Technical Report No. 2000-438

Pursuit and Evasion on a Ring: An Infinite Hierarchy for Parallel Real-Time Systems*

Stefan D. Bruda and Selim G. Akl
Department of Computing and Information Science
Queen's University
Kingston, Ontario, K7L 3N6 Canada
Email: {bruda,akl}@cs.queensu.ca

September 4, 2000

Abstract

We present a new complexity theoretic approach to real-time parallel computations. Based on the theory of timed ω-languages, we define complexity classes that capture the intuitive notion of resource requirements for real-time computations in a parallel environment. Then, we show that, for any positive integer n, there exists at least one timed ω-language L_n which is accepted by a $2n$-processor real-time algorithm using arbitrarily slow processors, but cannot be accepted by a $(2n-1)$-processor real-time algorithm. It follows therefore that real-time algorithms form an infinite hierarchy with respect to the number of processors used. Furthermore, such a result holds for any model of parallel computation.

1 Introduction

Consider the following question:

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

Although 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. Furthermore, 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 theoretical point of view (specifically, from the point of view of parallel real-time complexity theory).

There are noteworthy results in the area of real-time parallel computation [3, 4, 5, 6, 7, 11, 12, 22], but all of them make the assumption of equal computational speed in the parallel and sequential cases, and thus none of them appear to properly address question 1. The closest to an answer for this question is probably a result concerning real-time Turing machines, conforming to which a k-tape real-time Turing machine is strictly more powerful that a $(k-1)$-tape one [1, 13]. However, the model of choice here creates another problem. Indeed, for one thing, Turing machines appear not to be an expressive enough model for either sequential or parallel real-time computations. We shall elaborate on this idea shortly. On the other hand, this time from a parallel point of view, it is not clear that the concept of tape in a Turing machine can

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

be equated with the concept of processor in a parallel algorithm. Indeed, the Turing machine is essentially a sequential model, because of its finite control, even if elementary processes can be carried in parallel on multiple tapes. In conclusion, although a result with respect to Turing machines is interesting, we believe that a truly significant result should be based on a more realistic model of real-time parallel computation.

As for the choice of such a model, there are some formal models for real-time computations, but many of them fail to take into consideration aspects that are important in practice. Specifically, the already mentioned *real-time Turing machine* [27] (which is probably the oldest and the most widely studied such a model) cannot express specific deadlines, such as "this computation should take less than 4 seconds." As well, the *real-time producer/consumer paradigm* [19] can express neither real-time events that occur acyclically, nor variable arrival rates for the input. By contrast, the *ω-regular languages* [8] seem to be particularly well suited for modeling real-time problems. Nonetheless, the device used for the recognition of such languages is not sufficiently powerful to take into account all the real-time applications. Indeed, it is easy to see that there are real-time problems that cannot be modeled by ω-regular languages. Finally, *timed ω-languages* appear to be expressive enough in order to capture all the practically important aspects of real-time computations. This model was proposed in [14], and its expressiveness is illustrated in [15], where various aspects of practical importance are modeled using this formalism.

An intuitive aside.

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

> 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, such that each of C and D moves at $1/x$ the speed of A (and B), $x > 1$, then C and D *always* catch B.

This modified version of the pursuit/evasion problem was mentioned for the first time in [13].

Starting from this intuition, we lay in this paper the basis for a parallel real-time complexity theory. Specifically, we start by defining the underlying notions for such a theory, in particular real-time complexity classes, and a notion of input size suitable for the real-time domain. Then, we construct a timed ω-language that models the geometric problem presented above, and we extend this language as to model an "n-dimensional circle," $n \geq 1$, 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 solving real-time problems is infinite. To our knowledge, this is the first time such a result is obtained. Finally, we show that this result can be extended to any model of parallel computation, that is, the infiniteness of the parallel real-time hierarchy is invariant with the model of parallel computation involved.

We organize this paper as follows: We start by presenting in section 2 a review of the theory of timed ω-languages, summarizing at the same time the notations used throughout the paper. Then, in section 3, we outline the basis of our complexity theory. Specifically, we identify what we believe to be the most important complexity classes for real-time computations, and outline at the same time a general way of describing such classes. A natural notion of input size is also presented in section 3. The main result of this paper (that is, the infiniteness of the parallel real-time hierarchy) is presented in section 4. We conclude in section 5, with a discussion and some open problems.

2 Preliminaries

Given some finite alphabet Σ, the set of all words (that is, sequences of symbols) of finite (but not necessarily bounded) length over Σ is denoted by Σ^*. The cardinality of $\mathbb{N}$, the set of natural numbers, is denoted by

ω. It should be noted[1] that $\omega \notin \mathbb{N}$ [16]. Then, the set Σ^ω contains exactly all the words over Σ of length ω. The length of a word σ is denoted by $|\sigma|$. Given a set $A \subseteq \Sigma$, $|\sigma|_A$ denotes the length of σ after all the symbols that are not in A have been erased. By abuse of notation, we often write $|\sigma|_a$ instead of $|\sigma|_{\{a\}}$. The set Σ^k is defined recursively by $\Sigma^1 = \Sigma$, and $\Sigma^i = \Sigma \times \Sigma^{i-1}$ for $i > 1$. By convention, $\Sigma^0 = \{\lambda\}$, where λ denotes the empty word.

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 $\sigma' \subseteq \sigma$) iff (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 $i > k$ iff $j > l$.

The following summary conforms to [14], with some additions that make the subsequent presentation clearer. 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 every $t \in \mathbb{N}$, there exists some finite $i \geq 1$ such that $\tau_i > t$. In should be noted that a time sequence may be finite or infinite, while a well-behaved time sequence is always infinite.

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.

Timed ω-languages were defined in [14], as an extension of timed ω-regular languages presented in [8]. In [14], a timed ω-word is defined as a tuple with two components: an infinite sequence of symbols, and a well-behaved time sequence. Then, the finite variant is considered as well. This appeared to be a reasonable notation at the time, since it is our thesis [14] that such languages model all the real-time computations. On the other hand, those timed words whose time sequence is not well-behaved were not given a name in [14], except for the finite variant. However, it turns out that, even if these words by themselves do not model real-time computations, they may be useful as intermediate tools in building real-time models. Therefore, we slightly changed the terminology in this paper. What was called in [14] a timed ω-language, we now call a well-behaved timed ω-language, while the absence of the "well-behaved" qualifier denotes those timed words whose time sequence does not necessarily respect the progress condition, including finite timed words. As a consequence, the thesis formulated in [14] is now expressed as follows in the terminology of this paper: well-behaved timed ω-languages model all real-time computations.

As another difference from the approach used in [14], we consider time to be discrete, since in essence the time perceived by a computer is discrete as well. Furthermore, one can define a granularity of time as fine as desired.

In the following, 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 an *output tape* (that is, an *output 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 output tape. Furthermore, the output tape is *write-only*, that is, A cannot read any symbol previously written on the output tape. The content of the output tape of some real-time algorithm A working on some input w is denoted by $o(A, w)$. There exists some designated symbol $f \in \Delta$. In addition, A may have access to an infinite amount of working storage space (working tape(s), RAM memory, etc.) outside the input and output 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, we 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/output tapes.

[1] Also note that the cardinality of $\mathbb{N}$ is denoted by either ω [16] or $\aleph_0$ [23]. We chose the first variant in order to be consistent with the notation used in [8].

The designated symbol f from Δ has the same meaning as the final state used in [8]. There, only those timed ω-languages accepted by finite automata are considered. Such an automaton has a designated state f, and the current input is accepted by the automaton iff the state f is entered infinitely many times during the accepting process. The same approach is used in [14] for more complex machines. However, since the sate of a general machine may be hard to work with, we prefer to change the state with output. That is, a real-time algorithm accepts some word iff some designated symbol f appears infinitely many times on the output tape. Formally, a real-time algorithm A *accepts* the timed ω-language L if, on any input w, $|o(A,w)|_f = \omega$ iff $w \in L$. One can note that the models that were constructed in [14, 15] can be trivially converted to this variant of acceptor.

It is worth mentioning that the actual meaning of the symbol f on the output tape might be different from algorithm to algorithm. However, 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 keeps writing f on the output tape. That is, the first appearance of f 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 f). 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, f might appear on the output tape each time an occurrence of the query is successfully served (obviously, a failure could prevent further occurrences of f, should the specification of the problem require that all the queries be served). In this case, each occurrence of f signals a successfully served query. However, even if the actual meaning of the f's on the output 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.

Finally, we assume 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. Furthermore, 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, $t < \tau_i$.

2.1 Operations on timed 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.

Furthermore, one can rely on the semantics of timed words in defining a meaningful concatenation operation. More precisely, 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:

Definition 2.1 **[14]** 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 $(\sigma,\tau) = (\sigma',\tau')(\sigma'',\tau'')$, iff

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, $i \leq k < l \leq j$, there exists m such that, for any $0 \leq \iota \leq j-i$, $(\sigma_{m+\iota},\tau_{m+\iota}) = (\sigma^d_{i+\iota},\tau^d_{i+\iota})$, 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 $L = \{w_1 w_2 | w_1 \in L_1, w_2 \in L_2\}$. $\Box$

In addition to the mentioned order of the resulting sequence of symbols (formalized in item 1 of definition 2.1), two more constraints are imposed in definition 2.1. 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. 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.

Given n (timed ω-)words $w_1, w_2, \ldots, w_n$, $n > 1$, the notation

$$w = \prod_{i=1}^{n} w_i$$

is a shorthand for $w = w_1 w_2 \cdots w_n$ (that is, w is the concatenation of all the words w_i, $i > 0$). By extension, the (timed ω-)language L obtained by concatenating n (timed ω-)languages $L_1, L_2, \ldots, L_n$ is denoted by

$$L = \prod_{i=1}^{n} L_i.$$

The reason for using $\prod$ as the operator for concatenation is that such an operation on languages is similar to the cross product on sets. Furthermore, language concatenation is sometimes called *product* [18]. One should also note that the concatenation of timed ω-words as defined by definition 2.1 is associative.

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

Definition 2.2 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 $L^* = \cup_{0 \leq k < \omega} L^k$. We call L^* the Kleene closure of L. $\Box$

The following result is immediate:

Theorem 2.1 *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 will make use of the following two operations: Given some timed ω-word w, $w = (\sigma, \tau)$, let $\text{detime}(w) = \sigma$ and $\text{time}(w) = \tau$.

3 Sizing up real-time computations

3.1 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. Given some function $f : \mathbb{N} \to \mathbb{N}$, the class TIME($f$) denotes those algorithms working on input of size n and whose running time is upper bounded by $f(n)$. From this definition, more general classes can be easily derived, for example PTIME (polynomial running time). On the other hand, in the real-time area, time is in most cases predetermined by the existence of deadlines imposed to the computation or by other similar time constraints. Admittedly, there are classes of real-time algorithms for which running time actually makes sense as a measure of performance. Furthermore, one may 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. For this reason, we will not consider real-time classes with respect to running time.

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. With the same notations as in the above paragraph, an algorithm from the class SPACE(f) uses an effective storage space upper bounded by $f(n)$ when working on an input of length n. We therefore introduce the corresponding classes $rt-\mathrm{SPACE}(f)$, with its immediate extensions, such as $rt-\mathrm{LOGSPACE}$, $rt-\mathrm{PSPACE}$, etc. In general, we prefix all the real-time complexity classes by "$rt-$" (from "real-time"), in order to avoid any possible confusion.

A third measure of interest is the *number of processors*. In classical complexity theory this measure received less attention than the other measures. However, in the real-time paradigm, a parallel algorithm has been shown to make up for the limited time that is available, and solve problems that are not solvable by a sequential implementation [3, 11, 12, 22]. Therefore, we will consider the classes $rt-\mathrm{PROC}(f)$. An algorithm pertaining to such a class solves the given problem using at most $f(n)$ processors on an input of size n.

However, another issue is worth considering with respect to the number of processors. Indeed, consider the Parallel Random Access Machine (PRAM) model, as opposed to, say, some interconnection network [2]. 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. Except for a complete network, 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 classes $rt-\mathrm{PROC}(f)$ are different from model to model. If this is the case, given some model of parallel computation M, we denote the corresponding class by $rt-\mathrm{PROC}^M(f)$, with the superscript often omitted when either the model is unambiguously understood from the context, or the class is invariant to the model.

By abuse of notation, we write $rt-\mathrm{PROC}(c)$ (or $rt-\mathrm{SPACE}(c)$, etc.) instead of $rt-\mathrm{PROC}(f)$, whenever f is a constant function, such that $f(x) = c$ for all $x \in \mathbb{N}$. In order to formalize the above discussion, we offer the following definition.

Definition 3.1 Given a total function $f : \mathbb{N} \to \mathbb{N}$, and some model of parallel computation M, the class $rt-\mathrm{SPACE}^M(f)$ 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 $rt-\mathrm{PROC}^M(f)$ 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 $rt-\mathrm{PROC}^M(1)$ (that is, the class of sequential real-time algorithms) is invariant with the model M. □

Note that we do not consider nondeterminism. This is because real-time computation is a highly practical area, and thus nondeterministic computations, which are rather of theoretical interest, seem to be of little importance.

3.2 How to measure

We intentionally left unexplained in definition 3.1 the notion of input size. In classical complexity theory, the input size is the length of the current input, that is, the number of symbols that are available as input. However, such a notion cannot be naturally extended to ω-languages. Indeed, using such a definition, all well-behaved timed ω-words have length ω. That is, any ω-word is infinite. Therefore, a new notion of input size should be developed.

In the most general case of real-time applications the input data are received in bundles. Indeed, take for example the domain of real-time database systems. Here, the most time consuming operation is answering queries. Such queries appear at the input one at a time. That is, at any moment when some new input

arrives, this input consists in the n symbols that encode a query. Such a model of input arrival seems to appear in most real-time applications. Motivated by this, we propose the following definition for input size:

Definition 3.2 Let w be some timed ω-word over an alphabet Σ, $w = (\sigma, \tau)$. Let $\tau = \tau_1\tau_2\tau_3\ldots$ and $\sigma = \sigma_1\sigma_2\sigma_3\ldots$. Fix $i_0 = 0$. For any $j > 0$, let $s_j = \sigma_{i_{j-1}+1}\sigma_{i_{j-1}+2}\ldots\sigma_{i_j}$, such that the following hold:

1. $\tau_{i_{j-1}+1} = \tau_{i_{j-1}+2} = \cdots = \tau_{i_j}$, and
2. $\tau_{i_j+1} \neq \tau_{i_j}$.

Then, the size of w, denoted by $|w|$, is $|w| = \max_{j>0}|s_j|$.

Furthermore, given some alphabet Σ', $\Sigma' \subseteq \Sigma$, the size of w over Σ' is naturally defined by $|w|_{\Sigma'} = \max_{j>0}|s_j|_{\Sigma'}$, with s_j defined as above. □

In other words, the size of some ω-word w is given by the largest bundle of symbols that arrive as input together, at the same time. It should be noted that such a definition makes sense only in those cases in which the real-time algorithms manifest the *pseudo-on-line* property, that is, when they process input data in bundles, without knowledge of future input. As mentioned in [14], it would appear that this is a common feature of such algorithms.

Yet, it might appear that there are computational paradigms that seem not to fit into such a definition. Take for example the case of d-algorithms [11, 21, 22]. The input of a d-algorithm (short for "data-accumulating algorithm") is not entirely available at the beginning of the computation. Instead, more data arrive while the computation is in progress, according to a certain arrival law expressed as a (strictly increasing) function of time. Furthermore, a computation terminates when all the already received data are processed before another datum arrives as input. Within this paradigm, the input size is defined as the amount of data that is processed during the computation (one should also note that d-algorithms pertain to the class where time complexity actually makes sense, provided that this definition for the input size is used). However, the amount of processed data is itself a function of the running time, which is an inconvenience as far as a consistent time analysis of such algorithms is concerned. Furthermore, when parallel implementations are considered [11], the number of processors as a function of input size loses significance, since this would imply that the number of processors used by the algorithm depends on the running time (in particular, this number increases with time), which we believe to be an unrealistic approach. Finally, when space complexity is studied, the dependency of the input size on the running time is likely to make such a complexity hard to express (to our knowledge, however, such a direction has not been explicitly pursued in the literature).

It should be noted, however, that the input size for d-algorithms as defined in [21] and summarized in the above paragraph was shown to be of reduced significance. Indeed, it was proved [11] that d-algorithms are actually *on-line* algorithms, on which some real-time constraints are imposed. That is, the true input size for such algorithms is 1, since on-line algorithms process each input datum without knowledge of future inputs.

For all these reasons, we believe that definition 3.2 is adequate for most of the real-time algorithms, and therefore we will use it henceforth. We shall be aware though of those particular cases in which the expressiveness of such a notion of input size is diminished. Should such cases appear in the future, one can provide alternative, meaningful definitions for input size that are specific to the variants in discussion.

A final remark with respect to concatenated timed ω-languages is in order. While in classical language theory, given some word $w = w_1w_2$, the equality $|w| = |w_1| + |w_2|$ holds, this is not the case with respect to timed ω-words. Indeed, assume that the set of time values in the two ω-words is disjoint. Then, the equality that holds is $|w| = \max(|w_1|, |w_2|)$. However, not even this relation is general, since $|w|$ may be larger than both $|w_1|$ and $|w_2|$ in those cases when the words to be concatenated have sets of time values that are not disjoint. For this reason, we provide the following notation:

Definition 3.3 Consider two timed ω-languages L_1 and L_2. Now, let w be some word in L_1L_2, that is, $w = w_1w_2$, $w_1 \in L_1$, and $w_2 \in L_2$. Then, the size of w over L_1 is defined as $|w|_{L_1} = |w_1|$. The size of w over L_2 (namely, $|w_2|_{L_2}$) is defined analogously. □

4 The hierarchy $rt - \mathrm{PROC}(f)$

In the following, given some arbitrary word w of length n, we find convenient to index it starting from 0. Specifically, for $0 \leq i \leq n-1$, w_i denotes the i-th symbol of w, w_0 being the first symbol and w_{n-1} the last. By extension, for any $0 \leq i \leq j \leq n-1$, we denote by $w_{i\ldots j}$ the subword $w_i w_{i+1} \ldots w_j$ of w, with $w_{i\ldots j} = \lambda$ whenever $j < i$. Note, however, that both the symbol and time sequences in a timed ω-word are indexed in [14] and in section 2 starting from 1, and we keep henceforth this convention.

4.1 Two processors

In the following, we develop a timed ω-language (call it L_1) which is accepted by a two-processor algorithm, but cannot be accepted by a sequential algorithm.

First, fix two constants r and p, $r > 2p$, and the alphabet $\Sigma = \{a, b, +, -\}$. Now, consider the following (finite) timed language:

$$L_o = \{(\sigma, \tau) | \sigma \in \{a, b\}^r, \tau_i = 0 \text{ for all } 1 \leq i \leq r\}.$$

Intuitively, a word in the language L_o (all of its symbols being available at the beginning of the computation) represents the initial value of a word that will be modified as time passes. Let us introduce such a modification:

$$\begin{aligned} L_t &= \{(\sigma, \tau) || \sigma| = j, 1 \leq j \leq p+1, \sigma_1 \in \{+, -\}, \\ &\quad \sigma_{2\ldots j} \in \{a, b\}^{j-1}, \tau_i = t \text{ for all } 1 \leq i \leq j\}. \end{aligned}$$

A word in L_t denotes a change to the initial word, that arrives at time t. A word in L_t has a $+$ or $-$ as its first symbol, followed by at most p a's and/or b's. The semantics of such a word will become clear shortly. The language that denotes all the changes over time is thus:

$$L_u = \prod_{i>0} L_{ci},$$

for a given positive integer c. The language L_1 will be constructed as a subset of $L_o L_u$. However, in order to precisely define L_1, we have to define some new concepts, namely *insertion modulo r* and *acceptable insertion zone*.

First, consider some words $w \in \{a, b\}^r$ and $u = u_0 u'$, such that $u_0 \in \{+, -\}$ and $u' \in \{a, b\}^j$, $j \leq p$. Then, for some integer i, $0 \leq i \leq r-1$, we define the concept of *insertion modulo r at point i* of u in w as a function ins_r that receives the three parameters w, u, and i, and returns a new word w and a new i. The behavior of ins_r is as follows: Let $i' = i + p$ if $u_0 = +$ and $i' = i - p$ otherwise. Then, $ins_r(w, u, i) = (w', i' \bmod r)$, where w' is computed as follows ($\overline{x}$ denoting the reversal of some word x):

1. If $i' < 0$, let $i'' = i' \bmod r$. 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 above description might seem complicated, but the intuition behind it (which is also suggested by the name of the operation) is simple. Indeed, 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 the value of u_0

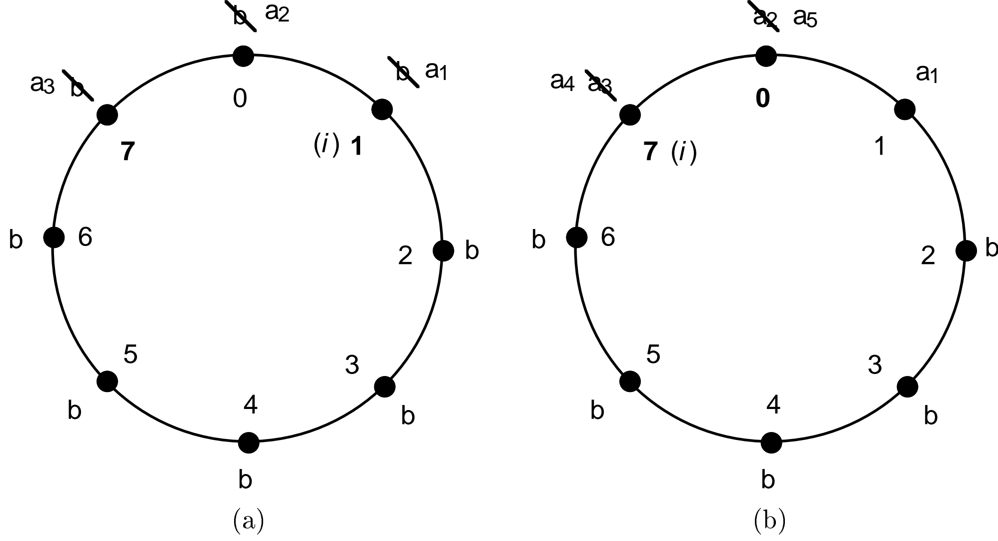

Figure 1: Insertion modulo r.

(+ for right, − for left). In other words, u models the moves of the entity which is pursued (the "pursuee") over the circle modeled by w. It follows that such an entity has a topmost velocity of p/r-th of the circle's circumference per time unit. As we shall see later, the algorithm that accepts L_1 (that is, the pursuer) needs to inspect all the symbols that are modified during such an insertion process. In other words, the pursuer has to match the move of the pursuee. The "velocity" of the accepting algorithm is given by its ability to inspect symbols stored in the memory, and thus this velocity is directly proportional to the speed of the processor(s) used by the algorithm. The modulo operation makes the two ends of w conceptually adjacent, thus the pursuit space becomes conceptually a circle, as desired.

More specifically, consider the example in figure 1. Here, $r = 8$ and $p = 3$. 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). Figure (1.a) 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). It is easy to see that we are in the case handled by item 1 in the above enumeration. In other words, 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 $u = +a_4a_5$. We are now in case 2 in the enumeration above, and the indices whose values are modified are 7 and 0. This processing is illustrated in figure (1.b). The final result is $i = 0$ and $w = a_5a_1bbbbba_4$.

Finally, the following generalization will prove useful later: 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

$$(w, i) \bigoplus_{j=\alpha}^{\beta} u^j = (w, i) \oplus u^\alpha \oplus u^{\alpha+1} \oplus \cdots \oplus u^\beta. \tag{1}$$

An intuitive aside.

Intuitively, the operator $\bigoplus$ is for $\oplus$ as $\sum$ is for $+$ in arithmetics. Specifically, such an operator receives some initial word w and some initial insertion point i, and successively inserts modulo r the words u^α, ..., u^β in w, modifying the insertion point accordingly after each such an insertion. For example, refer again to figure (1.b), 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 $(a_5a_1bbbbba_4, 0)$.

Let us consider now the second of the concepts we will use. Consider some word $w \in L_oL_u$, $w = w^0 \prod_{i>0} w^i$, with $w^0 \in L_o$, and $w^i \in L_{ci}$, $i > 0$. For some time value t and some i_0, $0 \leq i_0 \leq r-1$, let

$$s(w,t) = (\sigma^0, i_0) \bigoplus_{ci \leq t} \sigma^i, \tag{2}$$

where $\sigma^i = \text{detime}(w^i)$, $i \geq 0$.

Consider now some algorithm A that receives w as input and uses π processors, $\pi \geq 1$ (when $\pi = 1$, the algorithm is clearly sequential; otherwise, it is parallel). Furthermore, at any moment t, A may inspect (i.e., read from memory) the symbols stored at some index in $s(w,t)$. In the parallel case, many processors may inspect different indices in parallel. For each processor q, $1 \leq q \leq \pi$, let ι_t^q be the most recent index inspected by processor q up to time t. If some processor happens to inspect no symbols from $s(w,t)$, then, by convention, $\iota_t^q = -1$. In addition, let I_t^q be the "history" of symbols inspected by processor q up to time t, that is, $I_t^q = \bigcup_{t' \leq t} \iota_{t'}^q \setminus \{-1\}$. One may note that neither ι_t^q nor I_t^q depend on the actual processing performed by A. Indeed, 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, $\iota_t^q = -1$ and thus $I_t^q = \emptyset$, $1 \leq q \leq \pi$.

For some t, let now $lo = \min_{1 \leq q \leq \pi}(\iota_t^q)$, $hi = \max_{1 \leq q \leq \pi}(\iota_t^q)$, and $I = \bigcup_{1 \leq q \leq \pi} I_t^q$. Then, we define the *acceptable insertion zone* at time t (denoted by $z(w,t)$) as follows:

$$z(w,t) = \begin{cases} \{i | 0 \leq i < r\} & \text{if } lo = -1, \\ \{i | 0 \leq i < r, i \neq lo\} & \text{if } lo \neq -1 \text{ and there exists } j \notin I,\ j > hi \text{ or } j < lo, \\ \{i | lo \leq i \leq hi\} & \text{otherwise.} \end{cases} \tag{3}$$

That is, $z(w,t)$ is a set of indices, that has the following form: When all the indices in the area delimited by the latest inspected indices have been inspected in the past, then this area is excluded from $z(w,t)$. Otherwise, the acceptable insertion zone contains all the indices except one of the indices ι_t^q (specifically, the smallest of them).

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

An intuitive aside.

In order to support the intuition, we refer again to the geometric version of the problem (presented in section 1). In figure 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). Figure (2.a) 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 figure (2.b).

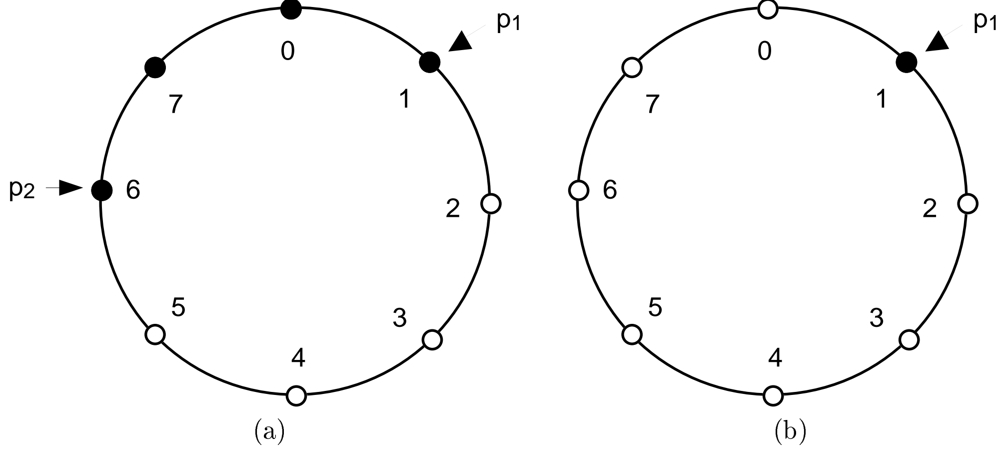

Figure 2: Acceptable insertion zone.

Furthermore, *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 figure (2.a), except that the index 7 is not inspected by any processor. Then, according to relation (3), the acceptable insertion zone is identical to the one shown in figure (2.b). 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 1. 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 L_1. For any word $w \in L_o L_u$ and for any time value ci, $i > 0$, denote by $z_i(w)$ the set of indices whose values are modified by the timed subword w^i of w, $w^i \in L_{ci}$. Then,

$$\begin{aligned} L_1 \quad = \quad & \{w \in L_o L_u | \text{for any } i > 0,\ z_i(w) \subseteq z(w, ci), \text{ and there exists some } t, \\ & t > 0, \text{ and some } i_0,\ 0 \leq i_0 \leq r-1, \text{ such that } |s(w,t)|_a = |s(w,t)|_b\}, \end{aligned}$$

with $s(w,t)$ and $z(w,t)$ defined as in relations (2) and (3), respectively.

Lemma 4.1 *L_1 is a well-behaved timed ω-language.*

Proof. Let w be some word in L_1, $w = (\sigma, \tau)$. Since L_1 is a subset of a concatenation of (well-behaved or not) timed languages ($L_1 \subseteq L_o \prod_{i>0} L_{ci}$), it follows by theorem 2.1 that time(w) satisfies the monotonicity condition.

Let $w = w^0 \prod_{i>0} w^i$, with $w^0 \in L_o$, and $w^i \in L_{ci}$, $i > 0$. Consider now some $s = (\sigma_j, \tau_j)$ such that s appears in some w^k. Then, we have $\tau_j = ck$. Furthermore, $j < r + (p+1)(k+1)$, since $|w^i| \leq p+1$ for any $1 \leq i \leq k$, and $|w^0| = r$. Now, for any finite $t > 0$, choose $k = (t+1)/c$. Then, $\tau_j = t+1 > t$, and j is finite (since $j < r + (p+1)(k+1)$). Thus w satisfies the progress condition, and therefore L_1 is well-behaved. □

Lemma 4.2 *There exists no sequential deterministic real-time algorithm that accepts L_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 L_1: First, we assume that A does not have to build the structure $s(w,t)$. 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 L_1. Indeed, A actually has to spend some extra time in order to build $s(w,t)$. Furthermore, the $|x|$ steps required for the above recognition problem constitute a lower bound for that problem, since all the symbols of x must be inspected in order to check whether $|x|_a = |x|_b$.

Consider now that the processor used by A has the ability to inspect c_1 symbols per time unit, $c_1 > 0$, and choose p such that $p = c \times c_1 + 1$. It is then immediate that, during each interval of c times units, A can inspect at most $p - 1$ 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 p 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 p symbols are inserted in $s(w,t)$. Without loss of generality, consider the insertion point at time $t = ci$ to be $j = \lfloor r/2 \rfloor$ (indeed, given the circularity of $s(w,t)$, another insertion point can be considered in the same manner, by performing a simple translation; we chose the "middle of the circle" in order to avoid modulo operations that would affect the clarity of the argument). Furthermore, according to observation 1, $z(w,t)$ 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 $|x| = p$. Otherwise (that is, if $j < z_t \leq r - 1$), chose $w^{i+1} = -x$, again with $|x| = p$. 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 $r > 2p$ 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}).

In summary, there exists an input word $w \in L_1$ which inserts p symbols in $s(w,t)$ at any time $t = ci$, and in this case there exists at least one symbol in $s(w,t)$ which is unknown to A. Thus, A cannot decide L_1, and we completed the proof. □

We have yet to prove the second part of the result (namely, that there exists a 2-processor real-time algorithm that accepts L_1). The crux of this proof is given by the following observation:

Observation 2 For some 2-processor 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 for the indices in $s(w,t)$, 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 4.2, and ϵ is a positive constant, arbitrarily close to 0.

Let $\iota_0^1 = 0$, and $\iota_0^2 = 1$. Thus, at each time t_1 when processor 1 inspects a new index in $s(w,t)$, let the newly inspected index be $\iota_{t_1}^1 + 1$ (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 $(\iota_{t_2}^2 - 1) \bmod r$ (and thus processor 2 advances only to the "left"). Then, according to relation (3), $|z(w,t)| \geq |z(w,t+1)|$ (specifically, if neither of the two processors inspected any index at time $t+1$, then $|z(w,t)| = |z(w,t+1)|$; otherwise, $|z(w,t)| > |z(w,t+1)|$). 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 the structure $s(w,t)$ as it is changed by input. However, $s(w,t)$ depends on $\prod_{cj \leq t} w^j$ only. Given that w is a well-behaved timed ω-language, $\prod_{cj \leq t} w^j$ 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 immediately. □

Lemma 4.3 *There exists a 2-processor PRAM deterministic real-time algorithm that accepts L_1 and that uses arbitrarily slow processors.*

Proof. Given observation 2, the ability of A' to accept L_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 f on the output 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 4.2, and ϵ is a positive constant, arbitrarily close to 0. That is, the processors used by A' are arbitrarily slow, as desired. □

Lemmas 4.1, 4.2, and 4.3 imply:

Theorem 4.4 $rt - \mathrm{PROC}(1) \subset rt - \mathrm{PROC}^{\mathrm{PRAM}}(2)$ *(strict inclusion).*

Theorem 4.4 is itself an important result, since it means that, in a real-time environment, a parallel 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 sequential implementation. To our knowledge, this is the first result of this nature to date. In fact, we can improve on the result stated in theorem 4.4.

4.2 n processors

In the following, we extend theorem 4.4. Specifically, we show that such a result holds for any number of processors n, $n > 1$. This way, we show not only that parallel real-time implementations are more powerful than sequential ones, but we also prove that such parallel algorithms form an infinite hierarchy with respect to the number of processors used. That is, 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.

For this purpose, we develop a language L_k similar to L_1. Intuitively, since L_1 models a one-dimensional, circular space, we extend such a space to k dimensions. Again, fix $k > 1$, $p > 0$, and $r > 2p$. For convenience, let $r' = kr$. Then, consider

$$L'_o = \{(\sigma, \tau) | \sigma \in \{a, b\}^{r'}, \tau_i = 0 \text{ for all } 1 \leq i \leq r'\}.$$

Let $\mathbb{N}_k = \{enc(i) | 1 \leq i \leq k\}$, where enc is a suitable encoding function from $\mathbb{N}$ to $\{I\}^*$, for some symbol[2] $I \notin \Sigma$. We do not insist on the actual form of enc, it can be, for example, the usual unary encoding of natural numbers [20]. However, 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

$$L'_t = L_t^{\mathbb{N}} L_t.$$

In addition to the direction of insertion and the word to be inserted, a word in L'_t now provides the "dimension" (from 1 to k) along which the insertion takes place. Finally, let

$$L'_u = \prod_{i>0} L'_{ci},$$

for a given constant c, $c > 0$. L_k will be a subset of $L'_o L'_u$.

[2]Recall from section 4.1 that $\Sigma = \{a, b, +, -\}$.

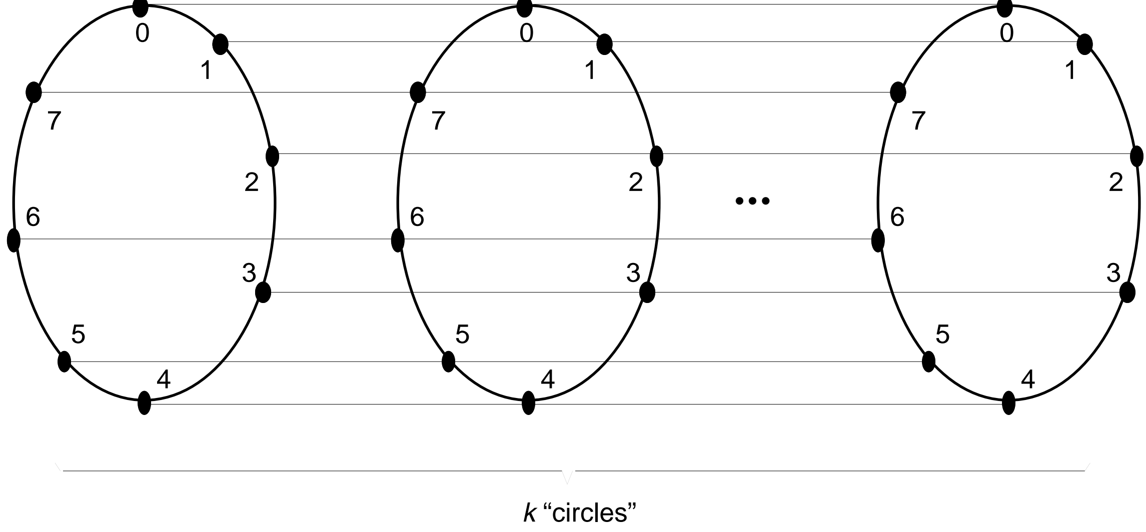

Figure 3: The k-dimensional circle.

We will now extend the notion of insertion modulo r to the new problem. Since the presentation is easier when we make use of the previously developed operators $\oplus$ and $\bigoplus$, we denote the new variant of insertion modulo r by $\otimes$, with its generalization $\bigotimes$.

Given some word $w \in \{a,b\}^{r'}$, we use the notation $w = w(1)w(2)\dots w(k)$, where $|w(i)| = r$, $1 \leq i \leq k$, and we call each $w(i)$ a *segment* of w. Let $w \in \{a,b\}^{r'}$, and $u = u'u''$, with $u' \in \mathbb{N}_k$, and $u'' \in \Sigma^j$, $1 \leq j \leq p+1$, $u''_1 \in \{+,-\}$, $u''_{2\dots p} \in \{a,b\}^{j-1}$. Then, for some i, $0 \leq i \leq r-1$, 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. Then, the operator $\otimes$ inserts (modulo r) u'' into that segment of w which is given by u'. The operator $\bigotimes$ is defined analogously to $\bigoplus$ (relation (1)), again, considering that $\otimes$ is left-associative.

Now, for some word $w \in L'_o L'_u$ ($w = w^0 \prod_{i>0} w^i$, with $w^0 \in L'_o$, and $w^i \in L'_{ci}$, $i > 0$), and for some i_0, $0 \leq i_0 \leq r-1$, let

$$s'(w,t) = (\sigma^0, i_0) \bigotimes_{ci \leq t} \sigma^i, \tag{4}$$

where $\sigma^i = \text{detime}(w^i)$, $i \geq 0$.

One should note that relation (4) is a generalization of relation (2). 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 π processors. Then, for some $t \geq 0$, define $z^j(w,t) = z(w(j),t)$, $1 \leq j \leq k$, with $z(w(j),t)$ defined as in relation (3), except for the following change: if, at time t, some processor inspects an index outside $s(w(j),t)$, then $\iota^q_t(j) = -1$ and $I^q_t(j) = \emptyset$. Finally, let

$$z'(w,t) = \bigcup_{j=1}^{k} z^j(w,t), \tag{5}$$

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 3. Each dimension is represented by a circle whose circumference has length r. There are, therefore, k such circles. Moreover, 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 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 do so. 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 2).

We have now all the concepts that are necessary for defining L_k:

$$\begin{aligned} L_k \quad = \quad & \{w \in L'_o L'_u | \text{for any } i > 0,\ z'_i(w) \subseteq z'(w, ci), \text{ and there exists some } t, \\ & t > 0, \text{ and some } i_0,\ 0 \leq i_0 \leq r - 1, \text{ such that } |s'(w,t)|_a = |s'(w,t)|_b\}, \end{aligned}$$

with $s'(w,t)$ and $z'(w,t)$ defined as in relations (4) and (5), respectively, and $z'_i(w)$ denoting the set of indices whose values are modified by the timed subword w^i of w, $w^i \in L_{ci}$, $i > 0$.

Lemma 4.5 *L_k is a well-behaved timed ω-language for any $k > 1$.*

Proof. Trivial generalization of the proof of lemma 4.1. Indeed, note that any word $u \in L_t^{\mathbb{N}}$ has a finite length (specifically, a length smaller than k). □

Lemma 4.6 *There exists no $(2n-1)$-processor PRAM deterministic real-time algorithm that accepts L_n, $n \geq 1$.*

Proof. The proof is by induction over n. Let A be an algorithm that attempts to accept L_n. The base case ($n = 1$) is established by lemma 4.2. Consider now a word $w \in L_n$ with its associated $s'(w,t)$. Furthermore, let $s'(w,t) = s_1 s_2$, such that $|s_1| = r$ and $|s_2| = r' - r$.

First, consider the case in which w modifies s_2 only. In order to successfully handle such a case, A must allocate at least $2(n-1)$ processors that inspect s_2, since $2(n-1)-1$ processors are not enough by inductive assumption. But then at most one processor inspects s_1, and therefore A cannot handle those inputs that modify s_1 exclusively, as shown by lemma 4.2. That is, whichever of the above processor allocations is chosen by A, there is an input word that A fails to accept.

Note that the version in which only a part of the computational power of some processor is allocated to s_1 is not acceptable, since such a processor has no influence on the acceptable insertion zone of s_1 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 p large enough (specifically, for $p > 2c \times c_1$, where c_1 is the maximal number of symbols that can be inspected by a processor in one time unit). That is, no processor allocation can lead to the acceptance of exactly all the words in L_n, and we completed the proof. □

Lemma 4.7 *There exists a $2n$-processor PRAM deterministic real-time algorithm that accepts L_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 4.3. Since there are $2n$ processors, such an allocation is clearly achievable. □

By lemmas 4.5, 4.6 and 4.7 we have

Theorem 4.8 *For any $n \in \mathbb{N}$, $n \geq 1$, $rt - \mathrm{PROC}^{\mathrm{PRAM}}(2n-1) \subset rt - \mathrm{PROC}^{\mathrm{PRAM}}(2n)$ (strict inclusion).*

That is, the hierarchy $rt - \mathrm{PROC}^{\mathrm{PRAM}}(f)$ is infinite.

4.3 Other parallel models of computation

One may wonder whether theorem 4.8 holds for other models of computation beside the PRAM. Indeed, except for a bounded number of processors, current technology does not allow a physical implementation of the PRAM. On the other hand, a model that allows a straightforward implementation is the *bounded-degree network* (BDN) [2], where processing elements have access to a local storage space only, and communication between them 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 Concurrent Read Concurrent Write PRAM [2]) can be simulated on a sparse interconnection network with bounded slowdown and bounded memory blowup. Specifically, there exists such a simulation [17] 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 4.8, 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 (arguable, but nevertheless often encountered) assumption: The BDN is the most elementary model of parallel computation. With this assumption in mind, the following result is an immediate corollary of theorem 4.8:

Theorem 4.9 *Given any model of parallel computation M, and for any $n \in \mathbb{N}$, $n \geq 1$, $rt - \mathrm{PROC}^M(2n-1) \subset rt - \mathrm{PROC}^M(2n)$ (strict inclusion).*

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

5 Conclusions

In one of our previous paper [14], we suggested as an interesting research direction the study of a realistic computational complexity theory for parallel real-time systems, that is, a theory based on a realistic model of real-time computations, that can be easily translated into practice. The concept of timed ω-languages was proposed in [14] as a possible foundation for this pursuit. This paper continues this idea. We started by defining complexity classes for the real-time domain. We believe that definition 3.1 captures the intuitive notion of resource (processors, storage space) bounds for real-time parallel algorithms. We also believe that these resources are the most important for the domain. However, section 3.1 offers the basis for the development of other complexity classes, should they prove to be useful. Since the notion of input size is different from the classical definition in the real-time domain, we presented an intuitive definition of such in section 3.2.

Besides defining the basis for our theory, we also proved what we believe to be an important result. Indeed, theorem 4.9, which is the central result of this paper, shows that the hierarchy of real-time algorithms with respect to the number of processors used is infinite. Furthermore, such a result is invariant to the model of parallel computation involved, and independent of the characteristics (that is, speed) of the particular processors used by the algorithms. To our knowledge, this is the first time such a result is proved. From a practical point of view, theorem 4.9 emphasizes the need for looking into parallel implementations, since this theorem shows that parallelism may add power, in a more general sense than mere speed, to a real-time application.

The languages L_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 paper hold even if the input is allowed to change (any number of) arbitrary indices within the acceptable insertion zone at any time ci, $i > 0$.

It should be noted that, even if the algorithms developed in this paper do not exhibit explicit deadlines, they are nonetheless real-time algorithms, since their inputs arrive in real-time. In order to justify this, note first that real-time input arrival implicitly imposes a sequence of deadlines (in a certain sense) on the

algorithm itself [10, 26] (for example, many algorithms with real-time input constraints should process the current input before another datum arrives in order to successfully handle the (real-time) input arrival [5, 26]; as another example, although no explicit deadlines are present in the case of d-algorithms, it is shown in [10] that the real-time arrival law for the input actually imposes a deadline on the length of the computation of any d-algorithm). Second, many practical domains such as real-time databases and industrial applications seem to include input into those factors that determine whether a given application deserves the real-time qualifier [9, 24, 26]. Finally, those algorithms with real-time constraints on their input are also recognized as real-time algorithms in [25].

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 also 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.

We believe that the direction started in this paper (namely, the real-time parallel complexity theory) is worth pursuing. In particular, an open problem we intend to address is the study of space hierarchies. Space in particular received a diminished attention from the parallel community, but we believe that interesting results can be found in this area. Another open problem, which is of less importance but nonetheless interesting, refers to the notion of real-time input size. Our measure is intuitive and faithful to the real world. However, given a concatenated timed ω-word $w = w_1w_2$, it would be nice to have $|w| = |w_1| + |w_2|$, as is the case in classical complexity theory. However, it is not possible to state such a property, since any well-behaved timed ω-word can be written as an infinite concatenation of timed ω-words, and thus its length would be infinite given the mentioned property. Therefore, we decided to offer instead a new notation (definition 3.3) for those cases in which complexity results need to be expressed as functions of many elements of an input which is the result of a concatenation operation. Such a somewhat cumbersome solution could be avoided if one can find a canonical representation of timed ω-words as concatenation of elementary words. While we don't expect that such a representation is absolutely necessary, it would simplify the formalisms, and it may also simplify the statement of some complexity results.

References

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

[3] ——, *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.

[4] 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/ ˜akl/ techreports/ beyond.ps.

[5] ——, *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/ ˜akl/ techreports/ realcrypto.ps.

[6] ——, *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/ ˜akl/ techreports/ realnum.ps.

[7] S. G. Akl. and L. Fava Lindon, *Paradigms admitting superunitary behaviour in parallel computation*, Parallel Algorithms and Applications, 11 (1997), pp. 129–153.

[8] R. Alur and D. L. Dill, *A theory of timed automata*, Theoretical Computer Science, 126 (1994), pp. 183–235.

[9] 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.

[10] 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. For a preliminary version see http:// www.cs.queensu.ca/ ~bruda/ www/ data_accum.

[11] ——, *The characterization of data-accumulating algorithms*, Theory of Computing Systems, 33 (2000), pp. 85–96. For a preliminary version see http:// www.cs.queensu.ca/ ~bruda/ www/ data_accum2.

[12] ——, *A case study in real-time parallel computation: Correcting algorithms*, in Proceedings of the Midwest Workshop on Parallel Processing, Kent, OH, August 1999. For a preliminary version see http:// www.cs.queensu.ca/ ~bruda/ www/ c-algorithms.

[13] ——, *On the necessity of formal models for real-time parallel computations*, in Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV, June 2000, pp. 1291–1297.

[14] ——, *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, March 2000, pp. 274–279. For a preliminary version see http:// www.cs.queensu.ca/ ~bruda/ www/ rt-def.

[15] ——, *Real-time computation: A formal definition and its applications*, Tech. Rep. 2000-435, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, 2000. http://www.cs.queensu.ca/ home/ akl/ techreports/ timed_lang.ps.

[16] J. H. Conway, *On Numbers and Games*, Academic Press, 1976.

[17] T. J. Harris, *A survey of PRAM simulation techniques*, ACM Computing Surveys, 26 (1994), pp. 187–206.

[18] J. E. Hopcroft and J. D. Ullman, *Formal languages and their relation to automata*, Addison-Wesley, Reading, MA, 1969.

[19] 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.

[20] H. R. Lewis and C. H. Papadimitriou, *Elements of the Theory of Computation*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[21] 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.

[22] ——, *Computing with time–varying data: Sequential complexity and parallel speed–up*, Theory of Computing Systems, 31 (1998), pp. 5–26.

[23] S. Swierczkowski, *Sets and Numbers*, Routledge & Kegan Paul, London, UK, 1972.

[24] M. Thorin, *Real–Time Transaction Processing*, Macmillan, Hampshire, UK, 1992.

[25] Usenet, *Comp.realtime: Frequently asked questions*, Version 3.4 (May 1998). http:// www.faqs.org/ faqs/ realtime-computing/ faq/.

[26] S. V. Vrbsky and S. Tomić, *Satisfying timing constraints of real-time databases*, Journal of Systems and Software, 41 (1998), pp. 63–73.

[27] H. Yamada, *Real-time computation and recursive functions not real-time computable*, IRE Transactions on Electronic Computers, EC-11 (1962), pp. 753–760.