SIZE MATTERS: LOGARITHMIC SPACE IS REAL TIME

S. D. Bruda* and S. G. Akl†

Abstract

We show that all the problems solvable by a nondeterministic machine with logarithmic work space (NL) can be solved in real time by a parallel machine, no matter how tight the real-time constraints are. We also show that several other real-time problems are in effect solvable in nondeterministic logarithmic space once their real-time constraints are dropped and they become non-real-time. We thus conjecture that NL contains exactly all the problems that admit feasible real-time parallel algorithms. The issue of real-time optimization problems is also investigated. We identify the class of such problems that are solvable in real time. In the process, we determine the computational power of directed reconfigurable multiple bus machines (DRMBMs) with polynomially bounded resources and running in constant time, which is found to be the same as the power of directed reconfigurable networks with the same properties. We also show that write conflict resolution rules such as Priority or even Common do not add computational power over the Collision rule, and that a bus of width 1 (a wire) suffices for any constant time computation on DRMBM.

Key Words: real-time computation, timed ω-language, parallel complexity, reconfigurable multiple bus machine, independence system, matroid.

1 Introduction

One direction in the area of algorithms and complexity theory for real-time computations started by the introduction of *well-behaved timed ω-languages* [1]. Unlike previous models of real-time computation (e.g., [2]), timed languages bridge the long standing gap between the complexity theorists and the real-time systems community: While timed ω-languages create a formal model, they also capture all the features of real-time computations as understood by the systems community. A computation is thus deemed real time if the notion of correctness is linked not only to the output but also to the notion of time; in particular, the real-time qualifier is introduced by timing constraints on either input (that arrives at moments in time determined precisely by external factors) or output (which has associated deadlines, this being the most common type of real-time constraints). Our claim that timed ω-languages capture all the features of real-time computations as understood by the systems community is supported by the work from [1], where the formalism is used to model real-time computations encountered in practical areas. Real-time complexity classes, as well as complexity theoretic properties of real-time computations, are studied in [3]. However, the computations analyzed in [3] do not exhibit explicit deadlines; the real-time qualifier is given by the input (and its real-time characteristics). Still, most practical applications do require that computations are carried out within deadlines.

For this reason, our main focus in this paper consists in computations with explicit deadlines: We study (classical) languages that can be recognized in nondeterministic logarithmic space, augmented with real-time constraints (including deadlines). We show that all such computations can be carried out in parallel in a feasible manner (i.e., with

*Department of Computer Science, Bishop's University, 2600 College St, Sherbrooke, Quebec, J1M 0C8 Canada; email: bruda@cs.ubishops.ca
†School of Computing, Queen's University, Kingston, Ontario, K7L 3N6 Canada; email: akl@cs.queensu.ca

polynomially bounded resources), no matter how tight the time constraints are. Conversely, we show that, although hard to recognize in real time, the languages developed in [3] can be accepted in logarithmic space once the time constraints are eliminated. Thus, we conjecture that (nondeterministic) logarithmic space contains exactly all the computations that admit feasible real-time parallel implementations (again feasible means polynomial bounded resources, or $poly(n)$ processors in this particular case).

Motivated by this conjecture, we identify the class of optimization problems over independence systems that are solvable in real-time, and we are able thus to extend the results obtained in [4]. We show that the solution obtained by a parallel algorithm is arbitrarily better than the solution reported by a sequential one not only for the real-time minimum-weight spanning tree (as shown in [4]), but for any real-time maximization problem over a matroid for which the size of the optimal solution can be computed in real time.

In passing we offer a tight characterization of constant time computations on *reconfigurable multiple bus machines* (RMBMs). 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 as they do not add computational power over the easily implementable Collision rule. As far as constant time RMBM computations are concerned, we also find that a unitary bus width is enough—a simple wire as bus will do for all constant time RMBM computations.

The results in this paper are presented as follows: In Section 3 we show that exactly all NL languages can be recognized in constant time using a directed fusing RMBMs with $poly(n)$ processors and buses, each of width 1. Our main results are the subject of Section 4, where we establish that any NL language is computable in real time on RMBMs, no matter how tight the real-time restrictions actually are, and we state the aforementioned conjecture. The issue of optimization problems over independence systems is considered in Section 5. We conclude in Section 6.

2 Preliminaries

The cardinality of $\mathbb{N}$, the set of natural numbers, is denoted by ω. Given some $f : \mathbb{N} \rightarrow \mathbb{N}$, we denote by $\mathrm{DSPACE}(f(n))$ [$\mathrm{NSPACE}(f(n))$] the set of languages [5] 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. L [NL] is a shorthand for $\mathrm{DSPACE}(\log n)$ [$\mathrm{NSPACE}(\log n)$]. $GAP_{i,j}$ denotes be the following problem: given a directed graph $G = (V, E)$, $V = \{1, 2, ..., n\}$, determine whether vertex j is accessible from vertex i.

2.1 Timed ω-languages

A sequence $\tau = \tau_1\tau_2 \ldots \in \mathbb{N} \cup \omega$ is a *time sequence* if it is an infinite sequence of positive values, and $\tau_i \leq \tau_{i+1}$ for all $i > 0$. Any subsequence of a time sequence is a time sequence. A *well-behaved* time sequence is a time sequence for which, for every $t \in \mathbb{N}$, there exists some (finite) $i \geq 1$ such that $\tau_i > t$. A (well-behaved) *timed ω-word* over some alphabet Σ is a pair (σ, τ), where $\tau \in \mathbb{N}^k$ is a (well-behaved) time sequence, $k \in \mathbb{N} \cup \{\omega\}$, and $\sigma \in \Sigma^k$. A value τ_i from τ represents the time at which the respective σ_i becomes available as input. For some timed ω-word $w = (\sigma, \tau)$, $\mathrm{detime}(w) = \sigma$. By abuse of notation, $\mathrm{detime}(L) = \{\mathrm{detime}(w) | w \in L\}$ for a timed ω-language L.

The concatenation of two timed words is defined as the merging of their sequences of symbols, ordered in nondecreasing order of their arrival time. Two further, disambiguating restrictions are imposed: any sequence of symbols arriving simultaneously in one of the two words being concatenated appears as a subsequence of the result, and any symbol arriving at some time t in the second word being concatenating appears after all the symbols that arrive at time t in the first word. The timestamps of all the symbols in the result of a concatenation are the original timestamps of the respective symbols in the two words being concatenated. Given two timed ω-languages L_1 and L_2, the concatenation of L_1 and L_2 is $L_1L_2 = \{w_1w_2 | w_1 \in L_1, w_2 \in L_2\}$. The notation $\prod_{i=1}^{n} w_i$ [$\prod_{i=1}^{n} L_i$] is a shorthand for $w_1w_2 \cdots w_n$ [$L_1L_2 \ldots L_n$]. It has been argued (and well supported) [1] that well-behaved timed ω-languages model exactly all the real-time computations, so we henceforth assume this to be the case.

A *real-time algorithm* A consists in a *finite control*, *internal storage*, an *input tape* that always contains a timed ω-word, and an *output tape* containing symbols from some alphabet Δ. The input tape has the same semantics as a timed ω-word. During any time unit, A may add at most one symbol to the output tape. The content of the output tape of A working on w is denoted by $\mathcal{O}(A, w)$. There exists a designated symbol $f \in \Delta$. A real-time algorithm A *accepts* the timed ω-language L if, on any input w, $|\mathcal{O}(A, w)|_f = \omega$ iff $w \in L$ (where $|x|_a$ denotes as usual the number of occurrences of a in x). When a real-time algorithm is run in a multi-processor environment the finite state control is understood to encompass all the processors and their interconnections. Similarly the internal storage is the ensemble of all the storage available to individual processors. The input of the machine is the sequence of symbols from the input tape, that are made available to the machine at moments in time determined by the associated time sequence.

Let $w = (\sigma, \tau)$ be some timed ω-word. 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 $|w| = \max_{j>0} |s_j|$ (the maximum number of symbols arriving simultaneously). Given a total function $f : \mathbb{N} \to \mathbb{N}$, and some model of parallel computation M, the class rt-$\mathrm{PROC}^M(f)$ 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-$\mathrm{PROC}^M(1)$ of sequential real-time algorithms is invariant with M.

Pursuit and evasion on a ring The languages L_k, $k > 0$, modeling the k-dimensional version of the pursuit and evasion on a ring problem are developed in [3]. These languages are somehow a special case of real-time problems, as their real time qualifier is given by the input that arrives in real time, whereas no explicit deadline is imposed on the output. They also make for an interesting real-time computation as they form an infinite hierarchy with respect to the number of processors used to accept them. We shall give here a rather brief overview of these languages, directing the interested reader to [3].

Fix $k, p > 0$, $r > 2p$, $r' = kr$. Put $L_o = \{(\sigma, \tau) | \sigma \in \{a, b\}^{r'}, \tau_i = 0, 1 \leq i \leq r'\}$. Let $\mathbb{N}_k = \{enc(i) | 1 \leq i \leq k\}$, where enc is a suitable encoding function from $\mathbb{N}$ to $\{I\}^*$. Define $L_t = \{(u_h u_d u_c, \tau) \mid u_h \in \mathbb{N}_k, u_d \in \{+, -\}, u_c \in \{a, b\}^{j-1}, \tau_i = t \text{ for all } 1 \leq i \leq |u_h u_d u_c|\}$. Let $L_u = \prod_{i>0} L_{ci}$, for a given constant $c > 0$.

For some $w \in \{a, b\}^{r'}$, let $w = w(1)w(2) \ldots w(k)$, $|w(i)| = r$, $1 \leq i \leq k$ ($w(i)$ is a *segment* of w), and let $u = u_h u_d u_c = detime(x)$ for some $x \in L_t$, as above. The word u will be "inserted" into w at some position i and contains three components: u_h gives the segment of w in which the insertion takes place, u_d specifies whether

the insertion happens to the left or to the right, and u_c is the actual word to be inserted. Then the insertion has the effect of replacing symbols to the left or to the right (depending whether u_d is $-$ or $+$) of index i in the specified segment of w with the content of u_c in a circular fashion. The result of the operation is a pair (w', i'), where w' is the modified w and i' is the index immediately following the last modified index in w. Denote this operation by $(w, i) \otimes u$. 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, $0 \leq i_0 \leq r-1$, let $s(w,t) = \mathrm{fst}((\sigma^0, i_0) \bigotimes_{ci \leq t} \sigma^i)$, where $\sigma^i = \mathrm{detime}(w^i)$, $i \geq 0$.

Let A be an algorithm that considers $s(w,t)(j)$ and uses π processors, $\pi \geq 1$. A may inspect (i.e., read from memory) the symbols stored at some indices in $s(w,t)(j)$. Many processors may inspect different indices in parallel. For each processor q, let ι_t^q be the most recent index inspected by processor q up to time t. If some processor inspects no symbols from $s(w,t)(j)$, then $\iota_t^q = -1$. Let I_t^q be the "history" of inspected symbols up to time t, i.e., $I_t^q = \bigcup_{t' \leq t} \iota_{t'}^q \setminus \{-1\}$. Let $lo = \min_{1 \leq q \leq \pi}(\iota_t^q)$, $hi = \max_{1 \leq q \leq \pi}(\iota_t^q)$, and $I = \bigcup_{1 \leq q \leq \pi} I_t^q$. Then, we define $z(w(j), t)$ as follows: if $lo = -1$ then $z(w(j),t) = \{i | 0 \leq i < r\}$; if $lo \neq -1$ and there exists $j \notin I$, $j > hi$ or $j < lo$ then $z(w(j),t) = \{i | 0 \leq i < r, i \neq lo\}$; otherwise $z(w(j),t) = \{i | lo \leq i \leq hi\}$; if, at time t, some processor inspects an index outside $s(w(j),t)$, then $\iota_t^q(j) = -1$ and $I_t^q(j) = \emptyset$. Finally, let $z(w,t) = \bigcup_{j=1}^{k} z(w(j),t)$, and call $z(w,t)$ the acceptable insertion zone at time t.

With $z_i'(w)$ the set of indices whose values are modified by the subword $w^i \in L_{ci}$ of w, $L_k = \{w \in L_o L_u |$ for $i > 0$, $z_i'(w) \subseteq z'(w, ci)$, and there exists $t > 0$ and i_0, $0 \leq i_0 < r$, s. t. $|s'(w,t)|_a = |s'(w,t)|_b\}$.

In order to eliminate the ambiguity generated by the somehow generic notations used in [3], we shall denote henceforth L_k by $\textsc{Pursuit}_k$, for any $k > 0$.

2.2 Models with reconfigurable buses

Two main models with reconfigurable buses have been developed in the literature: the *reconfigurable network* (or RN for short) [6] and the *reconfigurable multiple bus machine* (or RMBM) [7].

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 [6, 7] at the same time—indeed, similar constructs are already present in VLSI circuits [8]. Note that their directed variants (defined below) are just as feasible, considering that the switches that connect buses to processors contain active components anyway [6, 8].

The reconfigurable multiple bus machine An RMBM [9, 7] consists of a set of p *processors* and b *buses*. For each processor i and bus b there exists a *switch* controlled by processor i. Using these switches, a processor has access to the buses by being able to read or write from/to any bus. A processor may be able to *segment* a bus, obtaining thus two independent, shorter buses, and it is allowed to *fuse* any number of buses together by using a *fuse line* perpendicular to and intersecting all the buses. DRMBM, the *directed* variant of RMBM, is identical to the undirected model, except for the definition of fuse lines: Each processor features two fuse lines (*down* and *up*). Each of these fuse lines can be electrically connected to any bus. Assume that, at some given moment, buses i_1, i_2, ..., i_k are all connected to the down [up] fuse line of some processor. Then, a signal placed on bus i_j is transmitted in one time unit to all the buses

i_l such that $i_l \geq i_j$ [$i_l \leq i_j$].If some RMBM [DRMBM] is not allowed to segment buses, then this restricted variant is denoted by F-RMBM [F-DRMBM] (for "fusing" RMBM/DRMBM). The *bus width* of some RMBM (DRMBM, etc.) denotes the maximum size of a word that may be placed (and read) on (from) any bus in one computational step.

For CRCW (concurrent read, concurrent write; as opposed to CREW for concurrent read, exclusive write) RMBMs, the most realistic conflict resolution rule is Collision, where two values simultaneously written on a bus result in the placement of a special, "collision" value on that bus. We consider for completeness other conflict resolution rules such as Common, Arbitrary, Priority, and Combining. However, we find that all of these rules are in fact equivalent to the seemingly less powerful Collision rule (see Theorem 3.5(3)). We restrict only the Combining mode, requiring that the combining operation be associative and computable in nondeterministic linear space.

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

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

Observation 1 $XY\mathrm{RMBM}(x(n), y(n), z(n)) \subseteq XY\mathrm{DRMBM}(x(n), y(n), z(n))$ for any $x, y, z : \mathbb{N} \to \mathbb{N}$, $X \in \{\mathrm{CRCW}, \mathrm{CREW}\}$, and $Y \in \{\text{F-}, \varepsilon\}$.

The reconfigurable network An RN [6] is a connected graph whose vertices are the 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. DRN, the *directed* RN 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 $\mathrm{RN}(s(n), t(n))$ [$\mathrm{DRN}(s(n), t(n))$] is the set of problems solvable by RN [DRN] uniform families with $s(n)$ processors ($s(n)$ is also called the *size complexity*) and $t(n)$ running time.

3 RMBM and NL computations

Lemma 3.1 $GAP_{1,n} \in \mathrm{CRCW\,F\text{-}DRMBM}((n^2 - n)/2, n, 2)$ *with Collision resolution rule and bus width* 1.

Proof. The following RMBM algorithm is a variant of the algorithm that computes the shortest path in a directed graph [10]. Denote each processor by p_{ij}, $1 \leq i < j \leq n$, and let p_{ij} know the value of both I_{ij} and I_{ji}, where I is

the incidence matrix. The algorithm works as follows: (a) Each p_{ij}, $1 \leq i < j \leq n$ directionally fuses buses i and j iff $I_{ij} = True$; simultaneously, p_{ij} fuses buses j and i iff $I_{ji} = True$. (b) Then, 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 (the original or the collision one), and $False$ otherwise. It is easily proved by induction over n that, for any s, t, $1 \leq s, t \leq n$, a signal placed on bus s reaches bus t iff vertex t is accessible from vertex s. The content of the emitted signal is immaterial, so a bus width 1 suffices. □

It is worth mentioning that the algorithm presented in [9] uses a CREW DRMBM (as opposed to the CRCW F-DRMBM used in Lemma 3.1). Furthermore, this algorithm computes the shortest path between two vertices. Therefore, it implicitly computes $GAP_{1,n}$. This lets us conclude that $GAP_{1,n} \in \text{CREW DRMBM}(2mn, n^2, O(1))$ (where m is the number of edges in the given graph). 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.

Consider now some language $L \in \mathsf{NL}$. There exists a nondeterministic Turing machine $M = (K, \Sigma, \delta, s_0)$ that accepts L and uses $O(\log n)$ working space. Without loss of generality, consider that the working (and input) alphabet of M is $\Sigma = \{0, 1\}$. Let k be the number of states of M, i.e., $k = |K|$. The transition function is denoted by δ, $\delta : (K \times \Sigma \times \Sigma) \rightarrow 2^{(K \cup \{h\}) \times (\Sigma \cup \{L,R\}) \times \{L,R\}}$ (with h the halting state), and the initial state by s_0. M accepts an input string x iff M halts on x. A *configuration* of M working on input x is defined as a tuple (s, i, w, j), where s is the state, i and j are the positions of the heads on input and working tape, respectively, and w is the content of the working tape. For two configurations v_1 and v_2, we write $v_1 \vdash v_2$ iff v_2 can be obtained by applying δ exactly once on v_1.

The set of possible configurations of M working on x forms a directed graph $G(M, x) = (V, E)$ as follows: V contains one vertex for each and every possible configuration of M working on x, and $(v_1, v_2) \in E$ iff $v_1 \vdash v_2$. It is clear that $x \in L$ iff some configuration (h, i_h, w_h, j_h) is accessible in $G(M, x)$ from the initial configuration (s_0, i_0, w_0, j_0). For any configuration (s, i, w, j), i can take $n = |x|$ values; since $|w| = O(\log n)$, there are at most $poly(n)$ possible contents of the working tape. There are thus $poly(n)$ possible configurations of M. For any language $L \in \mathsf{NL}$ and for any x, determining whether $x \in L$ can be reduced to the problem of computing $GAP_{1,|V|}$ for $G(M, x) = (V, E)$, where M is some nondeterministic, logarithmic space bounded Turing machine deciding L (we consider without loss of generality that the initial state is represented by vertex 1 and the final state by vertex n in $G(M, x)$).

Lemma 3.2 *Let $M = (K, \Sigma, \delta, s_0)$ be an* NL *Turing machine that accepts $L \in$ NL. Then, given some word x, $|x| = n$, there exists a* CREW *or* CRCW F-DRMBM *algorithm that computes $G(M, x)$ (as an incidence matrix I) in $O(1)$ time, and using $poly(n)$ processors and $poly(n)$ buses of width* 1.

Proof. Put $n' = |V|$ ($n' = poly(n)$). The RMBM algorithm uses $(n + (n'^2 - n')/2)$ processors: The first n processors p_i, $1 \leq i \leq n$, contain x, i.e., each p_i contains x_i, the i-th symbol of x; p_i does nothing but write x_i on bus i. We shall refer to the remaining $(n'^2 - n')/2$ processors as p_{ij}, $1 \leq i < j \leq n'$. Each p_{ij} assembles first

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

the configurations corresponding to vertices v_i and v_j of $G(M,x)$ and then 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, and $False$ otherwise. There is no interprocessor communication between processors p_{ij}, thus any RMBM model is able to carry on this computation.

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

Some comments on the RMBM algorithm developed in the proof of Lemma 3.2 are in order. One can note that the constant running time of this algorithm may be quite large ($\Theta(2^{4(k+1)})$; furthermore it depends on the number of states in the initial Turing machine). On the other hand, the subsequent use of Lemma 3.2 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 to the number of states is not a desirable feature. However, given some nondeterministic Turing machine $M=(K,\Sigma,\delta,s_0)$, one can build using standard manipulation of states an equivalent Turing machine $M'=(K',\Sigma',\delta',s_0)$ such that $|\delta'(s)|\leq 2$ for any $s\in K'$. One can now construct the algorithm A from Lemma 3.2 based on M' instead of M. Then, although $G(M,x)$ may grow (still, $|V|$ remains $poly(n)$), the running time of A is now upper bounded by a very small constant, which no longer depends on the number of states of M (or M' for that matter).

Lemma 3.3 $\mathsf{NL}\subseteq$ CRCW F-DRMBM$(poly(n),poly(n),O(1))$, *with Collision resolution rule and bus width* 1.

Proof. Given some language $L\in\mathsf{NL}$, let M be an NL Turing machine accepting L. For any input x, the F-DRMBM algorithm that accepts L works as follows: Using Lemma 3.2, 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 3.1 in order to determine whether vertex n (halting 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} stored at p_{ij} after the algorithm from Lemma 3.2 are in the right place as input for p_{ij} in the algorithm from Lemma 3.1. It is immediate given the mentioned lemmas that the resulting algorithm accepts L within the prescribed time and space bounds. □

Conforming to Lemma 3.3, any NL language can be accepted in constant time by a directed RMBM. In fact, the relation between directed RMBMs and NL languages is even stronger:

Lemma 3.4 CRCW DRMBM$(poly(n),poly(n),O(1))\subseteq\mathsf{NL}$, *for any write conflict resolution rule and any bus width.*

Proof. Consider some $R\in$ CRCW DRMBM$(poly(n),poly(n),O(1))$ performing step d of its computation ($d\leq O(1)$). We need to find an NL Turing machine M_d that generates the description of R after step d using $O(\log n)$ space, and thus [11] an NL Turing machine M'_d that receives n' (the number of processors in R) and some i, $1\leq i\leq n'$, and outputs the ($O(\log n)$ long) description for processor i instead of the whole description. We establish the existence of M_d (and thus M'_d) by induction over d, and thus we complete the proof; indeed, once we have the machines M_d, we start with M_0 (that receives the input of the algorithm) which in turn uses M'_0 to generate (as detailed below) and run repeatedly M'_1 and so on [11].

M_0 exists by the definition of a uniform RMBM family. We assume the existence of M_{d-1}, M'_{d-1} and show how M_d is constructed. For each processor p_i and each bus k read by p_i during step d, M_d performs (sequentially) the following computation: M_d maintains two words b and ρ, initially empty. For every p_j , $1 \leq j \leq poly(n)$, M_d determines whether p_j writes on bus k. This implies the computation of $GAP_{j,i}$ (clearly computable in nondeterministic $O(\log n)$ space since it is NL-complete [12]). The local configurations of fused and segmented buses at each processor (i.e., the edges of the graph for $GAP_{j,i}$) are obtained by calls to M'_{d-1}. The computation of $GAP_{j,i}$ is necessary to ensure that we take p_j into account even when p_j does not write directly to bus k but instead to another bus that reaches bus k through fused buses.

If p_j writes on bus k, then M_d uses M'_{d-1} to determine the value v written by p_j , and updates b and ρ as follows: (a) If b is empty, then it is set to v (p_j is currently the only processor that writes to bus k), and ρ is set to j. Otherwise: $(b.1)$ If R uses the Collision resolution rule, the collision signal is placed in b. $(b.2)$ When the Common rule is used, M_d compares b and v. If they are different, the input is rejected. $(b.3)$ 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. $(b.4)$ Finally, if R uses the Combining resolution rule with $\circ$ as combining operation, b is set to the result of $b \circ v$ (since the operation $\circ$ is associative, the final content of b is indeed the correct combination of all the values written on bus k).

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

Given Observation 1 and that $\mathsf{NL} = \mathrm{DRN}(poly(n), O(1))$ [6], Lemmas 3.3 and 3.4 imply the following results:

Theorem 3.5 *1.* $\mathrm{CRCW\,DRMBM}(poly(n), poly(n), O(1)) = \mathsf{NL} = \mathrm{CRCW\,F\text{-}DRMBM}(poly(n), poly(n), O(1))$ *with Collision resolution rule and bus width* 1.

2. $\mathrm{DRMBM}(poly(n), poly(n), O(1)) = \mathrm{DRN}(poly(n), O(1))$.

3. For any problem P solvable in constant time by some (directed or nondirected) RMBM family using $poly(n)$ processors and $poly(n)$ buses, $P \in \mathrm{CRCW\,F\text{-}DRMBM}(poly(n), poly(n), O(1))$ with Collision resolution rule and bus width 1.

Part of Theorem 3.5 is an expected result. Indeed, a similar result for DRNs exists [6], and it is known that (nondirected) RNs are as powerful as (nondirected) RMBMs [7] (and the two models using polynomially bounded resources solve in constant time exactly all the problems in L). It is thus expected that such properties hold for the directed variants of the two models (this time combined with nondeterministic Turing machines, as formally shown in Theorem 3.5). The other part of Theorem 3.5 on the other hand is very interesting: For constant time computations on DRMBM, bus width does not matter; any problem can be solved using buses of width 1. As is the case of (undirected) RMBMs, it follows from Theorem 3.5(3) that segmenting buses does not add computational power over fusing buses, and that the collision rule is the most powerful write conflict resolution method.

4 Small space computations are real-time

We have now all the necessary ingredients to state our main 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, so that any deadline encompasses a constant (and small) number of processor cycles. For instance a processor operating at 1GHz will not need to accommodate deadlines measured in single-digit nanoseconds (but will cope well with say 100 nanoseconds deadlines). We believe that this is a reasonable assumption, for indeed we are not aware of any real-time application that requires such exceedingly small deadlines. In addition, such an assumption is not even necessary throughout the paper, but only for Theorem 4.1 and the subsequent Claim 1.

Note now that the potential existence of a *deadline* can be modeled as a well-behaved timed ω-word [1] by $W_d = (\sigma, \tau)$, with the following semantics: A special symbol ν is present whenever the current time does not exceed the deadline; if the deadline passed, then the symbols that arrive as input are all another designated symbol χ. If the computation is completed at a moment in which the input symbol is ν, then it has met the associated deadline; otherwise, the deadline has passed.

With this definition of W_d, we have the following relation linking NL with real-time computations.

Theorem 4.1 *Consider the timed ω-word $w = (\sigma, \tau)W_d$ with W_d some timed word modeling a deadline, σ some input for some problem $P \in$ NL, and $\tau_1 = \tau_2 = \cdots = \tau_{|\sigma|}$. Then $w \in$ rt-$\mathrm{PROC}^{\mathrm{CRCW\ F\text{-}DRMBM}}(poly(|\sigma|))$ for any w thus constructed.*

Proof. We note that the size complexity of an RMBM with $poly(n)$ processors and $poly(n)$ buses is $poly(n)$. Then all the processing implied by Theorem 3.5 takes constant time, and thus accommodates any reasonable time sequence τ associated with the computation. □

The relation between NL and real-time computations can be informally stated as follows: Suppose we have an input for a problem in NL. We impose some (any) deadline for this input, and we feed it to some machine. If that machine happens to be a CRCW F-DRMBM, then it is able to produce the results while meeting the respective deadline.

In some sense, one may argue that the inclusion relation from Theorem 4.1 is in fact an equality, conforming to Theorem 3.5. Indeed, NL 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), no computation outside NL can be successfully carried out. However, there might exist real-time computations (for example, not exhibiting explicit deadlines and thus not necessarily having constant time constraints) that are not in NL but can still be performed within the given resource bounds. Indeed, one candidate for such computations can be the family of timed ω-languages $\textsc{Pursuit}_k$, $k \geq 1$, presented in [3] and summarized in Section 2.1. We shall try to see what is the classical computation corresponding to this problem.

In Theorem 4.1, we *added* deadlines (that is, real-time constraints) to problems. We face now the reversed problem, namely how can one *eliminate* the real-time qualifier from the specification of some problem. Analyzing the form of

the word W_d 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 consist of symbols ν and χ that model the timing constraints imposed on the computation. Similarly, in a real-time problem for which the input is infinite, 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 4.1 Consider some well-behaved timed ω-language L; $i > 0$ is a *progression point* for $(\sigma, \tau) \in L$ iff[2] $\tau_i \neq \tau_{i+1}$. Let now $L_s = \{\sigma' |$ there exists some (finite) progression point n s.t. $(\sigma, \tau) \in L$ and $\sigma' = \sigma_{1...n}\}$ (each word in L_s is constructed by taking a word from L, restricting its length to a finite n, and discarding the time sequence). If $L_s \in C$ for some complexity class C, we say that $L \in C/rt$. L_s solves the same problem as L, but without real-time constraints; we thus say that L is the *real-time counterpart* of L_s, or L_s is the *static version* of L.

Note in passing that Definition 4.1 not only allows us to study the pursuit problem in the context of Theorem 4.1, but it offers a more concise formulation of Theorem 4.1 itself:

Theorem 4.2 $\mathsf{NL}/rt \subseteq \text{rt-PROC}^{\text{CRCW F-DRMBM}}(poly(n))$.

We now show that pursuing something is easy outside the real-time paradigm:

Theorem 4.3 *For any $k > 0$,* $\text{PURSUIT}_k \in \mathsf{L}/rt$.

Proof. A word w_s, $|w_s| = n$, in the static version of PURSUIT_k contains: (a) an initial word $w^0 \in \{a, b\}^r$, $r \leq n$ (the initial configuration), and (b) m moves by the pursuee (denoted by $w^i \in L_{ci}$, $1 \leq i \leq m$; each such a move changes a maximum of $p < r$ symbols from w^0).

Let M be a deterministic Turing machine accepting the static version of PURSUIT_k. M keeps two counters C_a and C_b, one for a's and the other for b's. As w^0 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. Finally, M identifies that portion of w^i that changes w^0 and scans it, incrementing C_a and/or C_b accordingly. 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 that are present in w^0 as it is changed by all w^i. When the end of the input is reached, M compares C_a and C_b and accepts the input iff they are identical. All the counters (C_a, C_b, two more pair of counters needed to delimit the current w^i and the portion of interest in w^0) clearly take $O(\log n)$ space. Manipulating these counters involves simple arithmetic operations on indices (that is, numbers bounded above by n), hence they are computable in L. The space required by the whole computation is $O(\log n)$. □

Theorem 4.3 is an interesting result: even if PURSUIT_k requires a lot of computational effort (in particular, it cannot be solved at all if less than $2k$ processors are available [3]), it becomes a very simple problem (not only in NL, but even in L) once the real-time constraints are eliminated. Thus, Theorem 4.3 justifies the following conjecture:

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

[2] 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...

algorithm GREEDYMAX $(E, ind; s_g)$
1. let $(e_1, e_2, \ldots, e_n)$ be an ordering of E with $c(e_i) \geq c(e_{i+1})$
2. $s_g \leftarrow \emptyset$
3. **for** $i \leftarrow 1 \ldots n$ **do**
3.1. **if** $ind(s_g \cup \{e_i\})$ **then** $s_g \leftarrow s_g \cup \{e_i\}$

(a)

algorithm PARALLELGREEDYMAX $(E, ind; s_g)$
1. sort E, obtaining $(e_1, e_2, \ldots, e_n)$ s.t. $c(e_i) \geq c(e_{i+1})$
2. $s_g \leftarrow \emptyset$; $r_0 \leftarrow 0$
3. **for** $i \leftarrow 1 \ldots n$ **do in parallel**
3.1. $r_i \leftarrow ur\{e_1, e_2, \ldots, e_i\}$
3.2. **if** $r_{i-1} < r_i$ **then** $s_g \leftarrow s_g \cup \{e_i\}$

(b)

Figure 1: Greedy algorithms for maximization problems.

5 Independence systems and real-time computation

We focus our attention now to optimization problems. In this context, we identify the class of such problems that can be computed in parallel real time. Based on this identification, we extend previous results [4].

$S \subseteq 2^E$ is a set (of *feasible solutions*) with elements from a finite set E. A mapping $c : E \rightarrow \mathbb{R}$ is defined, and then extended to $c : S \rightarrow \mathbb{R}$, $c(s) = \sum_{i \in s} c(i)$. Consider the *optimization (maximization) problem* over S of finding $\max\{c(s) | s \in S\}$. Without loss of generality let $c(i) \geq 0$, for all $i \in E$. The set of optimal solutions to the problem is thus not changed if one replaces S by its *hereditary* closure S^* defined as $S^* = S \cup \{s | s \subseteq s', s' \in S\}$. (E, S^*) is an *independence system*. The (algorithmic) input for a search problem can be considered in multiple ways [13]. In particular, the set S can be quite large and is thus impractical to give explicitly as input; so we consider instead that the input for a search problem is the set E (together with the associated weights), plus an "independence oracle" ind such that $ind(s) = true$ iff $s \in S$. The transition from an optimization problem to an equivalent language is standard [11], so we say by abuse of notation that a certain optimization problem is or is not in NL.

Definition 5.1 [14] Let E be a finite set and $S \subseteq 2^E$, 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: $lr(F) = \min\{|s| | s \in S; s \subseteq F \text{ and } s \cup \{e\} \notin S \text{ for all } e \in F \setminus \{s\}\}$; $ur(F) = \max\{|s| | s \in S; s \subseteq F\}$.

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

A *greedy algorithm* for maximization problems on general independence systems [14] is given in Fig. 1(a). The algorithm contains one statement which depends on the actual independence system being considered (the condition on line 3.1).

Let (E, S) be an arbitrary independence system, and consider a maximization problem with s_g the solution returned by GREEDYMAX, and s^* the optimal solution. Then [14] for any weight function $c : E \rightarrow \mathbb{R}^+$, $\min_{F \subseteq E} \frac{lr(F)}{ur(F)} \leq \frac{c(s_g)}{c(s^*)} \leq 1$. It follows that algorithm GREEDYMAX on a matroid (E, S) yields the optimal solution for a maximization problem for all objective functions $c : E \rightarrow \mathbb{R}^+$.

5.1 A real-time perspective

To put the definition of matroids in another way [15, 11], 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, shown in Fig. 1(b), is immediate [11]. The algorithm uses a *rank oracle*: The function $ur\{e_1, e_2, \ldots, e_i\}$ introduced by Definition 5.1 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 5.1 *Suppose* $ur\{e_1, e_2, \ldots, e_i\} \in \text{DRMBM}(poly(i), poly(i), t(i))$ *(i.e.,* $ur\{e_1, e_2, \ldots, e_i\}$ *can be computed by a DRMBM in time* $t(i)$ *using a polynomially bounded number of processors and buses). Then,* PARALLELGREEDYMAX $\in \text{DRMBM}(poly(n), poly(n), O(t(n)))$.

In particular, if $t(i) = O(1)$*, then* PARALLELGREEDYMAX $\in \text{DRMBM}(poly(n), poly(n), O(1))$.

Proof. The initial sorting (step 1) can be achieved in constant time on a DRMBM with polynomially bounded resources [16] and thus in constant time on a DRMBM using $poly(n)$ processors and buses by Theorem 3.5(3). Steps 2 and 3.2 are trivially computable in constant time with polynomially bounded resources. Each of the calls to ur in step 3.1 can be performed in $t(n)$ time by using n independent copies of the RMBM computing ur. 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 $\in \text{DRMBM}(poly(n), poly(n), O(t(n)))$, as desired. It is then immediate that PARALLELGREEDYMAX $\in \text{DRMBM}(poly(n), poly(n), O(1))$ if $t(i) = O(1)$. □

Lemma 5.2 *Let* (E, S) *be some independence system,* $E = \{e_1, e_2, \ldots, e_n\}$*, and let* A *be an algorithm solving a maximization problem 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 $ur(E) = k$. 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 processor p_i 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 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 [9]. Therefore, $t_{ur}(n) = O(t_A(n))$ (or $t_A(n) = \Omega(t_{ur}(n))$), as desired. □

Theorem 5.3 *Let* $\mathcal{M}$ *be the class of maximization problems that can be described as a matroid and for which* $ur \in \text{DRMBM}(poly(i), poly(i), O(1))$*. Let* P *be some maximization problem over some independence sys-*

tem (E, S). *Then* $P \in \mathcal{M}$ *iff* $P \in \mathrm{DRMBM}(poly(n), poly(n), O(1))$ *(equivalent in turn to* $P \in \mathsf{NL}$ *and* $\{P\}/rt \subseteq \text{rt-PROC}^{\text{CRCW F-DRMBM}}(poly(n))$*)*

Proof. In light of Lemmas 5.1 and 5.2 the only thing that needs further consideration is showing that $P \in \mathrm{DRMBM}(poly(n), poly(n), O(1))$ implies that (E, S) is a matroid. This is however a direct consequence of the lower bounds on optimization problems over independence systems that are not matroids [11, 13], which place these problems outside NL (and thus outside $\mathrm{DRMBM}(poly(n), poly(n), O(1))$). □

By Theorem 5.3 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: 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 5.3). 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 NL. On the other hand, Theorem 5.3 easily extends previous results, as we shall show in what follows.

5.2 Beyond speedup, revisited

The problem of computing the *minimum-weight spanning tree* (MST) of a connected, undirected, and weighted graph in real time is investigated in [4], where it is shown that the best approximate solution to the MST problem returned by a sequential algorithm can be arbitrarily worse than the solution obtained by a parallel algorithm (which actually returns the optimal solution). We shall not, however restrict ourselves to connected graphs, since the extension to unconnected ones (when the tree becomes a forest) is immediate.

MST can be trivially transformed into a maximization problem: just negate all the edge weights and add to every weight the absolute value of the maximum weight. It is also immediate that the MST problem can be expressed as a matroid [15]. Thus, using Theorem 5.3 we can both tighten and extend the result from [4].

For one thing, the result in [4] is not tight: Time up to n^{ϵ}, for some $0 < \epsilon < 1$, is allowed for each (parallel or sequential) real-time computation leading to the result. This running time asymptotically exceeds any (however large) constant deadline imposed to the computation by some real-time environment, so the settings used in [4] are too permissive for our environment. Fortunately, we are able to obtain precisely the same result for true real-time computations. 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 5.3, is given in Lemma 5.4 below.

Lemma 5.4 *Let MST denote the problem of computing the minimum-weight spanning forest on undirected and weighted graphs. Then,* $\mathrm{MST} \in \mathrm{DRMBM}(poly(n), poly(n), O(1))$ *(and thus* $\mathrm{MST} \in \mathsf{NL}$*,* $\{\mathrm{MST}\}/rt \subseteq$

rt-$\mathrm{PROC}^{\mathrm{CRCW\ F\text{-}DRMBM}}(poly(n))$*, and the best approximate solution to any problem from* $\{\mathrm{MST}\}/rt$ *returned by a sequential 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 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 NL-complete problem [12]). By Theorem 5.3, it follows that MST can be computed *exactly* in real time on an RMBM, no matter how tight the deadlines are. However [4], an optimal sequential algorithm solving 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; the guess can be arbitrarily bad. □

In fact, the second part of the proof of Lemma 5.4 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, the proof of Lemma 5.4 requires that ur is NL (clearly applicable to the whole class $\mathcal{M}$), that the underlying independence system is a matroid (ditto), plus the result from [4] (which immediately holds for any problem that does not admit a sequential algorithm with constant running time [4]; however no optimization problem can be solved in constant sequential time since at the very least the weights of all the elements of E need to be inspected). Therefore:

Corollary 5.5 *With* $\mathcal{M}$ *as in Theorem 5.3 and for any* $P \in \mathcal{M}$*, the best approximate solution to a problem in* $\{P\}/rt$ *returned by a sequential algorithm is arbitrarily worse than the solution obtained by a parallel RMBM algorithm with polynomially bounded resources.*

In other words, the results from [4] do hold even for the tightest real time environment. In addition, these results are not applicable only to the MST, but to a whole class of problems instead, namely $\mathcal{M}$ from Theorem 5.3. That is, there exists not only a problem, but a whole family of them for which a parallel implementation can do something other than speeding up computation, namely improve the offered solution.

6 Conclusions

We addressed previously a number of questions associated with real-time computations featuring implicit deadlines [3]. In this paper, we focused our attention on computations with explicit deadlines. Given any language in NL, we showed in Theorem 4.1 that such a language can be accepted by a parallel machine with polynomially bounded resources, in the presence of *any* (i.e., however tight) real-time constraints.

According to Theorem 4.3, even a language like $\textsc{Pursuit}_k$, whose acceptance requires considerable computational effort, can be accepted in logarithmic space once the real-time constraints are dropped. This allows us to state Claim 1, which offers a nice counterpart of the parallel computation thesis [11]. In this thesis, NC is conjectured to contain exactly all the computations that admit efficient parallel implementations; by contrast, we conjecture that NL contains exactly all the computations that admit efficient real-time parallel implementations.

As well, we considered the class of maximization problems over independence systems, showing that a problem pertaining to this class is solvable in real time iff it is a matroid and the size of an optimal solution is computable in real-time. Given this result, we showed that there exists not only a problem but a whole family of them for which a parallel implementation can do something other than speeding up computation, namely unboundedly improve the offered solution.

In light of Claim 1, 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 paper 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 NL 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 3.5.

We also determined the computational power of DRMBM running in constant time. We showed that DRMBM and DRN with constant running time have the same computational power. In addition, we showed that no conflict resolution rule is more powerful than Collision. According to this result, the discussion regarding the practical feasibility of rules like Priority or Combining on spatially distributed resources such as a buses is no longer of interest. Indeed, such rules are not only of questionable feasibility, but not necessary too. 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.

Another interesting open problem naturally arises from the characterization described in the above paragraph: does a form of Theorem 3.5(3) hold for other models of parallel computations? On one hand, we showed that unrealistic rules like Priority and Combining do not add computational power. However, this result is obtained for the restricted class of DRMBMs running in constant time. Thus, we wonder whether such a result holds for (a) DRMBMs in general, not only those with constant running time, and (b) for other models of parallel computation (RN, PRAM, etc.). On the other hand, we wonder whether the bus width can be bounded for DRNs running in constant time as it has been bounded in the case of DRMBMs. In other words, can the bus width in a DRN be bounded by a constant?

Acknowledgement

This research was supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] Stefan D. Bruda and Selim G. Akl. Real-time computation: A formal definition and its applications. *International Journal of Computers and Applications*, 25(4):247–257, 2003.

[2] Hisao Yamada. Real-time computation and recursive functions not real-time computable. *IRE Transactions on Electronic Computers*, EC-11:753–760, 1962.

[3] Stefan D. Bruda and Selim G. Akl. Pursuit and evasion on a ring: An infinite hierarchy for parallel real-time systems. *Theory of Computing Systems*, 34(6):565–576, 2001.

[4] Selim G. Akl and Stefan D. Bruda. Parallel real-time optimization: Beyond speedup. *Parallel Processing Letters*, 9:499–509, 1999.

[5] Eric Allender, Michael C. Loui, and Kenneth W. Regan. Complexity classes. In Mikhail J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, pages 27–1–27–23. CRC Press LLC, 1999.

[6] Y. Ben-Asher, K.-J. Lange, D. Peleg, and A. Schuster. The complexity of reconfiguring network models. *Information and Computation*, 121:41–58, 1995.

[7] Jerry L. Trahan, Ramachandran Vaidyanathan, and Ratnapuri K. Thiruchelvan. On the power of segmenting and fusing buses. *Journal of Parallel and Distributed Computing*, 34:82–94, 1996.

[8] J. P. Gray and T. A. Kean. Configurable hardware: A new paradigm for computation. In C. L. Seitz, editor, *Proceedings of the Tenth Carltech Conference on VLSI*, pages 279–295, Cambridge, MA, March 1989. MIT Press.

[9] Jerry L. Trahan, Ramachandran Vaidyanathan, and Chittur P. Subbaraman. Constant time graph algorithms on the reconfigurable multiple bus machine. *Journal of Parallel and Distributed Computing*, 46:1–14, 1997.

[10] Naya Nagy. The maximum flow problem: A real-time approach. Master's thesis, Department of Computing and Information Science, Queen's University, January 2001.

[11] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, New York, NY, 1995.

[12] Andrzej Szepietowski. *Turing Machines with Sublogarithmic Space*. Springer Lecture Notes in Computer Science 843, 1994.

[13] R. M. Karp, E. Upfal, and A. Wigderson. Are search and decision programs computationally equivalent? In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, pages 464–475. ACM Press, December 1985.

[14] Ravi Kannan and Bernhard Korte. Approximative combinatorial algorithms. In Richard W. Cottle, Milton L. Kelmanson, and Bernard Korte, editors, *Mathematical Programming*, pages 195–248. Elsevier Science Publishers, Amsterdam, The Nederlands, 1981.

[15] Thomas H. Cormen, Charles E. Leiserson, and Clifford Stein. *Introduction to Algorithms*. MIT press, Cambridge, MA, 2 edition, 2001.

[16] Selim G. Akl. *Parallel Computation: Models and Methods*. Prentice-Hall, Upper Saddle River, NJ, 1997.