and m memory locations. Without loss of generality we provide a Combining DRMBM that simulates the given BSR; for indeed, once such a construction is established a Collision DRMBM with polynomially bounded resources and $O(t(n))$ running time exists by Theorem 3.3.1.

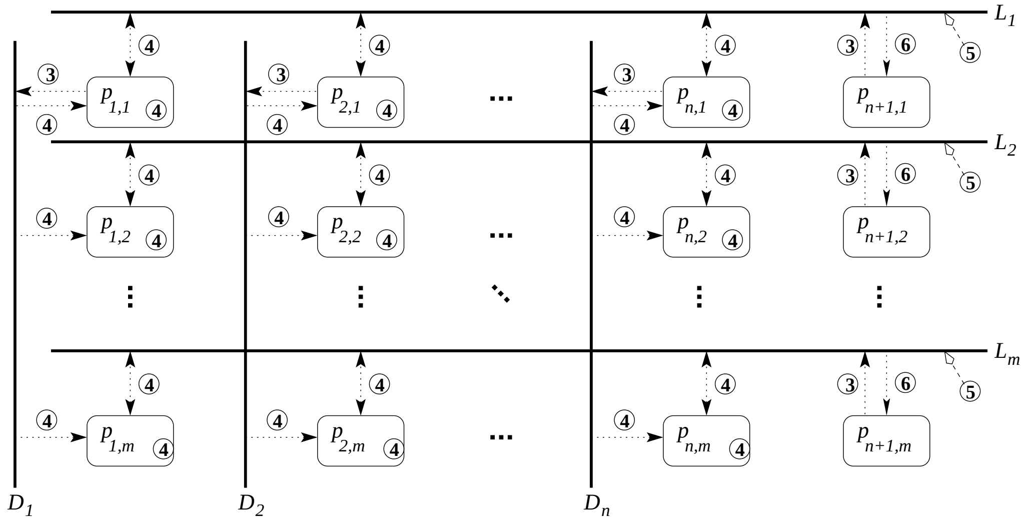

Figure 4.1: A DRMBM simulation of the BSR

The DRMBM that simulates the BSR is partially depicted in Figure 4.1. It has $(k + 1) \times m$ processors, which we denote by $p_{i,j}$, $1 \leq i \leq k + 1$, $1 \leq j \leq m$. The "real" processors $p_{i,1}$, $1 \leq i \leq k$ perform identically to the processors of the original BSR (except for the bus manipulation routines). The "memory" processors $p_{k+1,j}$, $1 \leq j \leq m$ will simulate the locations of the shared memory. They designate one register μ that will hold the data stored in the respective memory location.

As shown in the figure, the DRMBM features $k + m$ buses denoted by D_i, $1 \leq i \leq k$ and L_j, $1 \leq j \leq m$. In addition, every "memory" processor $p_{k+1,j}$ has a dedicated bus M_j, $1 \leq j \leq m$ (not shown in the figure). The DRMBM simulates one step of the BSR computation (meaning one read-compute-broadcast-select-reduce cycle of the BSR) in constant time as follows (the steps of the simulation—less the read and compute phases—are depicted by circled numbers in the figure):

1. At the beginning of every BSR step every "memory" processor $p_{k+1,j}$ puts the datum held in its designated register μ on bus M_j. Every "real" processor $p_{i,1}$ that is interested in some

memory data reads the bus of interest.

2. All the “real” processors perform the computation prescribed by the BSR for the current computational step.

3. Every “real” processor $p_{i,1}$ broadcasts the computed pair (d_i, g_i) by putting it on bus D_i. At the same time, every “memory” processor $p_{k+1,j}$ broadcasts its limit l_j by placing it on bus L_j.

4. The processors $p_{i,j}$, $1 \leq i \leq k$, $1 \leq j \leq m$ implement the selection phase as follows: $p_{i,j}$ reads the pair (d_i, g_i) and the limit l_j from the buses D_i and L_j, respectively. It then computes $t_i \, \sigma \, l_j$ and places d_i on bus L_i as appropriate (i.e., iff $t_i \, \sigma \, l_j =$ True).

5. The reduction is accomplished by the buses L_j, $1 \leq j \leq m$ which perform the combining (reduction) operation prescribed by the BSR algorithm.

6. Finally, every “memory” processor $p_{k+1,j}$ reads bus L_j and stores the datum thus obtained into its designated register μ.

It is immediate that the above steps complete in constant time and accomplish the desired computation. This sequence of steps is applied repeatedly for every step performed by the BSR. The running time of the whole simulation is then $O(t(n)$, as desired. We have $(k+1) \times m = poly(n)$ processors and $2m + k = poly(n)$ buses, so the DRMBM uses polynomial resources.

We have shown that one computational step of the BSR is computable in constant time by a DRMBM. The inclusion follows immediately, and the proof is thus complete. ■

Lemma 4.1.7 DRMBM$(poly(n), poly(n), t(n)) \subseteq$ BSR$(poly(n), O(t(n)))$.

Proof. The result is an almost immediate consequence of Proposition 2.3.2. Indeed, the capability of the BSR to compute GAP in constant time allows this model to simulate bus fusing, which is the only essential supplementary capability of the RMBM over the BSR.

As usual by now, we prove the inclusion by showing how one computational step of a DRMBM can be simulated in constant time by a BSR. According to Proposition 3.1.1, it is enough to offer a proof with respect to the Collision F-DRMBM.

Consider a Collision F-DRMBM with k processors and m buses. The BSR that simulates it uses k processors p_i, $1 \leq i \leq k$ to simulate the DRMBM processors and $O(m^2)$ processors (referred to collectively as P^c and individually as p^c_{ij}, $1 \leq i, j \leq m$) dedicated to the computation of the reflexive and transitive closure of an $m \times m$ graph. In terms of memory space, the notable areas include m memory locations b_j and another m memory locations c_j, $1 \leq j \leq m$ to simulate the buses, $m \times m$ memory locations G_{ij}, $1 \leq i, j \leq m$ to hold the connectivity graph for the buses, and another $m \times m$ memory location $C_{i,j}$, $1 \leq i, j \leq m$ to hold the reflexive and transitive closure of the aforementioned graph. We assume that there exists a value that is never placed on a bus by any DRMBM processor; we call this vale the collision marker (similar to the notion used in the Collision CRCW PRAM).

A computational step of a DRMBM processor consists in the following phases: read data from the buses, perform the prescribed computation (thus determining what buses to fuse and what values to write to buses), fuse buses, and write data to buses. The BSR simulation of one DRMBM step proceeds then as follows:

1. The P^c processors initialize in parallel G_{ij} to False and c_j to 0, $1 \leq i, j \leq m$

2. Every p_i reads the memory locations b_j as prescribed by the DRMBM algorithm (we replace reading buses with reading memory locations).

3. Every p_i performs the operations prescribed by the DRMBM algorithm.

4. Every p_i that wants to fuse buses in the DRMBM algorithm broadcasts True to all the memory location G_{uv} corresponding to buses u and v being fused by p_i. The writing is a Combining CRCW write with $\bigvee$ as combining operator.

5. The memory locations G_{uv}, $1 \leq u, v \leq m$ now hold the adjacency matrix of the graph of connected buses. The processors P^c now compute the reflexive and transitive closure of G, putting the result in $C_{i,j}$, $1 \leq i, j \leq m$.

6. Every processor p_i that wants to write some value to bus j writes the corresponding value into b_j using a Combining CRCW write with any combining operator, and writes 1 into memory location c_j using a Combining CRCW write with Σ as combining operator.

7. The processors P^c now alter the content of b_j as follows: If $C_{ji} =$ False, then p^c_{ji} does not perform anything. Otherwise, $p^c ji$ reads the value from b_j and writes it into the memory location b_i using a Combining CRCW write with any combining operator. p^c_{ji} also reads the value from c_j and writes it into the memory location c_i using a Combining CRCW write with Σ as combining operator.

8. Every processor p^c_{j1}, $1 \leq j \leq m$ reads c_j and places the collision marker into memory location b_j iff $c_j > 1$.

Reading data from the buses and performing the prescribed computation are simulated by Steps 2 and 3 of the above simulation, respectively. Fusing the buses is prepared in Step 1 and carried on in Steps 4 (G_{ij} contains True iff buses i and j are fused together directly by a processor) and 5 (C_{ij} contains True iff buses i and j are fused together either directly by a processor of via a chain of fused buses). Writing on the buses is performed in Step 6 (each bus receives the values written to it directly by the DRMBM processors) and 7 (the bus content is propagated according to the fused buses). At the end of this step, the value of c_j contains the number of processors that have written on bus j (either directly or via intermediate, fused buses); If c_j is one or zero, then the bus should be left alone; otherwise, a collision has happened, so the content of the bus should be replaced by the collision marker. This replacement is accomplished by Step 8.

We argue that Step 4 is achievable in constant time, as follows: there is no specification of how many buses can be fused and in what combination by a DRMBM. However, a DRMBM processor should be capable of computing the configuration of fused buses in constant time. It follows that the BSR processor simulating the DRMBM processor is capable of determining the corresponding broadcast parameters in constant time, since it is the same processor in terms of computational capabilities. All the other steps are immediately achievable in constant time, so the proof is established. ■

4.2 GAP, the Universality of Collision, and Real Time Considerations

As far as constant time computations are concerned, we noted a contrast between the power of conflict resolution rules for models with directed reconfigurable buses (DRMBM and DRN) on one hand, and for shared memory models (PRAM) on the other hand. According to our results, Collision is the most powerful rule on DRMBM and DRN. By contrast, we showed in Theorem 4.1.1 that the Combining CRCW PRAM is strictly more powerful than the Collision CRCW PRAM.

We also note that the ability of DRMBM (and DRN) to compute GAP in constant time is central to the constant time universality of the Collision rule, and also determines that exactly all DRMBM and DRN computations are in NL; we also note that GAP is NL-complete [20]. In light of this, we consider the classes $\mathbb{M}_{<GAP}$, $\mathbb{M}_{\equiv GAP}$, and $\mathbb{M}_{>GAP}$ of parallel models of computations using polynomially bounded resources (processors and, if applicable, buses), such that:

$\mathbb{M}_{<GAP}$ contains exactly all the models that cannot compute GAP in constant time, and cannot compute in constant time any problem not in NL.

$\mathbb{M}_{\equiv GAP}$ contains exactly all the models that can compute GAP in constant time, but cannot compute in constant time any problem not in NL.

$\mathbb{M}_{>GAP}$ contains exactly all the models that can compute GAP in constant time and can compute in constant time at least one problem not in NL. To our knowledge, no model has been proved to pertain to such a class.

As a direct consequence of Theorem 4.1.1, we can then populate these three classes (or at least the first two) in a meaningful manner:

Corollary 4.2.1 *1.* Combining CRCW PRAM$(poly(n), O(1))$ = BSR$(poly(n), O(1))$ = DRMBM$(poly(n), poly(n), O(1))$ = DRN$(poly(n), O(1))$ = NL.

2. X CRCW PRAM$(poly(n), t(n)) \in \mathbb{M}_{<GAP}$ *for any* $X \in \{\text{Collision}, \text{Priority}, \text{Common}\}$.

3. Combining CRCW PRAM$(poly(n), O(1))$, BSR$(poly(n), O(1))$,

DRMBM$(poly(n), poly(n), O(1))$, DRN$(poly(n), O(1)) \in \mathbb{M}_{\equiv GAP}$.

Proof. Immediate from definitions, Theorem 4.1.1, and Proposition 3.1.1. ■

Compare now the previous discussion on GAP with the following immediate generalization of Claim 1:

Theorem 4.2.2 *For any models of computation M_1, M_2, and M_3 such that $M_1 \in \mathbb{M}_{<GAP}$, $M_2 \in \mathbb{M}_{\equiv GAP}$, and $M_3 \in \mathbb{M}_{<GAP}$, it holds that*

$$\text{rt-PROC}^{M_1}(poly(n)) \subseteq \mathsf{NL}/rt \tag{4.1}$$

$$\text{rt-PROC}^{M_2}(poly(n)) = \mathsf{NL}/rt \tag{4.2}$$

$$\text{rt-PROC}^{M_3}(poly(n)) \supset \mathsf{NL}/rt \tag{4.3}$$

Proof. Minor variations of the arguments used previously [9] show that those computations which can be performed in constant time on M_i, $1 \leq i \leq 3$, can be performed in the presence of however tight time constraints (and thus in real time in general). Then, Relations (4.1) and (4.3) follow immediately from Claim 1. By the same argument, rt-PROC$^{M_2}(poly(n)) \supseteq$ rt-PROC$^{\text{CRCW F-DRMBM}}(poly(n))$ holds as well. The equality (and thus Relation (4.2)) is given directly by the arguments that support Claim 1 [9]. ■

Thus, the characterization of real-time computations established by Claim 1 does hold in fact for any machines that are able to compute GAP in constant time. The characterization presented in Theorem 4.2.2 emphasizes in fact the strength of Claim 1. Indeed, as noted above (Theorem 4.1.1), no model more powerful than the DRMBM is known to exist. That is, according to the current body of knowledge, $\mathbb{M}_{>GAP} = \emptyset$. Unless this relation is found to be false, Claim 1 states essentially that no problem outside NL can be solved in real time no matter the model of parallel computation being used (that is, Claim 1 holds for all the parallel models, not just the DRMBM).

Chapter 5

EW On Reconfigurable Buses And The Power of the MESH

The universality of Collision on models with directed reconfigurable buses was established earlier (as presented in Chapter 3). Similar arguments turn out to establish the same result for undirected reconfigurable buses. As it turns out, the universality of Collision can even be strengthened, though this depends on the definition of conflict on reconfigurable buses.

Indeed, a collision happens whenever two signals arrive simultaneously at the same bus, but it can also be defined as happening whether two signals *from two different processors* arrive at the same bus. Under the latter definition it turns out the EW is universal and the Collision conflict resolution rule is not necessary. Such a result has practical consequences, most notably in the design of VLSI circuits.

Still on the same practical level (design of VLSI circuits), we also note that MESH simulations exist for all the heavyweight models. This is practically significant given the ease of implementation of MESH over other models with reconfigurable buses.

5.1 The Universality of EW or Collision on Reconfigurable Buses

One can conceivably identify two variants of CW, as follows:

Definition 5.1.1 Strong CW: Any two signals arriving simultaneously at the same bus are considered concurrent-write.

Weak CW: Two signals from two different processors arriving simultaneously at the same bus are considered concurrent-write. However, a signal that is split and arrives two times at some bus is not considered concurrent-write.

Strong CW is implied earlier in this thesis. Weak CW also appears realistic, for indeed a bunch of fused buses form an electrical, longer bus; then it makes no sense to consider a signal that travels on two different paths; the signal is simply placed on the bus and propagates along it according to the physical laws.

As it turns out, the definition of CW makes a significant difference in terms of universality of Collision: under weak CW, Collision is unnecessary on reconfigurable buses; instead, EW becomes universal. This is all put together as follows:

Theorem 5.1.1

1. *Under strong CW, Collision is universal on reconfigurable buses (directed or undirected).*
2. *Under weak CW, EW is universal on reconfigurable buses (directed or undirected).*

Proof. The strong CW case has been established for the directed case earlier, as presented in Section 3.3. The undirected case is easily derived from the proofs supporting the directed case; one only need to replace GAP with undirected GAP and NL with L as done (in different contexts) earlier [5, 6].

The weak CW case has been established for undirected buses elsewhere [6]. A simple extension of this proof establishes it for directed buses as well. Alternatively, the proofs of the results presented in Section 3.3 establish the directed buses variant immediately, as we note that no collision (in the strong sense) happens in the constructions presented there. ■

In all, all of the heavyweight models of computation can in fact be simulated by EW DRMBMs (or EW DRNs).

5.2 MESH Simulations

We found out that exclusive-write is universal on reconfigurable buses. Another practically useful property of these models is their simulation as MESH. Indeed, a reconfigurable bus machine (be it RMBM or RN) can be always laid out as a MESH.

Proposition 5.2.1 **[17]** *A d-dimension* R-MESH$(r_1, r_2, \ldots, r_d)$ *can be simulated by a 2D R-MESH with* $poly(r_1 \times r_2 \times \ldots \times r_d) \times poly(r_1 \times r_2 \times \ldots \times r_d)$ *resources in constant time.*

Proposition 5.2.2 **[17]** *For* $X, Y \in \{C, E\}$*, each step of a* $\sqrt(p) \times \sqrt(p)$ $XRYW$ R-MESH *can be simulated on an* $XRYW$ F-RMBM$(4p, 5p)$ *in* $O(1)$ *time.*

Proposition 5.2.3 **[17]** *For* $X, Y \in \{C, E\}$*, a step of an* $XRYW$ F-RMBM(p, b) *can be simulated on a* $(b+1) \times 3p$ $XRYW$ R-MESH *in* $O(1)$ *time.*

The proof of Proposition 5.2.2 is sketched informally in Figure 5.2 and proceeds as follows: let p_i denote a processor of the simulated R-MESH, for $0 \leq i \leq p$. Divide the processors of the simulating RMBM into p teams T_i, each with four processors $\pi_{i,N}$, $\pi_{i,S}$, $\pi_{i,E}$, and $\pi_{i,W}$, for $0 \leq i < p$. The processors of T_i simulate the ports of R-MESH processor p_i.

The first $2p$ buses of the RMBM simulate external processor-to-processor connections in the R-MESH. The next $2p$ buses provide fusing locations to form internal connections. The last p buses provide channels for connection among processors on the same team.

The simulation stated in Proposition 5.2.3 is shown informally in Figure 5.2. Let p_i and b_j denote the processors and buses, respectively of the F-RMBM, where $0 \leq i < p$ and $0 \leq j < b$. Here the simulating R-MESH is of size $(b+1) \times 3p$. Denote the processors of the R-MESH by $pi_{k,g}$,where $0 \leq k < b+1$ and $0 \leq g < 3p$. Rows $1, 2, \ldots, b$ of the R-MESH simulate the b buses of RMBM. For any $0 \leq i < p$,columns $3i$,$3i+1$,and $3i+2$ simulate the write port, read port, and fuse line of processor p_i of the RMBM.

These simulations are trivially extensible to the directed case, so we have:

Corollary 5.2.4 *For $X, Y \in \{C, E\}$, each step of a $\sqrt(p) \times \sqrt(p)$ $XRYW$* DR-MESH *can be simulated on an $XRYW$* F-DRMBM$(4p, 5p)$ *in $O(1)$ time.*

Corollary 5.2.5 *For $X, Y \in \{C, E\}$, a step of an $XRYW$* F-DRMBM(p, b) *can be simulated on a $(b+1) \times 3p$ $XRYW$* DR-MESH *in $O(1)$ time.*

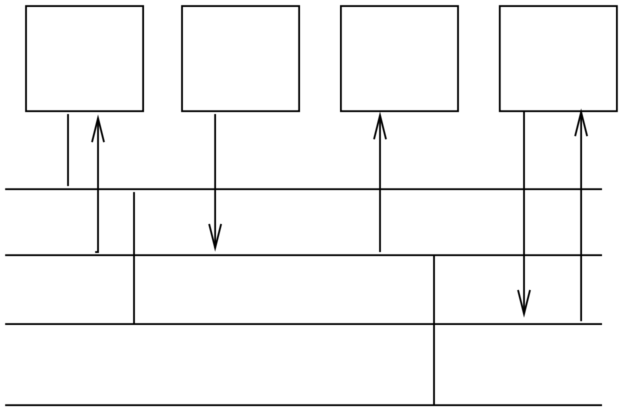

Figure 5.1: The simulated RMBM

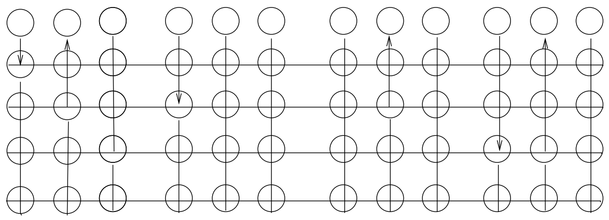

Figure 5.2: A reconfigurable MESH simulates an RMBM

Chapter 6

Conclusions

This thesis focused on two points. The major point is the establishment of a hierarchy for parallel models with shared memory and with reconfigurable buses. A secondary point is the investigation of the universality of EW on reconfigurable buses (along with other practical considerations).

6.1 The Relationships Between Several Parallel Computation Models

Our results are rather significant, as we essentially free the analysis of parallel algorithms and problems from a number of restrictions such as whether using the Broadcast instruction of the BSR or using the Combining resolution rule on distributed resources like buses diminishes the practicality of the analysis. At the same time, we also offer a strict delimitation between the two classes of heavyweight and lightweight models of parallel computation.

It was found earlier that there exists a very strong similarity between the two models with directed reconfigurable buses, the DRN and the DRMBM: Not only they solve the same problems (namely, exactly all the problems in NL), but in both cases (a) the smallest possible bus width is enough for all problems, and (b) the Collision resolution rule is the most powerful (even as powerful as the Combining rule). It was further shown that Collision is the most powerful on DRN and DRMBM for any running time. Accordingly, the discussion regarding the practical feasibility of rules like Priority or Combining on spatially distributed resources such as buses is no longer of interest. Indeed, such rules are not only of questionable feasibility, but also not necessary! This offers a powerful tool in the analysis of models with directed reconfigurable buses and in the design of algorithms for these models: One can freely use Combining (Priority, etc.) rules on these models, with the knowledge that they can be eliminated without penalty—the analysis or algorithm design uses an “unfeasible” model yet is fully pertinent to the real world. In fact we used such a technique ourselves in the proof of Lemma 4.1.6.

By contrast with models with directed reconfigurable buses, it was widely believed that the all-powerful Combining conflict resolution rule does add computational power to the PRAM model, and that the BSR’s Broadcast instruction adds further power. We are to our knowledge the first to establish formally a hierarchy of the PRAM variants that both confirms and contradicts the mentioned belief. Indeed, we showed that Combining does add computational power over “lesser” rules. However, we also showed that surprisingly enough the Broadcast instruction does not add computational power over Combining . In fact we established an intriguing collapse of the hierarchy of parallel models at the top of the food chain, where the Combining CRCW PRAM, the BSR, and the models with directed reconfigurable buses turn out to have identical computational power.

Once more, this result offers substantial support for the analysis of shared memory models.

Indeed, the power of BSR's Broadcast instruction has attracted attention in various areas of parallel algorithms. Algorithms with running time as fast as constant have been developed for various problems, most notably in the areas of geometric and graph computations [12, 13]. Of course, constant time algorithms for such practically meaningful problems are very attractive. Nevertheless, the model tends to be frowned upon given its apparent implementation complexity, even after efficient implementations have been proposed [2]. We now showed that whether the BSR is feasible or not is irrelevant, as a Combining PRAM with the same performance and with all the attractive properties of the BSR is automatically available. Same goes for, say the BSR versus the DRMBM: one can freely use the BSR model to design DRMBM algorithms (which is like using an abstract, powerful model to design VLSI circuits) and the other way around—in essence, one can freely choose between a number of models, depending on no matter what issues (ranging from practical feasibility to convenience to mere taste) with the formally supported knowledge that the results are portable to all the other models.

We also note that most of our proofs are constructive (and those which are not still offer constructive hints), so we also set the basis for automatic conversion back and forth between models. True, we did not have efficiency in mind, so our constructions are likely to be inefficient; however, traditionally inefficient algorithms have been optimized quite easily, so we believe that our however inefficient algorithms are nonetheless a significant contribution.

We also noted the central role of the graph accessibility problem (GAP) for the DRN and DRMBM results obtained here and also previously [9]. We further strengthened our previous results on real-time computations, eliminating to some degree their weak point (model dependence).

Having found that Collision is universal on all the models with reconfigurable buses from

$\mathbb{M}_{\equiv GAP}$ and given that GAP is not computable in constant time on models from $\mathbb{M}_{<GAP}$, we believe that the Collision resolution rule is *not* universal on models in $\mathbb{M}_{<GAP}$ (on which the notion of conflict resolution rule makes sense), such as models with undirected reconfigurable buses or the lightweight PRAM models. For instance, note that Priority has already been established by Lemmata 4.1.2 and 4.1.3 as being the most powerful rule on these models, but this does not yet exclude the possibility that Collision has equal power and is thus also universal; we believe however that Priority is strictly more powerful than Collision on the PRAM. Showing (or disproving) all of this is an intriguing open problem.

6.2 EW is Universal on Reconfigurable Buses

This thesis, as well as preceding work on the matter have established the universality of Collision on reconfigurable buses assuming implicitly a strong CW rule. Under weak CW however an almost identical proof establishes the universality of EW. We believe that weak CW is a realistic definition, given the physical (electrical) realization of reconfigurable buses.

VLSI design uses reconfigurable buses extensively. The universality of Collision or EW (depending on whether we choose the strong or weak CW rule) is significant for VLSI design, as both Collision and EW are easily implemented in silicon. Equally significant (but this time from a layout point of view) is that all the reconfigurable bus models can be all laid out as two-dimensional meshes. More work is necessary for the refinement of these processes (of converting a general machine into a Collision-only machine, or to lay out a general machine as a mesh) before they become useful in practice, but the most important step (or showing that they are possible) is done here.

Bibliography

[1] S. G. AKL, *Parallel Computation: Models and Methods*, Prentice-Hall, Upper Saddle River, NJ, 1997.

[2] S. G. AKL AND L. FAVA LINDON, *An optimal implementation of broadcasting with selective reduction*, IEEE Transactions on Parallel and Distributed Systems, 4 (1993), pp. 256–269.

[3] S. G. AKL AND G. R. GUENTHER, *Broadcasting with selective reduction*, in Proceedings of the IFIP 11th World Congress, G. X. Ritter, ed., San Francisco, CA, 1989, North-Holland, Amsterdam, pp. 515–520.

[4] E. ALLENDER, M. C. LOUI AND K. W. REGAN, *Complexity Classes*, DIMACS Technical Reports, 23 (1998).

[5] Y. BEN-ASHER, K.-J. LANGE, D. PELEG, AND A. SCHUSTER, *The complexity of reconfiguring network models*, Information and Computation, 121 (1995), pp. 41–58.

[6] Y. BEN-ASHER, D. PELEG, R. RAMASWAMI, AND A. SCHUSTER, *The power of reconfiguration*, Proceedings of the 18th International Colloquium on Automata, Languages and Programming, Madrid, Spain, Springer 1991, pp. 139–150.

[7] S. D. BRUDA, *The graph accessibility problem and the universality of the collision CRCW conflict resolution rule*, WSEAS Transactions on Computers, 10 (2006), pp. 2380-2387.

[8] S. D. BRUDA AND S. G. AKL, *Pursuit and evasion on a ring: An infinite hierarchy for parallel real-time systems*, Theory of Computing Systems, 34 (2001), pp. 565–576.

[9] ——, *Size matters: Logarithmic space is real time*, International Journal of Computers and Applications, 29:4 (2007).

[10] M. FURST, J. B. SAXE, AND M. SIPSER, *Parity, circuits, and the polynomial-time hierarchy*, Mathematical Systems Theory, 17 (1984), pp. 13–27.

[11] R. GREENLAW, H. J. HOOVER, AND W. L. RUZO, *Limits to Parallel Computation: P-Completeness Theory*, Oxford University Press, New York, NY, 1995.

[12] J.-F. MYOUPO AND D. SEME, *Work-efficient BSR-based parallel algorithms for some fundamental problems in graph theory*, The Journal of Supercomputing, 38 (2006), pp. 83–107.

[13] J.-F. MYOUPO, D. SEME, AND I. STOJMENOVIC, *Optimal BSR solutions to several convex polygon problems*, The Journal of Supercomputing, 21 (2002), pp. 77–90.

[14] I. PARBERRY, *Parallel Complexity Theory*, John Wiley & Sons, New York, NY, 1987.

[15] D. PELEG, Y. BEN-ASHER AND A. SCHUSTER, *The complexity of reconfiguring network models*, Theory of Computing and Systems, Springer Berlin / Heidelberg, 1992, pp. 79–90.

[16] P. RAGDE, F. E. FICH AND A. WIGDERSON, *Simulations among concurrent-write PRAMs*, Algorithmica, 3:1 (1988) pp. 43–51.

[17] R. VAIDYANATHAN AND J. L. TRAHAN , *Dynamic Reconfiguration: Architectures and Algorithms* , Plenum Publishing Co, 1 (2004), pp. 500–507.

[18] S. SAXENA, *Parallel integer sorting and simulation amongst CRCW models*, Acta Informatica, 33:7 (1996), pp. 607–619.

[19] L. STOCKMEYER AND U. VISHKIN, *Simulation of parallel access machines by circuits*, SIAM Journal on Computing, 13 (1984), pp. 409–422.

[20] A. SZEPIETOWSKI, *Turing Machines with Sublogarithmic Space*, Springer Lecture Notes in Computer Science 843, 1994.

[21] J. L. TRAHAN, R. VAIDYANATHAN, AND R. K. THIRUCHELVAN, *On the power of segmenting and fusing buses*, Journal of Parallel and Distributed Computing, 34 (1996), pp. 82–94.

[22] 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 (1990), pp. 500–507.

[23] ——, *Two-dimensional processor array with a reconfigurable bus system is at least as powerful as CRCW model*, Information Processing Letters, 36 (1990), pp. 31–36.

[24] ——, *Efficient Simulations Between Concurrent-Read Concurrent-Write Pram Models*, Information Processing Letters, 36 (1990), pp. 231–232.

[25] *Wiki.com*, http://www.wiki.com

[26] *The IBM Website*, http://www.ibm.com

[27] *The Guide to Parallel Computing*, http://ctbp.ucsd.edu/pc/html/intro4.html