Parallel Real-Time Complexity Theory

by

Stefan D. Bruda

A thesis submitted to the
Department of Computing and Information Science
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
April 2002

Copyright © Stefan D. Bruda, 2002

Abstract

We present a new complexity theoretic approach to real-time computations. We define timed ω-languages as a new formal model for such computations, that we believe to allow a unified treatment of all variants of real-time computations that are meaningful in practice. To our knowledge, such a practically meaningful formal definition does not exist at this time.

In order to support our claim that timed ω-languages capture all the real-time characteristics that are important in practice, we use this formalism to model the two most important features of real-time algorithms, namely the presence of deadlines and the real-time arrival of input data. We emphasize the expressive power of our model by using it to formalize aspects from the areas of real-time database systems and ad hoc networks.

We also offer a complexity theoretic characterization of parallel real-time computations. First, we define complexity classes that capture the intuitive notion of resource requirements for real-time computations in a parallel environment. Then, we show that real-time algorithms form an infinite hierarchy with respect to the number of processors used, and that all the problems solvable in nondeterministic logarithmic space (NLOGSPACE) can be solved in real time by a parallel machine, no matter how tight the real-time constraints are. As well, we show that, once real-time constraints are dropped, several other real-time problems are in effect in NLOGSPACE. Therefore, we conjecture that NLOGSPACE contains exactly all the computations that admit feasible ($poly(n)$ processors) real-time parallel implementations.

In the context of these results, the issue of real-time optimization problems is investigated. We identify the class of such problems that are solvable in real time, and we show that, for a large class of optimization problems, a parallel algorithm can report in real time a solution that is arbitrarily better than the solution reported by a sequential algorithm.

We also address the problem of real-time approximation algorithms. We identify problems that do not admit good approximation solutions in real time. We also show that bin packing admits a good real-time approximation algorithm.

Acknowledgments

Special thanks to my thesis supervisor, Professor Selim Akl. Without him, the research direction described in this thesis would be at best only a project. His research support is gratefully acknowledged, and is equaled only by his advice and support in countless other academic situations (from the publishing process to job applications). Many thanks again for his support and advice.

This thesis has benefited from the comments of the anonymous referees of our papers, especially the ones who reviewed our submissions to International Parallel and Distributed Processing Symposium, and to Parallel Processing Letters.

I also wish to thank Professor Kai Salomaa, Professor Henk Meijer, Professor David Rappaport, Professor Ivan Stojmenovic, and Professor Thomas Dean for comments and fruitful discussions on the material of this thesis.

Finally, I wish to express my gratitude for their emotional support to my parents as well as to all of my friends. In this respect, a truly special mention goes to Andreia and Flavius. Without them, this thesis might have been the same, but I would not.

Co-Authorship

The work presented in this thesis has been published in many papers [26, 27, 28, 29, 30, 34, 32, 33], all of them joint authored by myself (as principal author) and Selim G. Akl, my thesis supervisor.

Contents

1 Introduction **1**
1.1 Real Time in Practice: An Informal Definition 2
1.2 Formal Models of Real Time 3
1.3 The Problem 4
1.4 Towards a Solution 4
1.5 Thesis Summary 7

2 Preliminaries **10**
2.1 Notations 12
2.2 Parallel Models of Computation 16
2.3 The Data-Accumulating Paradigm 23

3 Previous Work: Theory and Practice of Real-Time Systems **33**
3.1 Real Time in Practice 34
3.2 Real-Time and On-Line Turing Machines 38
3.3 The Real-Time Producer/Consumer Paradigm 40
3.4 Real-Time Discrete Steepest Descent 41
3.5 Timed Automata 42

4 Defining Real-Time Computations: From Applications to Theory. . . **47**
4.1 Well-Behaved Timed ω-Languages 47
4.2 Accepting Timed ω-Languages 49
4.3 Operations on Timed ω-Languages 52
4.4 Sizing Up Real-Time Computations 54

5 . . . And Back [from Theory to Applications] **58**
5.1 Computing with Deadlines 59
5.2 Real-Time Input Arrival 61
5.3 Real-Time Database Systems 63
5.4 Ad Hoc Networks 70

6 Complexity of Real Time I: A Strong Infinite Hierarchy **78**
6.1 Two Processors are More Powerful than One . . . 79
6.2 The Hierarchy rt-PROC$^{\text{PRAM}}$ . . . 88
6.3 The Strong Hierarchy rt-PROC . . . 92
6.4 On Practical Issues and Why the Hierarchy rt-PROC does not Collapse . . . 93

7 Complexity of Real Time II: Logarithmic Space Computations are Real Time **96**
7.1 RMBM and NLOGSPACE Computations . . . 97
7.2 Small Space Computations Are Real-Time . . . 106

8 Complexity of Real Time III: Real Time Computations are Logarithmic Space? **109**
8.1 Non-Real-Time Pursuit Is Easy . . . 111
8.2 The Characterization of D-Algorithms . . . 113
8.3 The Characterization of C-Algorithms . . . 122
8.4 The Graph Accessibility Problem and Real Time . . . 129

9 Real-Time Characterization of Optimization Problems **132**
9.1 Independence Systems and Matroids . . . 133
9.2 A Real-Time Perspective . . . 135
9.3 Beyond Speedup, Extended . . . 138

10 On Real-Time Approximation Algorithms **143**
10.1 Real-Time Approximation Schemes and Problems not Admitting Real-Time Approximation Algorithms . . . 144
10.2 A Real-Time Approximation Scheme for Bin Packing . . . 145

11 The Characterization of Constant Time RN Computations **149**
11.1 Write Conflict Resolution Rules on RN . . . 150
11.2 Bus Width Bounds on RN . . . 151
11.3 Open Problems . . . 155

12 Conclusions and Open Problems **157**
12.1 Open Problems . . . 160
12.2 Incidental results . . . 164

Bibliography **167**

List of Figures

2.1 The reconfigurable multiple bus machine 19
2.2 Example of reconfigurable networks: (a) nondirected, and (b) directed . . . 23

6.1 PURSUIT$_1$: Insertion modulo r 80
6.2 PURSUIT$_1$: Acceptable insertion zone 83
6.3 PURSUIT$_k$: The k-dimensional circle 89

9.1 A RAM greedy algorithm for maximization problems 134
9.2 A parallel greedy algorithm for maximization problems 136

11.1 A mesh of $n \times n$ processors 152
11.2 A collection of n meshes connected together 153

Chapter 1

Introduction

To adapt an example presented in [57], consider a person driving a vehicle. The driver should perform some actions whose timing is critical to the task at hand (bringing the vehicle to the destination of the trip). For example, once a turn is approached, proper force should be applied to the steering wheel. This force should be applied *at a precise moment in time*: should it be applied too early (before entering the turn) or too late, the outcome (placing the vehicle off the road) is probably different from the desired result. We say that steering is a *real-time* task, since in this case the *timing* of the action is just as important as the action itself. By contrast, one non-real-time task may consist in filling up the gas tank of the vehicle. While it is desirable that the process be completed as soon as possible (and within a reasonable amount of time), the successful completion of the task is much more important than the timing.

The object of this thesis is the formal definition and analysis of real-time computations, that is, the computational equivalents of the driving process. Chances are, everybody has already used (directly or indirectly) a computer performing a real-time task. Consider again the automotive world: the engine of most of today's vehicles is driven by an embedded real-time computer. Various other computer driven systems, such as anti-lock braking or traction control, are becoming more and more common. In fact, the nowadays automobile contains so many electronic and electrical devices that the auto industry is in the

process of increasing the voltage standard for vehicles, from the current 12 volts to 42-volt systems [55]. Part of this extra voltage will undoubtedly be used by even more real-time computing devices. On a (both literally and technically) higher level, many modern aircraft use fly-by-wire systems, that are driven by real-time computers [56].

We have mentioned in the preceding paragraph only two examples, taken from day-to-day life. However, real-time systems do permeate the whole industrialized world, from the control of industrial processes, to global communication networks, to financial markets, to the military [21, 81]. Thus, the practice of real-time computing systems has reached a significant level of maturity.

1.1 Real Time in Practice: An Informal Definition

We used the term "real-time computation" many times, without providing a definition (save for the intuition from the first paragraph). Most computing scientists have an intuitive notion about what this concept means, but the absence of a definition is no accident: to our knowledge, the term does not have a precise, general definition. The systems literature either defines it partially (restricting the notion to the actual subdomain that is considered in the corresponding paper), or does not define it at all (for instance, one textbook on the subject [58] goes directly to describe the components of a real-time system and various design issues—no definition is ever provided).

Indeed, the domain of real-time systems is very complex, with requirements varying from application to application. For example, while in some applications the real-time component is the presence of deadlines imposed upon the computation, other applications require that input data are processed as soon as they become available, with more data to come while the computation is in progress. Variants (and combinations) of these two main requirements are often present. This complexity of the domain is probably best captured in a recent textbook on the subject [57], that candidly opens with the following statement: "After writing a book on real-time systems you might think that we would be able to give

you a precise, cogent statement of what a real-time system is. Unfortunately, we cannot."

An informal definition of a general, practical real-time system is actually possible. By putting the existing partial definitions together, one can easily reach the following conclusion[1]: The correctness of a classical (i.e., non-real-time) system is given by its ability to report the correct result for any set of input data available at the beginning of the computation. By contrast, a real-time system is considered to work correctly if it produces the correct result, *and* all the time restrictions (on input, output, or both) are met. In addition, time in a real-time system is measured in "real" (or human) time units (i.e., seconds, milliseconds, hours, ...) instead of machine units (i.e., processor cycles). In other words, a real-time system is essentially nothing more than a normal computing system, except for the notion of a correct computation, where the (human notion of) *time* becomes an inherent part.

1.2 Formal Models of Real Time

By inspecting the existing body of literature on the practice of real-time systems design, one can notice that all the real-time specifications conform to the definition presented above. One would therefore expect that the existing formal models conform, even partially, to the same definition. This is, unfortunately, not always the case. Indeed as we shall see in detail in Chapter 3 on page 33, most formal models (such as the real-time Turing machine [97]) revert to the machine sense of time. By contrast to practitioners, theorists tend to define real time as *on-line* and/or *linear time*[2]. Attempts at articulating a comprehensive definition for real time in general have also been made recently: An algorithmic definition of real-time computations is proposed in [4]. Although time is still taken in a machine sense, real time is no longer taken as a synonym for linear time. This is to our

[1] We anticipate a bit here. A detailed discussion on the matter is provided in Chapter 3 on page 33.

[2] When processing one input datum, an *on-line* algorithm does not know any of the subsequent input data (as opposed to an *off-line* algorithm, that has access to the entire input at any time). A *linear time* algorithm requires n steps to complete the computation on any input of length n. These notions are presented in more detail in Section 3.2 on page 38.

knowledge the most complete definition of real time.

Other recent formal models (such as the real-time producer/consumer paradigm [51] or timed automata [14]) offer a realistic view of the domain, but—expectedly given the lack of a unified practical definition—are not general enough to provide a uniform abstract characterization of all real-time computations.

1.3 The Problem

In summary, we are faced with the following state of the art in the area of real-time computations: They have a very strong practical grounding in domains like operating systems, databases, and the control of physical processes. Besides these practical applications however research in the area is primarily focused on formal methods and on communication issues in distributed real-time systems. Little work has been done in the direction of applied complexity theory. This is even more evident (if possible) when parallel implementations are considered.

In fact, the limited extent of this work is emphasized by the fact that a general complexity theoretic definition for (parallel or sequential) real-time computations is missing.

1.4 Towards a Solution

This thesis is an attempt at remedying the situation described in Section 1.3. We believe that a formal model and an associated complexity theory are essential tools in the process of specification and implementation of computer algorithms. Thus, we first propose a formal definition which, in our opinion, captures all the practically meaningful aspects of real-time computations.

We should emphasize that our definition is a *complexity theoretic model*, in the following sense: We consider real-time computations (or problems), but we are not concerned with any particular algorithm that carries out the given computation. Instead, for any such a computation, our model allows for the construction of a language (i.e., decision problem)

such that the computational resources required in order to accept this language are of the same order as the resources that are required to carry out the given real-time computation. Thus, by analyzing the model (i.e., the language), one is able to determine upper and lower bounds for the resources that are necessary to successfully handle the modeled computation in the real world.

However, constructing a formal model is not our only goal: It is also our intention to use such a model as the basis for a complexity theoretic characterization of real-time systems, similar to the one that has been in place for a long time for non-real-time computations (and which is based on formal languages). As such, we shall not take an algorithmic approach (as done in [4]). Instead, our model will be based on formal languages. This way, not only we are consistent with classical complexity results, but we are able to take advantage of existing results as well.

Once the model is in place, we will build the basis of a complexity theoretic approach to parallel real-time computations.

To our knowledge, this thesis is the first to articulate a formal and general definition of real world real-time applications, which is suitable for complexity theoretic analysis. As well, this is the first time it is shown that parallel real-time computations form an infinite hierarchy, and that there exists a strong relation between the class of real-time parallel problems and a classical complexity class. The main contributions of this thesis are the following:

1. We propose, under the form of *well-behaved timed ω-languages*, a general formal definition of real time as understood by the systems community. It is our thesis that *well-behaved timed ω-languages model exactly all real-time computations*.

2. Based on this model, we offer a complexity theoretic characterization of parallel real-time computations. Specifically,

    (a) We show that parallel real-time computations form an infinite hierarchy with

respect to the number of processors used.

   (b) We show that all the classical logarithmic space-bounded computations can be successfully carried out in parallel, in the presence of however tight real-time constraints.

   (c) We present strong evidence that logarithmic space-bounded computations are in effect *exactly all* the computations that can be performed in the presence of real-time constraints.

3. We also offer an incipient discussion on practical applications of our theory.

   (a) We model as well-behaved timed ω-languages the main ingredients that give the qualifier "real-time" to some computation. We also offer formal models for practical aspects from the areas of real-time database systems and ad hoc networks.

   (b) We offer a characterization of real-time optimization problems expressible as independence systems.

   (c) We show that, in a real-time environment, parallel algorithms can offer an arbitrarily better solution not only for the minimum weight spanning tree problem as previously known [9], but for a whole class of optimization problems.

   (d) Acknowledging the existence of problems that cannot be solvable exactly in real time, we provide results in the area of approximation real-time algorithms, .

Classical complexity theory is of central concern for practitioners. Indeed, a proven lower complexity bound for some problem cannot be overcome, no matter how clever a program is. In the area of real-time systems, the current absence of such a complexity theory implies that any question related to resource allocation for, or even solvability of a real-time problem is unique, in the sense that the answer to such a question should be developed from scratch. We believe that the importance of this thesis is that it is the first

work to address this issue and thus to bridge the long standing gap between the complexity theory and the practice of real-time computations. In a nutshell, our main contribution is that we offer a common (complexity theoretic) ground to which practitioners can refer in order to get readily available answers to their questions.

1.5 Thesis Summary

The remainder of this thesis is organized as follows: We present the necessary preliminaries (including descriptions of the models of computation that are used throughout the thesis) in Chapter 2 on page 10, and we survey the existing work on defining and characterizing real-time systems in Chapter 3 on page 33.

Chapters 4 to 10 constitute the body of this work: We offer in Chapter 4 on page 47 our formal definition of real-time computations expressed as the model of *well-behaved timed ω-languages*. We present a notion of acceptance for such languages, together with the structure of an acceptor for them. Based on these languages, we also define underlying complexity theoretic notions (input size, complexity classes). In order to support our thesis that timed ω-languages model all the practically meaningful aspects of real-time computations, we show in Chapter 5 on page 58 how those ingredients that, when present, give to some problem the "real-time" qualifier (namely, computing with deadlines, and input data that arrive in real-time during the computation) can be modeled using our formalism. The expressiveness of the formalism is also emphasized in Chapter 5, by modeling important practical problems. More precisely, we consider the domain of real-time database systems, and we offer a real-time variant of the *recognition problem* from the domain of classical database systems; then, we construct a formal model of the *routing problem* in ad hoc networks.

Chapters 6, 7, and 8 are dedicated to the complexity theoretic characterization of real-time computations. Intriguingly, we find that these computations are in some sense "hard" and "easy" at the same time. On one hand, we show in Chapter 6 on page 78 that there is

no such thing as too many processors, as real-time computations form an infinite hierarchy with this respect. In addition, this hierarchy is strong, in the sense that it is invariant with respect to the model of parallel computation involved. On the other hand, once the real time restrictions are eliminated, we show that such computations pertain to a class of relatively reduced computational power (namely, NLOGSPACE, the class of logarithmic space-bounded computations). We show in Chapter 7 on page 96 that all NLOGSPACE computations can be carried out successfully in parallel, no matter how tight the time constraints are. In fact, the way the result from Chapter 7 is obtained hints towards an even tighter relation between real time and NLOGSPACE computations. We investigate in Chapter 8 on page 109 whether such a tight relation really exists, that is, whether NLOGSPACE contains exactly all computations that can be successfully carried out in real time. We offer strong evidence that this is indeed the case.

Once the complexity theoretic characterization of real time is complete, we investigate aspects of real-time computations that are closer to practice. In Chapter 9 on page 132 we focus our attention on optimization problems that can be described as independence systems. In this context, we identify the class $\mathcal{M}$ of such problems that are solvable in real time. We also extend in this chapter previous results [9], conforming to which a parallel implementation can do more than merely speed up the computation. It is known [9] that the parallel solution for the real-time minimum-weight spanning tree problem can be made arbitrarily better than the solution reported by a sequential algorithm that solves the same problem. We show that, for all practical purposes, such a property does in fact hold for *any* optimization problem in $\mathcal{M}$.

One of the consequences of the complexity theoretic characterization of real-time systems is that there are problems that are simply unsolvable in real time. One of the possible approaches to such problems is giving up in the effort to find an exact solution, and seek approximate solutions instead. We offer an incipient discussion on real-time approximation algorithms in Chapter 10 on page 143. We first use the substantial body of knowledge

on approximation algorithms for P-complete problems to identify problems that do not admit "good" approximate, real-time computable solutions. Then, we show that the bin packing problem admits approximate solutions computable in real time.

The results of Chapters 7 and 8 are derived with respect to one particular model of parallel computation. We show in Chapter 11 on page 149 that these results do hold for other parallel models as well. The thesis wraps up with Chapter 12 on page 157, that hosts our conclusions, including a discussion on the potential research directions opened by our work.

Chapter 2

Preliminaries

Summary

We present in this chapter the notations, as well as the models of (parallel and sequential) computation that will be used throughout this thesis. We do not, however, linger over well known models such as the Turing machine [61], the finite automaton [49], the Random Access Machine (RAM) [89], the Parallel Random Access Machine (PRAM) [70], or the bounded degree interconnection network (BDN) [5]. Instead, we present a very brief description of such models (provided merely for summarizing the terminology), directing the interested reader to [5, 49, 61, 70, 89]. As well, previously studied models of real-time computation are not presented here, they being the subject of Chapter 3 on page 33.

The sequential models (RAM, Turing machine) are briefly presented in Section 2.1 on page 12, while the parallel models (PRAM, BDN) are the subject of Section 2.2 on page 16. We also use models with reconfigurable buses, namely the reconfigurable multiple bus machine (RMBM) [88] and the reconfigurable network (RN) [19, 20]. The RMBM and the RN are presented in detail in Section 2.2.

We also review the *data-accumulating paradigm*, with its main two variants d-algorithms and c-algorithms. Case studies from this paradigm will be used throughout the remainder of this thesis.

A note on the models of computation and their use We shall offer a formal definition of real-time computations in Chapter 4 on page 47, and we shall use it throughout the thesis. This definition consists in two parts: First, we offer a model for real-time problems by introducing the formalism of timed ω-languages. A problem is, of course, independent of the agent that solves it. Thus, the definition of timed ω-languages does not depend on any model of computation.

Then we offer a general definition of an acceptor for timed ω-languages. In other words, we model real-time algorithms, i.e., algorithms that solve real-time problems. For such an algorithm, the input has a different semantics than a conventional input, as it also includes time constraints. The computations that are performed internally are, however, the same as the ones performed by a conventional algorithm. Thus, our definition relies on the existence of some conventional (sequential or parallel) model of computation.

The definition of acceptors is, however, general, in the sense that it is applicable to any such an underlying model. That is, based on our definition, one can construct real-time algorithms running on, say, the Turing machine just as well as real-time algorithms on the Random Access Machine, or the Parallel Random Access Machine. The power of acceptors running on different models are likely to vary with the underlying model, but the general definition can be easily particularized to the model of choice.

Therefore, in our definition of timed ω-languages and their acceptors we do not mention any particular model of computation, with the understanding that these definitions apply to any reasonable such a model. Things are different though when we begin to present complexity results (in Chapter 6 on page 78 and the subsequent ones). Indeed, such complexity results are in general different from model to model. Thus, we fix the models used throughout the thesis as follows:

- *All the results regarding sequential computations are established with respect to the Random Access Machine* (RAM).

- *Results related to parallel computations are established with respect to the reconfigurable multiple bus machine* (RMBM), *unless they are invariant with the model of choice*[1] (see, for instance, the infinite hierarchy developed in Chapter 6). The power of reconfigurable buses is needed in most of the cases, hence the use of RMBM. A limited generalization is, however, presented in Section 8.4 on page 129.

- We shall, however use various other models (such as the Turing machine or the PRAM) in order to derive intermediate results whenever we find it convenient.

2.1 Notations

Given some alphabet (i.e., finite set) A, the set of all the words of finite (but not necessarily bounded) length over A is denoted by A^*. The cardinality of $\mathbb{N}$, the set of natural numbers, is denoted by ω. It should be noted[2] that $\omega \notin \mathbb{N}$ [36]. Then, the set Σ^ω contains exactly all the words over Σ of length ω. Given two words σ_1 and σ_2, $\sigma_1\sigma_2$ denotes their concatenation. The length of a word σ is denoted by $|\sigma|$. The empty word is denoted by λ. $\mathbb{R}$ denotes the set of real numbers. Given some alphabet A, the set A^k is defined recursively by $A^1 = A$, and $A^i = A \times A^{i-1}$ for $i > 1$. For some set Σ, $\mathcal{P}(\Sigma)$ stands for the power set of Σ, that is, $\mathcal{P}(\Sigma) = 2^\Sigma$.

Given two (infinite or finite) words $\sigma = \sigma_1\sigma_2\ldots$ and $\sigma' = \sigma'_1\sigma'_2\ldots$, we say that σ' is a *subsequence* of σ (denoted by $\sigma' \subseteq \sigma$) if and only if *both* the following two conditions hold: (a) for each σ'_i there exists a σ_j such that $\sigma'_i = \sigma_j$, *and* (b) for any positive integers i, j, k, l such that $\sigma'_i = \sigma_j$ and $\sigma'_k = \sigma_l$, it holds that $i > k$ if and only if $j > l$.

We use the following standard asymptotic notations [38]: Let f and g denote functions

[1] The notion of "invariant with the model of choice" will be precisely defined in Section 2.2 on page 16.

[2] Also note that the cardinality of $\mathbb{N}$ is denoted by either ω [36] or $\aleph_0$ [83]. We chose the first variant in order to be consistent with the notation used in [14].

over natural numbers, i.e., $f, g : \mathbb{N} \to \mathbb{N}$. Then,

$$\begin{array}{lcll} O(g(n)) & = & \{f(n) \mid & \text{there exist positive constants } c \text{ and } n_0 \\ & & & \text{such that } 0 \leq f(n) \leq c \times g(n) \text{ for all } n \geq n_0.\} \\ \Omega(g(n)) & = & \{f(n) \mid & \text{there exist positive constants } c \text{ and } n_0 \\ & & & \text{such that } 0 \leq c \times g(n) \leq f(n) \text{ for all } n \geq n_0.\} \\ \Theta(g(n)) & = & \{f(n) \mid & \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \\ & & & \text{such that } 0 \leq c_1 \times g(n) \leq f(n) \leq c_2 \times g(n) \text{ for all } n \geq n_0.\} \end{array}$$

A general *finite automaton* [49] is defined as the 5-tuple $A = (\Sigma, S, s_0, \delta, F)$, where Σ is the (finite) input alphabet, S is a (finite) set of states, s_0 is the initial state, δ is the transition relation, $\delta \in S \times S \times \Sigma$, and F is the set of accepting states, $F \subseteq S$. The accepting condition for a finite automaton A is as follows: If at the end of the input string, A is in some state from F, then the input is accepted. Otherwise, the input is rejected.

The sequential model used throughout this thesis is the *Random Access Machine* (RAM) [5, 70, 89]. A RAM consists in a set of registers (memory), and a processing unit (processor). Each register is bounded in size by $O(\log n)$ for any input of size n, but the memory (i.e., the number of registers) is unbounded. However, any successful computation must use only a finite amount of memory. The processor can read/write data from/to the memory, and perform elementary operations on data. We assume that the *restricted arithmetic instruction set* is available to the processor [70]. That is, a *time unit* is the time required by the processor to perform one addition, one subtraction, or one arbitrary length shift.

Sometimes, we will use the *Turing machine* as an intermediate model. For example, we may simulate the computation performed by a Turing machine on some other model. Such an approach is considered in Chapter 7 on page 96, where, rather than simulating a RAM computation, we find it more convenient to work with the graph structure (and the associated graph accessibility problem) formed by the configurations of a Turing machine. As well, it is sometimes the case that the class of problems we want to analyze (e.g., the class of on-line algorithms) is formally defined in the literature only in terms of Turing machines. Then, the results related to this class are also stated in terms of Turing machines (such a case is considered in Section 8.2.1 on page 113).

This, however, does not reduce the generality of our results. Indeed, the class of logarithmic space bounded computations, that we investigate in Chapter 7, is the same as the class of space bounded computations on the RAM [52]. Then, the idea of on-line algorithms is easily extended from Turing machines (that we use throughout Section 8.2.1) to the RAM. Finally, RAMs and Turing machines can simulate each other with polynomial overhead on the running time [89]. As a consequence, the results presented in this thesis with respect to the RAM are easily extended to Turing machines.

For some constant k, $k \geq 1$, a k-tape Turing machine is tuple $M = (K, \Sigma, W, \delta, s_0)$, where K is the (finite) set of states, s_0 is the initial state, $s_0 \in K$, Σ and W are the input and working alphabets, respectively. M has one, read-only input tape, and k working tapes.

The state transition function δ is defined as follows: If $\delta : (K \times \Sigma \times W^k) \rightarrow (K \cup \{h\}) \times (W \cup \{R, L\})^k \cup \{R, L, \lambda\}$, then M is deterministic. If, on the other hand, $\delta : (K \times \Sigma \times W^k) \rightarrow \mathcal{P}((K \cup \{h\}) \times (W \cup \{R, L\})^k \cup \{R, L, \lambda\})$, then M is nondeterministic. The halting state h is not a member of K.

A *configuration* of a k-tape Turing machine is a $(k + 2)$-tuple of the form $(q, x_\iota \underline{a_\iota} y_\iota, x_1 \underline{a_1} y_1, \ldots, x_k \underline{a_k} y_k)$, where q is a state, $x_\iota \underline{a_\iota} y_\iota \in \Sigma^*$ is the content of the input tape, and for any i, $1 \leq i \leq k$, $x_i a_i y_i$ is the content of the i-th working tape. For $i \in \{\iota\} \cup \{j | 1 \leq j \leq k\}$, a_i is the symbol that is currently scanned by the head of tape i (the head of the input tape if $i = \iota$). If a configuration C_1 yields another configuration C_2 by exactly one application of δ, we write $C_1 \vdash_M C_2$ (with the subscript M often omitted when understood from the context). As usual, $\vdash_M^*$ denotes the transitive and reflexive closure of $\vdash_M$. A Turing machine M is said to accept some language L if, for any input string w, $(s_0, w, \lambda, \ldots, \lambda) \vdash_M^* (h, w, x_1, \ldots, x_k)$ if and only if $w \in L$, for some $x_i \in W^*$, $1 \leq i \leq k$.

Many variants of the above definition of Turing machine exists, but they do not substantially affect the computational power of the model [61, 65]. For instance, it is sometimes convenient to consider Turing machines with only one tape; the input is placed on the (sole) tape at the beginning of the computation. This tape is read-write and is used as

working tape during the computation.

The notation $poly(n)$ expresses the upper bound for polynomial functions of one variable n, that is, $poly(n) = n^{O(1)}$. Given some total function $f : \mathbb{N} \rightarrow \mathbb{N}$, we denote by SPACE($f(n)$) [NSPACE($(f(n))$)] the set of languages that are accepted by a deterministic [nondeterministic] Turing machine which uses at most $O(f(n))$ space (not counting the input tape) on any input of length n. LOGSPACE [NLOGSPACE] is a shorthand for SPACE($\log n$) [NSPACE($\log n$)]. The class P [NP] contains exactly all the languages accepted in deterministic [nondeterministic] polynomial time. Finally, NC denotes the class of languages accepted in polylogarithmic time by some parallel machine using $poly(n)$ processors. Given some class C of languages (that is, boolean functions) and some (non-boolean) function f, we say by abuse of notation that $f \in C$ whenever the extension from language to function does not alter the complexity of computation.

The most common measure of the performance of a parallel algorithm is the *speedup* [5, 82]. Given a p_1-processor algorithm A_{p_1} for some problem π, $p_1 > 0$, let $\mathcal{A}$ be the *best* known sequential algorithm that solves π. Then, the speedup of A_{p_1} is the ratio $S(1, p_1) = T_\pi(1)/T_\pi(p_1)$, where $T_\pi(x)$ is the running time of the x-processor algorithm that solves π ($\mathcal{A}$ when $x = 1$). Note, however, that, using the above definition of speedup, one cannot directly compare the performance of two parallel algorithms. Speedup is therefore an absolute measure. This definition was extended to a relative measure in [12] where, given two parallel algorithms A_1 and A_2 which use p_1 and p_2 processors, respectively, $p_1 > p_2 > 1$, the speedup of A_1 over A_2 is $S(p_2, p_1) = T_\pi(p_2)/T_\pi(p_1)$. In the following we refer to the original definition each time when we write $S(1, p)$, and to the modified definition when we write $S(p', p)$, $p' \neq 1$. We also imply the original definition when we refer to "speedup" without any further clarifications.

In a conventional environment, the speedup (and hence the efficiency) of a parallel algorithm is bounded by the following results: The *speedup theorem* [82] states that, for any problem admitting a sequential algorithm and a p-processor parallel algorithm, $p > 1$,

it holds that $S(1, p) \leq p$. Conforming to *Brent's Theorem* [24], if a computation π can be performed with p processors in time $T_\pi(p)$ and with q processors in time $T_\pi(q)$, where $q < p$, then $T_\pi(p) \leq T_\pi(q) \leq T_\pi(p) + pT_\pi(p)/q$.

2.2 Parallel Models of Computation

For some $p > 1$, we consider that any (p-processor) parallel model of computation M meets the following minimal requirements:

- M has p RAM processors.
- M has access to $\Omega(p)$ registers (the memory of M).
- Each processor can access $\Omega(1)$ registers in constant time.
- Each processor can access any register from the memory of M in finite (but not necessarily bounded) time.

One can easily notice that all the models presented below (PRAM, BDN, RMBM, etc.) conform to the above description. Indeed, almost any reasonable model of parallel computation does conform to this definition, as a parallel machine is in effect a collection of sequential machines connected together. One apparent exception is the *combinational circuit* [38, 70]. However, such exception in only apparent, as it is well known that the complexity classes of combinational circuits have exact counterparts expressed in terms of parallel machines that meets the constraints presented above (see, for example, [5, 45, 70]).

One other exception to the above description is the *processor farm* [93, 96]. In a processor farm, tasks are distributed ("farmed out") by one "farmer" processor to several "worker" processors, and results are sent back to the farmer [96]. The "worker" processors do not communicate with each other, thus invalidating one of the requirements specified above. One may argue that the parallel machine on which a processor farm is implemented must have interprocessor communication capabilities, since the tasks are distributed and the

results are collected back. In other words, the lack of interprocessor communication in the processor farm paradigm is an algorithmic choice rather than a machine-level one. In any case, we shall exclude the processor farm paradigm from any subsequent discussions.

Therefore, we say without loss of generality that any parallel machine (with the exclusion of the processor farm) meets these minimum requirements. In particular, when we state results that hold for any model of parallel computation (e.g., the results established in Chapter 6 on page 78), we simply say henceforth that such results are *invariant to the parallel model used*, with the implicit assumption that they hold for any model that meets the requirements stated at the beginning of this section.

One of the most convenient models of parallel computation is the *parallel random access machine* (PRAM). Such a machine consists in p RAM processors that have access to a shared storage, forming thus a tightly coupled system. The other extreme is the *bounded degree interconnection network* (BDN), where the p RAM processors have their own storage space, and interprocessor communication is accomplished by an exchange of messages, transmitted through a sparse network connecting the processors. We assume that the reader is familiar with the PRAM and BDN. Therefore we do not define the terms that are usually covered in a textbook on such a subject (e.g., [5, 70]).

However, a third class of parallel models, featuring *reconfigurable buses*, is less known. We thus review in what follows this class. Two main models with reconfigurable buses have been developed in the literature: the *reconfigurable network* (or RN for short) [19, 20] and the *reconfigurable multiple bus machine* (or RMBM) [88]. While both models have similar characteristics, the RMBM features a clear separation between buses and processors.

We shall primarily use in Chapters 7 on page 96 and 8 on page 109 the RMBM (but we shall refer to RN as well). The reason for such a choice of computational model (with reconfigurable buses) is the fact that the concept of reconfigurable buses is both powerful (our constructions do use the power of reconfiguration) and feasible [20, 44, 88] at the same time.

We assume without loss of generality that the sequential equivalent of *all* the parallel models is the RAM. That is, any p-processor parallel machine (PRAM, RMBM, RN, etc.) becomes a RAM whenever $p = 1$. Indeed, issues that may or may not be relevant for some particular model (write or read conflicts, bus configuration, etc.) no longer apply to a machine using one processor. Thus, any such a machine can be considered equivalent to the RAM.

The following description of RMBMs closely follows the one provided in [34], that, in addition to [88], defines the concept of uniform family.

The reconfigurable multiple bus machine An RMBM [87, 88] consists a set of p (RAM) *processors* and b (electronic, nondirectional) *buses*. For each processor i and bus b there exists a *switch* controlled by processor i. By these switches, a processor has access to the buses by being able to read or write from/to any bus. As well, a processor may be able to *segment* a bus, obtaining thus two independent, shorter buses. Any processor is allowed to *fuse* any number of buses together by using a *fuse line* perpendicular to and intersecting all the buses. A fuse line can be electrically connected to any number of buses, simultaneously. Two buses that are connected to the same fuse line are said to be fused, and act as a unique, longer bus.

DRMBM, the *directed* variant of RMBM [88], is identical to the nondirected model (in particular, the buses continue to be nondirectional), except for the definition of fuse lines. In a DRMBM, each processor features two fuse lines (*down* and *up*) perpendicular to and intersecting all buses. At the processor's control, 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 $l \geq j$ [$l \leq j$]. It is argued [87] that the fuse lines must use active components anyway, such that a directional connection is as practically realizable as a nondirectional one.

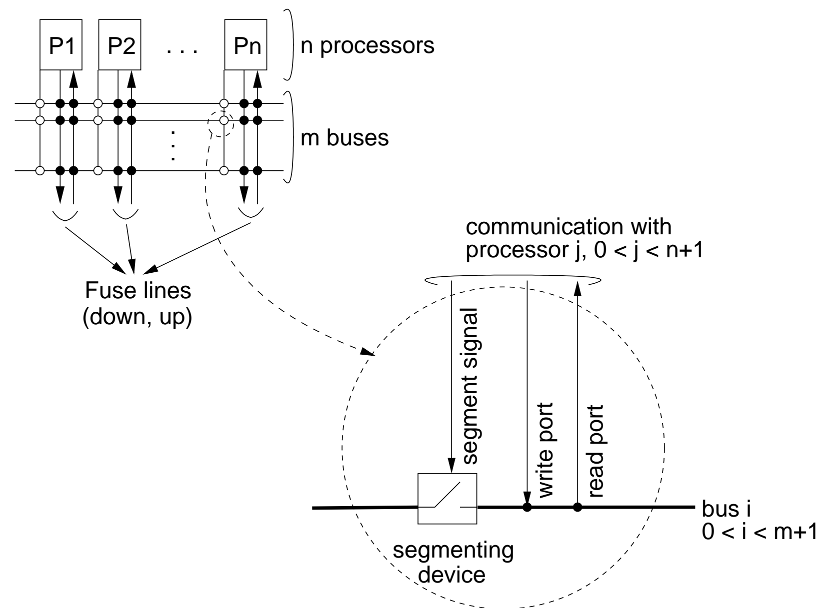

Figure 2.1: The reconfigurable multiple bus machine

For ease of presentation, one can consider RMBM as a special case of DRMBM, in which the up and down fuse lines are "synchronized," in the sense that the down fuse line of some processor p_i is connected to some bus j if and only if the up fuse line of p_i is connected to bus j. We shall adopt in the following this uniform characterization, and thus we assume that each processor in any RMBM variant has two (up and down) fuse lines, even if these fuse lines may in fact act as one bidirectional line. Furthermore, as we shall emphasize below, it is clear from this construction that, for any nondirectional RMBM there exists a DRMBM simulating it, that uses the same amount of resources (time, processors, buses, bus width).

If some RMBM [DRMBM] is not allowed to segment buses, then this restricted variant is denoted by F-RMBM [F-DRMBM]. Figure 2.1 illustrates the structure of a (directed or nondirected[3]) RMBM.

[3]Recall that the only difference between these two variants is that the up and down fuse lines are kept synchronized in the nondirected case.

As far as the process of reading and writing on the buses is concerned, one can distinguish between CREW (concurrent read, exclusive write) and CRCW (concurrent read, concurrent write) RMBMs. Theoretically, exclusive read, exclusive write (EREW) RMBMs are possible as well, but we shall not consider such, since we believe that the ability of all the processors to listen to a common bus is a trivial feature (that is, some extra effort in order to insure exclusive read appears to be necessary).

For CRCW (concurrent read, concurrent write) RMBMs, one should establish a conflict resolution rule for the process of writing a value to some bus. The most realistic such a rule is Collision (indeed, such a technique is widely used nowadays in the MAC network layer protocols, like CSMA-CS from which the Ethernet protocol is derived [85]), where two values simultaneously written on a bus result in the placement of a special collision value on that bus. Other conflict resolution rules (used for either RMBM or other models of parallel computation) are Common (two processors are allowed to simultaneously write on the bus only if the values written by them are identical), Arbitrary (some arbitrary processor succeeds in writing on the bus and the write request of all the others are discarded), Priority (the write request of the highest priority processor is the only one to succeed), and Combining (a combination of the values written by all the processors is placed on the bus). The use of the latter three rules for a bus (i.e., a spatially distributed resource) is indeed questionable. We will, however, consider all these possible rules. On one hand, this is done for completeness reasons. On the other hand, these rules are in fact equivalent, at least for the computational settings we are interested in (directed RMBMs with constant running time), as we shall show in Corollary 7.8 on page 105. We restrict only the Combining mode, requiring that the combining operation be associative and computable in nondeterministic linear space (NSPACE(n)). We believe that these are reasonable restrictions, as they clearly hold for any reasonable combining operation.

As for most models of computation, the word size of each processor in a [D]RMBM is limited to $O(\log n)$ [88]. Furthermore, we are interested in constant time computations.

Thus, we can assume without loss of generality that a processor has only a constant number of internal registers (indeed, even if there are an infinite number of registers, a processor can access only a constant number of them given the time restrictions). It follows that the *internal configuration* or *internal state* c_i of some processor p_i (which contains the content of p_i's registers and the state of p_i's finite control) in an RMBM can be expressed by a word of size $O(\log n)$. For similar reasons ($O(\log n)$ word size and constant running time) and by information theoretic arguments, it follows that, at any given time, one can fully describe which buses are fused together or segmented by a given processor, using a word of size $O(\log n)$. These limitations can be formally captured by introducing the concept of *uniform family* of RMBMs, similar to the concept of RN family [19].

An RMBM [DRMBM, F-DRMBM, etc.] *family* $\mathcal{R} = (R_n)_{n\geq 1}$ is a set containing one RMBM [DRMBM, F-DRMBM, etc.] construction for each $n > 0$. A family $\mathcal{R}$ solves a problem π if, for any n, R_n solves all inputs for π of size n.

A description of some [D]RMBM family using $p(n)$ processors and $b(n)$ buses is a list of $p(n)$ tuples $(i, c_i, up_i, down_i, segment_i)$, $1 \leq i \leq p(n)$. Such a tuple describes the configuration of processor p_i. Specifically, c_i denotes the internal configuration of p_i, and up_i [$down_i$, $segment_i$] represents a set of rules that determine which buses are fused by the up fuse line [fused by the down fuse line, segmented], depending on c_i. In the case of F-RMBM or F-DRMBM, the set $segment_i$ is always empty (no buses are ever segmented).

We say that some RMBM family $\mathcal{R}$ is a *uniform RMBM family* (or that $\mathcal{R}$ is *uniformly generated* in SPACE$(\log p(n) \times b(n))$) if there exists a Turing machine M that, given n, produces the description of R_n using $O(\log p(n) \times b(n))$ cells on its working tape. Since we deal only with uniform families here, we henceforth drop the “uniform” qualifier, with the understanding that any RMBM family described in this thesis is uniform.

Assume that some family $\mathcal{R} = (R_n)$ solves a problem π, and that each R_n, $n > 0$, uses $p(n)$ processors, $b(n)$ buses, and has a running time $t(n)$. We say then that $\pi \in$ RMBM$(p(n), b(n), t(n))$ [or $\pi \in$ F-DRMBM$(p(n), b(n), t(n))$, etc.], and that $\mathcal{R}$ has *size*

complexity $p(n) \times b(n)$ (it is customary [67, 88] to consider the size of a network as being the product between the number of processors and the number of buses) and *time complexity* $t(n)$.

It should be noted that, as shown above, a directed RMBM can simulate a nondirected RMBM by simply keeping all the up and down fuse lines synchronized with each other:

Observation 1 For any $x, y, z : \mathbb{N} \to \mathbb{N}$, $X \in \{\text{CRCW}, \text{CREW}\}$, and $Y \in \{\text{F-}, \lambda\}$, it holds that $X\,Y\text{RMBM}(x(n), y(n), z(n)) \subseteq X\,Y\text{DRMBM}(x(n), y(n), z(n))$

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. It is immediate that the bus width of any RMBM from an RMBM family is upper bounded by $O(\log n)$.

The reconfigurable network An RN [19] is a network of processors that can be represented as a connected graph whose vertices are the (RAM) processors and whose edges represent fixed connections between processors. Each edge incident 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. 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. The class $\text{RN}(p(n), t(n))$ [$\text{DRN}(p(n), t(n))$] 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.

Figure 2.2 on the next page is a graphical example of nondirected (Part (a)) and directed (Part (b)) reconfigurable network.

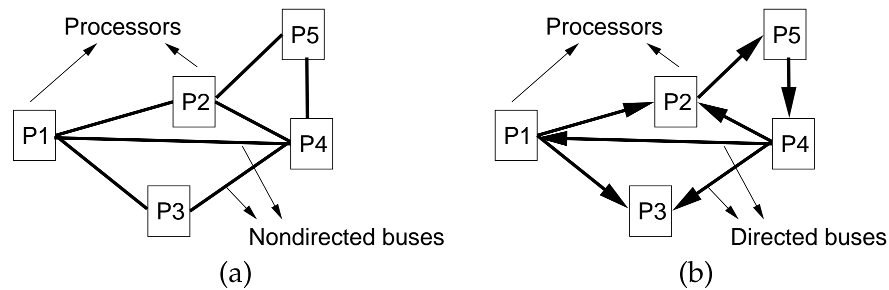

Figure 2.2: Example of reconfigurable networks: (a) nondirected, and (b) directed

2.3 The Data-Accumulating Paradigm

The data-accumulating paradigm was introduced in [62] and further studied in [27, 63, 64]. The two main variants of this paradigm are d-algorithms and c-algorithms. They were formally defined in [28] and [30], respectively. The following summary conforms to [28, 30], with some terminological modifications that clear up the subsequent presentation.

2.3.1 D-Algorithms

An algorithm for which the input data arrive while the computation is in progress, and the computation terminates when all the currently arrived data have been treated, is called a *data-accumulating algorithm* (d-algorithm for short) [63]. Formally,

Definition 2.1 An algorithm A is a d-algorithm if

1. A works on a set of data which is not entirely available at the beginning of computation. Data come while the computation is in progress (conforming to some specified data arrival law), and A terminates when all the currently arrived data have been processed before another datum arrives.

2. For any input data set, there is at least one data arrival law ϕ such that, for any value of n, A terminates in finite time, where ϕ has the following properties: (a) ϕ is increasing with respect to t, and (b) for any $n > 0$, $\phi(n, C(n)) > n$, where $C(n)$ is the

complexity of A, $C(n) > 0$. Moreover, A immediately terminates if the initial data set is null ($n = 0$).

The first condition in Definition 2.1 is implicitly given in [63]. The second condition means that A stops for some increasing data arrival law, such that *at least one* new datum arrives before A finishes the processing of the initial set of n data. If this condition is not stated, then any algorithm A_1 may be considered a d-algorithm.

The size of the set of processed data is denoted by N. The data arrival law is denoted by $\phi(n, \mathbb{t})$, where n denotes the number of input data available at the beginning of the computation, and $\mathbb{t}$ denotes the time. That is, $\phi(n, \mathbb{t})$ denotes the amount of input data that are accessible to the d-algorithm at any time $\mathbb{t}$. Moreover, we have the constraint $\phi(n, 0) = n$, since n denotes the size of the initial input. Given the termination condition, if we denote by t the running time of some d-algorithm, then $N = \phi(n, t)$.

The form proposed in [63] for the data arrival law is

$$\phi(n, \mathbb{t}) = n + kn^{\gamma}\mathbb{t}^{\beta}, \tag{2.1}$$

where k, γ, and β are positive constants. Such a form of the arrival law is rather flexible, and its polynomial form eases the reasoning about algorithms that use it. In particular, note that, when $\gamma = 0$, the amount of data that arrive in one time unit is independent of the size of initial data set. If $\beta = 1$, then the data flow is constant during the time, while in the case $\beta > 1$ the flow of data actually increases with time. Similarly, when $\beta < 1$, fewer and fewer data arrive as time increases.

Definition 2.2 Consider a given problem π, and let A be a d-algorithm for π, working on a varying set of data of size N. Consider now an algorithm A_s that performs the same computations as A, except that A_s works on the N data as if they are available at time 0. Then, if A and A_s use the same number of processors, A_s is called the *static version* of A.

Note that this definition of the static version of a d-algorithm is somehow different from the definition presented in [63]. Indeed, the original definition [63] is similar with

Definition 2.2, except that "perform the same computation" is replaced by "solves the same problem," and, in addition, A_s is the best known algorithm that solves the given problem. However, as we shall note shortly, such a definition is of little relevance, since there are d-algorithms whose performance is inherently worse than in the static case as defined in [63]. Therefore we chose the above definition, as suggested in [28]. In addition, this definition allows us to define the time complexity of a d-algorithm in a more elegant manner, as we shall see below.

Example 2.1 In order to make a clear distinction between the two definitions of the static version, let us consider the problem of sorting a sequence of numbers. Consider then the d-algorithm A that solves this problem and performs the following computations: Sort by some efficient method the initial data set, putting the result in an array; then, insert each datum that arrive into the sorted array. Conforming to Definition 2.2, the static version A_s of A is an algorithm that receives the amount of data processed by A, sorts what was the initial sequence for A, and then inserts each additional datum into the already sorted array. Considering that $N > n$, the time complexity of A_s is $\Theta(N^2)$. On the other hand, the static version of A as defined in [63] takes the whole sequence of length N and sorts it using some efficient method. The complexity is now $\Theta(N \log N)$. ∎

It is important to emphasize the difference between the time complexity and the running time in the data-accumulating paradigm. In the static case, the time complexity $C(n)$ of some algorithm A_s on input w, $|w| = n$, is defined as the running time of A_s. Such a complexity is a function of n, and there is no difference between the notions of time complexity and running time. We define the time complexity in the data-accumulating case in a similar manner, that is, as a function of the size N of the processed input.

Definition 2.3 Given some d-algorithm A, some data arrival law ϕ, and some initial data set of size n, suppose that A terminates at some time t, where t depends on both n and ϕ. We call t the *running time* of A.

Given the amount of data N that is processed by A, the *time complexity* $C(N)$ of A (or just *complexity* for short) is defined as the termination time of A_s, expressed as a function of N, where A_s is the static version of A.

Note that Definition 2.3 is valid in both the sequential and parallel cases. However, for clarity, we use the notations t and $C(N)$ for the sequential case. On the other hand, when explicitly referring to the parallel case, we add the subscript p. That is, when speaking of a parallel d-algorithm, we denote its running time by t_p and its (time) complexity by $C_p(N)$.

By contrast to the static case, in the case of d-algorithms the time complexity and the running time are different from each other. Indeed, consider some d-algorithm A, working on some input where the size of the initial data set is n and new data arrive while the computation is in progress, according to some arrival law ϕ. Starting from the analysis of the static version A_s of A, one can easily compute the complexity $C(N)$ of A. However, the complexity is not a very useful measure, since N itself is a function of time. The running time is obtained by solving an implicit equation of the form $t = C(N)$. Similarly, the parallel time complexity is different from the parallel running time.

In general, as shown in [63], in the case of a sequential d-algorithm, the running time is given by the solution of the following implicit equation:

$$t = c_d(n + kn^{\gamma}t^{\beta})^{\alpha}, \tag{2.2}$$

where the static counterpart of the d-algorithm in discussion has a complexity of $c_d N^{\alpha}$, for a positive constant c_d. The complexity of such a d-algorithm is also $c_d N^{\alpha}$. In the parallel case, we have a similar equation for the running time:

$$t_p = \frac{c_{dp}(n + kn^{\gamma}t_p^{\beta})^{\upsilon}}{S'(1,p)}, \tag{2.3}$$

conforming to a result given in [63] and improved in [28], where $S'(1, p)$ is the speedup offered in the static case by an algorithm that uses p processors, and c_{dp} is a positive constant. The parallel complexity is in this case $c_{dp}N^{\upsilon}/S'(1, p)$.

The size of the whole input data set will be denoted by N_ω. Since the input data set is virtually endless in the data-accumulating paradigm, we will consider N_ω to be either large enough or tending to infinity. When considering N_ω to be infinite, it is obvious that some d-algorithm terminates in finite time if and only if it terminates before considering the whole input data set. By abuse of notation we also say this when N_ω is considered finite (that is, we say that the d-algorithm *terminates in finite time* if and only if it terminates before considering all its N_ω input data, no matter whether N_ω is finite or not).

A Turing machine model We shall investigate the relation between d-algorithms and on-line algorithms in Section 8.2.1 on page 113. Since, to our knowledge, the only formal definitions of on-line algorithms are expressed in terms of Turing machines [48, 71, 77], we provide a Turing machine model for the class of d-algorithms as well.

We shall use the following notations: We denote by D_i the i-th datum in the input stream. The ordering is naturally defined as follows: D_j is examined before D_i is examined for the first time if and only if $i > j$. We say that an algorithm A [Turing machine M] *is able to terminate at point* k if, before visiting any $D_{k'}$, $k' > k$, it has built a solution identical to the solution returned by A [M] when working on the input set $D_1, \ldots, D_k$. Note that N (the amount of data processed by a d-algorithm) is also a termination point for that d-algorithm.

Definition 2.4 A Turing machine M which models an algorithm that is able to terminate at some point other than N_ω is the tuple (K, Σ, δ, h'), K being the (finite) set of states, Σ the (finite) tape alphabet, δ the transition function, and h' the initial state. The machine M has two tapes, as in [48]: The first tape is the (read-only) input tape, and the second one is the working tape. In addition, M is deterministic, except that it has to model the ability to terminate at some point. For this purpose, we allow a designated state h' to have two output transitions as follows: $\delta(h', x, y) = (h, x, y)$, and $\delta(h', x, y) = (q, z, u)$, where h denotes the halting state. With the above exception, δ is deterministic. Moreover, no other

state is allowed to go directly to h. That is, the halting state h is replaced by an "optional halting" one (namely, h'). Note that the optional halting state h' is also the initial state.

Definition 2.4 clearly models a d-algorithm, less the real-time characteristics of the data arrival law. More precisely, the algorithm A corresponding to such a machine M can terminate before the whole input is considered, namely, when M enters the state h'. Once in h', M's choice of halting or continuing to work models the ability of A to terminate eventually when it is able to output a solution for the currently arrived data and there is no arrived but yet unprocessed datum. Note that it is required that the state h' be entered at least once before the end of input data in order for A to be considered a d-algorithm (since, conforming to Definition 2.1 on page 23, there is at least one data arrival law for which A terminates, and this termination is modeled by the nondeterminism of h'). Since a d-algorithm should immediately terminate on an empty initial input, we impose h' as the initial state.

Generally, we assume that any algorithm (whether or not modeled by such a machine M) eventually terminates after considering all its input data. That is, when N_ω is finite, M's initial state h' is reached again some time after M visits all the data on the input tape.

2.3.2 C-Algorithms

A *correcting algorithm* (c-algorithm for short) is defined in [63] as being an algorithm that works on an input data set of n elements, all available at the beginning of computation, but $V(n, \mathbb{t})$ variations of the n input data occur with time. For the sake of consistency with the analysis of d-algorithms, we denote by $\phi(n, \mathbb{t})$ the sum $n + V(n, \mathbb{t})$. We call the the function V the *corrections arrival law*. More precisely, given some time $\mathbb{t}$ and some initial input of size n, the quantity $V(n, \mathbb{t})$ represents the number of corrections that arrived in the time interval $[0, \mathbb{t}]$. We consider that $V(n, 0) = 0$. That is, no corrections are present at time 0.

We propose a definition of a c-algorithm that is similar to Definition 2.1 on page 23 of

a d-algorithm.

Definition 2.5 An algorithm A is a c-algorithm if

1. A works on a set of n data which is available at the beginning of computation. However, $V(n, \mathbb{t})$ corrections to the initial input data occur with time, and A terminates at some time t when all the corrections received up to time t have been processed.

2. For any input data set, there is at least one corrections arrival law V such that, for any value of n, A terminates in finite time, where V has the following properties: (a) V is increasing with respect to t, and (b) for any $n > 0$, $V(n, C(n)) > 0$, where $C(n)$ is the complexity of A, $C(n) > 0$. Moreover, A immediately terminates if the initial data set is null ($n = 0$).

The rationale behind this definition is the same as in the d-algorithm case: The first item is the definition introduced in [63], while the second one implies that there is at least one corrections arrival law such that a correction occurs before the algorithm in question has finished processing the initial input data.

Note that algorithms that correct themselves when their input changes have been studied [73, 76], but they do not fall within the theory of c-algorithms, since this theory assumes a real-time component, namely the corrections arrival law. We will call such algorithms, where the corrections arrival law is not considered, *dynamic algorithms*[4]. The reader should not confuse the notion of c-algorithm (or d-algorithm) with the notion of dynamic algorithm. The former notion assumes a real-time arrival law for the input and a specific termination condition, while a dynamic algorithm has neither real-time restrictions on the input, nor special termination rules. The terminology is indeed confusing, but these terms were already introduced elsewhere [63, 73, 76], hence we will use them as they are.

It remains to define the form of a correction in order to complete the definition of a c-algorithm. Generally, we consider that a correction consists of a tuple (i, v), where i

[4]Sometimes called "incremental algorithms."

denotes the index of the datum that is corrected, $1 \leq i \leq n$, and v is the new value for that datum.

An important consideration regards the quantity in terms of which the complexity analysis is expressed. It is not reasonable to give complexity results with respect to n only, since processing the corrections generally takes an important part of the computation time (it is consistent to assume that most of the data that are corrected are already considered; therefore, a c-algorithm will have at least to insert the new (corrected) datum into the solution; however, such an algorithm may also have to remove the effects of the old value). Thus, it seems more consistent to consider the quantity $\phi(n,t)$ as the basis for complexity analysis, where t denotes the termination time. We will denote $\phi(n,t)$ by N, as in the case of d-algorithms.

The size of the whole input data set (including the corrections) will be denoted by N_ω. Since the input data set is virtually endless in the data-accumulating paradigm, we will consider N_ω to be either large enough or tending to infinity, as in the case of d-algorithms.

Throughout the rest of the thesis, it is assumed that an algorithm A_u which applies one update only has a complexity of $C_u(n)$, where $C_u(n) = c_u n^\varepsilon$ and ε is a nonnegative constant. However, it may be useful sometimes to consider simultaneously a bundle of corrections of size b. Let an algorithm that performs this processing be A_u^b, of complexity $C_u^b(n,b)$. Now, considering the parallel implementation of A_u and A_u^b, we make the following assumption:

Claim 1 For some problem π solvable by a c-algorithm, let A_u be the best p-processor algorithm that considers a correction and runs on some parallel model of computation[5], and let A_u^b be the best known p-processor algorithm (running on the same parallel machine as A_u) that considers a bundle of corrections of size b. Also, let the speedup manifested by A_u and A_u^b be $S_u(1,p)$ and $S_u^b(1,p)$, respectively. Then, $S_u(1,p) = S_u^b(1,p)$.

[5]Recall that "some" parallel model of computation implicitly assumes the minimal characterization presented at the beginning of Section 2.2 on page 16, and that the sequential algorithms used for speedup comparison are RAM algorithms.

In other words, the speedup manifested by A_u is the same as the speedup manifested by A_u^b. We believe that this is a reasonable assumption, since the computation performed by A_u^b is essentially the same as the computation performed by A_u (except that there may be some initial manipulation like sorting the bundle of corrections; however, such a manipulation manifests linear speedup, and hence does not affect the overall speedup of A_u^b).

Note that a c-algorithm has to perform some initial processing, that may perform more computations than required for merely building the solution in the static case, in order to facilitate a fast update. Generally, we consider the computations that are performed before considering any correction as having a complexity of $C'(n) = cn^{\alpha}$, for some positive constants α and c.

Definition 2.6 Consider a given problem π, and let A be a c-algorithm for π, working on a varying set of data of size N. Consider now an algorithm A_s that performs the same computations as A, except that A_s works on the N data as if they are available at time 0. Then, if A and A_s use the same number of processors, A_s is called the *static version* of A.

Given some corrections arrival law V, and some initial data set of size n, suppose that A terminates at some time t, where t depends on both n and V. We call t the *running time* of A.

Given the amount of data N that is processed by A, the *time complexity* $C(N)$ of A (or just *complexity* for short) is defined as the termination time of A_s, expressed as a function of N.

The static version of a c-algorithm, as well as the running time and the time complexity, are hence defined similarly to the case of d-algorithms. Again, Definition 2.6 is valid in both the sequential and parallel cases. We denote by $S'(1, p)$ the parallel speedup for the static case.

Since this was the arrival law considered when studying d-algorithms [27, 28, 63], we

use the following corrections arrival law:

$$V(n, \mathbb{t}) = kn^{\gamma}\mathbb{t}^{\beta}, \tag{2.4}$$

with k, γ, and β as in Relation 2.1 on page 24.

Chapter 3

Previous Work: Theory and Practice of Real-Time Systems

Summary

We survey in this chapter the existing work on defining and characterizing real-time systems. We consider first the ultimately most important point of view, the industry's (Section 3.1 on the next page). Although, to our knowledge, no explicit definition of real time exists, we are able to derive one from the many implicit definitions in the literature.

We reach the conclusion that, in the systems community, "real-time" refers to those computations in which the notion of correctness is linked to the notion of time. That is, a computation is considered correct if the output is correct *and* the specified time restrictions are met, no matter whether these restrictions are imposed on input or output. In addition, real time describes a *human* rather than a machine sense of time: real time is measured in seconds (nanoseconds, hours. . .) rather than the number of steps performed by a machine.

We then consider the mainstream real-time theoretical models and their relation with practical real-time computations. In this context, we start with what is probably the oldest and most studied such a model, the *real-time Turing machine* (Section 3.2 on page 38). According to this model and related variants, we note that, when theorists say "real time" they often mean *linear time* or even *on-line*. This is clearly different from the point of view of the systems community.

We then briefly describe the *real-time producer/consumer paradigm* (Section 3.3 on page 40). We show that it cannot constitute an all-encompassing definition of real time, as it is not general enough.

Finally, we survey a third formal model for real-time computations, the *timed ω-regular languages* and timed automata (Section 3.5 on page 42). Again, this model is not general enough, but it constitutes the basis of our general definition that shall be presented in Chapter 4 on page 47.

3.1 Real Time in Practice

From a practical point of view, there are many implicit definitions of real-time computations. To our knowledge, however no explicit, general definition exists. Instead, partial definitions (restricted to the studied subdomain) are presented. The characteristic on which all these definitions agree is the presence of *deadlines*. That is, each computation in a real-time system has associated a deadline for its termination time. The result of a computation is either useless if it is reported after the deadline (and then this deadline is *hard*), or the usefulness decreases with time after the deadline passed (and we have a *soft* deadline). Variants of this definition can be found for example in [69, 92].

However, other applications add a new dimension to real-time computation. The most notable domains where such a dimension is added are industrial applications and real-time database systems. In such applications, time restrictions are imposed not only on the result of the computation, but on its input as well. Indeed, it is noted in [91] that, besides transaction deadlines a real-time database system should have a time-sensitive image of the real world, that includes the current state. That is, the information in such a database should present a state that is up to date. The same idea is emphasized in [17]. Here, the Airborne Warning and Control System (AWACS) is used as an example, and it is noted that the trajectory of an aircraft should be sampled at a rate that is upper bounded by a constant. In the same direction, a specification language for real-time systems is presented

in [16]. This language allows the specification of not only deadline constraints, but also of the maximum/minimum time allowed/required for the events, including those events that determine the input.

Real-time industrial applications have restrictions imposed on the input as well. For example, the model for such applications proposed in [58] consists in an algorithm that communicates with the external world by means of sensors (as input devices), and actuators (as output). It is mentioned that the readings of the sensors must be taken into account in an expedited manner, which is specified by limits to the temporal difference between the real world time and the last time at which the readings from sensors were considered. Similar specifications are mentioned in [86].

All the properties that were summarized above are reiterated in [90]. Although we couldn't find any explicit definition of real-time systems (not even in [90]), a closer look at the issues presented in the above paragraphs can lead to an acceptable general characterization of such systems. Indeed, the most important characteristic seems to be the existence of deadlines imposed on the output. In addition, the input may have time restrictions as well.

However, the restrictions on the input as presented above may be considered themselves output deadlines. For example, the requirement that the reading of some sensor should be taken into consideration before the former reading is 400 milliseconds old can be interpreted as a deadline for the process that interprets the sensor's reading and makes it available to the main processing algorithm. On the other hand, the point of view of the mentioned main algorithm is that it has to cope with an input stream whose data arrive at a temporal distance of no more than 400 milliseconds from each other. This is clearly a restriction on the input.

The question that appears naturally but has not been explicitly answered is: Should an algorithm that has some imposed time restrictions on the input without any explicit deadline on the output be considered real-time? We believe that the answer is affirmative,

since input restrictions usually imply output restrictions as well. In order to see this, we offer some examples.

The routing problem in *ad hoc* networks consists in transporting a message between two nodes in a collection of wireless mobile hosts, that dynamically forms a temporary network without using any existing network infrastructure or centralized administration [25, 46]. Due to the limited transmission range of such nodes, multiple hops may be needed for one node to exchange data with another. The routing problem is the main difference between an ad hoc network and a conventional one. In such a network, each mobile node acts not only as a host, but as a router as well, forwarding packets to other mobile hosts in the network, that are not within the direct reach of the sender. Furthermore, all the hosts are mobile. Therefore, the set of those nodes that can be directly reached by some host changes with time.

Take now for example a distance vector routing algorithm for ad hoc networks [25]. In such an algorithm, each node maintains a routing table listing the next hop for each reachable destination. The algorithm labels each possible route with a sequence number, and compares two possible routes using this number. On the other hand, each node in the network advertises a monotonically increasing even sequence number for itself. When some node B decides that its route to some destination C has broken, it advertises this route with a sequence number that is larger than the latest number heard from C by one, making therefore an odd number. This situation remains unchanged until B hears from C again, and changes the corresponding number to the one just heard, making it an even number again. Each node performs the computations that are required in order to keep its routing table up to date according with the information received from other nodes.

Assume now that this is not a real-time problem, since no explicit deadline is imposed. On the other hand, the direct connectivity of each two nodes change with time, and the time during which two nodes are in transmission range from each other is clearly imposed as a real-time constraint from the outside world. Should there be no real-time constraint

for the algorithm that updates the routing tables, the whole routing process may never be able to deliver packets at all, since the time taken for updates may be too large, such that, by the time the algorithm succeeds in computing the new tables, the configuration of nodes changed completely. It turns out therefore that a computation with real-time restrictions on the input has implicit constraints of similar nature on the output as well. Thus, it seems logical to include the routing problem in the family of real-time problems.

Analogously, this time from a theoretical point of view, a d-algorithm is an algorithm whose input data arrive in real time, and the output should be reported at that time when all the currently arrived data have been processed before the arrival of another datum (see Section 2.3 on page 23). Here, no real-time restriction is imposed on the output. However, we shall show in Section 8.2 on page 113 that such a setting actually imposes a real-time constraint on the output, even if such a restriction is not explicit.

For all these reasons, we conclude that an algorithm with real-time input constraints is included in the class of real-time algorithms. This notion is captured by the informal definition that we state at the beginning of this chapter (on page 33). In fact, this definition is the basis of the formal definition that we shall present in Chapter 4 on page 47, so it is worth repeating: *In the systems community, "real-time" refers to those computations in which the notion of correctness is linked to the notion of time. That is, a computation is considered correct if the output is correct* and *the specified time restrictions are met, no matter whether these restrictions are imposed on input or output. In addition, real time describes a* human *rather than a machine sense of time: real time is measured in seconds (nanoseconds, hours...) rather than the number of steps performed by a machine.*

With this definition in mind, we are now ready to go in the theoretical area. We survey the known theoretical models for real-time computations, checking at the same time whether they are expressive enough to model real-time computations as per the above definition.

3.2 Real-Time and On-Line Turing Machines

One such model is the *real-time Turing machine*, proposed for the first time in [97]. Machines belonging to this model, and the languages accepted by them, called *real-time definable languages*, were further studied in many papers, e.g., [1, 41, 72, 77, 78]. The model used is a deterministic one, but nondeterministic extensions were also studied, like the real-time Turing machines with restricted nondeterminism [42], and nondeterministic real-time Turing machines [22] (the languages accepted by the latter model being called *quasi-real-time languages*).

As noted in the at the beginning of this chapter, real time is closely related to *on-line* in the world of Turing machines. In fact, a real-time Turing machine is a further restricted on-line Turing machine.

Therefore, we take the opportunity here to define on-line algorithms as well. The notion of an *on-line* algorithm was introduced in order to define a class of algorithms for which the size of the input may be unknown at the beginning of computation. Informally, such an algorithm processes each input datum without looking ahead (to the input data that follow the current one). By contrast, an algorithm that needs to know all the input in advance is called *off-line*. Many, equivalent definitions of on-line algorithms are found in the literature [23, 50, 54]. However, to our knowledge, the only formal definitions of such algorithms are expressed in terms of Turing machines [48, 71, 77].

Formally, on-line and real-time Turing machines are introduced by the following definition [77]:

Definition 3.1 1. For some constant k, $k \geq 1$, an *on-line Turing machine* is a deterministic $(k+1)$-tape Turing machine (with k working tapes and one input tape) $M = (K_p, K_a, \Sigma, W, \delta, s_0)$, where $K_p \cup K_a$ is the set of states, not containing the halt state h, s_0 is the initial state, Σ is the input alphabet, W is the alphabet of working symbols, containing the blank symbol #, and δ is the state transition function,

$\delta : (K_p \times \Sigma \times W^k) \cup (K_a \times W^k) \to (K_p \cup K_a \cup \{h\}) \times (W \cup \{R, L\})^k$. The head on the input tape is allowed to move only to the right.

A *configuration* of an on-line k-tape Turing machine is a $(k+2)$-tuple $C = (q, t, x_1\underline{a_1}y_1, \ldots, x_k\underline{a_k}y_k)$, where q is a state, $t \in \Sigma^*$ is the (not yet considered) content of the input tape, for any i, $1 \leq i \leq k$, $x_i a_i y_i$ is the content of the i-th working tape, and a_i is the symbol that is currently scanned by the head of tape i. If a configuration C_1 yields another configuration C_2, we write $C_1 \vdash_M C_2$. As usual, $\vdash_M^*$ denotes the transitive and reflexive closure of $\vdash_M$.

The set of states is divided into two subsets: the set of *polling* states K_p and the set of *autonomous* states K_a. All the states that lead to h in one step are polling states, and the initial state is a polling state.

The form of δ is restricted such that only the following relations are possible: $(q, abv, x_1, \ldots, x_k) \vdash_M (q', bv, x'_1, \ldots, x'_k)$, $(q'', abv, x_1, \ldots, x_k) \vdash_M (q', abv, x'_1, \ldots, x'_k)$, and $(q, \lambda, x_1, \ldots, x_k) \vdash_M (h, \lambda, x'_1, \ldots, x'_k)$, with $q \in K_p$, $q'' \in K_a$, and $q' \in K_p \cup K_a$.

M accepts the input w if and only if $(s_0, w, \lambda, \ldots, \lambda) \vdash_M^* (h, \lambda, x_1, \ldots, x_k)$.

2. A *real-time Turing machine* is an on-line Turing machine for which $K_a = \emptyset$. A language accepted by such a machine is called a *real-time definable language*.

In plain English, an on-line Turing machine has a unidirectional input tape. Therefore, it has no knowledge about further input data. Between reading two input symbols, such a machine is allowed to go into a number of autonomous states, where it performs some work without considering any input. In addition to these requirements, a real-time Turing machine has no autonomous state, it being forced to consume an input datum at every step. It follows therefore that a real-time Turing machine should spend constant time only between considering any two consecutive input data. The nondeterministic extension of a real-time Turing machine is immediate:

Definition 3.2 A nondeterministic real-time Turing machine is a machine that is identical to the one defined in Definition 3.1 on page 38, except that[1] $\delta : (K_p \times \Sigma \times W^k) \cup (K_a \times W^k) \to \mathcal{P}((K_p \cup K_a \cup \{h\}) \times (W \cup \{R, L\})^k)$. The languages accepted by nondeterministic real-time Turing machines are called *quasi-real-time* languages [22].

Almost the same definition, this time in terms of algorithms rather than Turing machines, can be found in [74]. Here, a *linear time algorithm* takes $O(n)$ steps to complete on any input of length n. A real-time algorithm is a linear-time algorithm which has the additional requirement that it spends only $O(1)$ steps on any input symbol.

We note that the critical shortcoming of real-time definable languages is that a real-world application usually specifies that the computation should take less than, say, 4 seconds. This is clearly impossible to specify in terms of real-time Turing machines or their variants. To put it in another way, the human sense of time required in the systems community cannot be expressed in terms of real-time Turing machines, which model the machine sense of time. Thus, these machines are not suitable for modeling real-time applications.

3.3 The Real-Time Producer/Consumer Paradigm

Another model for real-time computations is presented in [51]. This model is based on the producer/consumer paradigm. In such a paradigm, there are two entities, a producer, that produces messages, and a consumer, that consumes the produced messages. They communicate through a buffer, that keeps those messages that were produced, but not consumed yet. Based on this model, the *real-time producer/consumer paradigm* (RTP/C) is introduced. Here, the producer produces messages at a given (real-time) rate, and the consumer must consume the messages at the rate they are produced (the buffer is thus eliminated). A real-time system is composed then by a set of such communicating processes, together with some storage space.

[1] Recall that $\mathcal{P}(()\Sigma)$ denotes the powerset of Σ.

The thesis advanced in [51] is that the RTP/C paradigm applies to a wide variety of interesting and important real-time applications, where all the data arriving from the external world must be processed in real-time. However, the concept of production rate may not be expressive enough in some cases. More precisely, given the railway crossing problem [60], the main event is the arrival of a train at the crossing, which does not happen at a specified rate (in fact, there is a possibility that the train never arrives). Another example where the RTP/C paradigm is not applicable is the case of d-algorithms (presented in Section 2.3 on page 23), where the arrival rate may vary over time. We conclude thus that this paradigm is not suitable for modeling real-time applications.

3.4 Real-Time Discrete Steepest Descent

A third, algorithmic definition of real-time computations has been developed recently in, e.g., [6, 7, 66, 67]. The most complete variant is the one presented in [4]. Conforming to this definition, input data for a real-time computation is not entirely available at the beginning of the computation, but arrives instead while the computation is in progress. Then, one or more of the following conditions may be imposed on a real-time algorithm:

- Each set of input data received simultaneously must be processed within a certain time after its arrival.
- Each output must be returned within a certain time (deadline) after the arrival of the corresponding input.

We note that, up to this point, the definition is in effect an extension of the RTP/C paradigm described in the previous section. In addition though, the input data for a real-time algorithm can depend on the processing performed by the algorithm itself. This is, indeed one of the features of real-time algorithms that is encountered in many practical applications (most notably, in those real-time algorithms controlling industrial processes). However, to our knowledge such a feature is acknowledged for the first time in [4].

By contrast to the RTP/C paradigm, the definition presented in [4] is also able to model problems such as the railway crossing [60], where input does not arrive periodically. To the best of our knowledge, this is the most comprehensive definition of real-time computations to date.

However, this definition still has some limitations that makes it unsuitable for our goal of constructing a general complexity theoretic approach to real-time computations. Most notably, the notion of time is still taken in the machine rather than human sense.

Even if we believe that changes which address these limitations can be incorporated in the model proposed in [4], we prefer to work with a model based on formal languages instead of an algorithmic model. Such a preference is purely pragmatic, as it would allow us to take full advantage of existing results from the classical complexity theory. Thus, our model will be based on the model of timed automata, that shall be described in the following section.

3.5 Timed Automata

Finally, another real-time model is based on the concept of *timed automata* [14].

The basis for the theory of timed automata is the ω-automaton. An *ω-automaton* is a usual finite state automaton $A = (\Sigma, S, s_0, \delta, F)$, whose accepting condition is modified, in order to accommodate input words of infinite length. More precisely, given an (infinite) word $\sigma = \sigma_1\sigma_2\ldots$, the sequence:

$$r = s_0 \xrightarrow{\sigma_1} s_1 \xrightarrow{\sigma_2} s_2 \xrightarrow{\sigma_3} \ldots$$

is called a *run* of A over σ, provided that $(s_{i-1}, s_i, \sigma_i) \in \delta$ for all $i > 0$. For such a run, $\inf(r)$ is the set of all the states s such that $s = s_i$ for infinitely many i.

Regarding the accepting condition, a *Büchi automaton* has a set $F \subseteq S$ of accepting states. A run r over a word $\sigma \in \Sigma^\omega$ is accepting if and only if $\inf(r) \cap F \neq \emptyset$. The acceptance of a *Muller automaton* on the other hand does not use the concept of final state. For such an

automaton, an *acceptance family* $\mathcal{F} \subseteq \mathcal{P}(S)$ is defined (recall that $\mathcal{P}(()\Sigma)$ stands for the powerset of Σ). Then, a run r over a word σ is an accepting run if and only if $\inf(r) \in \mathcal{F}$. A language accepted by some automaton (Büchi of Muller) consists of the words σ such that the automaton has an accepting run over σ.

Another ingredient of the theory developed in [14] is the *time sequence*. A time sequence $\tau = \tau_1\tau_2\ldots$ is an infinite sequence of positive real values, such that the following constraints are satisfied: (a) *monotonicity*: $\tau_i \leq \tau_{i+1}$ for all $i \geq 0$, and (b) *progress*: for every $t \in \mathbb{R}$, there is some $i \geq 1$ such that $\tau_i > t$. Then, a *timed ω-word* over some alphabet Σ is a pair (σ, τ), where $\sigma \in \Sigma^\omega$, and τ is a time sequence. That is, a timed ω-word is an infinite sequence of symbols, where each symbol has a time value associated with it. The time value associated with some symbol can be considered the time at which the corresponding symbol becomes available. A timed ω-language is a set of timed ω-words.

Finally, a *clock* is a variable over $\mathbb{R}$, whose value may be considered as being externally modified. Given some clock x, two operations are allowed: reading the value stored in x, and resetting x to zero. At any time, the value stored in x corresponds to the time elapsed from the moment that x has been most recently reset. For a set X of clocks, a set of *constraints* over X, $\Phi(X)$, is defined by: d is an element of $\Phi(X)$ if and only if d has one of the following forms: $x \leq c$, $c \leq x$, $\neg d_1$, or $d_1 \wedge d_2$, where c is some constant, $x \in X$, and $d_1, d_2 \in \Phi(X)$.

Starting from these notions, the concept of timed ω-regular languages is introduced. A *timed Büchi automaton* (TBA) is a tuple $A = (\Sigma, S, s_0, \delta, C, F)$, where C is a finite set of clocks. This time, the transition relation δ is defined as $\delta \subseteq S \times S \times \Sigma \times \mathcal{P}(C) \times \Phi(C)$. An element of δ has the form (s, s', a, l, d), where l is the set of clocks to be reset during the transition, and d is a clock constraint over C. The transition is enabled only if d is valued to true using the current values of the clocks in C.

A run r of a TBA $A = (\Sigma, S, s_0, \delta, C, F)$ over some timed ω-word (σ, τ) is an infinite

sequence of the form:

$$r = (s_0, \nu_0) \xrightarrow{\sigma_1,\tau_1} (s_1, \nu_1) \xrightarrow{\sigma_2,\tau_2} (s_2, \nu_2) \xrightarrow{\sigma_3,\tau_3} \cdots \quad (3.1)$$

where $\sigma = \sigma_1\sigma_2\ldots$, $\tau = \tau_1\tau_2\ldots$, $\nu_i \in \{f|f : C \to \mathbb{R}\}$ for all $i \geq 0$, and the following conditions hold:

- $\nu_0(x) = 0$ for all $x \in C$,
- for all $i > 0$, there is a transition $(s_{i-1}, s_i, \sigma_i, l_i, d_i) \in \delta$ such that $(\nu_{i-1} + \tau_i - \tau_{i-1})$ satisfies d_i, for all $x \in C - l_i$, $\nu_i(x) = \nu_{i-1}(x) + \tau_i - \tau_{i-1}$, and, for all $x' \in l_i$, $\nu_i(x') = 0$.

The notions of accepting run, and language accepted by a TBA are defined similarly to the case of Muller automata. A timed ω-language accepted by some TBA is a *timed regular language.*

It should be noted that, even if a timed regular language looks well suited for modeling general real-time computations, the TBA used in [14] for recognition of such languages is not sufficiently powerful for this purpose [26]:

Theorem 3.1 *There are languages formed by infinite words (ω-languages) that are not ω-regular.*

Proof. Let us consider the following language over the alphabet $\Sigma = \{a, b, c, d\}$: $L = \{a^u b^x c^v d^x | u, x, v > 0\}$. It is immediate that L is not regular. Now, consider the following ω-language: $L_\omega = \{l_1\$l_2\$l_3\$\ldots | l_i \in L$ for any $i > 0$, and $\$ \notin \Sigma\}$.

Assume now that L_ω is ω-regular. Then, there is a Büchi automaton $A = (\Sigma, S, s_0, \delta, F)$ that recognizes it. Let x be a word in L_ω, $x = x_1\$x_2\$x_3\$\ldots$. Therefore, there exists a run r of A over x such that $\inf(r) \cap F \neq \emptyset$.

In the run r, let A_i, $i \geq 1$ be finite automata constructed as follows: The initial [final] state of A_i is the state A is into immediately after parsing the symbol \$ that precedes x_i [immediately before parsing the symbol \$ that terminates x_i]. The set of states [transitions] of A_i contains exactly all the states [transitions] used by A while parsing

x_i. It is immediate that (a) A_i accepts x_i, and (b) A_i does not accept any word outside L (for assume that A_i accepts such a word x_i'; then, run r of A clearly accepts the ω-word $x' = x_1\$x_2\$\ldots x_{i-1}\$x_i'\$x_{i+1}\$\ldots$, which is not in L_ω; this contradicts our assumption that A recognizes L_ω).

Let now $\mathbb{A}_x$ be the set of all the automata A_i constructed as above for a word $x \in L_\omega$, $x = x_1\$x_2\$x_3\$\ldots$, and some accepting run r of A over x. Denote by $\mathbb{A}$ the set $\bigcup_{x \in L_\omega} \mathbb{A}_x$. Note that both the sets of states and transitions of some $A_* \in \mathbb{A}$ are subsets of S and δ, respectively, and both S and δ are finite. Therefore, the number of distinct automata $A_* \in \mathbb{A}$ is finite, i.e., $|\mathbb{A}|$ is finite. In addition, given Properties (a) and (b) above, it is immediate that (a') for any $v \in L$, there exists some $A_* \in \mathbb{A}$ that accepts v, and (b') no automaton $A_* \in \mathbb{A}$ accepts any word $v' \notin L$.

We can now construct then a finite automaton A' that recognizes L: let the initial [final] state of A' be some s' [s''], $s', s'' \notin S$; the set of states of A' is $S \cup \{s', s''\}$, and the transitions of A' are exactly all the transitions of the automata $A_* \in \mathbb{A}$, plus λ-transitions from s' to each initial state of some $A_* \in \mathbb{A}$, and from each final state of some $A_* \in \mathbb{A}$ to s''.

Clearly, A' recognizes L (given Properties (a') and (b') above and noting that we construct A' by performing a typical "parallel composition" [49] of the automata from $\mathbb{A}$). Moreover, A' is a finite automaton (given that $|\mathbb{A}|$ is finite). The existence of A' is a contradiction, since L is not regular. ∎

Corollary 3.2 *There are timed ω-languages that are not (timed) ω-regular.*

Proof. Simply attach to each word in the language L_ω some time sequence, and call the language obtained in this way L_ω'. Then, the proof by contradiction follows from the proof of Theorem 3.1 on the page before. Indeed, consider a TBA that is identical to A' from the mentioned proof, and for which $C = \emptyset$. Clearly, this TBA recognizes L_ω'. However, such an automaton is an impossibility. ∎

In passing, note that the language L_ω built in the proof of Theorem 3.1 is not uninteresting from a practical point of view. Indeed, it models a search into a database for a given key: the database is modeled by the word $a^u b^x c^v$, the key to search for is d^x, and the instance that matches the query is simulated by b^x. We just found hence some practical situation which does not pertain to the class of (timed) ω-regular languages.

Even if not powerful enough, we chose to present in a more extensive manner the formalism of timed ω-regular languages, as it is the basis of our model, that shall be defined in Chapter 4 on the following page.

We close here our survey of the existing work in the area of theoretical models for real-time computations, with the conclusion that the existing models are not suitable for a unified and realistic complexity theoretic formalization of real-time computations. We shall therefore introduce a new, expressive model that offers a consistent and practically meaningful characterization of real-time computations in Chapter 4.

Chapter 4

Defining Real-Time Computations: From Applications to Theory…

Summary

Even if the device used for recognition of timed ω-regular languages is not powerful enough to model all the real-time computations that are meaningful in practice, the notion of timed languages is very powerful. In this chapter, we introduce a model called *well-behaved timed ω-languages*, together with the structure of an acceptor for such languages. We also define the underlying notions for a complexity theory of real time based on timed ω-languages, offering definitions for a notion of input size suitable for this domain, as well as for real-time complexity classes that capture the intuitive notion of resource requirements for real-time computations in a parallel environment.

We believe that our construction captures all the practical aspects of real-time computations. That is, our thesis is that *well-behaved timed ω-languages model exactly all real-time computations.*

4.1 Well-Behaved Timed ω-Languages

Despite the limited scope of the finite state approach, the concept of timed languages is a very powerful one. We propose therefore a definition that is similar to the one in [14], but

is not restricted to finite state acceptors.

However, our presentation is clearer if we use a slightly modified concept of time sequence (we use the intuitive notion of subsequence that is formally defined on page 12):

Definition 4.1 A sequence $\tau \in \mathbb{N}^\omega$, $\tau = \tau_1\tau_2\ldots$, is a *time sequence* if it is an infinite sequence of positive values, such that the *monotonicity* constraint is satisfied: $\tau_i \leq \tau_{i+1}$ for all $i > 0$. In addition, a (finite or infinite) subsequence of a time sequence is also a time sequence.

A *well-behaved* time sequence is a time sequence $\tau = \tau_1\tau_2\ldots$ for which the *progress* condition also holds: for any $k, i \in \mathbb{N}^+ \cup \{\omega\}$ such that $\tau_k = \tau_{k+1} = \cdots = \tau_{k+i}$ it holds that i is bounded, and for every $t \in \mathbb{N}$, there exists some finite $j \geq 1$ such that $\tau_j > t$.

In should be noted that a time sequence may be finite or infinite, while a well-behaved time sequence is always infinite. In fact, a well-behaved time sequence in our terminology is similar to the concept of time sequence used in [14], except that, while time is considered dense in [14], we consider it to be discrete, since in essence the time perceived by a computer is discrete as well (besides, one can define a granularity of time as fine as desired).

Definition 4.2 A *timed ω-word* over an alphabet Σ is a pair (σ, τ), where τ is a time sequence, and, if $\tau \in \mathbb{N}^k$, then $\sigma \in \Sigma^k$, $k \in \mathbb{N} \cup \{\omega\}$. Given a symbol σ_i from σ, $i > 0$, then the associated element τ_i of the time sequence τ represents the time at which σ_i becomes available as input. A *well-behaved* timed ω-word is a timed ω-word (σ, τ), where τ is a well-behaved time sequence. A [well-behaved] timed ω-language over some alphabet Σ is a set of [well-behaved] timed ω-words over Σ.

Definition 4.2 is a natural extension of the definition of timed regular languages presented in [14], except that we added the "well-behaved" qualifier, generated by the modified terminology presented in Definition 4.1. We also take into consideration timed ω-words that are not well-behaved. Even if our thesis states that such words by themselves

do not model real-time computations, they may be useful as intermediate tools in building real-time models.

4.2 Accepting Timed ω-Languages

In light of the above definition, we can also establish the general form of an acceptor for timed languages:

Definition 4.3 A *real-time algorithm A* consists in a *finite control* (that is, a *program*), an *input tape* (that is, an *input stream*) that contains a timed ω-word, and a *decision tape* (that is, a *decision stream*) containing symbols from some alphabet Δ that are written by A. The input tape has the same semantics as a timed ω-word. That is, if (σ_i, τ_i) is an element of the input tape, then σ_i is available for A at precisely the time τ_i. During any time unit, A may add at most one symbol to the decision tape. Furthermore, the decision tape is *write-only*, that is, A cannot read any symbol previously written on the decision tape. We denote by $ans(A, w)$ the content of the decision tape of some real-time algorithm A working on some input w. A may have access to an infinite amount of working storage space (working tape(s), RAM memory, etc.) outside the input and decision tapes, but only a finite amount of this space can be used for any computation performed by the algorithm.

It should be noted that the concept of working space has the same meaning as in classical complexity theory. Like a classical algorithm, a real-time algorithm can make use of some storage space in order to carry out the desired computation. When considering space-bounded real-time computations, one can analogously consider the space used by the real-time algorithm as the amount of this storage space that is used during the computation, without counting the (content of) input and decision tapes.

Let us fix some designated symbol $\mathbb{y} \in \Delta$. The symbol $\mathbb{y}$ has the same meaning as the final state used in [14], where only those timed ω-languages accepted by finite automata are considered (see Section 3.5 on page 42). However, since the configuration of a general

machine may be hard to work with, we prefer to change the state with output on the decision tape. That is, a real-time algorithm accepts some word if and only if some designated symbol $\mathbb{y}$ appears infinitely many times on the decision tape. Formally,

Definition 4.4 A real-time algorithm A *accepts* the timed ω-language L if, on any input w, $|ans(A,w)|_{\mathbb{y}} = \omega$ if and only if $w \in L$.

It is worth mentioning that the actual meaning of the symbol $\mathbb{y}$ on the decision tape might be different from algorithm to algorithm, but such a distinction is immaterial for the global theory of timed ω-languages. Indeed, consider an aperiodic real-time computation, e.g., a computation with some deadline. If, for some particular input, the computation meets its deadline, then, from now on, the real-time algorithm that accepts the language which models this problem may keep writing $\mathbb{y}$ on the decision tape. That is, the first appearance of $\mathbb{y}$ signals a successful computation, and the subsequent occurrences of this symbol do not add any information, they being present for the sole purpose of respecting the acceptance condition (infinitely many occurrences of $\mathbb{y}$). On the other hand, consider the timed language associated with a periodic computation, e.g., a periodic query in a real-time database system. Then, $\mathbb{y}$ might appear on the decision tape each time an occurrence of the query is successfully served (obviously, a failure could prevent further occurrences of $\mathbb{y}$, should the specification of the problem require that all the queries be served). In this case, each occurrence of $\mathbb{y}$ signals a successfully served query. However, even if the actual meaning of the $\mathbb{y}$'s on the decision tape can vary from application to application, it is easy to see that the acceptance condition remains invariant throughout the domain of real-time computations.

It is assumed that the input of a real-time algorithm is always a (not necessarily well-formed) timed ω-word. That is, any real-time algorithm is fed with two sequences of symbols σ and τ, the first being a (possibly infinite) word over some alphabet, and the latter being the associated time sequence. It should be emphasized that a symbol σ_i with

the associated time value τ_i is not available to the algorithm at any time t, $t < \tau_i$.

Note that a TBA (presented in Section 3.5 on page 42) is equipped with a set of clocks, since a finite automaton does not have access to any amount of storage space. However, a TBA has to keep track of time. Thus, clocks were provided as a convenient way of achieving this. One should also note that the restricted access a TBA has to its set of clocks prevents uses of this set in other ways than for time keeping. On the other hand, such a mechanism (that is, the set of clocks) is not mentioned in Definition 4.3 on page 49. This omission is intentional. As opposed to a TBA, a real-time algorithm has access to storage space, hence it can use (part of) this storage for time-keeping purposes.

The absence of clocks implies that, by contrast to TBA, there are no time constraints on state transitions in a real-time algorithm. Such constraints are made, however, immaterial by the semantics of the input tape. Indeed, Definition 4.3 states that an input symbol is not available to the algorithm at a time smaller than the element from the time sequence associated to that symbol. We believe that this constraint is sufficient since, conforming to Definition 4.3, time restrictions are imposed by the input itself. One should note that real world real-time applications have the same property, namely that their behavior should conform to time restrictions imposed on their input and/or output rather than internal temporal constraints.

Should the need to define other classes of timed acceptors (where storage space is limited and/or the access to that storage space is restricted) arise, the set of clocks as a time keeping tool is likely to be needed again. For example, the definition of timed push-down automata can be obtained by naturally restricting Definition 4.3, but one will have to add clocks to the model, given the limited (stack-like) nature of the storage space access of such a device. We believe that such models can be easily derived. However, the intended use of our model is building a complexity theory for general real-time computations, hence we will not consider such automata.

4.3 Operations on Timed ω-Languages

Traditional formal language theory provides tools for generating new languages from existing ones, like union, concatenation, Kleene closure, etc. We therefore conclude the section that describes our model by defining equivalent operations on timed ω-languages.

The union, intersection, and complement for timed ω-languages are straightforwardly defined. Moreover, it is immediate that the language that results from such an operation on two [well-behaved] timed languages is a [well-behaved] timed language as well.

On the other hand, the concatenation is a more complex issue. Indeed, the naive operation of concatenation of two (finite) timed words (that simply concatenates together the pair of sequences of symbols and the pair of time sequences) fails to produce a timed word, since the result of the time sequence concatenation is likely not a time sequence. This naive approach is even worse in the case of well-behaved timed words, where concatenating two infinite sequences makes little sense.

However, one can rely on the semantics of timed words in defining a meaningful concatenation operation. Recall that a timed word means a sequence of symbols, where each symbol has associated a time value that represents the moment in time when the corresponding symbol becomes available. Then, it seems natural to define the concatenation of two timed words as the union of their sequences of symbols, ordered in nondecreasing order of their arrival time. Intuitively, such an operation is similar to merging two sequences of pairs (symbol, time value), that are sorted with respect to the time values. Formally, we have the following definition (we use the intuitive notion of subsequence that is formally defined on page 12):

Definition 4.5 Given some alphabet Σ, let (σ', τ') and (σ'', τ'') be two timed ω-words over Σ. Then, we say that (σ, τ) is the *concatenation* of (σ', τ') and (σ'', τ''), and we write $(\sigma, \tau) = (\sigma', \tau')(\sigma'', \tau'')$, if and only if

1. τ is a time sequence, that is, $\tau_i \leq \tau_{i+1}$ for any $i > 0$; both $(\sigma'_1, \tau'_1)(\sigma'_2, \tau'_2)\ldots$ and

$(\sigma''_1, \tau''_1)(\sigma''_2, \tau''_2)\ldots$ are subsequences of $(\sigma_1, \tau_1)(\sigma_2, \tau_2)\ldots$; furthermore, for any $i > 0$, there exists $j > 0$ and $d \in \{','' \}$ such that $(\sigma_i, \tau_i) = (\sigma^d_j, \tau^d_j)$,

2. for any $d \in \{','' \}$ and any positive integers i and j, $i < j$, such that $\tau^d_k = \tau^d_l$ for any k, l, $i \le k < l \le j$, there exists m such that, for any $0 \le \iota \le j - i$, $(\sigma_{m+\iota}, \tau_{m+\iota}) = (\sigma^d_{i+\iota}, \tau^d_{i+\iota})$, and

3. for any positive integers i and j such that $\tau'_i = \tau''_j$, there exist integers k and l, $k < l$, such that $(\sigma_k, \tau_k) = (\sigma'_i, \tau'_i)$ and $(\sigma_l, \tau_l) = (\sigma''_j, \tau''_j)$.

Given two timed ω-languages L_1 and L_2, the concatenation of L_1 and L_2 is the timed ω-language $L = \{w_1 w_2 | w_1 \in L_1, w_2 \in L_2\}$.

In addition to the mentioned order of the resulting sequence of symbols (formalized in Item 1 of Definition 4.5 on the page before), two more constraints are imposed in Definition 4.5. These constraints order the result in the absence of any ordering based on the arrival time, in order to eliminate the nondeterminism. First, if either of the two ω-words contains some subword of symbols that arrive at the same time, then this subword is a subword of the result as well, and this is expressed by Item 2 of Definition 4.5. That is, the order of many symbols that arrive at the same time is preserved. Then, according to Item 3, if some symbols σ_1 and σ_2 from the two ω-words that are to be concatenated, respectively arrive at the same moment, then we ask that σ_1 precedes σ_2 in the resulting ω-word.

The concept of Kleene closure for timed languages can be then defined based on the concatenation operation:

Definition 4.6 Given some timed ω-language L, let $L^0 = \emptyset$, $L^1 = L$, and, for any fixed $k > 1$, $L^k = LL^{k-1}$. Furthermore, let $L^* = \cup_{0 \le k < \omega} L^k$. We call L^* the Kleene closure of L.

The following result is immediate:

Theorem 4.1 *The set of [well-behaved] timed ω-languages is closed under intersection, union, complement, concatenation, and Kleene closure, under a proper definition of the latter two operations. Furthermore, a subset of a [well-behaved] timed ω-language is a [well-behaved] timed ω-language.*

Finally, we provide a way of separating the sequence of symbols and the timed sequence from a timed ω-word: For some timed ω-word $w = (\sigma, \tau)$, let $\mathrm{detime}(w) = \sigma$ and $\mathrm{time}(w) = \tau$.

4.4 Sizing Up Real-Time Computations

We are ready now for a complexity theoretic approach to real-time computations. First, we define complexity classes for timed ω-languages, that capture an intuitive notion of real-time efficiency. In the subsequent chapters, we study the relations between these classes and between them and existing complexity classes.

What to measure Classical complexity theory measures the amount of resources required for the successful completion of some algorithm. Such resources are running time, storage space, and, to a lesser degree, the number of processors used by a parallel algorithm. Let us analyze them one by one:

Time is probably the most used measure in complexity theory. On the other hand, in the real-time area, time is in most cases predetermined by the existence of deadlines imposed on the computation or by similar time constraints. Admittedly, there are classes of real-time algorithms for which running time actually makes sense as a measure of performance. One may also consider algorithms that terminate before the deadline as compared to those that terminate right when the allowed computation time expires (still, while quicker algorithms may, say, free some resources that can be used by other concurrent processes, those (slower) algorithms that still meet their deadlines are by no means less correct or useless). However, by contrast to its importance in classical complexity theory, time is

no longer a universal performance measure in the real-time environment, and thus we do not introduce real-time classes related to time complexity.

Things are different as far as *space* is concerned though. Indeed, space as a performance measure bears the same significance in a real-time environment as it does in classical complexity theory. We therefore introduce the corresponding classes rt-SPACE(f) (corresponding to SPACE(f) in classical complexity theory, that is, containing problems solvable in real time by some algorithm that uses no more than $f(n)$ working space for any input of size n). In general, we prefix all the real-time complexity classes by "rt-" (from "real-time").

A third measure of interest is the *number of processors*. In classical complexity theory this measure received less attention than the other measures. However, parallel real-time algorithms have been shown to make up for the limited time that is available, and solve problems that are not solvable by sequential implementations [8, 30, 63]. Thus, we consider the classes rt-PROC(f), containing those problems solvable in real time by a parallel machine that uses $f(n)$ processors for any input of size n. We also note an additional issue regarding the number of processors. Indeed, consider the PRAM model, as opposed to, say, the bounded degree interconnection network. In the first case, communication between two processors is accomplished by writing to, and reading from the shared memory, at a time cost equal to the cost of accessing a memory cell. By contrast, in an interconnection network, interprocessor communication uses message passing. Such a communication may involve many steps (if the two communicating processors are not directly connected), at an increased temporal cost. It is therefore reasonable to consider that the class $rt - \text{PROC}(f)$ is different from model to model. Therefore, given a model of parallel computation M, we denote the corresponding rt-PROC(f) class by rt-PROC$^M(f)$, with the superscript often omitted when either the model is understood from the context, or the class is invariant to the model (in the sense described on page 17 and assumed throughout the thesis). We write rt-PROC(c) [or rt-SPACE(c), etc.] instead of rt-PROC(f), whenever $f(x) = c$ for all

$x \in \mathbb{N}$.

The classes rt-SPACE(f) and rt-PROC(f), informally introduced in the above discussion, are formally defined as follows:

Definition 4.7 Given a total function $f : \mathbb{N} \rightarrow \mathbb{N}$, and some model of parallel computation M (meeting the minimum requirements stated at the beginning of Section 2.2 on page 16), the class rt-SPACE$^M(f)$ consists in exactly all the well-behaved timed ω-languages L for which there exists a real-time algorithm running on M that accepts L and uses no more than $f(n)$ space, where n is the size of the current input. Analogously, the class rt-PROC$^M(f)$ includes exactly all the well-behaved timed ω-languages L for which there exists a real-time algorithm running on M that accepts L and uses no more than $f(n)$ processors on any input of size n. By convention, the class rt-PROC$^M(1)$ (that is, the class of sequential real-time algorithms) is invariant with M (recall from Section 2.2 that all the parallel models become the RAM in the sequential case).

How to measure The notion of input size is not explained in Definition 4.7. In the classical theory, the input size is the (total) length of the input. However, using such a definition, all well-behaved timed ω-words have length ω. A new notion of input size should be therefore developed.

In the most general case of real-time applications the input data are received in bundles. Take for instance the domain of real-time database systems. Here, the most time consuming operation is answering queries that appear as input. That is, at any moment when some new input arrives, this input consists in the n symbols that encode a query. Motivated by this, we propose the following definition for input size: The size of some timed ω-word w is given by the largest bundle that arrives as input at the same time. Indeed, such a definition makes sense only when the real-time algorithms manifest the *pseudo-on-line* property (when they process input data in bundles, without knowledge of future input), but it would appear that this is a common feature of such algorithms [29].

Definition 4.8 Let $w = (\sigma, \tau)$ be some timed ω-word, $\tau = \tau_1\tau_2\tau_3\ldots$, $\sigma = \sigma_1\sigma_2\sigma_3\ldots$. For $i_0 = 0$ and any $j > 0$, let $s_j = \sigma_{i_{j-1}+1}\sigma_{i_{j-1}+2}\ldots\sigma_{i_j}$, such that (a) $\tau_{i_{j-1}+1} = \tau_{i_{j-1}+2} = \cdots = \tau_{i_j}$, and (b) $\tau_{i_j+1} \neq \tau_{i_j}$. Then, the size $|w|$ of w is $|w| = \max_{j>0} |s_j|$.

Chapter 5

. . . And Back [from Theory to Applications]

Summary

Our thesis is that the theory of timed languages presented in Chapter 4 on page 47 covers all the practically meaningful aspects of real-time computations, while doing so in a formal, unified manner. In order to further support this thesis, we take some meaningful examples, and we construct timed ω-languages that model them.

First, we model two general concepts that are central to real-time systems, namely *computing with deadlines* (Section 5.1 on the next page), and *real-time input arrival* (Section 5.2 on page 61). The appearance of either of these concepts in the specification of some problem gives the real-time characteristic to that problem (see Section 3.1 on page 34). Thus, being able to construct a suitable model for these concepts is crucial to the usefulness of the theory of timed ω-languages, and supports our thesis that the theory of timed languages covers all the practically relevant aspects of real-time computations.

Once these concepts are successfully modeled, one can use those models in order to analyze practical real-time applications. Indeed, we provide in Sections 5.3 on page 63 and 5.4 on page 70 timed ω-languages that model problems from two highly practical areas, namely real-time databases (where we identify a real-time variant of the *recognition problem*) and ad hoc networks (where we offer a formal model for the *routing problem*).

5.1 Computing with Deadlines

One of the most often encountered real-time features is the presence of *deadlines*. The deadlines are typically classified into *hard* deadlines, when a computation that exceeds the deadline is useless, and *soft* deadlines, where the usefulness of the computation decreases as time elapses [59, 86].

For example, a hard deadline may be expressed as "this transaction must terminate within 20 seconds from its initiation." By contrast, a soft deadline may be "the usefulness of this transaction is *max* before 20 seconds elapsed; after this deadline, the usefulness is given by the function $u(t) = max \times 1/(t-20)$."

For some problem, let the input [output] alphabet be $\mathbb{I}$ [$\mathbb{O}$]. We denote by n and m the sizes of the input $\mathbb{i} \in \mathbb{I}^*$ and of the output $\mathbb{o} \in \mathbb{O}^*$. We consider without loss of generality that $\mathbb{I}$, $\mathbb{O}$, and $\mathbb{N}$ are disjoint.

Let π be a problem whose instances can be classified into three classes: (i) no deadline is imposed on the computation; (ii) a hard deadline is imposed at time t_d; (iii) a soft deadline is imposed at time t_d, and the usefulness function is u after this deadline, $u : [t_d, \omega) \rightarrow \mathbb{N} \cap [0, max]$. We build for each instance a timed ω-word (σ, τ) over $\mathbb{I} \cup \mathbb{O} \cup (\mathbb{N} \cap [0, max]) \cup \{w, d\}$, $w, d \notin \mathbb{I} \cup \mathbb{O}$ as follows:

(i) $\sigma_1 \ldots \sigma_m = \mathbb{o}$, $\sigma_{m+1} \ldots \sigma_{m+n} = \mathbb{i}$, $\sigma_i = w$ for $i > m+n$, $\tau_i = 0$ for $1 \leq i \leq m+n$, and $\tau_i = i - m - n$ for $i > m+n$.

(ii) $\sigma_1 \in \mathbb{N} \cap (0, max]$, $\sigma_2 \ldots \sigma_{m+1} = \mathbb{o}$, $\sigma_{m+2} \ldots \sigma_{m+n+1} = \mathbb{i}$, $\tau_i = 0$ for $1 \leq i \leq m+n+1$; if $\tau_i < t_d$ and $i > m+n+1$, then $\tau_i = i - m - n - 1$ and $\sigma_i = w$. Let i_0 be the index such that $\tau_i = t_d$. Then, for all $i \geq i_0$, $\tau_i = i_0 + \lfloor (i - i_0)/2 \rfloor$, and:

$$\sigma_i = \begin{cases} d & \text{if } i - i_0 \text{ is even} \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

(iii) This case is the same as case (ii), except that Relation 5.1 on the preceding page becomes:

$$\sigma_i = \begin{cases} d & \text{if } i - i_0 \text{ is even} \\ \lfloor u(\tau_i) \rfloor & \text{otherwise} \end{cases} \tag{5.2}$$

Let the language formed by all the ω-words that conforms to the above description be L. Basically, a timed ω-word in L has the following properties: At time 0, a possible output and a possible input for π are available. Then, up to the deadline d, the symbols that arrive are w. After that, each time unit brings to the input a pair of symbols, the first component being d (signaling that the deadline passed), and the second one being the measure of usefulness the computation still has (which is 0 for ever when the deadline is hard). When a deadline is imposed over the computation (cases (ii) and (iii)), a minimum acceptable usefulness estimate is also present at the beginning of the computation.

Let then $L(\pi)$ be the language of successful instances of π, $L(\pi) \subseteq L$, in the following sense: Let A be an algorithm for solving π, and let $x \in L$. Suppose that A works on the input encoded in x. Then, $x \in L(\pi)$ if and only if A produces the output encoded in x either (a) at a time that is within the imposed deadline, or (b) at a time when the usefulness of the output is not below the acceptable limit encoded in x.

We are ready to present now an acceptor for $L(\pi)$. For simplicity, we consider that this acceptor is composed of two "processes," P_w and P_m. P_w is an algorithm that solves π, which works on the input of π contained in the current input ω-word, and stores the solution in some designated memory space upon termination. If there is more than one solution for the current instance, then P_w nondeterministically chooses that solution that matches the proposed solution contained in the ω-word, if such a solution exists. Meantime, P_m monitors the input. If, at the moment P_w terminates, the current symbol is w, then P_m compares the solution computed by P_w with the proposed solution, and imposes to the whole acceptor some "final" state s_f if they are identical, or some other designated state s_r (for "reject") otherwise.

On the other hand, if at the moment P_w terminates, the current symbol is d, then the deadline passed. Then, P_m compares the current usefulness measure with the minimum acceptable one. If the usefulness is not acceptable, then P_m imposes the state s_r on the whole acceptor. Otherwise, P_m compares the result computed by P_w with the proposed solution, and imposes either the state s_f or s_r, accordingly.

Once in one of the states s_f or s_r, the acceptor keeps cycling in the same state. In state s_f, the acceptor writes $\mathtt{y}$ on the decision tape (with $\mathtt{y}$ being the "accept" symbol, as in Definition 4.4 on page 50). The decision tape is not modified in any other state.

It is immediate that the language accepted by the above acceptor is exactly $L(\pi)$. It is also immediate that $L(\pi)$ is well-behaved. Thus we completed the modeling of computations with deadlines in terms of ω-languages. Note that we assumed here that all the input data are available at the beginning of computation. However, the case when data arrive while the computation is in progress is easily modeled by modifying the time value that corresponds to each input datum. This case is covered in more detail by our discussion in Section 5.2.

5.2 Real-Time Input Arrival

One of the computational paradigms that feature real-time input arrival is the *data accumulating paradigm* (see Section 2.3 on page 23), that has been extensively studied in [27, 28, 62, 63]. The shape of input in this paradigm is very flexible, and any practically important form of real-time input arrival can be modeled by a particular class of problems within this paradigm. Therefore, we use in what follows the data-accumulating paradigm to illustrate the concept of real-time input arrival. Further models of this concept are also presented in Sections 5.3 on page 63 and 5.4 on page 70.

A *d-algorithm* works on an input considered as a virtually endless stream. The computation terminates when all the currently arrived data have been processed before another datum arrives. In addition, the arrival rate of the input data is given by some function

$\phi(n, \mathbb{t})$ (called the *data arrival law*), where n denotes the amount of data that is available beforehand, and $\mathbb{t}$ denotes the time. The family of arrival laws most commonly used as example was introduced by Relation 2.1 on page 24:

$$\phi(n, \mathbb{t}) = n + kn^{\gamma}\mathbb{t}^{\beta}$$

where k, γ, and β are positive constants. Any successful computation of a d-algorithm terminates in finite time.

Again, we denote the input [output] alphabet by $\mathbb{I}$ [$\mathbb{O}$], while the input and the output are denoted by $\mathbb{i}$ and $\mathbb{o}$, respectively ($\mathbb{i} \in \mathbb{I}^*$, $\mathbb{o} \in \mathbb{O}^*$).

Given a problem π pertaining to this paradigm, we can build the corresponding timed ω-language $L(\pi)$ similarly to Section 5.1 on page 59. More precisely, given some (infinite) input word $\mathbb{i}$ for π (together with a data arrival law $\phi(n, \mathbb{t})$ and an initial amount of data n), and a possible output $\mathbb{o}$ of an algorithm that solves π with input $\mathbb{i}$, a timed ω-word (σ, τ) that may pertain to $L(\pi)$ is constructed as follows: $\sigma_1 \ldots \sigma_m = \mathbb{o}$, $\sigma_{m+1} \ldots \sigma_{m+n} = \mathbb{i}_1 \ldots \mathbb{i}_n$, $\tau_i = 0$ for $1 \leq i \leq m + n$. Note that, since both the arrival law and the initial amount of data are known, one can establish the time of arrival for each input symbol $\mathbb{i}_j$, $j > n$. Let us denote this arrival time by t_j. Also, let $i_0 = m + n + 1$. Then, the continuation of the timed ω-word is as follows: for all $i \geq 0$, $\sigma_{i_0+2i} = c$ (where c is a special symbol, $c \notin \mathbb{I} \cup \mathbb{O}$), and $\sigma_{i_0+2i+1} = \mathbb{i}_{i_0+i}$; moreover, $\tau_{i_0+2i+1} = t_{i_0+i}$, and $\tau_{i_0+2i} = \tau_{i_0+2i+1} - 1$.

Now, an acceptor for $L(\pi)$ has a structure which is identical[1] to the one used in Section 5.1: It consists in the two processes P_w and P_m. P_w works exactly as the P_w from Section 5.1, except that it emits some special signal to P_m each time it finishes the processing of one input datum. Note that, since any d-algorithm is an on-line algorithm (as we shall show in Section 8.2.1 on page 113), it follows that, once such a signal is emitted the p-th time, P_w has a (partial) solution immediately available for the input word $\mathbb{i}_1 \ldots \mathbb{i}_p$.

Then, suppose that P_m received p signals from P_w, and it also received the input symbol

[1] In particular, if there is more than one solution for the current instance, then P_w nondeterministically chooses that solution that matches the proposed solution contained in the ω-word, if such a solution exists.

$\sigma_{i_0+2(p-1-i_0)}$, but it didn't receive yet the input symbol $\sigma_{i_0+2(p-i_0)}$. This is the only case when P_m attempts to interfere with the computation of P_w. In this case, P_m compares the current solution computed by P_w with the solution proposed in the input ω-word; if they are identical, the input is accepted, and the input is rejected otherwise (in the sense that either state s_f or s_r is imposed upon the acceptor, accordingly).

Again, in state s_f, the acceptor writes $\mathtt{y}$ on the decision tape (with $\mathtt{y}$ being the "accept" symbol, as in Definition 4.4 on page 50), and the decision tape is not modified in any other state. As well, once in one of the states s_f or s_r, the acceptor keeps cycling in the same state. It is immediate that $L(\pi)$ is well-behaved and contains exactly all the successful instances of π, therefore we succeeded in modeling d-algorithms using timed ω-languages.

Other related paradigms, like c-algorithms (that are similar with d-algorithms, except that data that arrive during the computation consist in corrections to the initial input rather than new input—see Section 2.3 on page 23) can be easily modeled using the same technique.

5.3 Real-Time Database Systems

We start by briefly reviewing the main concepts of the relational and real-time database systems theory in order to summarize the notations and concepts that are used later, directing the interested reader to [2, 91] for a more detailed presentation. Then, we use our formalism for modeling the recognition problem for real-time database systems (RTDBS for short).

Relational Databases Fix two countably infinite sets **att** (of attributes) and **dom** (the underlying domain, disjoint from **att**). A relation is given by its name and its ordered set of attributes (its *sort*). Given a relation R, the arity of R is $arity(R) = |sort(R)|$. A *relation schema* is a relation name R. A *database schema* is a nonempty finite set $\mathbf{R}$ of relation names. Let R be a relation of arity n. A *tuple* over R is an expression $R(a_1, a_2, \cdots, a_n)$,

where $a_i \in \textbf{att}$, $1 \leq i \leq n$. A *relation instance* over R is a finite set of tuples over R. A *(database) instance* $\mathbf{I}$ over some database schema $\mathbf{R}$ is the union of relation instances over $\mathbf{R}$. The sets of instances over a database schema $\mathbf{R}$ [relation schema R] are denoted by $inst(\mathbf{R})$ [$inst(R)$]. The interrogation of a database is accomplished by *queries*. A query q is a partial mapping from $inst(\mathbf{R})$ to $inst(S)$, for fixed database schema $\mathbf{R}$ and relation schema S.

Complexity of queries We are mainly concerned with *data complexity* of queries, namely the complexity of evaluating a fixed query for variable database inputs [2], since the usual situation is that the size of the database input dominates by far the size of the query (and therefore this measure is most relevant).

The complexity of queries is defined based on the *recognition problem* associated with the query. For a query q, the recognition problem is: Given an instance $\mathbf{I}$ and a tuple u, determine if u belongs to the answer $q(\mathbf{I})$. That is, the recognition problem of a query q is the language:

$$\{enc(\mathbf{I})\$enc(u) | u \in q(\mathbf{I})\} \tag{5.3}$$

where *enc* denotes a suitable encoding over queries and tuples, and $ is a special symbol.

The *(data) complexity* of q is the (conventional) complexity of its recognition problem. Then, for each conventional (time, space, processors) complexity class C, one can define a corresponding *complexity class of queries QC*.

Another way to define the complexity of queries is based on the complexity of actually constructing the result of the query. The two definitions are in most cases interchangeable [2].

Real-Time Databases An *active databases* support the automatic triggering of updates (rules) in response to (internal or external) events. The typical form of a rule is "**on** *event* **if** *condition* **then** *action*." An action may in turn generate other events and hence trigger other rules. A fundamental issue in active databases addresses the choice of an execution

model (i.e., a semantics for rule application), with an important dimension of variation given by the moment the rules are fired: *immediate firing* (a rule is fired as soon as its event and condition become true), *deferred firing* (rule invocation is delayed until the final state in the absence of any rule is reached) and *concurrent firing* (a separate process is spawned for the rule action and is executed concurrently). In the most general model, each rule has an associated firing mode.

RTDBSs add temporal and timeliness dimensions to active databases. Indeed, a real-time database interacts with the physical world, and the database is thus active. In addition, data in a real-time database are time sensitive, and the transactions must be timely, i.e, they must complete within their time constraints (deadlines). We briefly present in the following the data model used in [91], which is itself derived from the historical relational data model [35].

In the following, the *valid time* associated to some database object is the time at which the fact represented by that object is true in the real world, and the *transaction time* is the time at which the fact represented by the corresponding database object has been recorded in the database and is available for retrieval. It is assumed that the difference between the valid time and the transaction time is small, so that we will refer to both the valid time and the transaction time simply as "valid time." Time is considered discrete and linear.

The objects from the database are grouped in three categories: *Image objects* are those objects that contain information obtained directly from the external environment. Associated with an image object is the most recent sampling time. A *derived object* is computed from a set of image objects and possibly other objects. The time stamp associated with a derived object is the oldest (i.e., smallest) valid time of the data objects used to derive it. Finally, an *invariant object* is a value that is constant with time. Then, a real-time database instance is defined as $B = (I_1, I_2, ..., I_n, D, V)$, where I_n is the most recent set of image objects, and $I_1, I_2, \ldots, I_{n-1}$ are archival variants of this set. D is the set of derived objects, and V is the set of invariant ones. The valid time associated with each temporal object in the

database instance is called the *lifespan* of the object. The lifespan of a data object is defined as a finite union of intervals, that form a boolean algebra. A lifespan can also be associated with a set of objects, in a natural manner. Based on these notions, a variant of relational algebra can be defined as a query language for real-time databases [91].

Additional issues in RTDBSs include the pattern of queries (periodic, sporadic, aperiodic), the nature of deadlines (hard, soft), and the way the updating rules are fired. While the first two issues received both theoretical and practical attention in the literature, to our knowledge, there is no special theoretical treatment of the last issue, except for the one that was spawned by the theory of active databases. The immediate firing in the case of image objects is implied in [91] and therefore in the above paragraphs.

An aperiodic query q is a partial function from B to $inst(S)$, where S is some relation. A periodic query returns an answer each time it is issued, therefore such a query is a function from B to $(inst(S))^\omega$.

5.3.1 Real-time Database Systems as Timed Languages

Recall that one of the ways of assessing the complexity of queries and query languages is based on the reduction of such problems to the problem of language recognition.

However, the real-time component in a real-time database system adds a new dimension to the model, namely the time. It seems natural therefore to try to model such database systems using timed languages. We describe such a modeling in what follows. We consider that there is a suitable encoding function *enc* that encodes objects and sets of objects, without giving much attention to how such a function is constructed, since such functions were widely used (see for example [2, 61]). Let $ be a symbol that is not in the codomain of enc.

Let us ignore the queries for the moment. Recall that a real-time database instance is a tuple $B = (I_1, I_2, ..., I_n, D, V)$. Moreover, assume for now that the database contains exactly one immediate object, called o_k, and that the value of o_k is read from the external world

each t_k time units. Let D and V be some sets of derived and invariant objects, respectively, with $m = |enc(V)|$ and $p = |enc(D)|$, and $o_k(t)$ be the value of o_k that is read at time t from the external world. Consider then the timed ω-word $db_k = (\sigma, \tau)$, where σ and τ has the following form: let[2] $q = |enc(o_k)|$; then, for every $i \geq 0$, $\sigma_{\alpha+i(q+1)+1} \ldots \sigma_{\alpha+(i+1)(q+1)} = enc(o_k(t_i))\$$, where $\alpha = m + p + 2$; moreover, $\tau_j = it_i$ for $\alpha + i(q+1) + 1 \leq j \leq \alpha + (i+1)(q+1)$.

Furthermore, let $db_0 = (\sigma, \tau)$, such that $\sigma_1 \ldots \sigma_m = enc(V)$, $\sigma(m+1) = \sigma(m+p+2) = \$$, and $\sigma_{m+2} \ldots \sigma_{m+p+1} = enc(D)$; in addition, $\tau_i = 0$, $1 \leq i \leq m + p + 2$.

In other words, the sets of both invariant and derived objects are specified at time 0, as modeled by the word db_0. Then, each t_k time units a new value for o_k is provided. This is modeled by db_k. It is clear that the database instance is completely specified by the word $db_0 db_k$, since this word models both the invariant and derived objects (by db_0), as well as all the updates for the sole image object (by db_k).

Now, let us consider the general case of a real-time database. That is, we do not restrict ourselves to one invariant object anymore. Therefore, let the database instance contain r such objects, called o_k, $1 \leq k \leq r$. However, if we consider a word db_k corresponding to each object o_k, $1 \leq k \leq r$, then it is immediate that the database is described by the word:

$$db_B = db_0 db_1 \ldots db_r \tag{5.4}$$

We have now a model for real-time databases, which it trivially a well-behaved timed ω-language. Now, all that we have to do is to consider the queries. Again, we assume without further details that there is a function enc_q for encoding queries and their answers, whose codomain is disjunct from the codomain of enc. Real-time queries can be classified in two classes: periodic and aperiodic [69].

Let us focus on aperiodic queries first. Each such query q may have a hard of soft deadline. However, it seems natural to also consider queries without any deadline, since

[2] We assume for clarity of presentation that the length of the encoding of o_k is constant over time. The extension to a variable length is straightforward.

they might be present even in a real-time environment. Therefore, the encoding of a query should include

1. the time t at which the query is issued,
2. the (encoding of) the query itself $enc_q(q)$,
3. a tuple s that might be included in the answer to the query,
4. the deadline t_d of the query, if any.

Note that a similar problem is the presence of deadlines, that was presented in Section 5.1 on page 59, except that the first item is not modeled (the computation always starts at time 0). Therefore, our construction is similar to the construction of the language that models computations with deadlines.

We have thus a query for which either (i) there is no deadline, (ii) a hard deadline is present, or (iii) a soft deadline is present. The deadline (if any) is imposed at some relative time t_d (that is, the moment in time at which the deadline occurs is $t + t_d$), and the usefulness function is denoted by u. For each query q and each candidate tuple s we can build similarly to Section 5.1 an ω-word $aq_{[q,s,t]} = (\sigma, \tau)$ as follows, where $m = |enc_q(s)\$|$, $n = |enc_q(q)\$|$, and \$, w_q, d_q are not contained in the codomain of enc_q:

(i) $\sigma_1 \dots \sigma_m = enc_q(s)\$$, $\sigma_{m+1} \dots \sigma_{m+n} = enc_q(q)\$$, $\sigma_i = w_q$ for $i > m+n$, $\tau_i = t$ for $1 \le i \le m+n$, and $\tau_i = t + i - m - n$ for $i > m+n$.

(ii) $\sigma_1 \in \mathbb{N} \cap (0, max]$, $\sigma_2 \dots \sigma_{m+1} = enc_q(s)\$$, $\sigma_{m+2} \dots \sigma_{m+n+1} = enc_q(q)\$$, $\tau_i = t$ for $1 \le i \le m+n+1$; if $\tau_i < t_d$ and $i > m+n+1$, then $\tau_i = t + i - m - n - 1$ and $\sigma_i = w_q$. Let i_0 be the index such that $\tau_i = t + t_d$. Then, for all $i \ge i_0$, $\tau_i = t + i_0 + \lfloor (i - i_0)/2 \rfloor$, and:

$$\sigma_i = \begin{cases} d_q & \text{if } i - i_0 \text{ is even} \\ 0 & \text{otherwise} \end{cases} \tag{5.5}$$

(iii) This case is the same as case (ii), except that Relation 5.5 on the preceding page becomes:

$$\sigma_i = \begin{cases} d_q & \text{if } i - i_0 \text{ is even} \\ \lfloor u(\tau_i) \rfloor & \text{otherwise} \end{cases} \tag{5.6}$$

Let q be a periodic query now. More precisely, q is issued for the first time at time t, and then it is reissued each t_p time units. Each time q is issued, we have to consider a tuple whose inclusion into the result of q is to be tested. Let s_i be such a tuple for the i-th invocation of q, and let $s = (s_1, s_2, s_3, \ldots)$. It is easy to see that such a query is modeled by the word $pq_{[q,s,t,t_p]} = aq_{[q,s_1,t]}aq_{[q,s_2,t+t_p]}aq_{[q,s_3,t+2t_p]}\ldots$. However, there is no guarantee that the resulting word $pq_{[q,s,t,t_p]}$ is well-behaved, since the concatenation of an infinite number of well-behaved timed ω-words is not necessarily well-behaved. In our case, however, the result of the concatenation is a well-behaved timed ω-word, and this follows immediately from the following observation.

Lemma 5.1 *For a word $pq_{[q,s,t,t_p]} = (\sigma, \tau)$, and for any finite positive integer k, there exists a finite integer k' such that $\tau_{k'} \geq k$.*

Proof. Without loss of generality, we assume that $k = t + it_p$ for some $i \geq 0$. However, the symbols for which $\tau_j < k$ can be counted as follows: there are $i + 1$ occurrences of some word of the form $enc_q(q)\$enc_q(s_v)\$$, $0 \leq v \leq i$, and at most k occurrences of symbols from $\{w_x, d_x | x = t + lt_p, 0 \leq l \leq i\}$. Therefore, $j \leq (i+1)|enc_q(q)\$enc_q(s)\$| + 2ki$, for some tuple s such that $|s| = \max_{0 \leq v \leq i} s_v$. Clearly, the upper bound for j is finite and therefore so is the number of symbols for which $\tau_j < k$. ∎

We modeled therefore the main ingredients of a real-time database system. All we have to do then is to put the pieces together.

Definition 5.1 Let B be some real-time database instance. Then, given some aperiodic

query q from B to $inst(S)$ (where S is some relation schema), issued at time t, the recognition problem for q on B is the (well-behaved) timed ω-language:

$$L_{aq} = \{db_B aq_{[q,s,t]} | s \in q(B)\} \tag{5.7}$$

Analogously, given a periodic query q from B to $(inst(S))^{\omega}$, issued at time t and with period t_p, the recognition problem for q on B is the (well-behaved) timed ω-language:

$$L_{pq} = \{db_B pq_{[q,s,t,t_p]} | s \in q(B)\} \tag{5.8}$$

Note that the recognition problem for real-time queries is similar to the same problem for conventional queries, shown in Relation 5.3 on page 64, except that the (conventional) words used in Relation 5.3 are replaced by timed ω-words.

5.4 Ad Hoc Networks

We direct our attention now to another real-time problem, namely the routing problem in ad hoc networks. We show how to model this problem using the theory of timed ω-languages. In the process, we also identify an interesting variant of real-time algorithms, which we believe to be useful in modeling parallel distributed real-time systems.

An *ad hoc network* is a collection of wireless mobile *nodes*, that dynamically forms a temporary network without using any existing network infrastructure or centralized administration [25, 46]. Due to the limited transmission range of such nodes, multiple hops may be needed for one node to exchange data with another.

The main difference between an ad hoc network and a conventional one is the routing protocol. In such a network, each host is mobile. Therefore, the set of those nodes that can be directly reached by some host changes with time. Furthermore, because of this volatility of the set of directly reachable nodes, each mobile node should act not only as a host, but as a router as well, forwarding packets to other mobile hosts in the network.

Although the concept of ad hoc networks is relatively new, many routing algorithms were developed (see, for example, [18, 25] and the references therein). However, little is

known about the performances of these algorithms. A comparative performance evaluation was proposed for the first time in [25], where several routing algorithms are compared based on discrete event simulation. To our knowledge, no analytical model has been proposed to date.

On the other hand, an ad hoc network is obviously a real-time system. Indeed, since the positions (and implicitly the connectivity) of all the hosts are functions of time, such a network is close to the correcting algorithms paradigm (see Section 2.3 on page 23). Therefore, conforming to our claim that timed languages can model all the meaningful aspects of real-time computations, one can model ad hoc networks using this formalism. This is what we are attempting in the following.

Assumptions and notations When speaking about ad hoc networks, we assume that, if a message is emitted by some node at some time t and received by another node that is in the transmission range of the sender at time t', then $t' = t + 1$. That is, transmitting a message takes one time unit. Note that we actually established in this way a granularity of the time domain. This granularity seems appropriate, since transmitting a message is an elementary operation in a network.

We introduce a notation for the transmission range. We denote this characteristic by the predicate $range(n_1, n_2, t)$. That is, a node n_2 is in the transmission range of another node n_1 at time t if and only if $range(n_1, n_2, t) = true$. We do not give any specific way of computing this predicate, since such a computation depends on the characteristics of the particular application. Indeed, this predicate depends on the characteristics of both n_1 and n_2, as well as on the geographical characteristic of the area between the two nodes.

5.4.1 Nodes as Timed ω-Words

The main component of a model for ad hoc networks is the mobile host (or the node). It is consistent to assume that each node in a network is uniquely identified (for example, by

its unique IP address). For convenience, we label such a node by an integer between 1 and n, where n is the number of nodes in the given network.

We assume that there is an encoding function e of the properties of any node i (like the label i of the node, the position of i, and other properties that will be explained below) over some alphabet Σ, with $@, \$ \notin \Sigma$. Denote by P the set of all possible properties. Then, we say that x is the *encoding* of some property ρ of node i if and only if $x = enc(i, \rho)$, where $enc : \mathbb{N} \times P \to \Sigma$,

$$enc(i,\rho) = \begin{cases} \$e(i)\$ & \text{if } \rho = i, \\ \$e(i)@e(\rho)\$ & \text{otherwise} \end{cases}$$

In other words, we have a standard encoding, except that each property of some node i (except i itself) is prefixed by an encoding of i. This will be useful when we put together the models of all the nodes that form an ad hoc network.

Each node i is characterized by its position, a variable that changes with time. We denote by $p_i(t)$ the (encoding of the) position of node i at time t. In addition, each node has a set of characteristics that are invariant with time (for example, the transmission range). The structure of this set is, however, immaterial for the present discussion. Therefore, we consider that these characteristics are encoded by some string q_i for each node i. Finally, it is sometime assumed that each node has a constant velocity [25]. However, the constant velocity assumption is made for simulation purposes, and is not necessarily a feature of the real world. Indeed, the velocity of some node usually varies with time, and/or is unknown to the other nodes. Such a case is considered in [18], where the only thing known by any node is its current position. We consider here the most general case, where the only thing known about some node at some moment in time is its position at that moment.

Given a series of (conventional) words $w_1, w_2, \ldots$, we denote by $w_1 w_2$ the concatenation of w_1 and w_2. Moreover, we denote by $\prod_{i=1}^{\omega} w_i$ the (infinite) word obtained by successively concatenating the words w_i, $i \geq 1$

We are ready now to consider a timed ω-word that models some mobile host. A node

i is modeled by the word $h_i = (\sigma, \tau)$, where $\sigma = (q_i)(\prod_{t=0}^{\omega} p_i(t))$, and $\tau = \tau_1\tau_2\ldots$, with $\tau_j = 0$ for $1 \leq j \leq |q_i p_i(0)|$, and, for any $k > 1$, $\tau_j = k$, $1 + |q_i| + \prod_{l=0}^{k-1} |p_i(l)| \leq j \leq |q_i| + \prod_{l=0}^{k} |p_i(l)|$.

In other words, the first part of h_i contains the invariant set of characteristics, together with the initial position of the object that is modeled. The time values associated with this subword are all 0. Then, the successive positions of the node are specified, labeled with their corresponding time values. It is immediate that all the necessary information about node i is contained in the word h_i.

5.4.2 A Model for Messages

Now that we have a model for the set of nodes, all we have to do is to connect them together. We have that is to model message exchanges between nodes.

Consider a message u issued at some time t. Such a message should contain the source node s and the destination node d. In addition, such a message may contain its type (for example, message or acknowledgment), the data that is to be transmitted, etc. All this content (referred to as the *body* of the message) is, however, immaterial, and we denote it by b_u as a whole. Considering that the encoding function e introduced above encodes messages over Σ as well, let the encoding of a message be $\$e(t)@e(s)@e(d)@e(b_m)\$$, and $k = |\$e(t)@e(s)@e(d)@e(b_u)\$|$. Then, the timed (finite) word that models u is $m_u = (\sigma, \tau)$, where $\sigma_1 \ldots \sigma_k = \$e(t)@e(s)@e(d)@e(b_u)\$$ and $\tau_j = t$ for $1 \leq j \leq k$.

Note that m_u is not a well-behaved timed ω-word. On the other hand, it is easy to see that, for any node i, $h_i m_u$ is such a word. However, for a message to exist, there must be at least one node in the network, namely the node that sends it. That is, a model of a message would always be concatenated to the model of at least one node, and therefore the above construction is sufficient for our purposes.

Finally, one has to consider the model for the receiving event. For this purpose, assume

that some message u (generated at time t_u, by a source s) is received by its intended destination d at some time t'_u. We model such an event by the timed word $r_u = (\sigma, \tau)$, where $\sigma_1 \ldots \sigma_{k'} = \$e(t)@e(s)@e(d)\$$ and $\tau_j = t'_u$ for $1 \leq j \leq k'$, with $k' = |\$e(t)@e(s)@e(d)\$|$. Again, such a word is not well-behaved, but the above argument still holds (namely, some "acknowledgment" cannot exist in a network with no hosts).

5.4.3 The Routing Problem

It is immediate that an ad hoc network with n nodes and without any message is modeled by the timed ω-word $a_n = h_1 h_2 \ldots h_n$. Then, a network of n nodes and some messages $u_1, u_2, \ldots, u_k$, $k \geq 1$, will be modeled by the word $w_{n,k} = h_1 h_2 \ldots h_n m_{u_1} m_{u_2} \ldots m_{u_k}$, and the model that includes the event of receiving u_i, $1 \leq i \leq k$, is $wr_{n,k} = h_1 h_2 \ldots h_n m_{u_1} r_{u_1} m_{u_2} r_{u_2} \ldots m_{u_k} r_{u_k}$. Moreover, given some countably infinite series of messages $u_1 u_2 \ldots$, the model of the network in which these messages are transmitted is $wr_{n,\omega} = h_1 h_2 \ldots h_n m_{u_1} r_{u_1} m_{u_2} r_{u_2} \ldots$. Note that $w_{n,\omega}$ is a well-behaved timed ω-word under the reasonable assumption that any node can generate only a bounded number of messages per time unit.

In the following we may refer to the encoding m_u of a message u simply by "the message m_u." Whether the term message refers to a message or an encoding of a message will be clear from the context. For a fixed n, denote by N_n the set of all the words of the form $w_{n,k}$, $k \in \mathbb{N} \cup \{\omega\}$.

We are ready now to state the routing problem in ad hoc networks as a timed ω-language. Consider a network with n nodes, and a message u generated at time t, with body b, that is to be routed from its source s to the destination d. Then, a route of u is a word in the timed ω-language $R_{n,u} \subseteq N_n$, where, for some finite positive integer f, there exists a set of messages $u_1, u_2, \ldots, u_f$, and possibly a set of messages $rt_1, rt_2, \ldots, rt_g$, with g a positive, finite integer, such that any word $w \in R_{n,u}$ has the form $w = h_1 h_2 \ldots h_n m_{u_1} r_{u_1} \ldots m_{u_f} r_{u_f} m_{rt_1} r_{rt_1} \ldots m_{rt_f} r_{rt_f}$. Furthermore, for each message

u_i, $1 \leq i \leq f$, denote by t_i, t'_i, s_i, d_i, and b_i the generation time, receiving time, source, destination, and body of u_i, respectively. Then,

1. $b_1 = b_2 = \ldots = b_f = b$, $s_1 = s$, $d_f = d$, $t_1 = t$,

2. for any i, $1 \leq i \leq f-1$, $d_i = s_{i+1}$, $t'_i = t_{i+1}$, and $range(s_i, d_i, t_i) = true$,

3. t'_f is finite.

In other words, the routing process generates f intermediate messages ($u_1, \ldots, u_f$). These are one-hop messages that contain the same information as the original message. Moreover, the time at which one such message arrives at the intended destination of u is finite (otherwise, the message is never received, and the routing process is hence unsuccessful). In addition, there might exist a finite number of additional messages ($rt_1, \ldots, rt_g$), that are exchanges between nodes in the routing process (for example, when the routing tables at each node are built/updated). In the following, we refer to some language $R_{n,u}$ as an (instance of a) *routing problem*, while some particular word $w \in R_{n,u}$ will be called an *instance* of $R_{n,u}$, or just routing instance when $R_{n,u}$ is understood from the context. Note that the actual routing (performed by some routing algorithm) of message u in some n-node network is modeled by a word in the corresponding routing problem.

Clearly, the language $R_{n,u}$ models all the relevant characteristics of a routing problem. Note that two routing algorithms may be compared by comparing their corresponding words from $R_{n,u}$. Moreover, more than one measure of performance may be considered. The measures of performance that are considered in [25] are the routing overhead (the total number of messages transmitted), path optimality (the difference between the number of hops a message took to reach its destination versus the length of the shortest possible path), and the message delivery ratio (the number of messages generated versus the number of packets received).

The first two measures have immediate analogs in our model. Indeed, considering some word $w \in R_{n,u}$ corresponding to a routing algorithm, the routing overhead is given

by $f + g$, the total number of messages that are generated. The number of hops a message traveled is given by $t'_f - t_1$. The message delivery ratio on the other hand needs some changes in our model, since we defined the routing problem as consisting in the successful deliveries of messages. Consider for this purpose the language $R'_{n,u} \subseteq N_n$, where any word $w \in R'_{n,u}$ has the same properties as above, except the finiteness of t'_f. This models a routing problem where the possibility of a message to be lost (that is, never received by its intended destination) exists. This property is modeled by the cases where $t'_f = \omega$.

However, note that in practice an infinite delivery time usually means that the delivery time exceeds some finite threshold T. This situation is modeled by our initial construction, where a lost message is a message for which $t'_f - t_1 > T$.

5.4.4 On Routing Algorithms

Up to this moment, we modeled the routing problem. Such an approach offers a basis for comparing routing algorithms, once the results of these algorithms are modeled as words from $R_{n,u}$. On the other hand, nothing is said about the routing algorithm itself. The immediate variant for such a model takes the form of real-time algorithm that accepts the language $R_{n,u}$. However, further restrictions to such an acceptor must be imposed: The real world router consists in n independent algorithms, that have limited means of communication. That is, two such nodes can communicate only by messages exchanged between them. A model for a routing algorithm must take this feature into account.

However, there is a second approach to this model: A node in such a network is unaware of the properties of another node, unless it receives a message from (or about) that node. Based on this intuition, we can propose a model for an n-node ad hoc network. For specificity, we model a routing instance $w = h_1 h_2 \ldots h_n m_{u_1} r_{u_1} \ldots m_{u_f} r_{u_f} m_{rt_1} r_{rt_1} \ldots m_{rt_f} r_{rt_f}$.

Such a model has n components, one for each node. These components are the timed ω-words H_i, $1 \leq i \leq n$. Each H_i consists in a "local" component $\mathcal{L}_i$ and a "remote"

component $\mathcal{R}_i$, where:

$$\mathcal{L}_i = h_i m_{u_{j_1}} m_{u_{j_2}} \ldots m_{u_{j_x}} m_{rt_{k_1}} m_{rt_{k_2}} \ldots m_{rt_{k_y}} \tag{5.9}$$

where $0 \leq x \leq f$, $0 \leq y \leq g$, $1 \leq j_l \leq f$ for any l, $1 \leq l \leq x$, and $1 \leq k_l \leq g$ for any l, $1 \leq l \leq y$. Moreover, the source of each message u_{j_l} or rt_{k_l} is i. That is, the local component consists only in those messages that are sent by the corresponding node, together with the space coordinates of that node.

Given $\mathcal{L}_i$, for each $j \neq i$, $1 \leq j \leq n$, denote by $M_{i,j}$ the set $\{r_{u_{j_l}} | 1 \leq l \leq x, d_{u_{j_l}} = j\} \cup \{r_{rt_{j_l}} | 1 \leq l \leq y, d_{rt_{j_l}} = j\}$. That is, the set $M_{i,j}$ contains the receiving events for all the messages that are sent from node i to node j. Then:

$$\mathcal{R}_i = v_1 \ldots v_k \tag{5.10}$$

where $v_1 \ldots v_k$ are exactly all the elements in the set $\cup_{l=1}^{n} M_{l,i}$.

Finally, $H_i = \mathcal{L}_i \mathcal{R}_i$. In other words, the component H_i contains only those messages that are sent by the corresponding node, and those messages that are received by the node. Besides this information, no knowledge about the external world exists.

In passing, we notice that such a construction is not limited to routing algorithms in ad hoc networks. Indeed, the concept of n independent processes that communicate with each other using messages (as opposed to, say, a shared storage) defines a *distributed system*. One of the approaches in developing formal characterizations for such systems [39] is to model them as n independent words (one for each process), and to provide a mechanism modeling interprocessor communication. The distributed model of routing algorithms in ad hoc networks that is presented in this section can be easily generalized [26] to offer a good basis for the analysis or real-time distributed systems.

Chapter 6

Complexity of Real Time I: A Strong Infinite Hierarchy

Summary

Consider the following question [75]:

Question 1 Can one find any problem that is solvable by an algorithm that uses p processors, $p > 1$, and is not solvable by a sequential algorithm, even if this sequential algorithm runs on a machine whose (only) processor is p times faster than each of the p processors used by the parallel implementation?

Though it is standard to assume that each processor on a parallel computer is as fast as the single processor on the sequential computer used for comparison, Question 1 does make sense in practice. Besides, questions of this kind are crucial for the process of developing a parallel real-time complexity theory. Indeed, a meaningful such theory should be invariant to secondary issues like the speed of some particular machine. Thus, an answer to the above question is also important from a complexity theoretic point of view.

An intuitive aside.

On the intuitive level, a positive answer to Question 1 for $p = 2$ is provided by (a slightly modified version of) the *pursuit and evasion on a ring* example presented in [5]:

> An entity A is in pursuit of another entity B on the circumference of a circle, such that A and B move at the same top speed. Clearly, A *never* catches B. Now, if two entities C and D are in pursuit of entity B on the circumference of a circle, then C and D *always* catch B, even if each of them moves at $1/x$ the speed of A (and B), $x > 1$.

This modified version of the pursuit/evasion problem was mentioned for the first time in [31].

Starting from this intuition, we construct a timed ω-language that models the geometric problem presented above. We extend this language to a "stack of n identical circles[1]" $n \geq 1$, and we show that such a language is accepted by a $2n$-processor PRAM, but there does not exist any $(2n-1)$-processor algorithm that accepts the language. Thus, we prove that the hierarchy of parallel machines that solve real-time problems is infinite. To our knowledge, this is the first time such a result is obtained. Finally, we show that the infiniteness of the parallel real-time hierarchy is invariant with the model of parallel computation involved (in the sense described on page 17 and assumed throughout the thesis). Theorem 6.9 on page 93 is the main result of this chapter, establishing the aforementioned hierarchy.

As several of the concepts in this chapter may not be immediately obvious, we provide a number of explanatory asides (as we did above) to help the reader develop some intuition.

6.1 Two Processors are More Powerful than One

Given some arbitrary word w of length n, we shall index it starting from 0 (however, both the symbol and time sequences in a timed ω-word are indexed in Chapter 4 on page 47

[1] For ease of presentation, we shall informally refer to "the stack of n circles," as "the n-dimensional case."

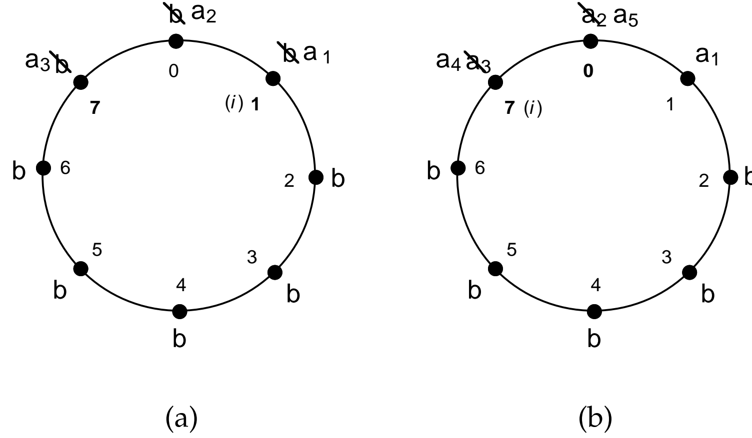

Figure 6.1: PURSUIT$_1$: Insertion modulo r

starting from 1, and we shall keep this convention). For any $0 \leq i \leq j < n$, we denote by $w_{i...j}$ the subword $w_i w_{i+1} \ldots w_j$ of w.

We construct in what follows a timed ω-language[2] PURSUIT$_1$ which is accepted by a two-processor algorithm, but cannot be accepted by a sequential algorithm.

Fix r and q, $r > 2q$, and $\Sigma = \{a, b, +, -\}$. Let $L_o = \{(\sigma, \tau) | \sigma \in \{a, b\}^r, \tau_i = 0$ for all $1 \leq i \leq r\}$. A word in L_o represents an initial value that will be modified as time passes. Such a modification is given by $L_t = \{(\sigma, \tau) || \sigma| = j, 1 \leq j \leq q+1, \sigma_1 \in \{+, -\}, \sigma_{2...j} \in \{a, b\}^{j-1}, \tau_i = t$ for all $1 \leq i \leq j\}$. A word in L_t denotes a change arriving at time t; the first symbol is $+$ or $-$, followed by at most q a's and/or b's. The semantics will become clear shortly. Let $L_u = \prod_{i>0} L_{ci}$, for a given positive c. PURSUIT$_1$ will be constructed as a subset of $L_o L_u$. However, we need some new concepts to precisely define PURSUIT$_1$.

Let $w \in \{a, b\}^r$ and $u = u_0 u'$, $u_0 \in \{+, -\}$ and $u' \in \{a, b\}^j$, $j \leq q$. We define the *insertion modulo r at point i* of u in w, $0 \leq i < r$, as a function ins_r that receives w, u, and i, and returns a new word w and a new i as follows: Let $i' = i + q$ if $u_0 = +$ and $i' = i - q$ otherwise. Then, $ins_r(w, u, i) = (w', i' \mod r)$, where w' is computed as follows

[2]The languages PURSUIT$_k$, $1 \leq k$, were called L_k in [32, 33]. We change here the notation in order to eliminate any confusion.

($\overline{x}$ denoting the reversal of some word x):

1. If $i' < 0$, let $i'' = i' \mod r$. Note that, in this case, $u_0 = -$. Then, $w' = \overline{u'_{0\ldots i}} w_{i+1\ldots i''-1} \overline{u'_{i+1\ldots j-1}}$.
2. Analogously, if $i' > r-1$ (and thus $u_0 = +$), then $w' = u'_{r-i\ldots j-1} w_{i''+1\ldots i-1} u'_{0\ldots r-i-1}$.
3. Otherwise (that is, when $0 \leq i' \leq r-1$), let $i_1 = \min(i, i')$, $i_2 = \max(i, i')$, and $x = u'$ if $u_0 = +$ and $x = \overline{u'}$ otherwise. Then, $w' = w_{0\ldots i_1-1} x w_{i_2+1\ldots r-1}$.

AN INTUITIVE ASIDE.

The intuition behind ins_r (also suggested by its name) is simple: Picture the word w as a circle, in which w_0 is adjacent to the right to w_{r-1}. Then, u replaces j consecutive symbols in the "circle" w, starting from w_i, and going either to the left or to the right, depending on u_0 (+ for right). u models the moves of the *pursuee* over the "circle" w. The pursuee has a topmost velocity of q/r-th of the circle's circumference per time unit. The algorithm that accepts PURSUIT$_1$ (the pursuer) will have to match the moves of the pursuee, and its "velocity" is proportional to the speed of the processor(s) used.

To clear things up, consider the example in Figure 6.1 on the preceding page. Here, $r = 8$ and $q = 3$. Initially, $w = bbbbbbbb$, and the insertion point is $i = 1$. For clarity, w is represented as a circle, with 8 identified locations that corresponds to the eight symbols stored in w. These locations are labeled with their indices (inside the circle), and with the values stored there (outside). Part (a) of Figure 6.1 shows the insertion of the word $u = -a_1a_2a_3$ (all the a's are the same symbol, the subscripts of symbols in u being provided solely for illustration purposes). Here, the pursued entity moves to the left (or counterclockwise), rewriting the symbols at indices 1, 0, and 7, in this order. After such a processing, the new insertion point becomes $i = 7$, and $w = a_2a_1bbbbba_3$. Consider now that the next word to be inserted is $u = +a_4a_5$. Now, the indices whose values are modified are 7 and 0. This processing is illustrated in Part (b) of Figure 6.1. The final result is $i = 0$ and $w = a_5a_1bbbbba_4$.

Denote $ins_r(w, u, i)$ by $(w, i) \oplus u$, and let $\oplus$ be a left-associative operator. Then, for some integers α and β, $1 \leq \alpha \leq \beta$, for appropriate words w, u^α, $u^{\alpha+1}$, ..., u^β, and for some i, $0 \leq i \leq r-1$, we define $(w, i) \bigoplus_{j=\alpha}^{\beta} u^j = (w, i) \oplus u^\alpha \oplus u^{\alpha+1} \oplus \cdots \oplus u^\beta$.

AN INTUITIVE ASIDE.

Intuitively, the operator $\bigoplus$ is for $\oplus$ as $\sum$ is for $+$ in arithmetic. Specifically, such an operator receives some initial word w and some initial insertion point i, and successively inserts modulo r the words $u^\alpha, \ldots, u^\beta$ in w, modifying the insertion point accordingly after each such an insertion. For example, refer again to Part (b) of Figure 6.1 on page 80, which illustrates the result of $(bbbbbbbb, 1) \bigoplus_{j=1}^{2} u^j$, where $u^1 = -a_1a_2a_3$, and $u^2 = +a_4a_5$. Incidentally, the result of this operation is $(a_5a_1bbbbba_4, 0)$.

We now introduce the second useful concept: Consider $w \in L_oL_u$, $w = w^0 \prod_{i>0} w^i$, with $w^0 \in L_o$, and $w^i \in L_{ci}$, $i > 0$. For some t and some i_0, $0 \leq i_0 \leq r-1$, let $s(w, t) = (\sigma^0, i_0) \bigoplus_{ci \leq t} \sigma^i$, where[3] $\sigma^i = \text{detime}(w^i)$, $i \geq 0$.

Let A be an algorithm that receives w and uses p processors, $p \geq 1$ (A is sequential if $p = 1$ and parallel otherwise). A may inspect (i.e., read from memory) the symbols stored at some indices in $s(w, t)$. Many processors may inspect different indices in parallel. For each processor l, $1 \leq l \leq p$, let ι_t^l be the most recent index inspected by processor l up to time t. If some processor inspects no symbols from $s(w, t)$, then $\iota_t^l = -1$. Let I_t^l be the "history" of inspected symbols up to time t, i.e., $I_t^l = \bigcup_{t' \leq t} \iota_{t'}^l \setminus \{-1\}$. Assume that A does not inspect any symbol whatsoever from $s(w, t)$ (e.g., A doesn't even bother to maintain $s(w, t)$ in memory). Then, for any t, $\iota_t^l = -1$ and thus $I_t^l = \emptyset$, $1 \leq l \leq p$. Let $lo = \min_{1 \leq l \leq p}(\iota_t^l)$, $hi = \max_{1 \leq l \leq p}(\iota_t^l)$, and $I = \bigcup_{1 \leq l \leq p} I_t^l$. Then, we define $z(w, t)$, the

[3]Recall that c is a positive constant and that, for some timed ω-word $w = (\sigma, \tau)$, $\text{detime}(w) = \sigma$.

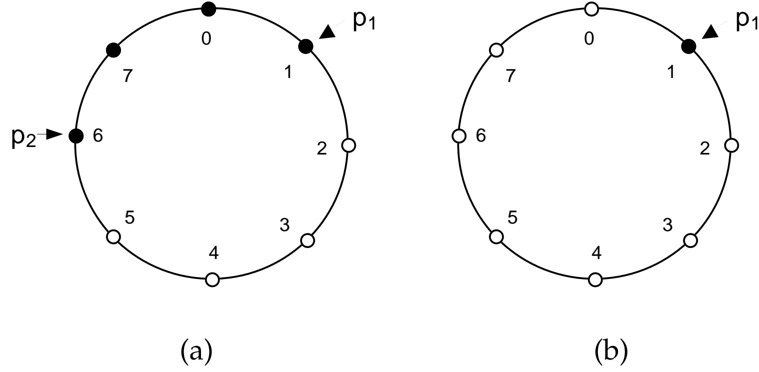

Figure 6.2: PURSUIT$_1$: Acceptable insertion zone

acceptable insertion zone at time t, as follows:

$$z(w,t) = \begin{cases} \{i|0 \le i < r\} & \text{if } lo = -1, \\ \{i|0 \le i < r, i \ne lo\} & \text{if } lo \ne -1 \text{ and there exists } j \notin I \\ & \text{such that } j > hi \text{ or } j < lo, \\ \{i|lo \le i \le hi\} & \text{otherwise.} \end{cases} \tag{6.1}$$

When the area delimited by the latest inspected indices has been seen, then this area is excluded from $z(w,t)$. Otherwise, $z(w,t)$ contains all the indices, except the smallest (if any) positive ι_t^l.

Observation 2 If $p = 1$ and at least one index has been inspected, then $|z(w,t)| = r - 1$ for any $t > 0$. Generally, if $p = 1$, then $z(w,t) \ge r - 1$.

AN INTUITIVE ASIDE.

In order to support the intuition, we refer again to the geometric version of the problem. In Figure 6.2, the acceptable insertion zone is denoted by white bullets, while those indices that do not pertain to this zone are represented by black bullets. Consider first that there are two pursuers (as we shall see in a moment, this means two processors used by the accepting real-time algorithm). Part (a) of Figure 6.2 represents the moment in which the two processors inspect indices 1 and 6, respectively. This figure

shows the acceptable insertion zone, provided that, say, processor p_1 started from index 0 and inspected only indices 0 and 1, and processor p_2 inspected only indices 7 and 6, in this order. On the other hand, when only one processor is available, the acceptable insertion zone is always the whole circle, except the most recently inspected index. This case is shown in Part (b) of Figure 6.2.

We should emphasize that *all* the indices outside the acceptable insertion zone must have been inspected by at least one processor. Indeed, consider that two processors are available and we have the same case as the one in Part (a) of Figure 6.2, except that the index 7 is not inspected by any processor. Then, according to Relation 6.1 on the page before, the acceptable insertion zone is identical to the one shown Part (b) of Figure 6.2 on the preceding page. The reason for such a constraint is the desire to faithfully model the geometric problem. First, we described the insertion operation such that it models the moves of the pursuee, by introducing the notion of insertion point, which defines the current position of the pursuee and moves accordingly after each insertion, as illustrated in Figure 6.1 on page 80. Then, since the circle is "unidimensional," neither the pursuer(s) nor the pursuee can jump over each other. This inability is modeled in the pursuee case by the presence of the acceptable insertion zone. However, the pursuers' inability to jump is another matter. Indeed, since our result is general, we cannot impose any restriction on the processing performed by the real-time algorithm that accepts the language. We therefore created a "levelled field of play," by making the algorithm lose the advantage of two pursuers if it decides to jump wherever it wants on the circle. Indeed, if the pursuers jump all over the place, then the definition of the acceptable insertion zone allows the pursuee to change almost any index in the circle. As we shall see, this makes it uncatchable.

We are now ready to define the language $\textsc{Pursuit}_1$. For $w \in L_o L_u$, let $z_i(w)$ be the set of indices whose values are modified by $w^i \in L_{ci}$ (recall that c is a fixed, positive constant,

as defined at the beginning of Section 6.1 on page 79). Then,

$$\text{PURSUIT}_1 = \{ \quad w \in L_o L_u \quad | \quad z_i(w) \subseteq z(w, ci), i > 0, \text{ and there exists } t > 0 \text{ and } i_0, 0 \leq i_0 < r, \text{ such that } |s(w,t)|_a = |s(w,t)|_b \quad \}.$$

Lemma 6.1 PURSUIT$_1$ *is a well-behaved timed ω-language.*

Proof. Consider some $w = (\sigma, \tau) \in$ PURSUIT$_1$, $w = w^0 \prod_{i>0} w^i$, with $w^0 \in L_o$ and $w^i \in L_{ci}$, $i > 0$. Monotonicity of τ follows from Theorem 4.1 on page 53, since PURSUIT$_1 \subseteq L_o \prod_{i>0} L_{ci}$. Progress of τ is immediate, since all w^i, $i \geq 0$, are bounded in length. ∎

Recall now from Definition 4.7 on page 56 that the class rt-PROC(1) is invariant with the model of computation that is considered (and is in effect the same as the class rt-PROC$^{\text{RAM}}(1)$ of sequential real-time computations). Since our results in this chapter refer exclusively to the rt-PROC hierarchy, we do not specify a particular model of computation when stating real-time sequential results, with the understanding that these results are stated in terms of RAM processing.

Lemma 6.2 *There exists no RAM deterministic real-time algorithm that accepts* PURSUIT$_1$.

Proof. Assume that there exists such an algorithm and call it A. Denote by w the current input.

In order to simplify the proof, we make the following changes to the problem of accepting PURSUIT$_1$: First, we assume that A does not have to build $s(w,t)$. Instead, this structure is magically updated, and the algorithm has access to the up-to-date $s(w,t)$ at any given time t. Next, we consider that, given some string x, A is able to decide whether $|x|_a = |x|_b$ in $|x|$ steps. Note that these assumptions make the problem easier, so that proving the nonexistence of an algorithm for this version implies the nonexistence of an algorithm deciding PURSUIT$_1$.

Consider now that the processor used by A has the ability to inspect c_1 symbols per time unit, $c_1 > 0$, and choose q such that $q = c \times c_1 + 1$. It is then immediate that, during

each interval of c times units, A can inspect at most $q-1$ symbols, and therefore, at any time t, there exists at least one symbol in $s(w,t)$ whose value is unknown to A (and thus A cannot decide whether $|s(w,t)|_a = |s(w,t)|_b$), provided that the input inserts q symbols at any time ci, $i > 0$.

Therefore, in order to complete the proof, it is enough to show that there exists a word w such that, at each time ci, $i > 0$, exactly q symbols are inserted in $s(w,t)$. Without loss of generality, consider the insertion point at time $t = ci$ to be $j = \lfloor r/2 \rfloor$ (given the circularity of $s(w,t)$, another insertion point can be considered in the same manner, by performing a simple translation). According to Observation 2 on page 83, $z(w,t)$ contains exactly all the indices in $s(w,t)$, except one (denote the latter by z_t). If $0 \leq z_t < j$, then choose $w^{i+1} = +x$, with $|x| = q$. Otherwise (that is, if $j < z_t \leq r-1$), choose $w^{i+1} = -x$, again with $|x| = q$. Note that, since A is deterministic, the index z_t is uniquely determined at any time t. It is clear that w^{i+1} is legal, since $r > 2q$ and thus its insertion does not affect any index outside $z(w,t)$ (informally, z_t is "in the other half of the circle" than w^{i+1}). ∎

We have yet to prove the second part of the result (namely, that there exists a 2-processor real-time algorithm that accepts PURSUIT$_1$). The crux of this proof is given by the following observation:

Observation 3 For some 2-processor PRAM algorithm A' and some input w ($w = w^0 \prod_{i>0} w^i$, with $w^0 \in L_o$, and $w^i \in L_{ci}$, $i > 0$), and under a judiciously chosen order of inspection of $s(w,t)$, it holds that: (a) $|z(w,t)|$ is decreasing with respect to t, and (b) for any $x \geq 0$, there exists a finite t such that $|z(w,t)| < x$. Therefore, there exists a finite time t_f for which $|z(w,t_f)| = 0$.

Proof. Consider that the two processors used by A' are able to inspect εc_1 symbols per time unit, where c_1 is as in the proof of Lemma 6.2 on the preceding page, and ε is a positive constant, arbitrarily close to 0.

Let $\iota_0^1 = 0$, and $\iota_0^2 = 1$ (recall that we denote by ι_t^l the most recent index inspected by

processor l up to time t; this notation is introduced on page 82). Thus, at each time t_1 when processor 1 inspects a new index in $s(w,t)$, let the newly inspected index be $\iota^1_{t_1}+1$ (that is, processor 1 advances always to the "right," whenever it has a chance to do so). Analogously, at each time t_2 when processor 2 inspects a new index, let this index be $(\iota^2_{t_2}-1) \mod r$. Then, according to Relation 6.1 on page 83, $|z(w,t)| \geq |z(w,t+1)|$. It follows that $|z(w,t)|$ has property (a).

As far as property (b) is concerned, let us look at the processing that A' needs to perform. First, A' needs to update $s(w,t)$ as it is changed by the timed ω-word w. However, $s(w,t)$ depends on $\prod_{cj\leq t} w^j$ only and, by Lemma 6.1 on page 85, $\prod_{cj\leq t} w^j$ is a finite word. Therefore, $s(w,t)$ can be built in finite time. Then, A' needs to keep track of the number of a's and b's that it already inspected. This is clearly achievable in finite time. Hence all the other processing that A' is required to perform (except inspecting indices in $s(w,t)$) takes finite time. Thus, after some finite time, A' inspects at least one new index. As shown above, any newly inspected symbol decreases $|z(w,t)|$. Condition (b) follows. ∎

Lemma 6.3 *There exists a 2-processor PRAM deterministic real-time algorithm that accepts* PURSUIT$_1$ *and that uses arbitrarily slow processors.*

Proof. Given Observation 3, the ability of A' to accept PURSUIT$_1$ is immediate. Indeed, note that at time t_f the acceptable insertion zone is empty. That is, at that time, no index in $s(w,t_f)$ can be changed, and A' can compare the number of a's and b's in $s(w,t_f)$. In other words, A' caught the input (or the pursuee) at time t_f. After this moment in time, A' keeps writing $\mathtt{y}$ on the decision tape.

In addition, recall that the two processors used by A' are able to compare εc_1 symbols per time unit, where c_1 is as in the proof of Lemma 6.2 on page 85, and ε is a positive constant, arbitrarily close to 0. That is, the processors used by A' are arbitrarily slow, as desired. ∎

Lemmas 6.1, 6.2, and 6.3 imply:

Theorem 6.4 rt-PROC$(1) \subset$ rt-PROC$^{\text{PRAM}}(2)$ *(strict inclusion).*

Theorem 6.4 states that a parallel real-time algorithm is more powerful than a sequential one, even if the speed of the processors that are used by the former is arbitrarily smaller than the speed of the unique processor used by the latter. To our knowledge, this is the first result of this nature to date. In fact, we can improve on the result stated in Theorem 6.4.

6.2 The Hierarchy rt-PROC$^{\text{PRAM}}$

A form of Theorem 6.4 holds for any number of processors n, $n > 1$; i.e., not only parallel real-time implementations are more powerful than sequential ones, but they also form an infinite hierarchy with respect to the number of processors used: Given any number of processors available to a parallel real-time algorithm, there are problems that are not solvable by that algorithm, but that are solvable if the number of available processors is increased, even if each processor in the new (augmented) set is (arbitrarily) slower than each processor in the initial set. To show this, we develop a language PURSUIT$_k$ similar to PURSUIT$_1$, that extends the "circle" expressed by PURSUIT$_1$ to k stacked such circles (the "k-dimensional version"). Fix $k > 1$, $q > 0$, $r > 2q$, $r' = kr$. Put $L'_o = \{(\sigma, \tau) | \sigma \in \{a, b\}^{r'}, \tau_i = 0 \text{ for all } 1 \leq i \leq r'\}$.

Let $\mathbb{N}_k = \{enc(i) | 1 \leq i \leq k\}$, where enc is a suitable encoding function from $\mathbb{N}$ to $\{I\}^*$, $I \notin \Sigma$. It is assumed that $|enc(j)| \leq j$ for any $j \in \mathbb{N}$, and that enc^{-1} is defined everywhere and computable in finite time (these properties clearly hold for any reasonable encoding function). Define $L_t^{\mathbb{N}} = \{(\sigma, \tau) | \sigma \in \mathbb{N}_k, \tau_i = t \text{ for all } 1 \leq i \leq |\sigma|\}$. Then, the multi-dimensional version of L_t is $L'_t = L_t^{\mathbb{N}} L_t$ (a word in L'_t also provides the "dimension," from 1 to k, along which the insertion takes place). Let $L'_u = \prod_{i>0} L'_{ci}$ (recall that c is a fixed, positive constant, as defined at the beginning of Section 6.1 on page 79). PURSUIT$_k$ will be a subset of $L'_o L'_u$.

We will now extend the notion of insertion modulo r to the new problem. For $w \in$

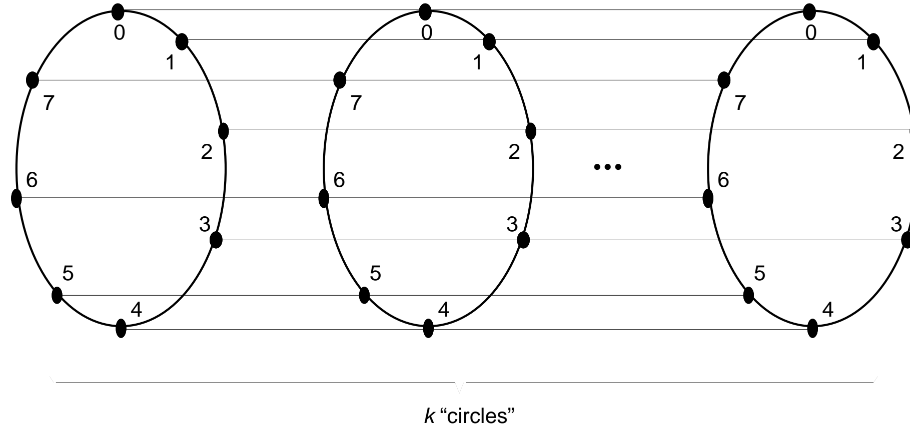

Figure 6.3: PURSUIT$_k$: The k-dimensional circle

$\{a,b\}^{r'}$, let $w = w(1)w(2)\dots w(k)$, $|w(i)| = r$, $1 \le i \le k$ ($w(i)$ is a *segment* of w), and $u = u'u''$, $u' \in \mathbb{N}_k$ and $u'' \in \Sigma^j$, $1 \le j \le q+1$, $u''_1 \in \{+,-\}$, $u''_{2\dots q} \in \{a,b\}^{j-1}$. For $0 \le i < r$, define $(w,i) \otimes u = \left(\prod_{j=1}^{d-1} w(j)\right) ((w(d),i) \oplus u'') \left(\prod_{j=d+1}^{k} w(j)\right)$, where $d = enc^{-1}(u')$. In other words, the word to be inserted contains two components, one of them (u') encoding a number, and the other one (u'') denoting the actual word that is to be inserted; the left-associative operator $\otimes$ inserts (modulo r) u'' into that segment of w which is given by u'. $\bigotimes$ is defined analogously to $\bigoplus$.

For some $w \in L'_o L'_u$ ($w = w^0 \prod_{i>0} w^i$, with $w^0 \in L'_o$, and $w^i \in L'_{ci}$), and for some i_0, $0 \le i_0 \le r-1$, let $s'(w,t) = (\sigma^0, i_0) \bigotimes_{ci \le t} \sigma^i$, where $\sigma^i = \text{detime}(w^i)$, $i \ge 0$.

One should note that $s'(w,t)$ is a generalization of $s(w,t)$ defined in Section 6.1. As a consequence, the concept of acceptable insertion zone can be naturally extended. Indeed, consider the same algorithm A that receives some $w \in L'_o L'_u$ as input and uses p processors. Then, for some $t \ge 0$, define $z^j(w,t) = z(w(j),t)$, $1 \le j \le k$, with $z(w(j),t)$ defined as in Relation 6.1 on page 83, except for the following change[4]: if, at time t, some processor l inspects an index outside $s(w(j),t)$, then $\iota^l_t(j) = -1$ and $I^l_t(j) = \emptyset$. Finally, let $z'(w,t) =$

[4]Recall that we denote by ι^l_t the most recent index inspected by processor l up to time t (-1 if no symbol is inspected), and I^l_t is the "history" of inspected symbols up to time t, i.e., $I^l_t = \bigcup_{t' \le t} \iota^l_{t'} \setminus \{-1\}$. These notation are introduced on page 82.

$\bigcup_{j=1}^{k} z^j(w,t)$, and call $z'(w,t)$ the acceptable insertion zone at time t.

AN INTUITIVE ASIDE.

The k-dimensional geometric version is a straightforward extension of the one-dimensional one. In order to present the intuitional support, we refer to Figure 6.3 on the page before. Each dimension is represented by a circle whose circumference has length r. There are, therefore, k such circles. Each collection of k identical indices (one on each circle) is connected by a special path (there are r such paths, represented by thinner lines in Figure 6.3). These paths can be used by the pursuee at no cost. However, the pursuers are too bulky to take such narrow paths, such that they are prohibited to use them. More precisely, pursuers *can* travel on them, but such a thing is suicidal: Once a pursuer uses such a path, it looses the advantage gained by the existence of the acceptable insertion zone, similarly to the case of jumping pursuers in the one-dimensional case (see Figure 6.2 on page 83).

We have now all the concepts that are necessary for defining PURSUIT_k: Recall from the beginning of Section 6.2 on page 88 that PURSUIT_k is a subset of $L'_o \prod_{i>0} L'_{ci}$ for a fixed positive constant c.Then, with $z'_i(w)$ the set of indices whose values are modified by the subword $w^i \in L_{ci}$ of w, let

$$\text{PURSUIT}_k = \{ \quad w \in L'_o L'_u \quad | \quad \text{for } i > 0,\ z'_i(w) \subseteq z'(w, ci), \text{ and there exists } t > 0 \text{ and } i_0,\ 0 \leq i_0 < r, \text{ such that } |s'(w,t)|_a = |s'(w,t)|_b \quad \}.$$

Lemma 6.5 PURSUIT_k *is a well-behaved timed ω-language for any $k > 1$.*

Proof. Trivial generalization of the proof of Lemma 6.1 on page 85. Indeed, note that any word $u \in L_t^{\mathbb{N}}$ has a finite length (specifically, a length smaller than k). ∎

Lemma 6.6 *There exists no $(2n-1)$-processor PRAM deterministic real-time algorithm that accepts* PURSUIT_n*, $n \geq 1$.*

Proof. The proof is by induction over n. Let A be a $(2n-1)$-processor PRAM deterministic real-time algorithm that accepts PURSUIT$_n$. The inexistence of A in the base case ($n = 1$) is established by Lemma 6.2 on page 85.

Given some word $s'(w,t)$, let $s'(w,t) = s_1(w,t)s_2(w,t)$, such that $|s_1(w,t)| = r$ and $|s_2(w,t)| = r' - r$.

We first note that PURSUIT$_1$ and PURSUIT$_{n-1}$ are both (strict) subsets of PURSUIT$_k$. Indeed, nothing prevents some word from PURSUIT$_k$ to change only a segment of $s(w,t)$ (and we can consider that words from PURSUIT$_1$ [PURSUIT$_{n-1}$] change only $s_1(w,t)$ [$s_2(w,t)$]).

Now, since A accepts PURSUIT$_n$, it accepts all the words $w_1 \in$ PURSUIT$_1$ (that change indices from $s_1(w,t)$). By Lemma 6.2, A has to allocate at least two processors to inspect $s_1(w,t)$. However, A also accepts PURSUIT$_k$ (whose words change indices from $s_2(w,t)$). Thus, A should allocate at least $2(n-1)$ processors for the inspection of $s_2(w,t)$ (since $2(n-1)-1$ processors are not enough by inductive assumption). In all, A should use at least $2n$ processors, a contradiction.

Therefore, the only possible processor allocation remaining must require that (at least) one of the processors that inspect $s_1(w,t)$ also inspects from time to time indices from $s_2(w,t)$. Let's say that such an inspection takes place each t_p time units. Then, given the definition of the acceptable insertion zone of $s_1(w,t)$, each t_p time units an input word is allowed to change any index of $s_1(w,t)$. By the same argument as the one used to prove Lemma 6.2, it follows that A can no longer decide whether at some moment in time the number of a's equals the number b's in $s_1(w,t)$. Therefore, A is no longer able to accept PURSUIT$_n$. In other words, the processor allocation in which only a part of the computational power of some processor is allocated to $s_1(w,t)$ is not acceptable, since such a processor has no influence on the acceptable insertion zone of $s_1(w,t)$ and thus such a processor becomes useless. Indeed, without the restrictions imposed by the acceptable zone, the algorithm cannot keep up with the changes for q large enough (specifically, for $q > 2c \times c_1$, where c_1 is the maximal number of symbols that can be inspected by a processor in one

time unit). Again, we reach a contradiction.

We exhausted all the possible processor allocation schemes, so the induction is complete. ∎

Lemma 6.7 *There exists a* $2n$*-processor PRAM deterministic real-time algorithm that accepts* PURSUIT_n *and that uses arbitrarily slow processors,* $n \geq 1$.

Proof. By allocating two processors for each $s'(w,t)(j)$, $1 \leq j \leq n$, it is possible to handle the changes, as shown by Lemma 6.3 on page 87. Since there are $2n$ processors, such an allocation is clearly achievable. ∎

By Lemmas 6.5, 6.6 and 6.7, the hierarchy $\text{rt-PROC}^{\text{PRAM}}$ is infinite:

Theorem 6.8 *For any* $n \in \mathbb{N}$, $n \geq 1$, $\text{rt-PROC}^{\text{PRAM}}(2n-1) \subset \text{rt-PROC}^{\text{PRAM}}(2n)$ *(strict inclusion).*

6.3 The Strong Hierarchy rt-PROC

One may wonder whether Theorem 6.8 holds for other models of computation beside the PRAM. A model that allows a straightforward implementation, as opposed to the PRAM, is the *bounded-degree network* (BDN) [5], where communication between processors is achieved using a sparse interconnection network of fixed degree. However, it is well-known that even the most powerful version of the PRAM (namely, the CRCW PRAM [5]) can be simulated on a BDN with bounded slowdown and bounded memory blowup. Specifically, there exists a simulation [47] for which the slowdown is $O(\log^2 n / \log \log n)$, and the memory blowup is $O(\log m / \log \log m)$, where n is the number of processing elements, and m is the amount of memory used by the PRAM.

However, a bounded slowdown does not affect the result in Theorem 6.8, since this result is invariant to the speed of the processors involved. Furthermore, the PRAM algorithm uses a finite amount of memory; thus, a bounded memory blowup results in a

finite amount of memory as well for the BDN that simulates the PRAM algorithm. In addition, given that the BDN allows for an immediate physical implementation, we make the following assumption: The BDN is the most elementary model of parallel computation. Indeed, such a claim is consistent with the notion of "invariant with the model of computation" used throughout the thesis and described on page 17. With this assumption in mind, the following result is an immediate corollary of Theorem 6.8:

Theorem 6.9 *Given any model of parallel computation M, and for any $n \in \mathbb{N}$, $n \geq 1$,* $\text{rt-PROC}^M(2n-1) \subset \text{rt-PROC}^M(2n)$ *(strict inclusion).*

That is, we have not only an infinite hierarchy $\text{rt-PROC}^{\text{PRAM}}$, but such a result holds for rt-PROC^M as well, for any model of parallel computation M.

It should be noted that the languages PURSUIT_k, $k \geq 1$, faithfully model the geometrical variant of the problem. We chose this direction in order to preserve the clear and intuitive support provided by the geometrical case. However, the notion of insertion point (that moves after each insertion) is not necessary. It is immediate that the results in this chapter hold even if the input is allowed to change (any number of) arbitrary indices within the acceptable insertion zone at any time ci, $i > 0$.

6.4 On Practical Issues and Why the Hierarchy rt-PROC does not Collapse

Questions could be raised regarding whether the infinite hierarchy developed in this chapter contains any practically interesting problems. In addition, one may also wonder why is the case that such a hierarchy does not collapse when standard simulation techniques (of several processors by one) are used. We thus conclude this chapter by a discussion on these two issues.

We address first the possible collapse of the hierarchy by simulation techniques. The standard simulation of a parallel machine is applicable as long as the input is *oblivious*

to the machine that accepts it. Once the input is aware of the algorithm that drives the machine (and vicious enough to take advantage of it), such a simulation is no longer be applicable (because the input becomes aware that a simulation takes place as soon as this happens). In particular, the real-time processing described by the languages PURSUIT_k, $k > 0$, implies that

1. the input depends on the actions performed by the algorithm (the system is *interactive*), and
2. the input reacts to the number of *physical* processors that are used by the algorithm.

We showed in fact that, in such a (time-sensitive and interactive) setting, no simulation can collapse the hierarchy—in other words, simulation techniques are not necessarily applicable in such environments.

As for the practicality of such a computational setting, we note that both Property 1 and Property 2 do appear in practice. Indeed, algorithms whose input depends on the actions performed by the algorithm are common for example in systems that control *industrial processes*: Such a system receives input (from various sensors); based on the received data, it then alters the state of the controlled process. Future input (from the same sensors) depends on the state of the process and thus on the output produced in previous computational steps. This example clearly illustrates the existence of Property 1 in practice.

Following the same idea, this time in the processor hierarchy, the input depends on the processing performed by the acceptor. Specifically, the way the "circle" is inspected determines which symbols from that circle can be changed by the input (formally, input words that modify symbols outside the permitted locations are immediately rejected, as they do not belong to the language). The way in which the permitted locations are established does depend on the number of processors used by the acceptor (and the way those processors inspect the circle).

One should note that the real-time task described by some PURSUIT_k language is at

the same time more complex and more restricted than algorithms that control industrial processes. In particular, no industrial process cares about the structure of the computer that controls it, and thus Property 2 on the page before in not applicable. However, such a property does manifest itself in *real-time operating systems*. Indeed, it is likely that the (real-time) operating system of a parallel machine will behave differently when the number of processors it manages varies (scheduling, concurrency, and calibration of kernel loops are just some processes that depend on the number of actual, physical processors driven by the operating system). It is immediate that a real-time operating system is a real-time computation itself, and thus it is a relevant example of processing for which the pursuit hierarchy may be relevant.

Chapter 7

Complexity of Real Time II: Logarithmic Space Computations are Real Time

Summary

Based on the theory of timed ω-languages, we study (classical) languages that can be recognized in nondeterministic logarithmic space (NLOGSPACE), augmented with real-time constraints (including but not limited to deadlines). The main result of this chapter is Theorem 7.10 on page 108, showing that all such computations can be carried out successfully in parallel on the reconfigurable bus machine (RMBM), no matter how tight the time constraints are (refer to page 18 for a definition of RMBM, its variants, and its properties).

Besides the main result, we also offer a tight characterization of constant time computations on RMBM. We show that constant time directed RMBMs have the same computational power as the directed *reconfigurable networks*, and that there is no need for such powerful write conflict resolution rules as Priority or Common. Indeed, they do not add computational power over the easily implementable Collision rule. We also find an interesting gap result. Indeed, as far as constant time computations on RMBMs are concerned, we show that a unitary bus width is enough. That is, a simple wire as bus will do for all constant time computations on directed RMBM.

7.1 RMBM and NLOGSPACE Computations

In this section, we first show that the *graph accessibility problem* (GAP) can be solved by a DRMBM in constant time. Then, we investigate the relation between RMBM and NLOGSPACE computations. We show that directed RMBMs with polynomially bounded resources and constant running time recognize exactly all the languages in NLOGSPACE. Recall from Section 2.2 on page 16 that, in the Collision resolution rule for a CRCW RMBM (CRCW RN, etc.), two values simultaneously written on a bus result in the placement of a special collision value on that bus.

Definition 7.1 $GAP_{1,n}$ denotes be the following problem: Given a directed graph $G = (V, E)$, $V = \{1, 2, ..., n\}$ (expressed, for example, by the (boolean) incidence matrix I), determine whether vertex n is accessible from vertex 1. In general, the problem of determining whether vertex j is accessible from vertex i is denoted by $GAP_{i,j}$.

Lemma 7.1 $GAP_{1,n} \in$ CRCW F-DRMBM$((n^2 - n)/2, n, 2)$. *Furthermore, the F-DRMBM family that solves* $GAP_{1,n}$ *uses the Collision resolution rule and has bus width* 1.

Proof. The following RMBM algorithm is a variant of the algorithm that computes the shortest path in a directed graph [67] (which is itself an adaptation of the algorithm for the minimum spanning tree [87]). However, we are not interested in the length of an eventual path, so that our construction requires considerably less resources.

For convenience, each processor is denoted by p_{ij}, $1 \leq i < j \leq n$. When we say that some processor fuses buses k and l, we imply that this fusion is directional, such that a signal placed on bus k is seen on bus l, but not vice versa. We assume that each processor p_{ij} knows the value of both I_{ij} and I_{ji}, where I is the incidence matrix. Then, the algorithm performs the following steps:

1. Each processor p_{ij}, $1 \leq i < j \leq n$, fuses buses i and j if and only if $I_{ij} = True$. Simultaneously, p_{ij} fuses buses j and i if and only if $I_{ji} = True$.

2. p_{13} places a signal on bus 1, and p_{12} listens to bus n. p_{12} reports[1] *True* if it receives some signal (either the original one emitted by p_{13} or the signal corresponding to a collision), and *False* otherwise.

Note that, even if only one processor writes on the buses, the algorithm cannot be implemented on an exclusive-write RMBM, as the signal emitted by p_{13} may reach some bus on more than one path. We must show that p_{12} reports true if and only if vertex n is accessible from vertex 1. In fact, it can be easily proved by induction on the length of the path from s to t that, for any s, t, $1 \leq s, t \leq n$, a signal placed on bus s reaches bus t if and only if vertex t is accessible from vertex s, and this completes the proof (just put $s = 1$ and $t = n$). Indeed, both steps of the algorithm can be clearly performed in one machine cycle each. As well, note that the content of the signal emitted by p_{13} is immaterial, so that a bus width of 1 suffices. ∎

Corollary 7.2 *If the input graph $G = (V, E)$ of $GAP_{1,n}$ is given by a list of vertices L instead of an incidence matrix, then $GAP_{1,n} \in$ CRCW F-DRMBM$(m, n, O(1))$, where $m = |E|$ and $n = |V|$.*

Proof. Identical to the algorithm in the proof of Lemma 7.1 on the preceding page, except that, at step 1 of this algorithm, processor p_{ij} fuses buses i and j if and only if $(i, j) \in L$. ∎

It is worth mentioning that the algorithm presented in [87] uses a CREW DRMBM (as opposed to the CRCW F-DRMBM used in Lemma 7.1 on the page before and Corollary 7.2). Furthermore, this algorithm computes the shortest path between two vertices. Therefore, it implicitly computes $GAP_{1,n}$. This lets us conclude that $GAP_{1,n} \in$ CREW DRMBM$(2mn, n^2, O(1))$. However, in what follows, we will use the result based on the CRCW F-DRMBM since, on one hand, it uses resources more efficiently, and, on the other hand, we believe that a Collision conflict resolution rule is just as realistic as exclusive write.

[1] In fact, neither p_{13} nor p_{12} have any special characteristics, and any pair of distinct processors will do.

Consider now some language L in NSPACE$(\log n)$. It follows that there exists a nondeterministic Turing machine $M = (K, \Sigma, \delta, s_0)$ that accepts L and uses $O(\log n)$ working space (by abuse of notation, we call M an NSPACE$(\log n)$, or NLOGSPACE, Turing machine). Without loss of generality, we consider that the working and input alphabet of M are both $\Sigma = \{0, 1\}$. Let k be the number of states of M, that is, $k = |K|$. The transition function is denoted by δ, $\delta : (K \times \Sigma \times \Sigma) \to \mathcal{P}((K \cup \{h\}) \times (\Sigma \cup \{L, R\}) \times \{L, R, \lambda\})$, and the initial state by s_0 (recall that $\mathcal{P}(()\Sigma)$ stands for the powerset of Σ). For the sake of simplicity, we consider that M has one working tape only (the extension for multiple working tapes is immediate [45, 84]).

Recall from Section 2.1 on page 12 that $poly(n)$ represent the upper bound for polynomial functions of one variable n, that is, $poly(n) = n^{O(1)}$. As well, recall that a *configuration* of M working on input x is defined as containing the current state, the content of its tapes, and the head position on each tape. Denote such a configuration by (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. Note that the content of the input tape is established at the beginning of the computation (indeed, the input tape contains the input x) and does not change. Therefore, the input tape does not change the configuration, except for its head position.

Since M is nondeterministic, the set of possible configurations of M working on x forms a directed graph (denote it by $G(M, x) = (V, E)$) as follows: V contains one vertex for each and every possible configuration of M working on x, and $(v_1, v_2) \in E$ if and only if the configuration corresponding to v_2 can be reached from the configuration corresponding to v_1 in one step of M (that is, if and only if $v_1 \vdash v_2$). In the following, we refer to both a configuration and the vertex denoting that configuration in the associated graph simply as "configuration," since there exists a one to one correspondence between vertices and configurations.

It is clear that $x \in L$ if and only if 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). One should also note that there are $poly(n)$ possible configurations of M. Indeed, for any configuration (s, i, w, j), i can take $n = |x|$ values. Furthermore, since $|w| = O(\log n)$, there are at most $poly(n)$ possible contents of the working tape, and j can take $O(\log n)$ values. Given that the set of states K is fixed, the number of possible configurations is $poly(n)$.

Therefore, for any language $L \in \text{NSPACE}(\log n)$ and for any x, determining whether $x \in L$ can be reduced to the problem of computing the graph accessibility problem (GAP) for the graph $G(M, x) = (V, E)$, where M is some Turing machine deciding L, $M \in \text{NSPACE}(\log n)$. In fact, a stronger result is immediate: Given x, L, M, and $G(M, x)$ as above, we consider without loss of generality that the initial state is represented by vertex 1 and the (unique) final state by vertex n in $G(M, x)$. Then, any problem in $\text{NSPACE}(\log n)$ can be reduced to $GAP_{1,n}$. Indeed, we are interested only in the reachability of vertex n (final state) from vertex 1 (initial state).

Lemma 7.3 *Fix a language $L \in \text{NSPACE}(\log n)$. Let $M = (K, \Sigma, \delta, s_0)$ be an $\text{NSPACE}(\log n)$ Turing machine that accepts L. Then, given some word x, $|x| = n$, there exists a CREW F-DRMBM algorithm that computes $G(M, x)$ (as an incidence matrix) in $O(1)$ time, and uses $poly(n)$ processors and $poly(n)$ buses of width 1.*

Proof. The configurations of $G(M, x)$ do not depend on x, but only on M. Therefore, we consider that these configurations are known in advance. That is, the set V of vertices of $G(M, x)$ is known beforehand, even if the set E of edges changes with x. In addition, the transition function δ is known to all the processors.

Put $n' = |V|$ ($n' = poly(n)$). Then, the RMBM algorithm uses $(n + (n'^2 - n')/2)$ processors, as follows: The first n processors, denoted by p_i, $1 \leq i \leq n$, contain the current input x (in the sense that each p_i contains x_i, the i-th symbol of x). At the beginning of each computational step, p_i writes x_i to bus i. Since $x_i \in \{0, 1\}$, a bus width of 1 is enough.

We shall refer to the remaining $(n'^2 - n')/2$ processors as p_{ij}, $1 \leq i < j \leq n'$. Initially, a processor p_{ij} holds a false initial value for the elements I_{ij} and I_{ji} of the incidence matrix I. Then, each p_{ij} considers the (potential) edges (v_i, v_j) and (v_j, v_i) corresponding to I_{ij} and I_{ji}, respectively. If such edge(s) exist, p_{ij} writes *True* to I_{ij} and/or I_{ji} as appropriate. Otherwise, it does nothing. There is no interprocessor communication between processors p_{ij}, $1 \leq i < j \leq n'$, thus any RMBM model is able to carry on this computation.

It remains to show that determining whether there exists an edge (v_i, v_j) is computable in constant time by one processor (p_{ij} or p_{ji}). Clearly, given a configuration v_i, p_{ij} can compute in constant time any configuration v_l accessible in one step from v_i (if $v_i = (s, z, w, y)$, then v_l is obtained by possibly changing the state s, incrementing, decrementing or keeping z and/or y unchanged, and changing at most one symbol from w, everything according to δ). Recall now that $\delta : (K \times \Sigma) \rightarrow \mathcal{P}((K \cup \{h\}) \times (\Sigma \cup \{L, R\}) \times \{L, R\})$, and note that $|\mathcal{P}((K \cup \{h\}) \times (\Sigma \cup \{L, R, \lambda\}) \times \{L, R\})| = O(2^k)$ (since $|\Sigma| = 2$, and $|K| = k$). That is, the number of configurations that are accessible from some given configuration is constant ($O(2^k)$). In other words, p_{ij} computes (in constant time) a constant number (at most $O(2^k)$) of possible configurations. Note that, in addition, p_{ij} can hold s and w in two of its registers, and it has access to any symbol x_i of the input by simply reading bus i. After this, p_{ij} can decide whether v_j is accessible from v_i in constant time by simply checking the membership of v_j in the set of the newly computed configurations. It follows that p_{ij} computes I_{ij} and I_{ji} in constant time, and this completes the proof. ∎

Some comments on the RMBM algorithm developed in the proof of Lemma 7.3 are in order. One can note that the constant running time of this algorithm may be quite large ($O(2^k)$; furthermore it depends on the number of states in the initial Turing machine). On the other hand, the subsequent use of Lemma 7.3 will emphasize the need for the RMBM algorithm to be as fast as possible. Thus, even if theoretically sound, the dependency of the running time on the number of states is not a desirable feature.

However, given some nondeterministic Turing machine $M = (K, \Sigma, \delta, s_0)$, one can build an equivalent Turing machine $M' = (K', \Sigma', \delta', s_0)$ such that, for any $s2 = \delta'(s1)$, $|s2| \leq 2$. Indeed, take some state $s \in K$ such that $S' = \delta(s, \alpha, \beta)$ for some $\alpha, \beta \in \Sigma$, and $|S'| > 2$. Then, introduce a set K_s of new, distinct states (which do not change the tapes' content or head positions) to K', such that the graph corresponding to δ' restricted to $K_s \cup \{s\}$ is a binary tree rooted at s, with exactly all the terminal nodes in S', and with all the nonterminals (except the root) from K_s. Clearly, M' is equivalent to M, in the sense that they accept the same language and use the same amount of space.

One can now build the algorithm A from Lemma 7.3 based on M' instead of M. Then, although $G(M, x)$ may grow (still, $|V|$ remains $O(n)$), the running time of A is now upper bounded by a very small constant, and this constant no longer depends on the number of states of M (or M' for that matter).

From Lemma 7.1 on page 97 and Lemma 7.3 on page 100, it follows that

Lemma 7.4 NLOGSPACE $\subseteq$ CRCW F-DRMBM$(poly(n), poly(n), O(1))$, *with the Collision resolution rule and bus width* 1.

Proof. Given some language L in NSPACE$(\log n)$, let M be the (NSPACE$(\log n)$) Turing machine accepting L. For any input x, the F-DRMBM algorithm that accepts L works as follows: Using Lemma 7.3, it obtains the graph $G(M, x)$ of the configurations of M working on x (by computing in effect the incidence matrix I corresponding to $G(M, x)$). Then, it applies the algorithm from Lemma 7.1 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} and I_{ji} computed by (and stored at) p_{ij} in the algorithm from Lemma 7.3 are in the right place as input for p_{ij} in the algorithm from Lemma 7.1. It is immediate given the aforementioned lemmas that the resulting algorithm accepts L and uses no more than $poly(n)$ processors and $poly(n)$ buses of constant width. ∎

Conforming to Lemma 7.4, any NLOGSPACE language can be accepted in constant time by a directed RMBM. In fact, the relation between directed RMBMs and NLOGSPACE languages is even stronger:

Lemma 7.5 CRCW DRMBM$(poly(n), poly(n), O(1)) \subseteq$ NLOGSPACE, *for any write conflict resolution rule and any bus width.*

Proof. Let R be some RMBM in CRCW DRMBM$(poly(n), poly(n), O(1))$ performing step d of its computation ($d \leq O(1)$). Suppose that there exists a Turing machine M_d that generates the description of R after step d using $O(\log n)$ space. Then, by standard techniques [45], one can modify M_d (obtaining M'_d) such that M'_d receives n 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.

Clearly, M_0 exists by the definition of a (uniform) RMBM family. We now assume the existence of M_{d-1} (and thus M'_{d-1}) and show how M_d is constructed. For each processor p_i and for 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 $GAP_{j,i}$. $GAP_{j,i}$ is clearly computable in nondeterministic $O(\log n)$ space (it is a simplification of the Graph Accessibility Problem, which is NLOGSPACE-complete [84]; the local configurations of fused and segmented buses at each processor are obtained by calls to M'_{d-1}). 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: If b is empty, then it is set to v (p_j is currently the only processor that writes something to bus k), and ρ is set to j. Otherwise,

1. If R uses the Collision resolution rule, the collision signal is immediately placed in b. The value of ρ is immaterial.

2. When the Common rule is used, M_d compares b and v. If they are different, the input is rejected immediately. The value of ρ is again immaterial.

3. 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 Arbitrary rule is handled similarly, except that the decision whether to modify b and ρ is made arbitrarily instead of being based on the values of j and ρ.

4. If R uses the Combining resolution rule with $\circ$ as combining operation, b is set to the result of $b \circ v$. The operation can be performed in $O(\log n)$ space, since the length of both b and v is $O(\log n)$, and $\circ$ is computable in linear space. As well, the operation $\circ$ is associative. It follows that, once all the processors p_j have been considered, the content of b is 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 space required by M_d is the space for the configuration of p_i itself, plus the space for the configuration of one other processor, plus the space required by b, v, and ρ. The latter three values cannot be of size larger than $O(\log n)$ (since the word size of any processor is $O(\log n)$ and the number of processors is $poly(n)$), and the configurations clearly take $O(\log n)$ space. Thus, the whole computation of M_d takes $O(\log n)$ space, and the induction is complete. ∎

Lemma 7.4 on page 102 and Lemma 7.5 on the page before imply the following results:

Theorem 7.6 CRCW DRMBM$(poly(n), poly(n), O(1))$ = NLOGSPACE, *for any write conflict resolution rule and any bus width. For any write conflict resolution rule and any bus width,* CRCW DRMBM$(poly(n), poly(n), O(1))$ = CRCW F-DRMBM$(poly(n), poly(n), O(1))$ *with the Collision resolution rule and bus width* 1.

Corollary 7.7 DRMBM$(poly(n), poly(n), O(1)) =$ DRN$(poly(n), O(1))$.

Proof. Immediate from Theorem 7.6, since NLOGSPACE = DRN$(poly(n), O(1))$ [19]. ∎

The following is a generalization of Theorem 7.6:

Corollary 7.8 *Consider a problem π solvable in constant time by some (directed or nondirected) RMBM family using $poly(n)$ processors and $poly(n)$ buses. For any such a problem π, it holds that $\pi \in$ CRCW F-DRMBM$(poly(n), poly(n), O(1))$ with the Collision resolution rule and bus width* 1.

Proof. From Theorem 7.6 and Observation 1 on page 22. ∎

We note that the power of (nondirected) RMBMs has been investigated in [88], where it is shown that nondirected RMBMs are exactly as powerful as nondirected RNs, and that the Collision, Common, Arbitrary, and Priority rules are equivalent in power. In addition, RNs (and thus RMBMs) solve in constant time exactly all the problems in LOGSPACE [19]. By Theorem 7.6 on the page before and Corollary 7.7 we extend these results to the directed variants of RMBMs and RNs running in constant time. As expected, DRMBMs, DRNs, and logarithmic space bounded nondeterministic Turing machines are found to have the same computational power. Corollary 7.8 shows that, again, the Collision, Common, Arbitrary, and Priority rules are equivalent to each other. In addition though, we show[2] in Corollary 7.8 that a resolution rule apparently much more powerful than the others, namely Combining, adds no computational power either. Then, for constant time computations on DRMBM, bus width does not matter; any problem can be solved using buses of unitary width. Finally, as is the case of (nondirected) RMBMs, it follows from Corollary 7.8 that segmenting buses does not add computational power over fusing buses.

[2]Also see the proof of Lemma 7.5 on page 103.

7.2 Small Space Computations Are Real-Time

We have now all the necessary ingredients to state the first result linking real time with logarithmic space computations. First though, we have to make an additional assumption: We henceforth consider that the deadlines imposed on real-time computations are reasonably large compared to the processor clock frequency. We believe that this is a reasonable assumption. Indeed, nowadays processors operate at frequencies around (and sometimes exceeding) 1GHz; still, we are not aware of any real-time application that requires deadlines measured in nanoseconds.

We have shown in Section 5.1 on page 59 that the potential existence of a *deadline* can be modeled as a well-behaved timed ω-word (denote such a word by W_d). Recall that W_d has the following semantics: The special symbol w is present whenever the current time does not exceed the deadline; if the deadline passed, then the symbols that arrive as input are all d. If the computation is completed at a moment in which the input symbol is w, then it has met the associated deadline; otherwise, the deadline has passed.

With this definition of W_d, and for any problem $\pi \in \text{NSPACE}(\log n)$, let

$$\pi_\tau = \{ \ (\sigma\sigma_d, \tau) \ | \ \sigma \text{ is some input for } \pi,\ \sigma_d = \text{detime}(W_d) \text{ for some timed word } W_d \text{ modeling a deadline, and } \tau \text{ is some well-behaved time sequence with } \tau_1 = \tau_2 = \cdots = \tau_{|\sigma|} \ \}.$$

In other words, π_τ represents the problem π in the (potential) presence of deadlines. Then, the relation between NLOGSPACE and real-time computations can be informally stated as follows: Suppose one has a (possibly infinite) set of inputs for a collection of problems in NLOGSPACE. We impose some (any) deadline for each of these inputs, and we feed them at various time moments to some machine. If that machine happens to be a CRCW F-RMBM, then it is able to handle the input successfully. Formally, given Theorem 7.6 on page 104 (and noting that the size complexity of an RMBM with $poly(n)$ processors and $poly(n)$ buses is $poly(n)$), we have the following relation linking NLOGSPACE with real-time computations.

Theorem 7.9 $\bigcup \left(\prod_{\pi \in \text{NSPACE}(\log n)} \pi_\tau \right) \subseteq \text{rt-PROC}^{\text{CRCW F-DRMBM}}(poly(n))$, *where* n *is the maximum input size for problems* π.

Proof. All the processing implied by Theorem 7.6 on page 104 (namely, the algorithms from Lemmas 7.1 and 7.3) takes very little (and constant) time, and thus accommodates any reasonable (in the sense of the above assumption) time sequence τ associated with the computation. ∎

In Theorem 7.9, we *added* deadlines (that is, real-time constraints) to problems. We conclude this chapter by considering the reversed problem, namely how can one *eliminate* the real-time qualifier from the specification of some problem. This will allow us to offer a more concise formulation of Theorem 7.9, and will also form a basis for strengthening Theorem 7.9, as we shall see in Chapter 8 on page 109.

Analyzing the form of the word W_d (see Section 5.1 on page 59) modeling deadlines offers the clue. Indeed, one can notice that, from some time on, the symbols from W_d no longer represent the input. Instead, they consists of symbols w and d that model the timing constraints imposed on the computation. Similarly, in a real-time problem for which the input is virtually endless, a prefix of that input represents the same problem, except that in the case of such a prefix, the input "stops coming" at some time. This is the most general restriction to a classical environment one can model, since the input is finite in such an environment:

Definition 7.2 Consider some well-behaved timed ω-language L. For some $(\sigma, \tau) \in L$, $i > 0$ is a *progression point* if and only if[3] $\tau_i \neq \tau_{i+1}$.

Let $L_s = \{\sigma' |$ there exists some finite progression point n such that $(\sigma, \tau) \in L$ and $\sigma' = \sigma_{1\ldots n}\}$ (each word in L_s is constructed by taking a word from L, restricting its length to some finite n, and discarding the time sequence). If, for some complexity class C, $L_s \in C$, then we say that $L \in C/rt$ (L is the *real-time counterpart* of L_s; alternatively, L_s solves the

[3] One does not want to split a bunch of symbols arriving at the same time, since such a bunch often represents a nondivisible piece of the input...

same problem as L, but without real-time constraints, and thus L_s is the *static version* of L; by abuse of notation, we also say that $L = L_s/rt$).

Definition 7.2 allows us to study the pursuit problem in the context of Theorem 7.9 on the preceding page (as we shall do in Chapter 8 on the next page). As promised, it also offers a more concise formulation of Theorem 7.9:

Theorem 7.10 $\text{NSPACE}/rt(\log n) \subseteq \text{rt-PROC}^{\text{CRCW F-DRMBM}}(poly(n))$.

It is immediate that the two formulations are equivalent, while the one expressed by Theorem 7.10 is easier to understand.

Chapter 8

Complexity of Real Time III: Real Time Computations are Logarithmic Space?

Summary

In some sense, one may argue that the inclusion relation from Theorem 7.10 on the preceding page is in fact an equality, conforming to Theorem 7.6 on page 104. Indeed, NLOGSPACE computations are *the only* computations in the classical sense that can be performed in constant time by DRMBMs, no matter how many processors and buses are used; thus, given any deadline (in effect imposing a constant upper bound on the running time), it follows that no computation outside NLOGSPACE can be successfully carried out.

Still, there might exist real-time computations (not exhibiting explicit deadlines and thus not necessarily having constant time constraints) that are not in NLOGSPACE but can still be performed within the given resource bounds (that is, a polynomial number of processors and buses). We now study this issue. In other words, we ask whether it holds that any real-time problem is in NLOGSPACE once real-time constraints are eliminated, and thus the inclusion from Theorem 7.10 is actually an equality. We present strong evidence that this is in fact the case.

Indeed, one candidate for such computations can be the family of timed ω-languages

PURSUIT$_k$, $k \geq 1$, presented in Chapter 6 on page 78. Those languages, modeling the k-dimensional version of the *pursuit and evasion on a ring* problem, do not feature explicit deadlines. The real-time qualifier is instead given by the "movements of the pursuee," that is, by the real-time input arrival. We show in Section 8.1 on the next page that the non-real-time versions of languages PURSUIT$_k$ are solvable in deterministic logarithmic space, even if these languages are quite hard to solve in real time (recall that they form an infinite hierarchy with respect to the number of processors).

Another paradigm pertaining to the same class of real-time problems (without explicit deadlines) is the *data-accumulating paradigm*. In Section 8.2 on page 113 we give further consideration to d-algorithms (introduced in Section 2.3.1 on page 23), and we show first that d-algorithms are nothing more than on-line algorithms, over which some real-time constraints are imposed. Since on-line algorithms are a restricted case of general algorithms, this is already good evidence that successful d-algorithms are based on NLOGSPACE computations according to the results from Chapter 7 on page 96. We can, however, go even further, showing that the real-time input constraints of a d-algorithm impose in effect *explicit* output deadlines. Thus, we are closer to showing that Theorem 7.10 on page 108 can be strengthened.

Do real-time input restrictions impose deadlines on other computations than d-algorithms (which are of quite reduced relevance in practice)? Indeed, they do. We conclude this chapter with Section 8.3 on page 122, where we consider *correcting algorithms* (introduced in Section 2.3.2 on page 28), a more realistic class of real-time computations (indeed, one of the most general settings for the real-time input arrival paradigm). We show that they have the same properties as d-algorithms. In particular, this implies that real-time input arrival implies explicit output deadlines.

All these results allow us to state the following:

Claim 2 $\text{NSPACE}/rt(\log n) = \text{rt-PROC}^{\text{CRCW F-DRMBM}}(poly(n))$.

In other words, we conjecture that NLOGSPACE contains exactly all computations that can be carried out in real time on RMBM.

Theorem 7.10 on page 108 and Claim 2 are stated for RMBM computations. However, RMBM is a feasible model [20, 44, 88], so we feel in fact confident in dropping the "on RMBM" qualifier from the statement above, claiming that, in general, *NLOGSPACE contains exactly all computations that can be carried out in real time*.

According to the current body of knowledge regarding the power of various computational models, it appears that the power offered by reconfigurable buses is needed in order to establish Theorem 7.10 and Claim 2. However, similar results may hold for other models of parallel computation. We address this issue in Section 8.4 on page 129, where we offer a generic characterization of minimum requirements for models on which Theorem 7.10 and Claim 2 hold.

8.1 Non-Real-Time Pursuit Is Easy

We show now that pursuing something is easy outside the real-time paradigm: Recall from Chapter 6 on page 78 that $\textsc{Pursuit}_k$ denotes the "k-dimensional version" of the pursuit and evasion problem. Then[1],

Theorem 8.1 *For any $k > 0$, $\textsc{Pursuit}_k \in \text{SPACE}/rt(\log n)$.*

Proof. Let C be a class such that $\textsc{Pursuit}_k \in C/rt$. We shall show that $C = \text{LOGSPACE}$ and we are done. According to Definition 7.2 on page 107, a word w_s in the static version of $\textsc{Pursuit}_k$ has the following structure: Denote $|w_s|$ by n; then, w_s contains

- An initial word $w^0 \in \{a, b\}^r$ for some $r \leq n$; this is the initial configuration, which the pursuee modifies as time passes.

[1] Recall from Definition 7.2 on page 107 that, for some complexity class C, C/rt denotes the class of problems from C in the presence of real-time constraints.

- Some number m of moves by the pursuee (denoted by some words $w^i \in L_{ci}$, $1 \leq i \leq m$); such a move in effect changes a maximum of p symbols from w^0, $p < r$.

It is clear that $r, p, m \leq n$, since n is the length of the whole input. Consider now a deterministic Turing machine M accepting the static version of PURSUIT$_k$. In order to determine the number of a's and b's in w^0, M simply keeps two counters C_a and C_b, one for a's and the other for b's, respectively. As the input is scanned, the two counters are incremented accordingly.

Once the end of w^0 is reached, M performs the following step for each w^i, $1 \leq i \leq m$: M identifies that portion of w^0 which is changed by w^i. Then, M scans this portion, decrementing C_a or C_b for each a or b it encounters during this procedure. Finally, M identifies that portion of w^i that changes w^0 and scans it, incrementing C_a and/or C_b accordingly. It is clear that, at the end of step m of such a computation, C_a and C_b contain precisely the number of a's and b's, respectively, that are present in w^0 as it is changed by all w^i, $1 \leq i \leq m$. Therefore, when the end of the input is reached, M simply compares C_a and C_b and accepts the input if and only if they are identical.

Clearly, C_a and C_b take $\log r$ space each (since there are at most r a's and at most r b's in w_0). The identification procedure mentioned above uses two pairs of counters, each pair delimiting the portions of interest in w^0 and the current w^i, respectively. Each of these four counters holds an index in the current input, hence it can be stored in $\log n$ space. Finally, setting these counters involves simple arithmetic operations on indices (that is, numbers bounded above by n), hence they are computable in LOGSPACE. Therefore, the space required by the whole computation is $O(\log n)$, as desired. ∎

Theorem 8.1 is an interesting result. Indeed, even if PURSUIT$_k$ is a problem that requires a lot of computational effort (in particular, it cannot be solved at all if less than $2k$ processors are available according to Theorem 6.9 on page 93), it becomes a very simple problem (not only in NLOGSPACE, but even in LOGSPACE) once the real-time constraints are eliminated. Thus, Theorem 8.1 justifies Claim 2 on page 110. We can offer even stronger

evidence for this conjecture though.

8.2 The Characterization of D-Algorithms

Recall from Section 2.3 on page 23 that a *d-algorithm* works on an input considered as a virtually endless stream. The computation terminates when all the currently arrived data have been processed before another datum arrives. In addition, the arrival rate of the input data is given by some function $\phi(n, \mathbb{t})$ (called the *data arrival law*), where n denotes the amount of data that is available beforehand, and $\mathbb{t}$ denotes the time. The family of arrival laws most commonly used as example was introduced by Relation 2.1 on page 24:

$$\phi(n, \mathbb{t}) = n + kn^{\gamma}\mathbb{t}^{\beta}$$

where k, γ, and β are positive constants. Any successful computation of a d-algorithm terminates in finite time.

8.2.1 D-Algorithms Are On-Line

The notion of *on-line* algorithm was introduced in Section 3.2 on page 38. In essence, such an algorithm (or Turing machine) processes each input datum without looking ahead (by contrast to an *off-line* algorithm). From the above informal characterization for the on-line class, one can already identify a strong similarity between on-line algorithms and d-algorithms. In this section we formally show that these two classes are in fact identical.

Recall that on-line algorithms were formally defined (in terms of Turing machines) in Item 1 of Definition 3.1 on page 38, which is, to our knowledge, the only formal definition of this class. As well, recall that Definition 2.4 on page 27 provides a Turing machine model for d-algorithms. The remainder of this section is based on these two definitions.

As in Section 2.3.1 on page 23, we denote by D_i the i-th datum in the input stream. The ordering is naturally defined as follows: D_j is examined before D_i is examined for the first time if and only if $i > j$. As well, we say that an algorithm A [Turing machine M] is able to

terminate at point k if, before visiting any $D_{k'}$, $k' > k$, it has built a solution identical to the solution returned by A [M] when working on the input set $D_1, \ldots, D_k$. It is immediate that an example of termination point for some d-algorithm is N (the amount of data processed by that d-algorithm).

Lemma 8.2 *A Turing machine M as in Definition 2.4 on page 27, working on any sufficiently large input data set of size N_ω, is able to terminate at some point $N_1 < N_\omega$, N_1 being constant with respect to N_ω, if and only if it is able to terminate at two finite points N_1 and N_2 strictly smaller than N_ω and constant with respect to N_ω.*

Proof. The "if" part is immediate. We provide a proof for the "only if" part.

When M terminates at point N_1 it must have reached the special state h'. Obviously, this happened after some constant number of steps (since both K and Σ are of constant size, and the number of tape cells visited is N_1 which is constant as well). Therefore, we have a cycle, from h' (the initial state) back to h', after a number of steps bounded by some constant ζ. Assume now that M chooses not to halt at the point N_1 and instead goes to another state q. The state h' is accessible from q (otherwise, M won't terminate even after processing all the N_ω input data) and, since M already reached h' for an arbitrary input, it will reach it again, after a number of steps bounded by ζ and after visiting a constant number of new tape cells, because M is deterministic. This point is the point N_2 whose existence we want to prove. ∎

Theorem 8.3 *A Turing machine M as in Definition 2.4 on page 27, working on any input data set of size N_ω, where N_ω tends to infinity, is able to terminate at some finite point N_1 if and only if it is able to terminate at all of the points in a countably infinite set $S \subseteq \{1, 2, \ldots, N_\omega\}$, where S has the following properties: (a) the least element of S is upper bounded by a finite constant ζ, and (b) the distance between any two consecutive elements in S is upper bounded by ζ.*

Proof. Again, the "if" part is immediate. The "only if" part is easily proved by induction over the size of S, using Lemma 8.2. ∎

For any alphabet X and positive integer y, let X^y be the set of all the words of length y over the alphabet X. Given a constant ζ, one can compact a Turing machine's tape by simply considering $\Sigma^\zeta \cup \{\#\}$, where # is the blank symbol, as the tape alphabet instead of Σ, then "folding" each sequence of ζ non-blank tape cells into one cell, and finally modifying the function δ accordingly (see for example the proof given in [61] of the fact that a k-tape Turing machine can be simulated by a one-tape Turing machine). We have thus the following corollary:

Corollary 8.4 *A Turing machine M as in Definition 2.4 on page 27, working on any input data set of size N_ω, where N_ω tends to infinity, is able to terminate at some finite point N_1 if and only if it is able to terminate at all of the points in the set $\{1, 2, \ldots, N_\omega\}$.*

Corollary 8.4 and Item 1 of Definition 3.1 on page 38 establish the main result of this section ($\mathcal{D}$ and $\mathcal{O}$ denote the classes of d-algorithms and on-line algorithms, respectively):

Theorem 8.5 $\mathcal{D} = \mathcal{O}$.

Proof. Let us take a closer look at our model from Definition 2.4 on page 27. Let M be a Turing machine that conforms to this definition. Then, it is easy to build an on-line Turing machine M' such that M' performs the same computation as M: just make h' the only polling state of M', and modify the transition function of M such that h' lead to h in one step if and only if the end of the input is reached. Clearly M' is deterministic and performs the same computation as M, except that it halts only at the end of the input. Note that Corollary 8.4 implies that M does not need to move its head on the left on the input tape. The reverse transformation (that is, the transformation of an on-line Turing machine to a Turing machine conforming to Definition 2.4) is analogous, except that new states may need to be added.

The above argument proves the inclusion $\mathcal{D} \subseteq \mathcal{O}$. M's nondeterministic choice of halting or continuing to work (modeled by the state h') should be viewed as the decision

made conforming to the first item in Definition 2.1 on page 23 (that is, whether no new data arrived during the current computation). However, when showing how to transform M into an on-line Turing machine, we lost this feature (the on-line Turing machine halts at the end of the input only). Hence, we also proved $\mathcal{O} \subseteq \mathcal{D}$, except that the second point of Definition 2.1 is not accounted for. Therefore, in order to complete the proof, we have to show that, for any on-line algorithm A and any size n of the initial data set, there is a data arrival law ϕ such that, when working on a data-accumulating input set, A terminates in finite time, and considers at least $n+1$ data.

Let the complexity of A be $C(n)$. In general, $C(n)$ depends on the actual values of the input data. For any positive integer n_1, denote by t_1 a lower bound on $C(n_1)$, and let t_2 be an upper bound on $C(n_1+1)$, for any possible input data sets of size n_1 and n_1+1, respectively. It is easy to build a function $\phi(n, \mathfrak{t})$, strictly increasing with respect to its second argument, such that $\phi(n_1, 0) = n_1$, $\phi(n_1, t_1) = n_1 + 1.1$, and $\phi(n_1, t_2) = n_1 + 1.5$ (for example, this could be done by interpolation). Then, the behavior of A working on an initial data set of size n_1, for any value of n_1, and under the data arrival law ϕ clearly satisfies the requirements stated in the second item of Definition 2.1. ∎

8.2.2 Real-Time Input Imposes Deadlines

We continue our investigation into the computational power of data-accumulating algorithms. We found that these algorithms are in effect on-line algorithms. We further limit their computational power by showing that deadlines are implicitly imposed on their running time. It has been shown [63] that, if the data arrive *fast enough*, then a successful algorithm (i.e., one that terminates) must have a running time upper bounded by a constant; when the running time exceeds that constant, the algorithm never terminates. Our results indicate that the qualifier “fast enough” is not necessary, and a constant upper bound for the running time exists for *any* polynomial data arrival law, and for any (parallel of sequential) d-algorithm. This is a negative, yet important result as it establishes a limit that was

not previously known. Such a limit, however, is consistent with the conjectured inclusion of real-time problems in NLOGSPACE (Claim 2 on page 110).

We begin by studying sorting d-algorithms, introduced in Example 2.1 on page 25. Note that a similar problem is investigated in [63], but there the result is a search tree. Conceivably, there exist applications working in real time on large sequences of data for which the $O(\log n)$ access time to the elements in a search tree is not acceptable. Therefore, the sorting algorithms discussed here output an *array* of sorted elements (the access time is thus $O(1)$). Henceforth, we refer to such processing as *sorting on a linear structure* (or simply *sorting* when there is no ambiguity). For comparison, we use an optimal static sorting algorithm whose running time is $\Theta(n \log n)$ [89].

By definition, data accumulates as the computation proceeds. Generally, we consider that the incoming data are buffered until some amount q is reached, and then the buffered data are inserted into the already sorted sequence (for the non-buffered case simply set q to 1). The time complexity of such an operation is given by the following lemma.

Lemma 8.6 *Let the length of the already sorted sequence be l. Then, the time complexity of inserting q new elements into the sorted sequence is $\Theta(q \log q + l)$, for any q and l such that either $q = 1$ or $l \geq (q \log q)/(q-1)$.*

Proof. The upper bound is immediate. For the lower bound, we have the following: Supposing that the distance between the insertion point of some element and the end of the buffer is $\Omega(l)$ (the general case), the time required for inserting a datum into some sorted sequence is $\Omega(l)$. If the newly arrived data are not sorted, then one should insert them into the buffer one by one. Therefore, the insertion time for the whole sequence is $\Omega(ql)$. On the other hand, the number of operations needed to sort q data is $\Omega(q \log q)$ [89], while merging the sorted sequences requires $\Omega(q + l)$ operations [89]. ∎

We did not consider the case in which more than q elements come before the processing of the already arrived data is finished. However, this case is equivalent to the one in

which the data arrive too fast and the d-algorithm never stops. Indeed, as seen below, the necessary condition for the algorithm to terminate in finite time given by Relation 8.1 covers this case.

Let t_q be the time in which a buffer of size q is filled. At some time t_x, when the buffer is filled and is ready to be inserted, the length of the already sorted sequence will be $l = n + kn^{\gamma}(t_x^{\beta} - t_q^{\beta})$, because all the arrived data have been inserted, except the data which are in the buffer (otherwise, some elements are lost). We will consider for simplicity that $\beta = 1$, but similar results can be obtained for other values as well. In this context, the time interval between two arrivals is $dt = 1/kn^{\gamma}$. Suppose that the algorithm stops at time t. This means that, at time t, all the buffered data have been inserted before another datum arrives. Thus, the time required to insert the buffer (given by Lemma 8.6 on the page before) should be no larger than dt, i.e.,

$$dt \geq q \log q + n + kn^{\gamma}(t - t_q). \tag{8.1}$$

Considering the data arrival law of Relation 2.1 on page 24, the time t_q in which the buffer is filled is given by $q = kn^{\gamma} t_q^{\beta}$. Then, simple calculations let us derive the following bound on the computation time:

$$t \leq \frac{1}{(kn^{\gamma})^2} + \frac{1}{kn^{\gamma}} q(1 - \log q) - \frac{n}{kn^{\gamma}}. \tag{8.2}$$

This imposes a limit on the running time of any sequential sorting d-algorithm that terminates for the polynomial data arrival law given by Relation 2.1. Henceforth, the right-hand side of Relation 8.2 will be denoted by t'_B[2]. We are now ready to determine the complexity of sorting on a linear structure.

Theorem 8.7 *Under the polynomial data arrival law given by Relation 2.1 on page 24, the running time of any RAM d-algorithm for sorting on a linear structure is* $\Theta(N^2)$.

[2] This notation was chosen in order to be consistent with the notation used in [63]. There, t_B denotes an upper bound on the running time. Obviously, Relation 8.2 also defines an upper bound.

Proof. Assume first that q is constant with respect to the running time (however, if it is a function of n, then the proof is not affected). By Lemma 8.6 on page 117, the time required to insert the q elements from the buffer is $\Theta(q \log q + l)$, where l is the number of already sorted elements. This time is $\Theta(l)$, since q is constant. Thus, the total time required for sorting N elements is $\Theta(\sum_{l=n}^{N/q} ql) = \Theta(N^2)$, as desired, since l increases by q each time a new buffer is inserted.

Assume now that the size q of the buffer varies with time. We have $\frac{\partial t'_B}{\partial t} = -\frac{1}{kn^\gamma} \log q \frac{\partial q}{\partial t}$. Obviously, $q \geq 1$ and $t'_B > 0$ (otherwise, the algorithm never terminates). If there is some constant (with respect to the time) Q such that $q \leq Q$ everywhere, then the relation derived for constant buffer size holds (as q may then be approximated by Q and the rate of growth does not change). Therefore, it is enough to consider the case of q being an increasing function. Then, $\frac{\partial q}{\partial t} > 0$, and this implies $\frac{\partial t'_B}{\partial t} < 0$, because $\log q > 0$ for $q > 1$. That is, t'_B is a decreasing function. Since $t'_B(0)$ is finite, there exists some Q such that $t'_B(q) \leq 0$, for all $q \geq Q$. By Relation 8.2 on the page before, the termination time of the algorithm is less than t'_B. Therefore, when $q \geq Q$, the upper limit for the termination time is negative, which means that the algorithm never stops. Thus the values q for which the algorithm terminates are bounded again and we are in the case covered by constant buffer size. ∎

Note that a side consequence of the proof of Theorem 8.7 is that the best value for the buffer size q is the minimal possible, i.e., 1, because t'_B is a decreasing function with respect to q. In other words, there is no reason to buffer data; it is better to insert each arrived datum in linear time. Hence, we will consider $q = 1$. Second, the time required to insert one element ($q = 1$) into the already sorted sequence is cl, for some constant c. The expression for t'_B becomes in this case $t'_B = \frac{1}{c(kn^\gamma)^2} - \frac{n}{kn^\gamma}$.

We have considered $\beta = 1$. This implies that the product $\alpha\beta$ is larger than 1, and the existence of a limit on the running time in this case was established in [63]. More interesting is the situation where $\beta \leq 1/2$, for $\alpha\beta \leq 1$. Under these conditions, no limit on the running time is known. We now study this case.

Theorem 8.8 *For the polynomial data arrival law given by Relation 2.1 on page 24, if a sorting RAM d-algorithm terminates, then its running time is upper bounded by a constant T that does not depend on n.*

Proof. We have the restriction $\gamma = 1$ for easier calculations. The time dt after which a new datum arrives is given by $1 = kn^{\gamma}((t+dt)^{\beta} - t^{\beta})$, for some moment t. That is, $(t+dt)^{\beta} - t^{\beta} = 1/kn^{\gamma}$. On the other hand, Relation 8.1 on page 118 in the general case becomes $dt \geq q\log q + n + kn^{\gamma}(t^{\beta} - q/kn^{\gamma})$. From these two relations, $1/kn^{\gamma} \geq (q(\log q - 1) + n + kn^{\gamma}t^{\beta} + t)^{\beta} - t^{\beta}$. In particular, for $\gamma = 1$,

$$\frac{1}{kn} \geq (q(\log q - 1) + n + knt^{\beta} + t)^{\beta} - t^{\beta}. \tag{8.3}$$

The complexity of the sorting algorithm is $O(N^2)$ by Theorem 8.7 on page 118. That is, for $\gamma = 1$, $n = t^{1/2}/c(1+kt^{\beta})$. By substituting this value in Relation 8.3 and manipulating the obtained expression,

$$\frac{qc}{k} \geq b(t) \times a(t). \tag{8.4}$$

where $a(t) = (q(\log q - 1) + \frac{t^{1/2}}{c} + t)^{\beta} - t^{\beta}$ and $b(t) = t^{1/2}/(1+kt^{\beta})$. Then, $\frac{\partial b(t)}{\partial t} = \frac{t^{-1/2}}{(1+kt^{\beta})^2}(1/2 + k(1/2-\beta)t^{\beta})$, and hence, for $\beta \leq 1/2$, $\frac{\partial b(t)}{\partial t} > 0$. That is, $b(t)$ is an increasing function. Analogously, $a(t)$ is an increasing function as well:

$$\begin{aligned}
\frac{\partial a(t)}{\partial t} &= \beta\left(\frac{t^{-1/2}}{c}+1\right)\left(q(\log q - 1) + \frac{t^{1/2}}{c} + t\right)^{\beta-1} - \beta t^{\beta-1} \\
&> \beta\left(\left(q(\log q - 1) + \frac{t^{1/2}}{c} + t\right)^{\beta-1} - t^{\beta-1}\right) \text{ [because } t^{-1/2}/c > 0] \\
&> 0 \text{ [because } q(\log q - 1) + t^{1/2}/c > 0 \text{ for large enough } t].
\end{aligned}$$

Therefore, $b(t) \times a(t)$ is increasing. Moreover, it is easy to see that, for $\beta < 1/2$, $\lim_{t\to\omega} b(t) = \omega$, and $\lim_{t\to\omega} a(t) > 0$. Therefore, $\lim_{t\to\omega} b(t) \times a(t) = \omega$ for $\beta < 1/2$. For $\beta = 1/2$, $lim_{t\to\omega} b(t) = 1/k$, and, for large enough t, $a(t) \geq t^{1/8}$ and thus $lim_{t\to\omega} a(t) = \omega$. Then again, $\lim_{t\to\omega} b(t) \times a(t) = \omega$. Since $b(t) \times a(t)$ is an increasing function and its limit

is infinite, there exists some finite T such that $b(t) \times a(t) > \frac{qc}{k}$ for any $t > T$. Then, such a t larger than T will contradict the necessary condition for algorithm termination given by Relation 8.4 on the page before. Hence, T is an upper bound for the running time and this completes the proof. ∎

Note that the theorem implicitly gives an upper bound for the maximum amount of data that can be processed, because this amount is given by $N = n + kn^{\gamma}t^{\beta}$ and its upper bound is obviously $n + kn^{\gamma}T^{\beta}$. We contradict by Theorem 8.8 the results derived in [63], where it is claimed that such a bound does not exist for $\alpha\beta < 1$.

In the case of sorting on a linear structure we found an upper bound on the running time for any data arrival law. Sorting is not the only case in which such a bound exists though.

Theorem 8.9 *For the polynomial data arrival law given by Relation 2.1 on page 24, let A be any RAM d-algorithm with time complexity $\Omega(N^{\alpha})$, $\alpha > 1$. If A terminates, then its running time is upper bounded by a constant T that does not depend on n.*

Proof. We consider only the case $\beta \leq 1/\alpha$, because the limit has been already found for $\alpha\beta > 1$ [63]. Let $\varepsilon = \alpha - 1$, $\varepsilon > 0$. If the algorithm terminates at some finite time t, then N data have been processed, $N = n + kn^{\gamma}t^{\beta}$. That is, the time for processing one datum is $cN^{\alpha}/N = cN^{\varepsilon}$ for some positive constant c. Following the same idea as the one used for deriving Relation 8.1 on page 118, we obtain $dt \geq c(n + kn^{\gamma}t^{\beta})^{\varepsilon}$, which is similar to Relation 8.1. Therefore, analogously to the proof of Theorem 8.8 on the preceding page, we obtain for $\gamma = 1$

$$\frac{qc}{k} \geq \frac{t^{1/2}}{(1 + kt^{\beta})} \left((c(n + knt^{\beta})^{\varepsilon} + t)^{\beta} - t^{\beta} \right). \tag{8.5}$$

The left-hand side of this relation is increasing, because $c(n + knt^{\beta})^{\varepsilon} > 0$, and it is immediate that the limit of the right-hand side is infinite. Hence, the limit T is derived in the same way as in the proof of Theorem 8.8. ∎

We now consider parallel d-algorithms. Recall that p is the number of processors in the parallel model. It is immediate that the process described in Lemma 8.6 on page 117 admits linear speedup. Indeed, sorting q elements admits linear speedup [5] (page 179), and inserting the buffer into the previously sorted sequence may be achieved by using an optimal merging algorithm [5] (page 209). Thus, Relation 8.2 on page 118 becomes in the parallel case $t \leq p/(kn^{\gamma})^2 + q(1 - \log q)/kn^{\gamma} - n/kn^{\gamma}$. As expected, this relation is similar to Relation 8.2 for the sequential case. Therefore, all the above sequential results hold for the parallel case as well. That is, the best value for q is 1 (buffering does not help), and a limit $t''_B(p)$, similar to t'_B, for the running time can be found, $t''_B(p) = \frac{p}{c_p(kn^{\gamma})^2} - \frac{n}{kn^{\gamma}}$. It is then easy to see that Theorem 8.9 holds for the parallel case as well.

Theorem 8.10 *For the polynomial data arrival law given by Relation 2.1 on page 24, let A be any p-processor d-algorithm, running on some parallel model of computation (meeting the minimum requirements stated at the beginning of Section 2.2 on page 16), with time complexity $\Omega(N^{\alpha})$, $\alpha > 1$. If A terminates, then its running time is upper bounded by a constant T that does not depend on n but depends on p.*

Proof. It is enough to replace the first term from the right-hand side of Relation 8.5 on the page before in the proof of Theorem 8.9 by $p \times t^{1/2}/(1 + kt^{\beta})$. The proof is then analogous, as this replacement introduces a multiplicative constant, which does not change the sign of the derivative. As well, the appearance of p does not change the limit. ∎

8.3 The Characterization of C-Algorithms

Theorems 8.9 and 8.10 support Claim 2 on page 110. We show in what follows that such a feature does hold for the class of correcting algorithms (c-algorithms for short). We believe that c-algorithms form the most general setting for the real-time input arrival paradigm, so that we are confident in the validity of Claim 2 and thus this section concludes the chapter.

It should be noted that we show the validity of Theorems 8.9 and 8.10 in the case of c-algorithms in an indirect way. Indeed, Theorem 8.16 on page 128 shows that the analysis of c-algorithms can be reduced to the analysis of d-algorithms. The validity of the mentioned theorems follows immediately.

Recall from Section 2.3.2 on page 28 that a c-algorithm is defined as being an algorithm that works on an input data set of n elements, all available at the beginning of computation, but $V(n, \mathbb{t})$ variations of the n input data occur with time. We also denote by $\phi(n, \mathbb{t})$ the sum $n + V(n, \mathbb{t})$. We call the the function V the *corrections arrival law*.

We first consider the case in which the c-algorithm processes the incoming corrections one at a time.

Theorem 8.11 *Consider some problem π solvable by a dynamic algorithm. Let A_s be the best known dynamic RAM algorithm that solves π, and let A_{ps} be a parallel version of A_s, which uses p processors, runs on some parallel model of computation M (meeting the minimum requirements stated at the beginning of Section 2.2 on page 16), and exhibits a speedup $S'(1, p)$. Then, there exist a RAM c-algorithm A and a p-processor c-algorithm A_p running on M that solve π. Moreover, when A and A_p work on an initial set of data of size n and with the corrections arrival law given by Relation 2.4 on page 32, there exist n', k', and γ', where γ' and k' are constants and n' is a function of n, such that the properties of A and A_p (namely, time complexity, running time, and parallel speedup) are the same as the properties of some sequential d-algorithm A_d and some p-processor d-algorithm A_{pd} running on M, which work on n' initial data and with the data arrival law $\phi(x, \mathbb{t}) = x + k'x^{\gamma'}\mathbb{t}^{\beta}$, where the complexity of A_d is cN, and the parallel speedup manifested in the static case by A_{pd} is $S'(1, p)$.*

Proof. Assuming that such d-algorithms A_d and A_{pd} exist, recall that the running time t of A_d is given by $t = cN$, where $N = \phi(n', t)$. That is, the relations from which all the

properties of A_d and A_{pd} can be derived are

$$t = c(n' + k'n'^{\gamma'} t^{\beta}), \tag{8.6}$$

$$t_p = \frac{c_p(n' + k'n'^{\gamma'} t^{\beta})}{S'(1, p)}. \tag{8.7}$$

Therefore, if we show that there exist n', γ', and k' such that we can consider two c-algorithms A and A_p whose running times respect Relations 8.6 and 8.7, respectively, we complete the proof. We do this in what follows.

The computation performed by A_s can be split in two parts: the initial processing, and the processing of one correction. Let the algorithm A' that performs the initial processing be of complexity $C'(n) = cn^{\alpha}$, and the algorithm A_u that processes a correction be of complexity $C_u(n) = c_u n^{\varepsilon}$, $\varepsilon \geq 0$.

The algorithm A performs then the following computations: (a) compute the solution for the initial amount of data n, using A' (this takes $C'(n)$ time), and (b) for each newly arrived correction, compute a new solution, using A_u; terminate after this only if no new correction arrived while the solution is recomputed, otherwise repeat this step (the complexity of this step is $C_u(n)$).

The complexity of A is then $C(N) = C'(n) + C_u(n)(N - n)$, and this gives the following implicit equation for the termination time: $t = cn^{\alpha} + c_u n^{\varepsilon}(n + kn^{\gamma}t^{\beta} - n) = c(n^{\alpha} + \frac{c_u}{c} kn^{\gamma+\varepsilon} t^{\beta})$. Then, considering $n' = n^{\alpha}$, $\gamma' = \frac{\gamma+\varepsilon}{\alpha}$, and $k' = kc_u/c$, we obtain for the running time of A the implicit equation 8.6.

With regard to the parallel implementation, considering that the static version of A provides a speedup $S'(1, p)$ when using p processors, and that the complexity of A is $C(N)$, it is immediate that the complexity in the parallel case is $C(N)/S'(1, p)$. Relation 8.7 as the implicit equation for the running time of A_p follows immediately. This is enough to prove that the properties of A and A_p are identical to the properties of A_d and A_{pd}.

However, in order to finalize the proof, we have to take into consideration the second item from Definition 2.5 on page 29. That is, we have to prove that A terminates for at

least one corrections arrival law and for any value of n'. To do this, it suffices to notice that it has been proved in [63] that a d-algorithm of complexity cN terminates for any initial amount n of input data if $\beta < 1$. ∎

An algorithm A is given which solves the corresponding problem π correctly is given in Theorem 8.11 . However, nothing is said in the theorem about the optimality of A. This issue is now studied. We show in what follows that there are instances of the input for which there is indeed an algorithm better than A, but that a form of Theorem 8.11 holds for this new algorithm as well. Specifically, one may wonder why a c-algorithm cannot consider the incoming corrections in bundles of size b instead of one by one.

Recall that we denote by A_u^b the best known algorithm that applies b corrections. Let the complexity of A_u^b be $C_u^b(n, b)$, and the complexity of A_u, the (best known) algorithm that applies one correction only, as in Theorem 8.11, be $C_u(n)$.

Suppose that algorithm A of Theorem 8.11 terminates at some time t for some initial data and some corrections arrival law. Further, let A_{bw} be an algorithm that performs the same processing as A, except that, instead of applying each correction immediately, it waits until some number b of corrections have been accumulated and applies them at once using algorithm A_u^b described above. Note that we do not impose any restriction on b; it may be either a constant, or a function of either n or t or both. We have the following immediate observation:

Lemma 8.12 *Given some solution σ to a problem π and a set of input data of size n, let A_u^b be the best known RAM algorithm that receives σ and b corrections, and computes a solution σ_u for the corrected input. Also, let A_u (of complexity C_u) be the best known RAM algorithm that receives σ and one correction only and returns the corresponding corrected solution. Then, $bC_u \geq C_u^b \geq C_u$ for any strictly positive b.*

Let A_b be a variant of algorithm A_{bw} for which the buffer size is correctly chosen, such

that A_b does not wait until b corrections have arrived, but instead processes whatever corrections are available whenever it has time to do so (that is, as soon as it finishes processing the previous bundle). For simplicity, we assume that the complexity of the initial processing $C'(n)$ is the same for A, A_{bw}, and A_b. We believe that this is a reasonable assumption, since the processes are essentially the same in the two c-algorithms. It follows immediately from Claim 1 on page 30 that

Lemma 8.13 *Given A, A_{bw}, and A_b as above, let S_1, S_2 and S_3 be the speedup of the static version of A, A_{bw}, and A_b, respectively, when implemented using p processors on some parallel model of computation meeting the minimum requirements stated at the beginning of Section 2.2 on page 16. Then, $S_1 = S_2 = S_3$.*

Recall that we consider the complexity of A_u as being $C_u(n) = c_u n^{\varepsilon}$. Also, let the complexity of A_u^b be $C_u^b(n) = c_u b^{\varepsilon_b} n^{\varepsilon}$. Then, Lemma 8.12 on the page before implies that ε_b is a positive number, no larger than 1.

Lemma 8.14 *A_b is no better than A if and only if either $\varepsilon_b = 1$, or $\beta > 1$, or $\beta = 1$ and $\beta k c_u n^{\varepsilon+\gamma} \geq 1$.*

Proof. The running time t_b of A_b is given by $t_b = cn^{\alpha} + c_u n^{\varepsilon} \sum_{i=1}^{m} b_i^{\varepsilon_b}$, where m is the number of times A_u^b has been invoked, and, when A_u^b is invoked the i-th time, the size of the buffer b is b_i. Note that exactly all the buffered corrections are processed, that is, $\sum_{i=1}^{m} b_i = V(n, t_b)$. Then, it follows that

$$t_b \leq cn^{\alpha} + kc_u n^{\varepsilon+\gamma} t_b^{\beta}. \tag{8.8}$$

Let $R(n,t)$ be the function $R(n,t) = cn^{\alpha} + kc_u n^{\varepsilon+\gamma} t^{\beta} - t$. Moreover, since there is at least one i such that $b_i > 1$, then $\sum_{i=1}^{m} b_i^{\varepsilon_b} = \sum_{i=1}^{m} b_i$ if and only if $\varepsilon_b = 1$. Therefore, we have from Relation 8.8

$$R(n,t) < R(n,t_b) \quad \text{if} \quad \varepsilon_b < 1, \tag{8.9}$$

$$R(n,t) = R(n,t_b) \quad \text{if} \quad \varepsilon_b = 1. \tag{8.10}$$

When $\varepsilon_b = 1$, Relation 8.10 holds. Therefore, in this case, A_b is no better than A. For $\varepsilon_b < 1$, considering Relation 8.9, we have:

1. If $\beta > 1$, then $\frac{\partial R}{\partial t} > 0$ for t large enough. In this case, given Relation 8.9, it follows that $t < t_b$. In other words, A_b is no better than A in this case.

2. If $\beta < 1$, then, $\frac{\partial R}{\partial t} < 0$ for t large enough. Therefore, analogously, A is no better than A_b.

3. If $\beta = 1$, then $\frac{\partial R}{\partial t}$ is positive if and only if $\beta k c_u n^{\varepsilon+\gamma} > 1$. Therefore, if $\beta k c_u n^{\varepsilon+\gamma} \geq 1$, then we are in the same case as in item 1 above. Otherwise, A is no better than A_b.

∎

We found therefore that A is optimal if and only if either $\varepsilon_b = 1$, or $\beta > 1$, or $\beta = 1$ and $\beta k c_u n^{\varepsilon+\gamma} \geq 1$. In the other cases, A_b may be better than A. Recall here that A_{bw} is the algorithm that waits for b corrections to have accumulated before processing them (unlike A_b which processes any available corrections as soon as it has time to do so). It is somewhat intuitive that A_{bw} is no better than A_b, but we give a proof for this below.

Lemma 8.15 *A_{bw} is no better than A_b when $\beta \leq 1$.*

Proof. When speaking of A_{bw} we denote by m_w and b_{wi} the values that correspond to m and b_i in the case of A_b, respectively. We denote by t_{bw} the running time of A_{bw}, which has the form

$$t_{bw} = cn^{\alpha} + c_u n^{\varepsilon} \sum_{j=1}^{m_w} b_{wj}^{\varepsilon_b} + t_w, \tag{8.11}$$

where t_w is the extra time generated by the waiting process. We denote by t_{wi} the waiting time before each invocation of the algorithm that applies the corrections, and by a_i the number of corrections A_{bw} waits for before each such invocation. Note that $t_w = \sum_{i=1}^{m_w} t_{wi}$. Moreover, it can be shown that $a_i \leq k n^{\gamma} t_{wi}^{\beta}$.

For any i, $1 \leq i \leq m_w$, we have $b_{wi}^{\varepsilon_b} \geq \frac{b_{wi+1} - kn^{\gamma} t_{wi}^{\beta}}{a}$. Note that $\sum_{i=1}^{m_w} b_{wi} = kn^{\gamma} t_{bw}^{\beta}$, since all the corrections that arrive before the termination time must be processed. Then, by summation over i of the above relation and from Relation 8.11, $t_{bw} \geq cn^{\alpha} + \frac{b}{a} t_{bw}^{\beta} - \frac{b}{a} \sum_{i=1}^{m_w} t_{wi}^{\beta} + t_w$, where $b = c_u n^{\varepsilon} kn^{\gamma}$. Considering Relation 8.8 on page 126 we have then

$$\rho(t_{bw}) - \rho(t_b) \geq t_w - \frac{b}{a} \sum_{i=1}^{m_w} t_{wi}^{\beta} + \left(b - \frac{b}{a}\right) t_{bw}^{\beta}, \tag{8.12}$$

where $\rho(t) = t - (b/a)t^{\beta}$. Let us denote the right hand side of Relation 8.12 by $\mathcal{R}$. It can be proved that $\mathcal{R}$ is positive. Then, Relation 8.12 leads to $\rho(t_{bw}) \geq \rho(t_b)$ and it is immediate that $t_{bw} \geq t_b$, since ρ is an increasing function for any $t \geq 1$. ∎

The proof of Lemma 8.15 contains yet another interesting result. We show there that $b_{wi+1} \leq ab_{wi}^{\beta\varepsilon_b} + a_i$. Since A_b does not wait for any correction, we have analogously $b_i \leq ab_i^{\beta\varepsilon_b}$. Applying this relation recursively i times, we observe that $b_{i+1} \leq a^{(1-(\beta\varepsilon_b)^i)/(1-\beta\varepsilon_b)}(b_1)^{(\beta\varepsilon_b)^i}$. However, $0 \leq \beta\varepsilon_b \leq 1$, hence $b_{i+1} \leq b_1$. On the other hand, $b_1 = kn^{\gamma}(cn^{\alpha})^{\beta}$, since the first invocation of the algorithm that processes the corrections happens exactly after the initial processing terminates. Therefore, in the case of A_b, the size of the buffer is bounded by a quantity that does not depend on time, but only on n. Then, the running time of A_b is given by an equation of the form $t_b = c(n'' + k''n''^{\gamma''} t_b^{\beta})$, which is similar to the form obtained in the case of A in Theorem 8.11 on page 123. Starting from this expression, one can follow the same reasoning as in the proof of Theorem 8.11 to derive an implicit equation for the parallel case of A_b similar to Relation 8.7 on page 124. Lemmas 8.14 and 8.15 therefore imply:

Theorem 8.16 *The (parallel and sequential) c-algorithms of Theorem 8.11 on page 123 are optimal if and only if the corrections arrival law has the following property: Either $\beta > 1$, or $\beta = 1$ and $\beta kc_u n^{\varepsilon+\gamma} \geq 1$. If this property does not hold, then the optimal RAM algorithm is A_b, where A_b processes at once all the corrections that arrived and have not been processed yet.*

Moreover, let A_b work on n initial data and with the corrections arrival law given by Relation 2.4 on page 32, and let A_{bp} be the p-processor parallel implementation of A_b that runs on the

same parallel model of computation M from Theorem 8.11, where the static version of A_{bp} exhibit a speedup $S'(1,p)$. Then, there exist n'', k'', and γ'', where γ'' and k'' are constants and n'' is a function of n, such that the properties of A_b and A_{bp} (namely, time complexity, running time, and parallel speedup) are the same as the properties of some sequential d-algorithm A_d and some p-processor d-algorithm A_{pd} running on M, which work on n'' initial data and with the data arrival law $\phi(x, \mathbb{t}) = x + k''x^{\gamma''}\mathbb{t}^{\beta}$, where the complexity of A_d is cN, and the parallel speedup manifested in the static case by A_{pd} is $S'(1,p)$.

8.4 The Graph Accessibility Problem and Real Time

By Claim 2 on page 110, we establish a strong relation between NLOGSPACE and real-time computations. This relation, however is stated in terms of one particular model of parallel computation. True, this model (namely, the RMBM) is a feasible one—variants of it have been already implemented [44]. Still we can also extend this result to other models of computation.

Recall from Definition 7.1 on page 97 that $GAP_{1,n}$ denotes the following version of the Graph Accessibility Problem: Given a directed graph $G = (V, E)$, $V = \{1, 2, ..., n\}$, determine whether vertex n is accessible from vertex 1. Incidentally, note that $GAP_{1,n}$ is NLOGSPACE-complete [84].

Consider now the classes $M_{<GAP}$, $M_{\equiv GAP}$, and $M_{>GAP}$ of parallel models of computations using polynomially bounded resources (processors and, if applicable, buses), such that:

$M_{<GAP}$ contains exactly all the models that cannot compute $GAP_{1,n}$ in constant time, and cannot compute in constant time any problem not in NLOGSPACE. An example of such a model is the Common CRCW PRAM [88, 95] (and thus the less powerful PRAM variants).

$M_{\equiv GAP}$ contains exactly all the models that can compute $GAP_{1,n}$ in constant time, but

cannot compute in constant time any problem not in NLOGSPACE. RMBM and RN (refer to Section 2.2 on page 16 for definitions of these models and their properties) are two examples of such models.

$M_{>GAP}$ contains exactly all the models that can compute $GAP_{1,n}$ in constant time and can compute in constant time at least one problem not in NLOGSPACE. To our knowledge, no model has been proved to pertain to such a class. However, a good candidate is the *broadcast with selective reduction* model [13].

Theorem 8.17 *For any models of computation M_1, M_2, and M_3 such that $M_1 \in M_{<GAP}$, $M_2 \in M_{\equiv GAP}$, and $M_3 \in M_{<GAP}$, it holds that*

$$\text{rt-PROC}^{M_1}(poly(n)) \subseteq \text{rt-PROC}^{\text{DRMBM}}(poly(n)) = \text{NLOGSPACE}/rt \quad (8.13)$$

$$\text{rt-PROC}^{M_2}(poly(n)) = \text{rt-PROC}^{\text{DRMBM}}(poly(n)) = \text{NLOGSPACE}/rt \quad (8.14)$$

$$\text{rt-PROC}^{M_3}(poly(n)) \supset \text{rt-PROC}^{\text{DRMBM}}(poly(n)) = \text{NLOGSPACE}/rt \quad (8.15)$$

Proof. Minor variations of the arguments used to prove Theorem 7.9 on page 107 and Theorem 7.10 on page 108 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 8.13 and 8.15 follow immediately from Claim 2 on page 110.

By the same argument, $\text{rt-PROC}^{M_2}(poly(n)) \supseteq \text{rt-PROC}^{\text{CRCW F-DRMBM}}(poly(n))$ holds as well. The equality (and thus Relation 8.14) is given by the arguments that support Claim 2. ∎

Thus, the characterization of real-time computations established by Claim 2 on page 110 does hold in fact for any machines that are able to compute $GAP_{1,n}$ in constant time. One particular such a model is the DRN: by Corollary 7.7 on page 105 it is immediate that Relation 8.14 holds for $M_2 = \text{DRN}$.

The characterization presented in Theorem 8.17 emphasizes in fact the strength of Claim 2. Indeed, as noted above, no model more powerful than the RMBM is known to exist. That is, according to the current body of knowledge, $M_{>GAP} = \emptyset$. Unless this relation is found to be false, it follows from Claim 2 that no problem outside NLOGSPACE can be solved in real time in general, not only as far as RMBM computations are concerned.

Chapter 9

Real-Time Characterization of Optimization Problems

Summary

We have given in Claim 2 on page 110 a tight characterization of problems solvable in real time by a parallel machine. In this chapter, as well as the subsequent one, we focus our attention on applications of this characterization.

Specifically, we now consider optimization problems. In this context, we identify the class $\mathcal{M}$ of such problems that can be computed in real time if a parallel machine is used (Theorem 9.5 on page 137). It is well-known [45] that all the problems that can be expressed as matroids (a restricted case of independence systems) admit a fast (i.e., polylogarithmic running time) parallel greedy algorithm. Since real time intuitively means "faster than fast," it is expected that optimization problems that are solvable in real time form a subclass of problems admitting fast parallel algorithms. We show that this is indeed the case, but we also find that the relation between matroids and optimization problem solvable in real time is even stronger: By contrast to the independence systems with fast parallel algorithms (which are not restricted to matroids), we show that matroids for which the size of the optimal solution can be computed in parallel real time are *exactly all* the independence systems whose exact solution can be found in real time.

Such a precise characterization has important consequences on both the theory and

the practice of real-time optimization problems. From a practical point of view, once an optimization problem does not fall into the class $\mathcal{M}$, one knows for sure that it cannot be solved in real time. Thus, different approaches have to be found (for example, further restricting the problem, or finding a good real-time approximation algorithm for it).

The identification of class $\mathcal{M}$ leads to immediate generalizations of previous results. We note that it has been shown that a parallel implementation can do more than merely speed up the computation [9, 10, 11]. In particular, in certain real-time environments parallel means (arbitrarily) better [9]. Specifically, the parallel solution for the real-time minimum-weight spanning tree problem can be made arbitrarily better than the solution reported by a sequential algorithm that solves the same problem. We show that such a property holds in fact for a whole class of problems, and for any set of real-time constraints. In other words, we extend the results presented in [9] by showing that, given *almost any* optimization problem in $\mathcal{M}$, the solution obtained by a parallel algorithm is arbitrarily better than the solution reported by a sequential one.

9.1 Independence Systems and Matroids

If S is a set referred to as the *set of feasible solutions*, over which a mapping c is defined ($c : S \to \mathbb{R}$), then a problem of the form

$$\{\max c(s) | s \in S\} \quad \text{or} \tag{9.1}$$

$$\{\min c(s) | s \in S\} \tag{9.2}$$

is an *optimization problem* over S. Form 9.1 defines a *maximization problem*, while form 9.2 is a *minimization problem*; c is referred to as the *objective function*. In the following, we shall refer to maximization problems whose set of feasible solutions contains only elements of $\{0, 1\}^n$. In this case, S can be considered a subset of $\mathcal{P}(E)$, with $E = \{1, 2, \ldots, n\}$ (recall that $\mathcal{P}(()\Sigma)$ stands for the powerset of Σ). Therefore, problems of the form 9.1 can be

```
algorithm GreedyMax (E, S; s_g)
  1.    let (e_1, e_2, ..., e_n) be an ordering of E with c(e_i) ≥ c(e_{i+1})
  2.    s_g ← ∅
  3.    for i ← 1 ... n do
  3.1.      if s_g ∪ {e_i} ∈ S then s_g ← s_g ∪ {e_i}
```

Figure 9.1: A RAM greedy algorithm for maximization problems

restated as

$$\max \left\{ \sum_{i \in R} c_i \,\middle|\, R \in S \right\}. \tag{9.3}$$

Notice that in this case $c(s)$ is implicitly defined as $\sum_{i \in s} c_i$ for any set $s \subseteq E$. We consider without loss of generality that $c_i \geq 0$, $1 \leq i \leq n$. The set of optimal solutions to the maximization problem 9.3 is thus not changed if one replaces S by its *hereditary* closure S^* defined as $S^* = S \cup \{s | s \subseteq s', s' \in S \text{ for some } s' \subseteq E\}$. (E, S^*) is an *independence system* as per Definition 9.1.

Definition 9.1 [53] Let E be a finite set and $S \subseteq \mathcal{P}(E)$, such that S has the *monotonicity property*: $s_1 \subseteq s_2 \in S \Rightarrow s_1 \in S$. Then, (E, S) is an *independence system*, and members of S are said to be *independent*.

Let (E, S) be an independence system. For each $F \subseteq E$, the *lower rank* $lr(F)$ [*upper rank* $ur(F)$] of F (with respect to S) is defined as the cardinality of the smallest [largest] maximal independent subsets of F: $lr(F) = \min\{|s| | s \in S; s \subseteq F \text{ and } s \cup \{e\} \in S \text{ for all } e \in F \setminus \{s\}\}$; $ur(F) = \max\{|s| | s \in S; s \subseteq F\}$.

A *greedy algorithm* for Problem 9.3 on general independence systems is given in [53]. GreedyMax, a variant of this algorithm running on the RAM, is shown in Figure 9.1.

Proposition 9.1 [53] *Let (E, S) be an arbitrary independence system, s_g the solution returned by algorithm* GreedyMax *from Figure 9.1, and s^* the optimal solution of Problem 9.3. Then, for*

any weight function $c : E \rightarrow \mathbb{R}^+$,

$$\min_{F \subseteq E} \frac{lrF}{urF} \leq \frac{c(s_g)}{c(s^*)} \leq 1.$$

It should be noted that the algorithm GREEDYMAX from Figure 9.1 contains one statement which depends on the actual independence system being considered, namely the boolean expression on line 3.1. Indeed, for a general independence system one does not know how the test "$s_g \cup \{e_i\} \in S$" is done. Thus, in order to analyze the complexity of such an algorithm, one can assume the existence of an *oracle* that can answer whether some set s is in S or not.

Definition 9.2 [53] An independence system (E, S) is called a *matroid* if, for any $F \subseteq E$, it holds that $lr(F) = ur(F)$.

From Proposition 9.1 on the page before and Definition 9.2 it follows that:

Corollary 9.2 *Algorithm* GREEDYMAX *from Figure 9.1 on the preceding page on a matroid* (E, S) *yields the optimal solution for Problem 9.3 on the page before for all objective functions* $c, c : E \rightarrow \mathbb{R}^+$.

9.2 A Real-Time Perspective

To put Definition 9.2 in another way [38, 45], matroids are independence systems with the additional property that all the maximal independent subsets have the same size (therefore, since $c_i \geq 0$, $1 \leq i \leq n$, the greedy algorithm obtains the optimal solution). In light of this formulation, the parallel implementation of GREEDYMAX from Figure 9.1 on the page before is immediate [37, 45]. Algorithm PARALLELGREEDYMAX presented in Figure 9.2 on the following page is such a parallel implementation (where PARALLELGREEDYMAX runs on some model of parallel computation that meets the minimum requirements stated at the beginning of Section 2.2 on page 16).

```
algorithm ParallelGreedyMax (E, S; s_g)
1.     sort E, obtaining (e_1, e_2, ..., e_n) s.t. c(e_i) ≥ c(e_{i+1})
2.     s_g ← ∅; r_0 ← 0
3.     for i ← 1 ... n do in parallel
3.1.       r_i ← ur{e_1, e_2, ..., e_i}
3.2.       if r_{i-1} < r_i then s_g ← s_g ∪ {e_i}
```

Figure 9.2: A parallel greedy algorithm for maximization problems

Algorithm ParallelGreedyMax from Figure 9.2 uses a *rank oracle*: The function $ur\{e_1, e_2, \ldots, e_i\}$ introduced by Definition 9.1 on page 134 and used at step 3.2 gives the size of some (hence, whenever (E, S) is a matroid, any) maximal independent set over $\{e_1, e_2, \ldots, e_i\}$.

Lemma 9.3 *Suppose* $ur\{e_1, e_2, \ldots, e_i\} \in \mathrm{DRMBM}(poly(i), poly(i), t(i))$ *(i.e., a DRMBM with polynomially bounded number of processors and buses can compute* $ur\{e_1, e_2, \ldots, e_i\}$ *in time* $t(i)$*). Then,* ParallelGreedyMax $\in \mathrm{DRMBM}(poly(n), poly(n), O(t(n)))$.

In particular, ParallelGreedyMax $\in \mathrm{DRMBM}(poly(n), poly(n), O(1))$ *whenever* $t(i) = O(1)$.

Proof. The initial sorting (step 1) can be achieved in constant time on a DRMBM with polynomially bounded resources [5]. It follows that step 1 is computable in constant time on a DRMBM using $poly(n)$ processors and $poly(n)$ buses by Corollary 7.8 on page 105. Steps 2 and 3.2 are trivially computable in constant time with polynomially bounded resources.

However, each on the calls to ur in step 3.1 can be performed in $t(n)$ time by using n copies of the RMBM computing ur, each of them working independently from each other. Finally, each of the n RMBMs communicate with one other processor. These n new processors implement step 3.2 and report the result. Since both the argument of

ur and the result returned by this function are polynomial in size, $poly(n)$ buses suffice for such a communication. All the resources are polynomially bounded, and thus PARALLELGREEDYMAX $\in$ DRMBM$(poly(n), poly(n), O(t(n)))$, as desired.

If $t(i) = O(1)$, PARALLELGREEDYMAX $\in$ DRMBM$(poly(n), poly(n), O(1))$ is immediate by Corollary 7.8. ∎

Lemma 9.4 *Let (E, S) be some independence system, $E = \{e_1, e_2, \ldots, e_n\}$, and let A be an algorithm that solves a maximization problem of the form 9.3 on page 134 over (E, S). Denote by $t_A(n)$ ($t_{ur}(n)$) the running time of A (the time required to compute $ur(E)$) on a DRMBM using a polynomially bounded number of processors and buses. Then, $t_{ur}(n)$ is a lower bound for $t_A(n)$.*

Proof. Let $s^* = \{s_1, s_2, \ldots, s_k\}$ be the solution computed by A. Since s^* is an optimal solution, it follows that $ur(E) = k$. However, given s^*, k can be computed in constant time on a DRMBM: Assume without loss of generality that the elements of s^* are stored in the registers of n processors p_i, $1 \leq i \leq n$, such that exactly k processors hold one element from s^* each. Then, each processors p_i, $1 \leq i \leq n$, sets a designated register v_i such that $v_i = 1$ if p_i holds a value from s^* and $v_i = 0$ otherwise. Then, a prefix sum[1] over v_i, $1 \leq i \leq n$, computes k. It follows that $|s^*|$ (and thus $ur(E)$) can be computed in constant time given s^*, since prefix sum takes constant time on RMBM [87]. Therefore, $t_{ur}(n) = O(t_A(n))$ (alternatively, $t_A(n) = \Omega(t_{ur}(n))$), as desired. ∎

Theorem 9.5 *Let $\mathcal{M}$ be the class of maximization problems that can be described as a matroid and for which $ur \in$ DRMBM$(poly(i), poly(i), O(1))$. Let π be some maximization problem of form 9.3 on page 134 over some independence system (E, S). Then[2],*

- $\pi \in$ DRMBM$(poly(n), poly(n), O(1))$,

[1] The parallel prefix sum problem is defined as follows [70]: given n input numbers $(x_1, x_2, \ldots, x_n)$, compute $(y_1, y_2, \ldots, y_n)$, such that $y_i = \sum_{j=1}^{i} x_i$, $1 \leq i \leq n$.

[2] Recall from Definition 7.2 on page 107 that, for some problem π, π/rt denotes the problem π in the presence of real-time constraints.

- $\pi \in$ NLOGSPACE, *and*
- $\pi/rt \in$ rt-PROC$^{\text{CRCW F-DRMBM}}(poly(n))$,

if and only if $\pi \in \mathcal{M}$.

Proof. The "if" part follows from Lemma 9.3 on page 136, and the "only if" part is established by Lemma 9.4 on the preceding page. ∎

By Theorem 9.5 we have precisely identified—among those optimization problems that can be expressed as independence systems—the class of such problems solvable in parallel real time. We believe that this result may be of interest for at least two reasons:

1. On one hand, consider those independence systems—or problems that can be formulated as such—not in $\mathcal{M}$ (with $\mathcal{M}$ as defined in Theorem 9.5). For these problems, finding an exact solution in real time is asymptotically impossible, even if a parallel machine is available (in the sense that the running time of any ($poly(n)$-processor) algorithm solving such a problem exceeds for large enough input size any (implicit or explicit) constant deadline). In such a case, one should probably look for either further restricting the problem (in order to bring it within $\mathcal{M}$), or find a reasonable approximation algorithm that is in NLOGSPACE.
2. On the other hand, Theorem 9.5 easily extends previous results, as we shall show in what follows.

9.3 Beyond Speedup, Extended

For a connected and undirected graph $G = (V, E)$, a *spanning tree* of G is a tree (i.e., an acyclic, connected, and undirected graph) $T = (V, E')$, with $E' \subseteq E$. A graph $G = (V, E)$ is *weighted* if there exists a function $w : E \to \mathbb{R}$. For some edge $e \in E$, $w(e)$ is the weight of e. A *minimum-weight spanning tree* of a weighted graph $G = (V, E)$ is a spanning tree $T = (V, E')$ of G such that $\sum_{e \in E'} w(e)$ is minimum over all the spanning trees of G. The

minimum-weight spanning tree problem is defined as follows: given a weighted, connected, and undirected graph G, compute a minimum-weight spanning tree of G. Whenever the exact meaning will be clear from the context, we shall use the acronym MST for both a minimum-weight spanning tree and the problem of computing such a tree.

The real time variant of MST is investigated in [9], where it is shown that the best approximate solution returned by a sequential algorithm can be arbitrarily worse than the solution obtained by a parallel algorithm (which actually returns the optimal solution).

Specifically, the following incremental variant of MST is analyzed in [9]: A connected, undirected, and weighted graph $G = (V, E)$ with $|V| = n$, $n \geq 1$, is given. Initially, the minimum-weight spanning tree of G is also known; it consists of n vertices and the $n - 1$ weighted edges connecting them. Time is divided into intervals of cn^{ϵ} time units, where c is a positive constant and $0 < \epsilon < 1$. At the beginning of each such a time interval, a new vertex and its associated edges are received. A new MST, or a best approximation possible to it, incorporating the new data must now be computed. The new tree should be reported at the beginning of the next time interval. At most $n^2 - n$ new vertices are received in all.

A parallel CRCW PRAM algorithm using $O(|V|^2)$ processors can compute the exact updated MST within one time interval of cn^{ϵ} time units [9]. On the other hand, all a sequential algorithm can do in this limited time is to replace up to n^{ϵ} of the existing edges with an equal number of new edges of smaller weight. This, of course, does not guarantee that the resulting tree is a minimum-weight spanning one. In fact, the ratio of the weight of the sequential solution to the weight of the parallel (exact) solution can be made arbitrarily large. We also note that the above sequential algorithm is optimal. Indeed, it is immediate that, in general,

Observation 4 Replacing [adding, deleting] α edges in [to, from] a minimum-weight spanning tree takes $\Omega(\alpha)$ time on the RAM.

One can notice that MST can be trivially transformed from a minimization problem

into a maximization one: just negate all the edge weights, and then add to every weight the absolute value of the maximum edge weight. Furthermore, it is immediate that the MST problem can be expressed as a matroid [38]. Thus, we can both tighten and extend the result described in [9] by using Theorem 9.5 on page 137.

First, we shall not restrict ourselves to connected graphs, since the extension to unconnected ones (when the tree becomes a forest) is immediate (we will, however continue to denote the problem by MST for uniformity). As well, our result is not restricted to incremental version presented in [9]; indeed, our result holds for any real-time variant of the MST problem.

We note that the result in [9] is not tight: Time up to cn^{ε} is allowed for each (parallel or sequential) real-time computation leading to the result. This running time, however, asymptotically exceeds any (however large) constant deadline imposed to the computation by some real-time environment. Still, the same result holds for true real-time computations as well. Indeed, we show in what follows that, for any real-time environment one can encounter, a parallel algorithm can solve MST arbitrarily better than a sequential one. That is, while the parallel implementation is able to return an optimal solution, even an optimal sequential algorithm can only report an approximate result in the limited time which is available due to the real-time constraints. This result, an immediate consequence of Theorem 9.5 on page 137, is given in Lemma 9.6 below[3].

Lemma 9.6 *Let MST denote the problem of computing the minimum-weight spanning forest on nondirected and weighted graphs. Then,* MST $\in$ DRMBM$(poly(n), poly(n), O(1))$ *(and thus* MST $\in$ NLOGSPACE, MST$/rt$ $\in$ rt-PROC$^{\text{CRCW F-DRMBM}}(poly(n))$*), and the best approximate solution to* MST$/rt$ *returned by a RAM algorithm is arbitrarily worse than the solution obtained by a parallel RMBM algorithm with polynomially bounded resources.*

Proof. Function *ur* for MST can be computed in logarithmic space (and thus in real-time

[3] Recall from Definition 7.2 on page 107 that, for some problem π, π/rt denotes the problem π in the presence of real-time constraints.

on RMBM): $ur\{e_1, e_2, \ldots, e_i\}$ is simply i minus the number of connected components in the graph induced by $\{e_1, e_2, \ldots, e_i\}$, and can thus be computed by performing a reflexive and transitive closure (which is an NLOGSPACE-complete problem [84]). By Theorem 9.5, it follows that MST can be computed *exactly* in real time on an RMBM, no matter how tight the deadlines are.

However, an optimal sequential algorithm that solves the same problem has a running time that cannot accommodate even the most generous deadline, and thus a sequential algorithm to some real time variant of MST can only guess *some* solution, and the guess can be arbitrarily bad. Indeed, such a result is shown in [9] for the incremental version presented at the beginning of this section. The generalization for any variant of MST and any abstract machine follows from Observation 4 on page 139. ∎

In fact, the second part of the proof of Lemma 9.6 also proves that this type of behavior (namely a parallel algorithm being able to compute an arbitrarily better solution than the optimal sequential one) is not an exclusive feature of the MST problem, but it applies to many more real-time computations instead.

Indeed, consider those problems π in $\mathcal{M}$, with $\mathcal{M}$ as in Theorem 9.5 on page 137, for which any optimal solution incorporate $\Omega(|i|^{\xi})$ input data on any instance i of π, for some constant $\xi > 0$. Denote the class of such problems by $\mathcal{M}'$. We note that all but the most trivial optimization problems return a subset of the input data of non-constant size, and thus they belong to the class $\mathcal{M}'$. For any $\pi \in \mathcal{M}'$, consider now the (possibly non-optimal) solution returned by a sequential algorithm in constant time. Such an algorithm cannot inspect more than $O(1)$ input data. Therefore, the ratio between the sequential, constant time solution and the optimal one is of the order of n^{ξ} for any instance of size n. The following result is then immediate:

Corollary 9.7 *For any $\pi \in \mathcal{M}'$, the best approximate solution to π/rt returned by a RAM algorithm is arbitrarily worse than the solution obtained by a parallel RMBM algorithm with polynomially bounded resources.*

In other words, the results obtained in [9] do hold even for the tightest real time environment, as shown in Lemma 9.6. In addition, these results are not applicable only to the MST, but to a whole class of problems instead, namely $\mathcal{M}'$. Given that, for all practical purposes, $\mathcal{M}'$ and $\mathcal{M}$ are identical, Theorem 9.7 shows that there exists not only a problem, but a whole family of problems for which a parallel implementation can do something other than speed up computation, namely improve the offered solution.

Chapter 10

On Real-Time Approximation Algorithms

Summary

As a consequence of Claim 2 on page 110, the problem of finding approximate solutions computable in real time (in those cases when the exact solution cannot be computed within the given time restrictions) becomes a worthy pursuit. Such an approach is common in classical complexity theory. Indeed, in the sequential case, NP-hard problems are for all practical purpose (unless P equals NP) not computable but for the smallest instances, and thus deterministic polynomial time approximations are usually sought [43]. Similarly, this time in the context of parallel computations, efficient parallel approximations to P-complete (that is, inherently sequential unless NC equals P) problems were also investigated [45].

The identification from Claim 2 of NLOGSPACE as the class containing exactly all the problems solvable in real time naturally extends such a search for approximation algorithms: Once a problem is shown as being likely not solvable in real time (that is, not in NLOGSPACE), then an approximate solutions may become attractive. We now offer a incipient discussion on this matter. In particular, we show that many P-complete problems do not admit good real-time approximation algorithms, but that the *bin packing* problem

(an NP-complete one!) does admit such a good approximation parallel real-time algorithm.

10.1 Real-Time Approximation Schemes and Problems not Admitting Real-Time Approximation Algorithms

First, we define the notion of "good" approximation algorithms by adapting the definitions already used [43, 45] to our framework. The following definition is applicable to any (parallel or sequential) algorithm (running on any computational model).

Definition 10.1 Consider some algorithm A working on instance i of a minimization (maximization) problem, and suppose that A delivers a candidate solution with value $A(i)$. With $Opt(i)$ denoting the value of the optimal solution for input i, the *performance ratio* of A on i is $R_A(i) = A(i)/Opt(i)$ ($R_A(i) = Opt(i)/A(i)$). The *absolute performance ratio* of A is defined as $R_A = \inf\{r \geq 1 | R_A(i) \leq r$ for all instances $i\}$.

An algorithm A with inputs $\varepsilon > 0$ and $i \in \pi$ is an *approximation scheme* for π if and only if A delivers a candidate solution with performance ratio $R_A(i) \leq 1 + \varepsilon$ for all $i \in \pi$. In addition, if $A \in$ rt-PROC$(poly(|i|))$, then A is a *real-time approximation scheme* for π.

The body of knowledge regarding NC approximations [45] gives some negative results: Once it is proved that some problem does not admit an NC approximation algorithm, it follows that no NLOGSPACE (and thus real-time) approximation algorithm exists either, since NLOGSPACE $\subseteq$ NC.

Theorem 10.1 *If* P $\neq$ NC, *then there exists no real-time approximation scheme for the following problems:*

- *Lexicographically first maximal independent set [45].*
- *Unit resolution (the problem of whether the empty clause can be deduced from a given propositional formula in conjunctive normal form) [80].*

- *Generability (given a finite set W, a binary relation $\bullet$ on W, a subset $V \subseteq W$, and $w \in W$, determine whether w is in the smallest subset of W that contains V and is closed under $\bullet$) [80].*
- *Path systems (given a path system $P = (X, R, S, T)$, $S, T \subseteq X$, $R \subseteq X \times X \times X$, determine whether there exists an admissible vertex in S) [80].*
- *Circuit value (given a combinational circuit built from two-input Boolean gates and an assignment for its inputs, compute the output) [80].*
- *High degree subgraph (given a graph G and an integer k, does G contain an induced subgraph with minimum degree at least k?) for $k \geq 3$ [80].*
- *Linear programming, in both the following cases: the approximation solution should be a vector close to the optimal one, and the approximation solution seeks the objective function to have a value close to optimal [79].*

Proof. It has been proven (references to proofs are given within the theorem) that any approximation scheme for these problems is P-complete. Since P $\neq$ NC and NSPACE $\subseteq$ NC, Claim 2 on page 110 implies that the above problems do not admit any real-time approximation scheme. ∎

10.2 A Real-Time Approximation Scheme for Bin Packing

We focus now our attention to the *bin packing* problem. True, there is a wide gap between bin packing (an NP-complete problem) and the class of real-time computations (NLOGSPACE), so one may think that there is little hope to find a good *real-time* approximation scheme to such a problem. However, we note that good NC approximation schemes (still not real time but closer) for this problem already exist [15]. Besides, bin packing is closely related to certain scheduling problems (since the item to be packed can

be viewed as tasks to be scheduled), and it is thus conceivable that real-time approximation algorithms can be of use for scheduling tasks in real time on a parallel machine (the utility of such a processing being evident). There is thus a good motivation for seeking real-time approximation schemes for such a problem.

The input for the bin packing problem consists in n items, each of size within interval $(0, 1)$. The n items should be packed in a minimal number of bins of unit capacity.

One of the successful approaches in developing sequential (that is, in P) bin packing approximation algorithms is the use of simple heuristics. In this respect, one should mention the *first fit decreasing* (FFD) heuristic, which considers the items in nondecreasing order of their size, and places each item into the first available (that is, with enough free space) bin. Even if simple, the length of the packing returned by FFD, of at most $11/9 \times Opt + 3$ (where Opt is the length of the optimal solution), is a good approximation, qualifying FFD as an approximation scheme. Still, it is not only intuitive that FFD is inherently sequential (that is, P-complete):

Proposition 10.2 [15] *Given a list of items, each of which having a size in the interval* $(0, 1)$*, in nonincreasing order, and two indices i and b, it is P-complete to decide whether the FFD heuristic will pack the ith item into the bth bin. This is true even if the item sizes are represented in unary.*

Even if FFD is inherently sequential, an NC algorithm that achieves the same performance as FFD (although by using different techniques) is given in [15]. This algorithm works in two stages, as follows:

1. The first stage packs all the items that have a size of at least $1/6$. Such a stage starts by *sorting* the list of items in nonincreasing order. Then, a constant number of passes are performed, each pass involving two algorithms: (a) *merge* two sorted lists of n elements each into a sorted list, and (b) in a string of length n of *opening and closing parentheses*, find the matching pairs.

2. In the second stage, the remaining items are packed. This stage involves a (relatively large) number of *parallel prefix computations*. The parallel prefix problem [70] is defined as follows: given n input numbers $(x_1, x_2, \ldots, x_n)$, compute $(y_1, y_2, \ldots, y_n)$, such that $y_i = \sum_{j=1}^{i} x_i$, $1 \leq i \leq n$.

Theorem 10.3 *Bin packing admits a real-time approximation scheme A such that* $A(i) \leq 11/9 \times Opt(i) + 3$ *for any instance i.*

Proof. We follow the algorithm presented in [15], showing how this algorithm can be implemented in real time. According to Theorem 7.6 on page 104, we have a choice of showing that this algorithm is in NLOGSPACE or in DRMBM$(poly(n), poly(n), O(1))$. We chose the latter variant.

First, we note that sorting can be done in constant time on a (nondirected or directed) CREW RMBM using $poly(n)$ processors and $poly(n)$ buses [87]. Then, it is immediate that merging two sequences into a sorted sequence is also computable in constant time on RMBM. Indeed, the quick and dirty method of sorting (using the algorithm mentioned above) the two lists concatenated together will do the trick.

The problem of matching parentheses can be implemented in two steps as follows: First, the unmatched parentheses can be eliminated by a parallel prefix computation. Then, there exists a constant time algorithm on DRN using $poly(n)$ processors for matching the remaining sequence of parentheses [5]. However, this implies the existence of a similar algorithm on RMBM with polynomially bounded number of processors and buses, according to Corollary 7.7 on page 105.

Thus, the only algorithm that is still needed is the parallel prefix computation, which is in CREW RMBM$(poly(n), poly(n), O(1))$ according to [87] (in fact, such an algorithm is the basis for the aforementioned sorting algorithm).

In conclusion, all the algorithms used by the two stages on the NC approximation scheme described in [15] are in CREW RMBM$(poly(n), poly(n), O(1))$. Since

these algorithms are applied a constant number of times, the whole processing is in CREW RMBM$(poly(n), poly(n), O(1))$ and thus in rt-PROC$(poly(n))$. This completes the proof. ∎

In passing, one should note that the algorithm from the proof of Theorem 10.3 apparently requires a large (albeit constant) amount of time to complete. However, such a construction is enough to prove that bin packing admits a real-time approximation scheme. Indeed, the existence of an algorithm in CREW RMBM$(poly(n), poly(n), O(1))$ implies the existence of another algorithm, that solve the same problem, but this time in CRCW F-DRMBM$(poly(n), poly(n), O(1))$, and whose running time is very small, as shown in Corollary 7.8 on page 105. True, we do not offer a constructive proof for this corollary, and thus the CRCW F-DRMBM$(poly(n), poly(n), O(1))$ algorithm cannot be effectively constructed using only the results from this chapter. Still, if needed, we believe that, although not a trivial matter, developing such a constructive transformation is feasible.

Chapter 11

The Characterization of Constant Time RN Computations

Summary

In the process of characterizing real-time computations, we also determined in Section 7.1 on page 97 the computational power of DRMBM running in constant time. We showed that DRMBM and DRN with constant running time have the same computational power (refer to Section 2.2 on page 16 for a precise definition of these models). In addition, we showed that, in the case of constant time RMBM computations, no conflict resolution rule is more powerful than Collision, and that a unitary bus width is enough. These results are presented in Theorem 7.6 and Corollary 7.8 on page 105—also see the proof of Lemma 7.5 on page 103 that provides the crux of the whole result.

We now study whether such properties (Collision being the most powerful resolution rule and unitary bus width being sufficient) hold for the other model with reconfigurable buses, namely the RN. Indeed, we find that the same properties do hold, as shown in Theorems 11.2 on the next page and 11.3 on page 151.

Theorem 11.1 *All the results developed throughout this thesis on the relations between real-time computations and RMBM computations continue to hold if one replaces RMBM by RN.*

11.1 Write Conflict Resolution Rules on RN

The generality of the Collision resolution rule is not limited to RMBM computations. Indeed, the same property holds for constant time computations on RN as well.

Theorem 11.2 *For any* $X \in \{\text{CRCW}, \text{CREW}\}$*,* $Y \in \{\text{D}, \lambda\}$*, and for any write conflict resolution rule, it holds that* $X\,Y\text{RN}(poly(n), O(1)) \subseteq \text{CRCW}\,\text{DRN}(poly(n), O(1))$ *with the Collision resolution rule.*

Proof. First, note that $\text{CRCW}\,\text{DRN}(poly(n), O(1)) = \text{NLOGSPACE}$ for the Collision resolution rule [19]. Thus, we complete the proof by showing that, for any conflict resolution rule, $\text{CRCW}\,\text{DRN}(poly(n), O(1)) \subseteq \text{NLOGSPACE}$.

This result is however given by the proof of Lemma 7.5 on page 103. 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 a 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 7.5 follows then unchanged.

The extra space used in the process of generating bus k consists in 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. ∎

11.2 Bus Width Bounds on RN

We identified in Theorem 7.6 and Corollary 7.8 on page 105 a gap in the complexity hierarchy of RMBM computations: As far as constant time computations are concerned, there is no need for a large bus width; instead, buses composed of single wires are sufficient.

It is natural to wonder whether a similar result holds for DRNs, so we investigate now this matter. As expected, we find that a bus width of 1 is enough for all constant time computations on RNs:

Theorem 11.3 *For any problem π solvable in constant time on some variant of RN, it holds that $\pi \in \text{CRCW DRN}(poly(n), O(1))$ with Collision resolution rule and bus width* 1*.*

The proof of Theorem 11.3 follows from the following two lemmas. Recall from Definition 7.1 on page 97 that $GAP_{1,n}$ denotes the following version of the Graph Accessibility Problem: Given a directed graph $G = (V, E)$ (expressed by the (boolean) incidence matrix I), $V = \{1, 2, ..., n\}$, determine whether vertex n is accessible from vertex 1.

Lemma 11.4 $GAP_{1,n} \in \text{CRCW DRN}(n^2, O(1))$ *with Collision resolution rule and 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 11.1 on the following page. 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 incidence 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.

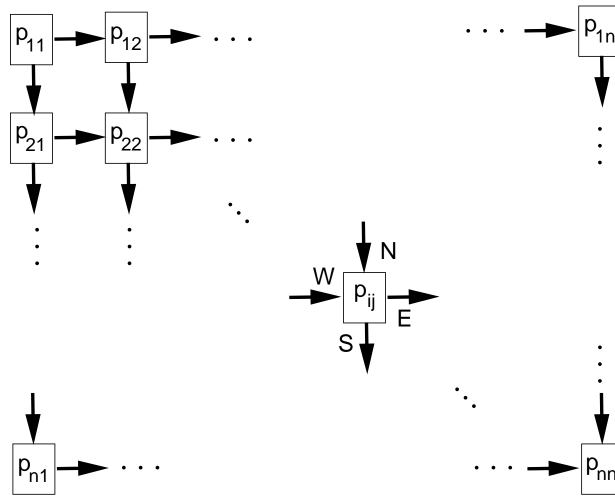

Figure 11.1: A mesh of $n \times n$ processors

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.

It is immediate that R solves $GAP_{1,n}$, by the same argument as in Lemma 7.1 on page 97 (also note that a similar construction is presented and proved correct in [94]). In addition, the content of the signal received by p_{nn} is clearly immaterial, so a bus of width 1 suffices. The proof is thus complete. ∎

Recall now that the graph $G(M, x)$ is the graph of configurations of the Turing machine M working on input x (see the discussion immediately following Corollary 7.2 on page 98 for a precise definition).

Lemma 11.5 *For any language $L \in$ NSPACE$(\log n)$ (with the associated NSPACE$(\log n)$ Turing machine M accepting L), and given some word x, $|x| = n$, there exists a constant time CREW DRN algorithm using $poly(n)$ processors and buses of width 1 that computes $G(M, x)$ (as an incidence matrix I).*

Proof. This fact is obtained by the same argument as the one presented in the proof of Lemma 7.3 on page 100. Indeed, except for the distribution of input x to processors, there

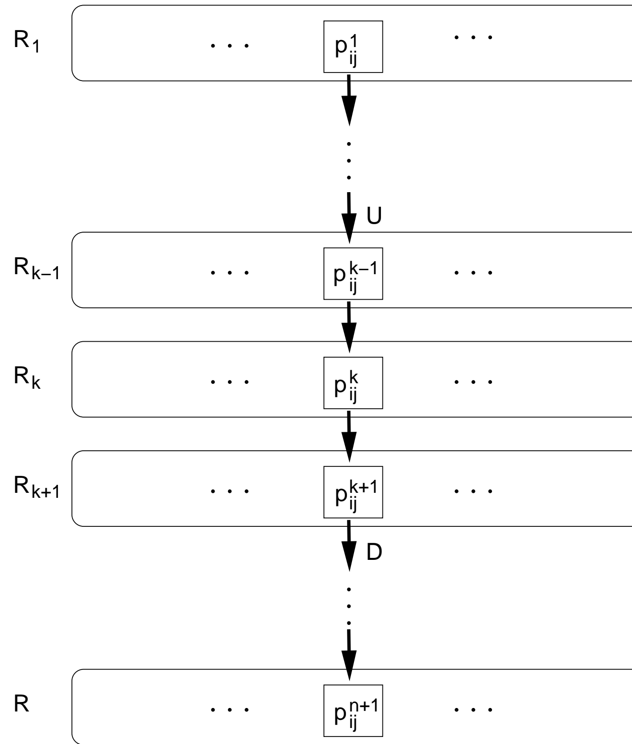

Figure 11.2: A collection of n meshes connected together

is no interprocessor 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 11.1 on the preceding page, 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 11.2.

At the beginning of the computation, x_k, the kth 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 7.3 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^k_{ij}, $1 \leq k \leq n$, know the configuration i (and thus the value of h_{ij}).

It remains therefore to show now how $x_{h_{ij}}$ reaches processor p^{n+1}_{ij} in constant time and we are done. Indeed, after this distribution is achieved, R is able to compute the incidence matrix I exactly as shown in the proof of Lemma 7.3. 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^k_{11} broadcasts x_k to all the processors in R_k. To do this, all processors p^k_{ij} fuse together their N, S, E, and W ports, and then p^k_{11} places x_k on its outgoing buses.

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

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

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

5. Finally, each p^{n+1}_{ij} 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 incidence 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^k_{ij}, $1 \leq k \leq n$, such that $k = h_{ij}$. Therefore, we realized the input distribution.

I_{ij} is then computed by processor p^{n+1}_{ij} without further communication, as shown in the proof of Lemma 7.3. 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 (as per the discussion immediately following Corollary 7.2 on page 98). ∎

Given Lemmas 11.4 and 11.5, the proof of Theorem 11.3 on page 151 is immediate:

Proof of Theorem 11.3. That the Collision resolution rule is the most powerful follows from Theorem 11.2 on page 150. It remains to be shown only that a bus width 1 suffices.

Given some language L in NSPACE$(\log n)$, let M be the (NSPACE$(\log n)$) Turing machine accepting L. For any input x, the DRN algorithm that accepts L works as follows: Using Lemma 11.5 on page 152, it obtains the graph $G(M,x)$ of the configurations of M working on x (by computing in effect the incidence matrix I corresponding to $G(M,x)$). Then, it applies the algorithm from Lemma 11.4 on page 151 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 11.5 are in the right place as input for p_{ij} in the algorithm from Lemma 11.4 (that uses only the mesh R). It is immediate given the aforementioned lemmas 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 RN are included in NLOGSPACE. ∎

11.3 Open Problems

By Theorem 11.2 on page 150 and Theorem 11.3 on page 151 we found that there exists a very strong similarity between the two models with reconfigurable buses, the RN and the RMBM: Not only they solve the same problems (namely, exactly all the problems in NLOGSPACE, or, conforming to Claim 2 on page 110, exactly all the problems solvable in real time), 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 more powerful than the Combining rule).

We conclude this chapter by mentioning a set of intriguing open problems related to

such a characterization:

First, the Collision resolution rule is established as the most powerful only for constant time algorithms. Minor changes to the proof of Theorem 11.2 also extend the result to the RMBM and RN algorithms using polynomially bounded resources and running in $poly(n)$ time. Thus, the extreme cases are established. As such, a possible generalization of our results is to investigate whether this property holds for intermediate cases, e.g., for RMBM/RN algorithms running in polylogarithmic time. We expect an affirmative answer.

As well, we note an apparent contrast between the power of conflict resolution rules for models with reconfigurable buses (RMBM and RN) on one hand, and for shared memory models (PRAM) on the other hand. According to our results (and with the expected generalization mentioned in the above paragraph), Collision is the most powerful rule on RMBM/RN. By contrast, it is widely believed that the Combining CRCW PRAM is more powerful than the CRCW PRAM using the equivalent of a Collision resolution rule. To our knowledge however no proof with this respect exists to date. We believe that an investigation in this direction is an interesting pursuit. We also believe that the contrast between RN and PRAM is not only apparent (that is, we believe that the Combining CRCW PRAM is indeed more powerful than the Collision CRCW PRAM).

Chapter 12

Conclusions and Open Problems

Classical complexity theory is of central concern not only for theorists, but also for practitioners. For example, the existence of a lower complexity bound for some problem is an important fact: No matter how clever a program is, the bound cannot be overcome. On the other hand, as noted in Chapter 2 on page 10, the term "real time" is used by the complexity theorists in a somewhat different manner than in the real-time systems community. Thus, as useful as a complexity theory is, practitioners in the real-time systems area do not have such a theory to refer to. As a consequence, any question related to resource allocation and even solvability for a real-time problem is unique, in the sense that the answer to such a question should be developed from scratch (either by experiments or using individualized proofs). A complexity theoretic approach to the real-time algorithms should offer a common ground to which practitioners can refer in order to get readily available answers to this kind of questions.

The model of well-behaved timed ω-languages intends to bridge the gap between theorists (i.e., complexity theory) and the systems researchers (i.e., practice): While it is a formal model, it captures all the features of real-time computations as understood by the systems community. Therefore, this model has little in common to the theorists' real-time concept (which more often than not stands for on-line and linear time). In particular, we believe

that well-behaved timed ω-languages accurately model the notion of real-time computation used in the systems community.

A note on the differences and similarities between timed ω-languages (that is, real-time algorithms) and classical formal languages (that is, classical algorithms) is in order. On one hand, it is immediate that formal languages are particular cases of timed ω-languages. Indeed, save for the time sequence, any word is a timed ω-word. If one relies on the semantics of the time sequence, one can add the time sequence $00\ldots0$ to a classical word and obtain the corresponding timed ω-word. However, none of the timed ω-words obtained in this manner is well-behaved. We have thus a crisp delimitation between real-time and classical algorithms, while keeping the formalisms as unified as possible. Moreover, the devices that are used for recognition of timed ω-languages enjoy the same crisp delimitation. Indeed, an acceptor for a timed language is simply an algorithm running on some existing model of computation. The timeliness constraints are given by the semantics of the timed ω-word that is received as input. Not only such a construction takes full advantage of the existing results from classical complexity theory, but it is also close to the real world, where real-time computations are carried out by normal computers under timing constraints that are imposed by the external environment (that is, imposed by the time sequence included in the input) over input and/or output.

We defined complexity classes for timed ω-languages, that capture an intuitive notion of real-time efficiency, and studied the relations between these classes and between them and existing complexity classes. In particular, we showed that parallel real-time computations form an infinite hierarchy with respect to the number of processors. In addition, this result is invariant with respect to the model of parallel computation involved, and independent of the characteristics (that is, speed) of the particular processors used by the algorithms. From a practical point of view, Theorem 6.9 on page 93 emphasizes the need for looking into parallel implementations, since this theorem shows that parallelism can

add computational power, in a more general sense than mere speed, to a real-time application.

Granted, the languages PURSUIT_k are primarily of theoretical interest. Most practical real-time applications feature explicit deadlines imposed on the computation. As such, we focused next on characterizing this latter class of computations. Given any language that can be accepted by a machine using logarithmic work space, we showed in Theorem 7.10 on page 108 that such a language can be accepted by a parallel machine with polynomially bounded resources, in the presence of *any* (that is, however tight) real-time constraints.

The way Theorem 7.10 has been obtained (by means of Theorem 7.6 on page 104) suggests an even stronger result. Indeed, according to Theorem 7.6, NLOGSPACE computations are *the only* computations in the classical sense that can be performed in constant time by DRMBMs. This suggests that the relation between real time and NLOGSPACE is even stronger, the two classes being in fact identical. If it were not, that is, for those real-time algorithms that do not feature explicit deadlines. . . We offered, however, good evidence that such class of computations in included in NLOGSPACE once the real-time constraints are eliminated. Specifically, we showed that pursuing outside the real-time domain is easy (indeed, we found in Theorem 8.1 on page 111 that, though hard to recognize, the languages PURSUIT_k are even in LOGSPACE once the real-time constraints are eliminated). Then, Theorems 8.5 on page 115, 8.9 on page 121, 8.10 on page 122, and 8.16 on page 128 show that both d-algorithms and c-algorithms (arguably the archetype of real-time input arrival) are little more than on-line algorithms with added real-time constraints on input, and that in both cases these restrictions on input impose in effect explicit deadlines on the output. This allows us to state Claim 2 on page 110, which offers a nice counterpart of the parallel computation thesis [45, 70]. In this thesis, NC is conjectured to contain exactly all the computations that admit efficient ($poly(n)$ processors and polylogarithmic running time) parallel implementations. By contrast, we conjecture that NLOGSPACE contains exactly

all the computations that admit efficient ($poly(n)$ processors) real-time parallel implementations.

To summarize, we believe that the two main points of the research described in this thesis are the following:

1. We offered a consistent, expressive, and realistic model for real-time computations.

2. We offered a complexity theoretic characterization for parallel real-time computations. Most importantly, we find that real-time computations pertain to a well studied class (NLOGSPACE).

12.1 Open Problems

We believe that this thesis opens several directions for future research. We presented some incidental open problems in Section 11.3 on page 155. We summarize now what we believe to be the major directions opened by our work.

12.1.1 Timed Languages In Practice

One of the conjectured properties of timed ω-languages is their applicability and usefulness in practice. We believe that this model would be a useful additional tool in characterizing problems from many areas of the real-time systems practice such as real-time and active database systems [16, 69, 68, 91, 92], industrial applications [58, 86], or routing in ad hoc networks [18, 25, 46].

This is supported by the models developed in Chapter 5 on page 58. Indeed, we have offered there a formulation of the recognition problem in *real-time database systems*. Since the recognition problem is an important tool for determining the complexity of queries in classical database systems [2], we believe that one is able to derive similar complexity results in the real-time domain using our model.

In particular, one of the practical areas that we are especially interested in is the problem of computing approximate answers to queries in real-time database systems whenever exact answers cannot be returned within the given time constraints (a data model and a query language—which is an extension of relational algebra—for such a processing is given in [91, 92]). At first sight, this is almost a non-problem, as relational algebra queries can be computed in constant time with a polynomially bounded number of processors [2]. That is, *exact* answers can be given even in the most constrained real-time environment. However, a more in-depth look reveals that, in practice, the number of processors that are available is likely to be lower. In addition, communication and synchronization issues are also likely to be significant, and, most important, the complexity of the rule-based system that is part of a real-time database systems is an unexplored component. Thus, additional interesting issues related to scalability, communication, and the active component of real-time databases have to be taken into account. We note that simulation-based results regarding the performance of real-time approximate query processing mechanisms show that such mechanisms do offer an improved predictability [92]. However, these results are restricted in scope for two reasons: On one hand, only one approximate query processing system is analyzed; thus, no general result on the upper and lower bounds for the improvement is available. On the other hand, no analytical expression of the improvement in predictability can be obtained by simulations. Thus, we note here, again, the problem that we signal the the beginning of this chapter: When faced with the problem of designing a real-time database system, a practitioner is currently faced with a unique problem; no general predictions of the system performance exists.

We thus identify the complexity analysis of real-time database systems (as a common ground to which practitioners can refer) to be a promising research direction continuing (and specializing) the work from this thesis. To our knowledge no attempt at such a complexity theoretic characterization exists to date.

In the same direction (that is, of building a common ground for practitioners), we note

that a comparative performance evaluation of routing algorithms was proposed for the first time in [25], where several routing algorithms are compared based on discrete event simulation. To our knowledge, no analytical model has been proposed to date. Our model of the routing problem in ad hoc networks (Section 5.4 on page 70) has the potential of becoming such an analytical tool. We believe that using this model in order to derive bounds for the performance of routing algorithms in ad hoc networks is another interesting research direction.

12.1.2 Approximation Algorithms

A consequence of Claim 2 on page 110 is that there are many problems that cannot be solved in real time. Thus, the following research direction becomes useful: Which are those problems that, although possibly not solvable in the real-time environment imposed by some real-time application, admit "good" approximate solutions provably achievable in any real-time environment? Do they form a well-defined complexity class? If so, which are the problems pertaining to such a class? This thesis offers a solid basis for the pursuit of this direction, since we identify here a class of candidates for approximating algorithms. In addition, this class of candidates is either NLOGSPACE or F-DRMBM$(poly(n), poly(n), O(1))$, whichever is more natural for the given problem, since they are in fact identical as shown by Theorem 7.6 on page 104.

As a starter for such a direction, along with identifying some problems not admitting real-time computable approximate solutions, we showed that real-time approximation schemes do exist. Interestingly enough, we found with relative ease such an approximation algorithm for quite a hard (in fact, NP-complete) problem, namely bin packing. This is a nice argument in favor of the relations that we discovered between NLOGSPACE,

RMBM, and real-time computations, and a good motivation for the use of timed ω-languages in the study of (approximate or not) real-time computations. However, Chapter 10 on page 143 offers only an incipient discussion on real-time approximation algorithms. We believe that the field is open to a multitude of future research subjects.

12.1.3 From Complexity Theory to Specification and Validation

Besides the immediate utility of complexity theory (which offers general results, including upper and lower bounds for resource requirements), it is often the case that such a theory can be used to generate tools for system specification and validation. For example, most work in compiler design is based on formal languages, and at least two stages of a compiler (lexical analysis and parsing) are based on formal grammars and their associated automata [3].

Specification and validation of real-time systems is not in the scope of this thesis, which is entirely dedicated to complexity theoretic results. Nonetheless, the formalism of timed ω-languages presented here supports the development of specification and validation tools. We believe that developing such tools is another promising future research direction. Specifically, this process involves combining the formalism of timed ω-languages with a precise semantics for their acceptors (based, for example, on temporal logic, as suggested in another context in [40]).

In the same direction, real-time distributed models are particularly interesting. Indeed, distributed systems are increasingly present in nowadays computing practice, including the area of real-time computations. The extension from one timed ω-word (that models a real-time computation in general) to a set of n such words communicating with each other (as an explicit model for distributed real-time computations) has been sketched in Section 5.4.4 on page 76. We believe that further developing this construction is yet another promising research direction.

12.2 Incidental results

We conclude this thesis by summarizing results that, although not an integral part or this work's main research interest (to model and characterize real-time computations), are nonetheless noteworthy.

Characterization of DRMBM and DRN computations We determined the computational power of DRMBM (directed reconfigurable multiple bus machines) running in constant time. We showed that DRMBM and DRN (directed reconfigurable networks) with constant running time have the same computational power. In addition, we showed that, for both RN and RMBM computations, no conflict resolution rule is more powerful than Collision (the RMBM result is given in Theorem 7.6 and Corollary 7.8 on page 105; as well, see the proof of Lemma 7.5 on page 103 which is the crux of the whole result; the RN result is established by Theorem 11.2 on page 150).

We note that the discussion on the comparative computational power of various resolution rules is of very reduced importance with respect to tightly coupled parallel machines such as the PRAM. Indeed, the combinatorial logic of Combining CRCW PRAM can be implemented using no more resources than the ones needed for the implementation of, say, the CREW PRAM [5]. Still, it is argued [88], and we agree with this argument, that the Priority rule is of questionable feasibility in the case of RMBM. In fact, the Combining rule on such systems is not even mentioned as a possibility, it being considered even less feasible. The reason for this statement is that RMBM is a loosely coupled system: The two (or more) processors that write on some bus are physically connected only through the bus itself. Thus, the sole responsibility of combining the the values written by the processors falls on the bus. Buses, however appear to take in the existing implementations of RMBM-like devices [44] a form equivalent to a set of plain electrical wires. Therefore, it is unlikely that performing the relatively complex computations required by the Combining rule (or, for that matter, Priority or Common rules) at the bus level is possible. In the case of RN,

some argument can be made for the feasibility of the Combining conflict resolution rules. Even so, the feasibility of such rules remains questionable to a high degree.

Questionable or not, the practical feasibility of rules like Priority or Combining on spatially distributed resources such as a buses is no longer of interest, at least in the area of constant time computations. Indeed, such rules are simply not necessary, since the Collision rule is all that is needed.

Finally, we identified a gap in the complexity hierarchy of RMBM computations as well: As far as constant time computations are concerned, there is no need for a large bus width; instead, buses composed of single wires are sufficient. This result is stated in Theorem 7.6 and Corollary 7.8 on page 105. As expected, the same result holds for constant time RN computations. This is formally shown in Theorem 11.3 on page 151.

The meaning of "parallel" is not exclusively "faster" Another incidental but nonetheless important result is a precise characterization of (real-time) maximization problems over independence systems, showing that a problem pertaining to this class is solvable in real time if and only if it is a matroid and the size of an optimal solution is computable in real-time. Incidentally, we note that our characterization of real time optimization problems is more precise than the characterization of the larger class of problems with fast parallel algorithms [45]: Both determine a subclass of independence systems (matroids with the size of optimal solution computable in real time, and matroids, respectively). However, independence systems that are not matroids but still admit fast parallel algorithms exist. By contrast, matroids with the size of optimal solution computable in real time are *exactly all* the independence systems solvable in real-time, as shown in Theorem 9.5 on page 137. We believe that such a tight characterization is important from a practical point of view, as non-membership in class $\mathcal{M}$ defined in Theorem 9.5 implies the impossibility of solving the given problem in real time. Thus, one should look in this case to either restricting the problem or finding reasonably good approximative solutions instead of exact ones.

Based on this characterization, we also generalized previous findings conforming to which parallel implementations can do more than speed up the computation [9]. Indeed, we use Theorem 9.5 to show that parallel means better for a large class of problems and for any set of real-time constraints. Indeed, Corollary 9.7 on page 141 shows that there exists not only a problem, but a whole family of problems for which a parallel implementation can unboundedly improve the offered solution.

Bibliography

[1] S. O. AANDERAA, *On k-tape versus $(k-1)$-tape real time computation*, in Complexity of Computation, R. Karp, ed., SIAM-AMS Proceedings, volume 7, 1974, pp. 75–96.

[2] S. ABITEBOUL, R. HULL, AND V. VIANU, *Foundations of Databases*, Addison-Wesley, Reading, MA, 1995.

[3] A. V. AHO, R. SETHI, AND J. D. ULLMAN, *Compilers, Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1985.

[4] S. G. AKL, *Discrete steepest descent in real time*, to appear in Parallel and Distributed Computing Practices.

[5] ——, *Parallel Computation: Models and Methods*, Prentice-Hall, Upper Saddle River, NJ, 1997.

[6] ——, *Nonlinearity, maximization, and parallel real-time computation*, in Proceedings of the Twelfth Conference on Parallel and Distributed Computing and Systems, Las vegas, NV, Nov. 2000, pp. 31 – 36.

[7] ——, *Superlinear performance in real-time parallel computation*, in Proceedings of the Thirteenth Conference on Parallel and Distributed Computing and Systems, Anaheim, CA, Aug. 2001, pp. 505 – 514.

[8] ——, *Secure file transfer: A computational analog to the furniture moving paradigm*, in Proceedings of the Conference on Parallel and Distributed Computing Systems, Cambridge, MA, November 1999, pp. 227–233.

[9] S. G. AKL AND S. D. BRUDA, *Parallel real-time optimization: Beyond speedup*, Parallel Processing Letters, 9 (1999), pp. 499–509. For a preliminary version see http://-www.cs.queensu.ca/home/akl/techreports/beyond.ps.

[10] ——, *Parallel real-time cryptography: Beyond speedup II*, in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV, June 2000, pp. 1283–1290. For a preliminary version see http://-www.cs.queensu.ca/home/akl/techreports/realcrypto.ps.

[11] ——, *Parallel real-time numerical computation: Beyond speedup III*, International Journal of Computers and their Applications, 7 (2000), pp. 31–38. For a preliminary version see http://www.cs.queensu.ca/home/akl/techreports/realnum.ps.

[12] S. G. AKL. AND L. FAVA LINDON, *Paradigms admitting superunitary behaviour in parallel computation*, Parallel Algorithms and Applications, 11 (1997), pp. 129–153.

[13] S. G. AKL AND G. R. GUENTHER, *Broadcasting with selective reduction*, in Proceedings of the IFIP Congress, 1989, pp. 515–520.

[14] R. ALUR AND D. L. DILL, *A theory of timed automata*, Theoretical Computer Science, 126 (1994), pp. 183–235.

[15] R. J. ANDERSON, E. W. MAYR, AND M. K. WARMUTH, *Parallel approximation algorithms for bin packing*, Information and Computation, 82 (1989), pp. 262–277.

[16] L. BÆKGAARD AND J. C. GODSKESEN, *Real-time event control in active databases*, Journal of Systems and Software, 42 (1998), pp. 263–271.

[17] S. K. BARUAH AND A. BESTAVROS, *Real-time mutable broadcast disks*, in Real-Time Database and Information Systems, A. Bestavros and V. Fay-Wolfe, eds., Boston, MA, 1997, Kluwer Academic Publishers, pp. 3–21.

[18] S. BASAGNI, I. CHLAMTAC, V. R. SYROTIUK, AND B. A. WOODWARD, *A distance routing effect algorithm for mobility (DREAM)*, in The Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking, Dallas, TX, 1998, pp. 76 – 84.

[19] 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.

[20] Y. BEN-ASHER, D. PELEG, AND A. SCHUSTER, *The power of reconfiguration*, Journal of Parallel and Distributed Computing, 13 (1991), pp. 139–153.

[21] A. BESTAVROS AND V. FAY-WOLFE, eds., *Real-Time Database and Information Systems*, Kluwer Academic Publishers, Boston, MA, 1997.

[22] R. V. BOOK AND S. A. GREIBACH, *Quasy–realtime languages*, Mathematical Systems Theory, 4 (1970), pp. 97–111.

[23] A. BORODIN AND R. EL-YANIV, *On-line Computation and Competitive Analysis*, Cambridge University Press, 1998.

[24] R. P. BRENT, *The parallel evaluation of general arithmetic expressions*, Journal of the ACM, 21 (1974), pp. 201–206.

[25] J. BROCH, D. A. MALTZ, D. B. JOHNSON, Y.-C. HU, AND J. JETCHEVA, *A performance comparison of multi-hop wireless ad hoc network routing protocols*, in The Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking, Dallas, TX, 1998, pp. 85–97.

[26] S. D. BRUDA AND S. G. AKL, *Real-time computation: A formal definition and its applications*, to appear in International Journal of Computers and Applications. For a preliminary version see http://turing.ubishops.ca/home/bruda/papers/timed-langs.

[27] ——, *On the data-accumulating paradigm*, in Proceedings of the Fourth International Conference on Computer Science and Informatics, Research Triangle Park, NC, Oct. 1998, pp. 150–153. For an extended version see http://turing.ubishops.ca/home/-bruda/papers/data_accum.

[28] ——, *The characterization of data-accumulating algorithms*, Theory of Computing Systems, 33 (2000), pp. 85–96. For a preliminary version see http://turing.ubishops.ca/-home/bruda/papers/data_accum2.

[29] ——, *Towards a meaningful formal definition of real-time computations*, in Proceedings of the ISCA 15th International Conference on Computers and Their Applications, New Orleans, LA, Mar. 2000, pp. 274–279. For an extended version see http://-turing.ubishops.ca/home/bruda/papers/timed-langs.

[30] ——, *A case study in real-time parallel computation: Correcting algorithms*, Journal of Parallel and Distributed Computing, 61 (2001), pp. 688–708. For a preliminary version see http://turing.ubishops.ca/home/bruda/papers/c-algorithms.

[31] ——, *On the necessity of formal models for real-time parallel computations*, Parallel Processing Letters, 11 (2001), pp. 353 – 361. For a preliminary version see http://-turing.ubishops.ca/home/bruda/papers/rttm.

[32] ——, *Parallel real-time complexity: A strong infinite hierarchy*, in Proceedings of VIII International Colloquium on Structural Information and Communication Complexity, Vall de Núria, Spain, June 2001, Carleton Scientific, pp. 45–59. For an extended version see http://turing.ubishops.ca/home/bruda/papers/pursuit.

[33] ——, *Pursuit and evasion on a ring: An infinite hierarchy for parallel real-time systems*, Theory of Computing Systems, 34 (2001), pp. 565–576. For an extended version see http://turing.ubishops.ca/home/bruda/papers/pursuit.

[34] ——, *On the relation between parallel real-time computations and logarithmic space*, in Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, Cambridge, MA, Nov. 2002, pp. 102–107. For an extended version see http://turing.ubishops.ca/home/bruda/papers/nlogspace.

[35] J. CLIFFORD AND A. CROCKER, *The Historical Relational Data model (HRDM) Revisited*, Benjamin/Cummings, CA, 1993, pp. 6–26.

[36] J. H. CONWAY, *On Numbers and Games*, Academic Press, 1976.

[37] S. A. COOK, *A taxonomy of problems with fast parallel algorithms*, Information and Control, 64 (1985), pp. 2–22.

[38] T. H. CORMEN, C. E. LEISERSON, AND C. STEIN, *Introduction to Algorithms*, MIT press, Cambridge, MA, 2 ed., 2001.

[39] E. CSUHAJ-VARJÚ, J. DASSOW, J. KELEMEN, AND G. PĂUN, *Grammar Systems. A Grammatical Approach to Distribution and Cooperation*, Gordon and Breach, London, 1994.

[40] E. A. EMERSON, *Real-time and the Mu-calculus (preliminary report)*, in Real-Time: Theory in Practice, J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, eds., 1991, pp. 176–194. Springer Lecture Notes in Computer Science 600.

[41] P. C. FISCHER, *Turing machines with a schedule to keep*, Information and control, 11 (1967), pp. 138–146.

[42] P. C. FISCHER AND C. M. R. KINTALA, *Real-time computations with restricted nondeterminism*, Mathematical Systems Theory, 12 (1979), pp. 219–231.

[43] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

[44] J. P. GRAY AND T. A. KEAN, *Configurable hardware: A new paradigm for computation*, in Proceedings of the Tenth Carltech Conference on VLSI, C. L. Seitz, ed., Cambridge, MA, Mar. 1989, MIT Press, pp. 279–295.

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

[46] Z. J. HAAS, *Panel report on ad hoc networks—Milcom'97*, Mobile Computing and Communications Review, 1 (1998), pp. 15–18.

[47] T. J. HARRIS, *A survey of PRAM simulation techniques*, ACM Computing Surveys, 26 (1994), pp. 187–206.

[48] J. HARTMANIS, P. M. LEWIS II, AND R. E. STEARNS, *Classification of computations by time and memory requirements*, in Proceedings of the IFIP Congress 65, Washington, DC, 1965, pp. 31–35.

[49] J. E. HOPCROFT AND J. D. ULLMAN, *Formal languages and their relation to automata*, Addison-Wesley, Reading, MA, 1969.

[50] S. IRANI AND A. R. KARLIN, *Online computation*, in Approximation Algorithms for NP-Hard Problems, D. Hochbaum, ed., International Thomson Publishing, 1997, pp. 521–564.

[51] K. JEFFAY, *The real-time producer/consumer paradigm: A paradigm for the construction of efficient, predictable real-time systems*, in Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing: States of the Art and Practice, 1993, pp. 796–804.

[52] N. D. JONES, *Computability and Complexity from a Programming Perspective*, MIT Press, Cambridge, MA, 1997.

[53] R. KANNAN AND B. KORTE, *Approximative combinatorial algorithms*, in Mathematical Programming, R. W. Cottle, M. L. Kelmanson, and B. Korte, eds., Elsevier Science Publishers, Amsterdam, The Nederlands, 1981, pp. 195–248.

[54] D. KNUTH, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Addison-Wesley, Reading, MA, 1969.

[55] G. KOBE, *The 42-volt revolution*, Automotive Industries, (Cover story, August 1998).

[56] C. KRAHE, *Airbus fly-by-wire aircraft at a glance: A pilot's first view*, in FAST, vol. 20, Airbus Industrie, December 1996, pp. 2–9. http://www.airbus.com/pdfs/customer-/fast20/p2to9.pdf.

[57] C. M. KRISHNA AND K. G. SHIN, *Real-Time Systems*, McGraw-Hill, New York, 1997.

[58] H. W. LAWSON, *Parallel Processing in Industrial Real-Time Applications*, Prentice Hall, Englewood Cliffs, NJ, 1992.

[59] M. R. LEHR, Y.-K. KIM, AND S. H. SON, *Managing contention and timing constraints in a real–time database system*, in Proceedings of the 16th IEEE Real–Time Systems Symposium, Pisa, Italy, Dec 1995, pp. 332–341.

[60] N. LEVESON AND J. STOLZY, *Analyzing safety and fault tolerance using timed Petri nets*, in Proceedings of the International Joint Conference on Theory and Practice of Software Development, 1985, pp. 339–355. Springer Lecture Notes in Computer Science 186.

[61] H. R. LEWIS AND C. H. PAPADIMITRIOU, *Elements of the Theory of Computation*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[62] F. LUCCIO AND L. PAGLI, *The p-shovelers problem (computing with time-varying data)*, in Proceedings of the 4th IEEE Symposium on Parallel and Distributed Processing, 1992, pp. 188–193.

[63] ——, *Computing with time–varying data: Sequential complexity and parallel speed–up*, Theory of Computing Systems, 31 (1998), pp. 5–26.

[64] F. LUCCIO, L. PAGLI, AND G. PUCCI, *Three non conventional paradigms of parallel computation*, in Parallel Architectures and Their Efficient Use, F. M. auf der Heide, B. Monien, and A. L. Rosenberg, eds., Springer Lecture Notes in Computer Science 678, 1992, pp. 166–175.

[65] A. MEYER AND P. C. FISCHER, *On computational speed-up*, in Conference Record of 1968 Ninth Annual Symposium on Switching and Automata Theory, Schenectady, New York, 15–18 Oct. 1968, IEEE, pp. 351–355.

[66] M. NAGY, *Parallelism in real-time computation*, Master's thesis, Department of Computing and Information Science, Queen's University, Oct. 2001.

[67] N. NAGY, *The maximum flow problem: A real-time approach*, Master's thesis, Department of Computing and Information Science, Queen's University, Jan. 2001.

[68] ÖZGÜR ULSOY, *Transaction processing in distributed active real-time database systems*, Journal of Systems and Software, 42 (1998), pp. 247–262.

[69] G. ÖZSOYOĞLU AND R. T. SONDGRASS, *Temporal real-time databases: A survey*, IEEE Transactions on Knowledge and Data Engineering, 7 (1995), pp. 513–532.

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

[71] W. PAUL, *On–line simulation of $k+1$ tapes by k tapes requires nonlinear time*, Information and Control, 53 (1982), pp. 1–8.

[72] M. O. RABIN, *Real time computations*, Israel Journal of Mathematics, 1 (1963), pp. 203–211.

[73] G. RAMALINGAM AND T. REPS, *On the computational complexity of dynamic graph problems*, Theoretical Computer Science, 158 (1996), pp. 233–277.

[74] S. RAO KOSARAJU, *Real-time pattern matching and quasi-real-time construction of suffix trees (preliminary version)*, in Proceedings of STOC 94, Montreal, Quebec, Canada, May 1994, pp. 310–316.

[75] D. RAPPAPORT, *Private communication*.

[76] J. H. REIF, *On dynamic algorithms for algebraic problems*, Journal of Algorithms, 22 (1997), pp. 347–371.

[77] A. L. ROSENBERG, *Real-time definable languages*, Journal of the ACM, 14 (1967), pp. 645–662.

[78] ——, *On the independence of real-time definability and certain structural properties of context-free languages*, Journal of the ACM, 15 (1968), pp. 672–679.

[79] M. J. SERNA, *Approximating linear programming is log-space complete for P*, Information Processing Letters, 37 (1991), pp. 233–236.

[80] M. J. SERNA AND P. G. SPIRAKIS, *The approximability of problems complete for P*, in Optimal Algorithms, International Symposium Proceedings, H. Djidjev, ed., Varna, Bulgaria, May–June 1989, pp. 193–204. Springer Lecture Notes in Computer Science 401.

[81] B. SHRIVER, *Foreword to [58]*.

[82] J. R. SMITH, *The Design and Analysis of Parallel Algorithms*, Oxford University Press, 1993.

[83] S. SWIERCZKOWSKI, *Sets and Numbers*, Routledge & Kegan Paul, London, UK, 1972.

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

[85] A. S. TANENBAUM, *Computer Networks*, Prentice Hall, Upper Saddle River, NJ, 3 ed., 1996.

[86] M. THORIN, *Real-Time Transaction Processing*, Macmillan, Hampshire, UK, 1992.

[87] J. L. TRAHAN, R. VAIDYANATHAN, AND C. P. SUBBARAMAN, *Constant time graph algorithms on the reconfigurable multiple bus machine*, Journal of Parallel and Distributed Computing, 46 (1997), pp. 1–14.

[88] 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.

[89] J. D. ULLMAN, A. V. AHO, AND J. E. HOPCROFT, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[90] USENET, *Comp.realtime: Frequently asked questions*, Version 3.4 (May 1998). http://-www.faqs.org/faqs/realtime-computing/faq/.

[91] S. V. VRBSKY, *A data model for approximate query processing of real-time databases*, Data and Knowledge Engineering, 21 (1997), pp. 79–102.

[92] S. V. VRBSKY AND S. TOMIĆ, *Satisfying timing constraints of real-time databases*, Journal of Systems and Software, 41 (1998), pp. 63–73.

[93] A. S. WAGNER, H. V. SREEKANTASWAMY, AND S. T. CHANSON, *Performance models for the processor farm paradigm*, IEEE Transactions on Parallel and Distributed Systems, 8 (1997), pp. 475–489.

[94] 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.

[95] ——, *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.

[96] WWW, *Processor farm*, in The Free On-Line Dictionary of Computing, http://-wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?farm.

[97] H. YAMADA, *Real-time computation and recursive functions not real-time computable*, IRE Transactions on Electronic Computers, EC-11 (1962), pp. 753–760.

Vita

Name: Stefan D. Bruda

Place and Year of Birth: Bucharest, Romania, 1971

Education:

- Polytechnic University of Bucharest, 1990–1996
  B.Sc. (honors, engineering) 1995; M.Sc. (computer science) 1996
- Queen's University at Kingston, 1997–2002
  Ph.D. 2002

Experience:

- Romanian Academy of Sciences (Bucharest, Romania), Center for Advanced Research in Machine Learning, Natural Language Processing and Conceptual Modeling
  Research Associate, 1995–1997
- Polytechnic University of Bucharest, Department of Computer Science
  Teaching Fellow, 1996–1997
- Queen's University, Department of Computing and Information Science
  Teaching Fellow, 2000–2002
- Queen's University, Department of Computing and Information Science
  Teaching and Research Assistant, 1997–2001

Awards:

- Governmental fellowship (Romania, 1991–1996)
- R. S. McLaughlin Fellowship (Queen's University, 1997–1998)
- Queen's Graduate Award (Queen's University, 1998–1999)

- CITO Award for Academic Excellence (May 1999)
- Ontario Graduate Scholarship (Province of Ontario, 1999–2002)

Publications:

- S. D. BRUDA AND M. CIOCOIU, *Generalized LR parser for unification based grammars*, in D. Tufis, ed., Language and Technology, Romanian Academy of Sciences, Bucharest, 1996, pp. 42–49 (in Romanian).
- S. D. BRUDA, *On the computational complexity of context-free parallel communicating grammar systems*, in G. Păun and A. Salomaa, eds., New Trends in Formal Languages, Springer Lecture Notes in Computer Science 1218, 1997, pp. 256–266.
- M. CIOCOIU AND S. D. BRUDA, *GULiveR: Generalized unification based LR parser for natural languages* in D. Tufis and P. Andersen, eds., Recent Advances in Romanian Language Technology, Romanian Academy of Sciences, 1997, pp. 38–51.
- D. TUFIS AND S. D. BRUDA, *Structure markup in CES and preliminary statistics on Romanian translation of Plato's "Republic"*, TELRI Newsletter, 5 (April 1997), pp. 23–27.
- S. D. BRUDA AND S. G. AKL, *On the data-accumulating paradigm*, in Proceedings of the Fourth International Conference on Computer Science and Informatics, Research Triangle Park, NC, October 1998, pp. 150–153.
- S. G. AKL AND S. D. BRUDA, *Parallel real-time optimization: Beyond speedup*, Parallel Processing Letters, 9 (1999), pp. 499–509.
- S. D. BRUDA AND S. G. AKL, *Towards a meaningful formal definition of real-time computations*, in Proceedings of the ISCA 15th International Conference on Computers and their Applications, New Orleans, LA, 2000, pp. 274–279.
- S. G. AKL AND S. D. BRUDA, *Parallel real-time cryptography: Beyond speedup II*, in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV, 2000, pp. 1283–1290.
- S. G. AKL AND S. D. BRUDA, *Parallel real-time numerical computation: Beyond speedup III*, International Journal of Computers and their Applications, 7 (2000), pp. 31–38.
- S. D. BRUDA AND S. G. AKL, *The characterization of data-accumulating algorithms*, Theory of Computing Systems, 33 (2000), pp. 85–96. (Preliminary version in Proceedings of the International Parallel Processing Symposium, San Juan, Puerto Rico, 1999, pp. 2–6.)
- S. D. BRUDA AND S. G. AKL. *A case study in real-time parallel computation: Correcting algorithms*, Journal of Parallel and Distributed Computing, 61 (2001), pp. 688–708.

- S. G. AKL AND S. D. BRUDA, *Improving a solution's quality through parallel processing*, Journal of Supercomputing 19 (2001), pp. 219–231.
- S. D. BRUDA AND S. G. AKL, *Parallel Real-Time Complexity: A Strong Infinite Hierarchy*, in Proceedings of VIII International Colloquium on Structural Information and Communication Complexity, Vall de Núria, Spain, June 2001, Carleton Scientific, pp. 45-59.
- S.D. BRUDA AND S.G. AKL, *Pursuit and evasion on a ring: An infinite hierarchy for parallel real-time systems (extended abstract)*, in Proceedings of the Thirteenth ACM Symposium on Parallel Algorithms and Architectures, July 2001, Crete Island, Greece, pp. 312–313.
- S. D. BRUDA AND S. G. AKL, *On the necessity of formal models for real-time parallel computations*, Parallel Processing Letters, 11 (2001), pp. 353–361.
- 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.
- S. D. BRUDA AND S. G. AKL, *The characterization of parallel real-time optimization problems*, to appear in Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications, June 2002, Moncton, NB.
- S. D. BRUDA AND S. G. AKL, *Real-time computation: A formal definition and its applications*, to appear in International Journal of Computers and Applications.

  Preliminary version in Proceedings of the Workshop on Advances in Parallel and Distributed Computational Models; in conjunction with the 15th Parallel and Distributed Processing Symposium, April 2001, San Francisco, CA. IEEE Computer Society Press, pp. [CD-ROM].
- S. D. BRUDA AND S. G. AKL, *On the Relation Between Parallel Real-Time Computations and Sublogarithmic Space*, Tech. Rep. 2001-446, Department of Computing and Information Science, Queen's University, Kingston, ON.