Technical Report 2008-553
Time-Sensitive Computational Models with a Dynamic Time Component[*]

Naya Nagy and Selim G. Akl

School of Computing

Queen's University

Kingston, Ontario K7L 3N6

Canada

Email: {nagy,akl}@cs.queensu.ca

Abstract

It is known that a parallel computer can solve problems that are impossible to be solved sequentially. That is, any general purpose sequential model of computation, such as the Turing machine or the random access machine (RAM), cannot simulate certain computations, for example solutions to real-time problems, that are carried out by a specific parallel computer. This paper extends the scope of such problems to the class of problems with uncertain time constraints. The first type of time constraints refers to uncertain time requirements on the input data, that is *when* and *for how long* are input data available. A second type of time constraints refers to uncertain deadlines for tasks. The main contribution of this paper is to exhibit computational problems in which it is very difficult to find out (read *compute*) what to do and when to do it. Furthermore, problems with uncertain time constraints, as described in this paper, prove once more, that it is impossible to define a universal computer, that is, a computer able to simulate all computable functions.

Keywords: real-time computation, unconventional computation, Turing machine, universal computer, parallel computing.

1 Introduction

Defer no time, delays have dangerous ends.
William Shakespeare, Henry VI, Part 1, Act 3, Scene 2

Time plays an important role in real-life computations. This may imply that some computational task has to meet a deadline or that the data to be processed has to be collected at specific time intervals around the clock. It is our daily experience that most (hopefully not all) of the tasks around us, computational or not, incur some form of time pressure. We talk about computing in real-time [7, 8] when computations are performed under time constraints.

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

To illustrate the idea of time constraint, the simplest model of computation suffices, that is, one in which each computational cycle consists of three steps:

Step 1. Data are read from the outside world.

Step 2. Some operations are performed with these data.

Step 3. The result data are written out.

Most often, computational time constraints refer to step 3. The output needs to be available before a certain deadline. If the result is produced too late, then it is either useless (the deadline is hard [7]) or less valuable (the deadline is soft [8]). The implications of a time deadline on the capacity of Turing-type computational models to simulate each other are studied in [2], where it is proven that a parallel model is theoretically superior to a sequential model.

Step 1 is also a candidate for time constraints. Input data may be available at a certain time and also only for some time, after which the values deteriorate or are no longer accessible. The following classes of input data subject to time constraints have been studied:

1. Time varying input data [2].

2. Interdependent data [2].

3. Data accumulating paradigm [6].

Step 2 is performed internally by the computer and is less likely to be directly under time constraints. At least, research to date does not take this option into consideration. Nevertheless, if computers are considered part of their environment, rather than operating as a closed system, time can affect the internal working of a computer. An obvious example in this category is that computers can operate as long as they have a power supply. If the power supply is time dependent, internal computation is also constrained by this dependency. Time constraints on step 2 will not be considered in this paper.

This paper presents two problems that exhibit time constraints. The first problem has a time constraint on the input of data, whereas the second problem has a constraint on the output. Each problem is solved on both a sequential machine and a parallel machine. Computing the time constraints in each case is computationally demanding. As such, the parallel machine can use its additional power to compute the time constraints on time and to monitor the environment exhaustively. The sequential machine performs worse than the parallel machine and therefore is not equivalent in effectiveness to the parallel machine.

The problems presented here define a new paradigm. They are easy to solve when it is known what to do and when to do it. Yet, computing what to do and when to do it is what presents the main challenge.

The rest of the paper is organized as follows: Section 2 introduces the concept of dynamic time requirements for tasks and incorporates this concept into the problem of task scheduling. Section 3 defines the models of computation used by our algorithms. Section 4 defines a novel model of computation that includes time in its definition. Sections 5 and 6 describe the paradigms of problems with time constraints on the input data and on the output data, respectively. Section 7 discusses the common issues of the problems presented and section 8 concludes the paper.

		tasks	
		static	dynamic
time requirements	static	[7, 14]	[1, 8]
	dynamic	studied in this paper	not yet studied

Table 1: Types of tasks versus types of time requirements.

2 Dynamic Time Constraints

A task scheduler is responsible for scheduling some arbitrary set of tasks in an efficient way. Here, a *task* is meant to be some computation or program. The scheduler has to decide on the order in which the tasks are to be executed, what resources are to be allocated to a specific task, and when a task is to start or to finish. Depending on the characteristics of the task set, the scheduler may work very differently, from one scheduling problem to another.

In particular, *static tasks* are tasks defined at the outset, before any computation or task execution starts. The scheduler has full knowledge of these tasks well in advance. *Dynamic tasks* arrive to the scheduler during the computation in an unpredictable way.

The time requirements of a task are defined as its deadline, or the time constraints on the input of data, or in general any time constraint concerning the task's connection to the outside world. *Static time requirements* on a task are requirements that are well defined before the task starts executing. If the time constraints of a task are defined during the task's execution, or are computed by the task itself, then they are called *dynamic time requirements*.

To date, in all problems with time constraints appearing in the literature [1, 7, 8, 14], the time constraint is defined outside of the computation. When a task arrives, it has all time constraints fully defined already. It is *known* when and how to acquire input and what time constraints apply on the output or completion of the task.

The dynamic aspect of a scheduling problem lies solely in whether the tasks are fully known to the scheduler before the computation even starts, or whether tasks are created and destroyed during the computation. Thus, the scheduler is faced with four (task,time requirements) scenarios shown in table 1.

The **first scenario** in table 1 is defined by static tasks and static time requirements. Thus, the number of tasks to be scheduled is known prior to the computation. Also, the characteristics of the tasks are given at the outset. Time constraints are defined once and for all and do not depend or vary over time, nor are they influenced by the execution of a task or the scheduler itself. Descriptions of how to schedule static tasks are given in [7, 14].

The **second scenario** in table 1 is defined by dynamic tasks and static time requirements. Dynamic systems exhibiting dynamic scheduling problems are treated in [1, 8]. In this case, the tasks and their characteristics are not fully known at the beginning of the computation. Tasks are generated and destroyed during the computation; they arrive to the scheduler in an unpredictable way and have to be scheduled on the fly. Still, the task itself is fully defined at its generation. A new task, when taken into the scheduler, comes with its own clear requirements of resources, clear requirements on the input data and specifically with a well defined deadline (hard or soft).

The **third scenario** in table 1 is defined by static tasks and dynamic requirements. The algorithms addressed in this paper fall into this category. The problem here is even more interesting: the task to be solved has uncertain characteristics at the outset. The time requirements of the task are not defined at the generation of the task. Time constraints on the acquisition of data and/or deadlines will be defined during the execution of the task. Of special interest is the case when considerable computational effort is required to establish the time for an input operation, or the deadline of an output. In this situation, the power of the computational device becomes crucial in order to learn which computational step necessary for the task to complete is to be performed, and when to perform it.

Let us consider a task with no clearly defined deadline. Its deadline will be computed while executing the task itself. In this case, the deadline will be known at some point during the computation and may fall into one of the situations:

1. The deadline is far and the task can be executed without taking its deadline into consideration.

2. The deadline is close, but the task still can be executed.

3. The deadline is so close that it is impossible for the task to be completed.

4. The deadline already passed.

In some situations, the computation reaches an unexpected state: completion was unsuccessful because the computation did not know the deadline. Alternatively, the main computation was easy to perform, but the time constraints were too difficult to compute.

The **fourth scenario** in table 1 is defined by dynamic tasks and dynamic time requirements. This category has not been studied yet, as it has many variable characteristics and is therefore difficult to formalize.

3 Models of Computation

3.1 The Sequential Machine

The sequential computational model used in this paper is a basic Random Access Machine (RAM) [10]. A RAM consists of a processor able to perform computations. Also, the processor has access to a memory and can read or write any arbitrary memory location. The memory M is divided into four logical units (fig. 1). The first unit M_{in} contains the input data and is meant to be read only. The processor writes final results in a dedicated unit M_{out}. The unit M_{comp} is used to store intermediate results during the computation. A last unit M_{prog} contains the program to be executed. The reason for the separation of the memory into units is that M_{in} and M_{out} have a specific connection to the outside world. By contrast, M_{comp} and M_{prog} are internal to the RAM and not visible to the outside world. The processor is able to access the entire memory $M = M_{\text{in}} \bigcup M_{\text{comp}} \bigcup M_{\text{out}} \bigcup M_{\text{prog}}$. The availability of input data, which means the time at which the input is written from outside into the input memory M_{in} is not under the control of the RAM. Likewise, special time requirements apply on M_{out}, as to when the output data is supposed to be written into M_{out} and thus to be available to the outside world. As such, M_{in} and M_{out} are subject to conditions outside of the control of the RAM.

A computational step on the RAM means that the processor may do some or all of the following:

1. An input datum is written from the outside world into the input memory M_{in}.

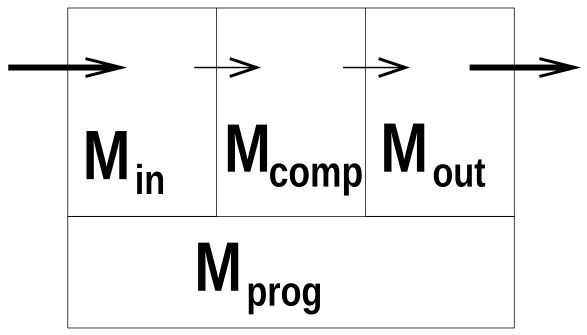

Figure 1: The four divisions of the memory.

2. The processor performs an operation on one or two values of the memory M and writes the result back to the memory. The input values may be taken from the input memory M_{in} or from the computation memory M_{comp}. If some input value is taken from M_{comp}, it is the result of a previous operation. The result of the current operation is written either into M_{comp} to be used in the future or into M_{out} in which case it is visible to the outside world.

Copying a value from one memory unit to another is also an operation. For example, transferring an input value from M_{in} to the output memory M_{out} is an operation.

3. Transfer a value from the output memory M_{out} to the outside world.

A computational step may contain all three phases described above, or only a subset of the three phases. The phases are considered to be executed in sequence from the first to the third. The time required to execute a computational step is considered to be one time unit. This one time unit is the same whether the computational step has all three phases or fewer.

3.2 The Parallel Machine

The parallel algorithms described in the next sections are supposed to run on a Parallel Random Access Machine (PRAM) [13]. The PRAM comprises a set of n processors that access a common memory. Each processor is similar to the processor of the RAM, in that it performs the same computational step.

The memory of the PRAM, as in the case of the RAM, is also divided into the same four divisions M_{in}, M_{comp}, M_{out}, and M_{prog}. Several processors can read from the same memory, permitting a concurrent read. Also several processors may write into the same memory location, defining a concurrent write. The convention for the concurrent write is that it is the sum of the (written) values that gets written into the memory location.

For the PRAM, all processors simultaneously execute one computational step in each time unit.

4 A Time-Aware Model of Computation

Programs will be written using a model that takes into consideration the time at which an operation is executed. This will help to define time constraints on input and output.

The new time-aware model of computation is defined by the quadruple $U = (C, O, M, P)$ where:

1. $C =< c_0, c_1, \ldots, c_i, \ldots >$ is a sequence representing a clock that keeps track of elapsed time. Each tick of the clock is one time unit; thus $c_{i+1} - c_i = 1$, for $i \geq 0$. In particular, c_0 is the time unit at which U begins a computation.

2. $O = \{R, W, CP, O_0, O_1, \ldots, O_{n-1}\}$, where $n \geq 1$, is a set of elementary operations:

   (a) R is a read operation which fetches inputs from the outside world. Whenever it is invoked, R reads one constant-size input.

   (b) W is a write operation which returns outputs to the outside world. Whenever it is invoked W writes one constant-size output.

   (c) CP is a copy operation that makes copies of memory locations. Whenever it is invoked CP copies a constant-size datum from one memory location to another.

   (d) O_i, $0 \leq i \leq n-1$, is an elementary arithmetic or logical operation such as addition, subtraction, comparison, logical AND, and so on.

3. $M = \{M_{\text{in}}, M_{\text{out}}, M_{\text{comp}}, M_{\text{prog}}\}$ is the memory consisting of four sections, each potentially of infinite size:

   (a) M_{in}, indexed $0, 4, 8, \ldots$, holds the input received from the outside world.

   (b) M_{comp}, indexed $1, 5, 9, \ldots$, holds intermediate computations.

   (c) M_{out}, indexed $2, 6, 10, \ldots$, holds the outputs delivered to the outside world.

   (d) M_{prog}, indexed $3, 7, 11, \ldots$, holds a program to be executed by U.

4. P is a processor capable of executing the elementary operations.

The program to be executed by U in solving a computational problem is a sequence of instructions, each requiring one time unit. An instruction is a quadruple

$$I_q = ((R, M_{\text{in}}, i, x), (O_r, M, k, l, m), (W, M_{\text{out}}, s), (a_q, b_q))$$

or

$$I_q = ((R, M_{\text{in}}, i, x), (CP, M, k, l), (W, M_{\text{out}}, s), (a_q, b_q))$$

where:

1. (R, M_{in}, i, x) means that the input value x of constant size is obtained from the outside world and stored in the location $M_{\text{in}}(i)$ of the memory. If $i < 0$ then no input is read.

2. (O_r, M, k, l, m) means that operation O_r is executed on up to two values stored in $M(k)$ and $M(l)$ and the result deposited in $M(m)$, depending on the values of k, l and m, respectively:

   (a) If $k \bmod 4 = 0$ and/or $l \bmod 4 = 0$ then the values $M(k)$ and/or $M(l)$ are obtained from M_{in}. If $k \bmod 4 = 1$ and/or $l \bmod 4 = 1$ then the values $M(k)$ and/or $M(l)$ are obtained from M_{comp}.

   (b) If $m \bmod 4 = 1$ then the value of $M(m)$ is written into M_{comp}. If $m \bmod 4 = 2$ then the value of $M(m)$ is written into M_{out}.

3. (CP, M, k, l) means that a value from $M(k)$ is copied to $M(l)$. This operation will be used to transfer measured input values to the output memory.

4. (W, M_{out}, s) means that the value of $M_{\text{out}}(s)$ is sent to the outside world, provided that s is nonnegative (if $s < 0$ nothing is done).

5. (a_q, b_q) is the time constraint for the instruction I_q to be executed, with $a_q, b_q \in C$:

   (a) $a_q < 0$ and $b_q < 0$, means that the instruction can be executed at any time (in other words, there is no time constraint on I_q).

   (b) $a_q < 0$ and $b_q \geq 0$, means that the instruction must be executed at time $t \leq b_q$, or else the computation fails.

   (c) $a_q \geq 0$ and $b_q < 0$, means that the instruction must be executed at time $t \geq a_q$, or else the computation fails.

   (d) $a_q \geq 0$ and $b_q \geq 0$, means that the instruction must be executed at time $a_q \leq t \leq b_q$, or else the computation fails. In particular, if $a_q = b_q \geq 0$, then the instruction must be executed *during* time unit a_q.

The following points are worth noting:

(a) If there exists a schedule of the instructions that satisfies all the time constraints, then it is assumed that the instructions have been already arranged sequentially according to that schedule. Otherwise, the computation is infeasible.

(b) Consider real-time applications with deadlines. The computation has to complete before a *deadline*. Real-time computations with deadlines are a special case of those computations afforded by the new time-aware model U in two ways:

   i. The deadline is reflected in those cases where there is an upper bound on the time at which an instruction is to complete, but not in the case where an instruction must be executed in a *given* time unit.

   ii. For all known real-time computation, deadlines are fixed a-priori, whereas in U the pair (a_q, b_q) may be computed by the algorithm itself.

4.1 Why a New Model?

The new model of computation is intended to fill both a theoretical and a practical void. The need for such a model will become increasingly important for computations that are increasingly unconventional [4]. We single out two specific motivations for our present purposes:

1. The presence of C in the model is what makes U unique. Without C, it is easy to see that U is equivalent to any of the traditional models of computation, such as the Turing Machine, the Random Access Machine, and so on. None of these conventional models takes real time into consideration. By contrast, every general-purpose computer in use today has a *clock*, and many computations on such machines are time-dependent. Examples of such computations that are aware of real time include: scheduled virus scans, regular backups, seasonal time changes [12], and so on. Therefore, U is intended as a theoretical model that captures the capabilities of realistic computers.

2. The new time-aware model U satisfies all the criteria for universality aspired to by all existing so-called universal models, such as the Turing Machine, the Random Access Machine, and so on, all of which are subsumed by U. However, contrary to general

belief, none of these conventional models of computation is universal [2, 3, 5], in the sense of being able to simulate any computation that is possible on any computer. As it turns out, U as well is not universal.

Thus, any existing computation, that can be expressed in Turing machine terms, can be reformulated into the formalism of the new time-aware model. Additionally, problems that have time as an explicit parameter of the computation can also be formalized on this model, but not on the Turing machine. Therefore, the time-aware model offers a broader perspective on the formal definition of computational problems. It can readily be applied to the following known paradigms:

1. Problems with time varying data [2] in which input data is unstable and varies with time.

2. Problems with time dependent complexity [6] in which the complexity of a certain computational step depends on the time when the step is executed.

In this paper, we show how the model allows us to study problems with uncertain time requirements. Here, time requirements depend on events or computations during the execution of a task.

5 Dynamic Time Requirements on the Acquisition of Data

If faced with a task to be executed, we are accustomed to know *what* input data we will need in order to perform the task and *when* to acquire the data necessary for the task's computations, in other words *when* to listen to the environment. This is actually an abstraction, a simplification of many real-life tasks. Even very simple paradigms can illustrate the idea that the nature of the input data (*what*) or the time requirements (*when*) of some data *cannot* be known unless some event *during* the task's computation happens.

We will call this paradigm, the *paradigm of the unexpected event*, where the cause of the event needs to be investigated. During a computation or the execution of a task, an unexpected event happens. By 'unexpected', it is meant not planned in advance and not predicted by the computation itself. The need immediately arises to investigate the cause of the event. This would mean to inspect the environment or, depending on the situation, inspect the computer itself. Yet, we are interested in the state of the environment *before* the event happened. This means that we need values of parameters describing the environment measured at a moment *prior* to the one in which we are now performing the investigation.

The following story might best suggest the paradigm.

Mr. Clueless is quietly reading in his study. He can hear the jolly hubbub of children outside. Suddenly, there is a loud crushing noise. He goes into the kitchen, just to find his kitchen window broken by a soccer ball. Quickly, Mr. Clueless gets out of the house and onto the playground, but ... the playground is empty. The children have all run away. This is not very helpful. Mr. Clueless would like to know, who was on the playground **before** *the kitchen window was broken. Investigating the playground now, is not giving any clues about who caused the damage of the window. What is worse, Mr. Clueless, while quietly reading, could have looked out of the window to see who is playing soccer, but he* **didn't know** *at the time, that this information would be of any value* **later**.

Real life examples of this paradigm are numerous and arise in a variety of contexts:

1. In a network (of streets) with a high traffic of objects (cars), a collision (accident) occurs. The problem is to find the cause of the collision, the objects (cars) responsible for the unexpected collision event.

2. A patient contracts a disease. It is required to determine what caused the health distress. A clear example for this idea would be the study of Sudden Infant Death Syndrome (SIDS) in children [19]. This research monitors a sample of healthy children with the intention of identifying among them some that will develop SIDS.

3. A nuclear reactor, after years of normal functioning, behaves erratically and even explodes. What was the cause of this behavior, what parameters reached critical or out of bounds values and why?

4. In the case of a natural disaster, such as an earthquake, the problem is to best describe the conditions and parameters that allow the prediction of the earthquake.

5. A computer gets infected by a virus and crashes. We want to restore the computer to a state prior to the virus infection ... backups would have been a good idea.

The next subsection describes a simple abstract problem that broadly subsumes the examples given above.

5.1 Implementation of the Paradigm of the Unexpected Event

The following example expresses the difficulty to perform a simple computation if time constraints are to be defined during the computation. In particular, the time constraints here refer to the acquisition of data.

A sensor has to monitor the environment in which it is located. The environment is defined by two different parameters: temperature T and pressure p. These parameters vary unpredictably with time: $T = T(t)$ and $p = p(t)$.

The computation starts at $t_0 = 0$ and continues for a time of $length = 10$ time units, that is, the monitor has to operate for this length of time. Computation has to finish at $t_f = t_0 + length$. When designing the algorithm, we may refer to any time unit of the monitor's operational time: $t_0 = 0$, $t_1 = 1$, $t_2 = 2$, ..., $t_{10} = t_f = 10$.

Requirements: The monitor has to measure, exactly once, one parameter of the environment and output this value at the end of its computation, that is, at time t_f. At the beginning of the task t_0, the task does not know yet what needs to be measured and when. The time at which the parameter is to be measured, denoted t_{input}, is anywhere during the computation

$$t_0 \leq t_{\text{input}} \leq t_f.$$

As a dynamic characteristic, t_{input} is not known at the beginning of the computation. Information to compute t_{input} will be available during the computation. Also, which of the two parameters to measure is not known at the beginning of the computation, that is, at time t_0. This information is represented by a binary value *which* and will also be computable later in the computation. If $which = 0$ then the monitor has to measure the temperature and if $which = 1$ then the monitor has to measure the pressure.

These two variables t_{input} and *which* will not be available easily, nor directly. There is no information in the environment, that the computer could read, until the middle of the computation, that is, at time $t_{\text{middle}} = t_0 + \frac{length}{2} = 0 + \frac{10}{2} = 5 = t_5$. Now, at t_{middle}, all of a sudden, several variables will make the computation of t_{input} and *which* possible. Three variables denoted x_0, x_1, x_2 are now available in the environment for the computer to read in. We chose the number of three variables to be smaller than the five time units left for computation, from t_{middle} to t_f. This will give the sequential computer some small chance to successfully complete the task. If there are more than five variables, the sequential computer will never be successful.

The time t_{input} and the boolean *which* will be computed from the sum of x_0, x_1, and x_2 according to the following formula.

$$t_{\text{input}} = (x_0 + x_1 + x_2) \bmod 10$$

$$which = (x_0 + x_1 + x_2) \bmod 2$$

We see that the problem here is to compute t_{input} and *which* on time. Although the monitoring is trivial otherwise, the difficulty is in knowing what to do and when.

We will see in the following subsections that the type of the computer employed to solve the problem will make the difference between success and failure.

5.1.1 Idle Sequential Solution

The first algorithmic solution is on a sequential computer. The computer does not start monitoring its environment unless it fully knows what to do. The computer is working "on request" or according to the principle: *Don't do anything unless you absolutely have to ...*

Thus, the monitor (i.e., the sequential computer) is a RAM, able to do one measurement, one computation and to output one value, all in one time unit. It starts its computation at $t_0 = 0$, but remains idle until it receives a measurement request at $t_{\text{middle}} = t_5$. At t_{middle}, the computer computes the parameters of the request. It takes the computer $\delta = 2$ time units to compute the values for t_{input} and *which*. The following situations can happen:

1. $t_{\text{input}} \geq t_{\text{middle}} + \delta$. The computer can satisfy the measurement request. At t_{input}, the computer measures the requested parameter, temperature or pressure, and outputs the value at t_f.

2. $t_{\text{middle}} \leq t_{\text{input}} < t_{\text{middle}} + \delta$. The computer has not finished calculating t_{input} and fails to read the input at the required time.

3. $t_{\text{input}} < t_{\text{middle}}$. The computer certainly fails to read the input at the required time. It cannot go back in time to measure the environment parameter at t_{input}.

Below are the computation steps performed using the time aware model:

$I_0 = ((R, M_{\text{in}}, 0, x_0), -, -, (5, 5))$
$I_1 = ((R, M_{\text{in}}, 4, x_1), (+, M, 0, 4, 1), -, (6, 6))$
$I_2 = ((R, M_{\text{in}}, 8, x_2), (+, M, 1, 8, 1), -, (7, 7))$
$I_3 = (-, (\text{mod}2, M, 1, \mathit{value}(10), 5), -, (8, 8))$
$I_4 = (-, (\text{mod}10, M, 1, \mathit{value}(10), 9), -, (9, 9))$
$I_5 = ((R, M_{\text{in}}, 12, T \text{ or } p = (1 - M(5)) \times T + M(5) \times p),$
$\quad (CP, M, 12, 2), (W, M_{\text{out}}, 2)$

This monitor has little chance of success, as it will try to get input only after all information concerning the input, t_{input} and *which*, is computed. This leaves only $t_f - t_{\text{middle}} - \delta$ time units at the end, in which the environment is actually listened to. If t_{input} falls in this range, the monitor is successful, otherwise it fails.

The overall success rate of this monitor is:

$$\mathit{success} = (t_f - t_{\text{middle}} - \delta)/(t_f - t_0) = 1/10 = 10\%.$$

5.1.2 Active (Smart) Sequential Solution

In this case, the monitor will try to do better by anticipating the request. The computer will monitor the environment, even though it does not know exactly what the requirements will be. The principle of this computer is: *Do as much as you can in advance ...*

The monitor is again a sequential computer as in the previous case. Instead of being idle while waiting for the request, the computer busily measures the environment. This is done ever since the computation started at t_0 until it receives the request at t_{middle}. Because the parameter of interest is unknown, the best the computer can do is to choose just one parameter, for example the temperature. The value of the temperature is recorded for all time units: t_0, t_1, t_2, t_3 and t_4. Then the request is received during $t_{\text{middle}} = t_5$, t_6, and t_7. The parameters t_{input} and *which* are computed during t_8 and t_9. During t_{10} it is still possible to read the correct environment parameter, if it so happens that $t_{\text{input}} = t_{10}$. The following situations can happen:

1. $t_{\text{input}} \geq t_{\text{middle}} + \delta$. The computer is able to measure the required environment parameter and output the value of interest after it received the request.

2. $t_{\text{middle}} \leq t_{\text{input}} < t_{\text{middle}} + \delta$. The computer is busy reading the variables x_0, x_1, and x_2 and has not recorded any history of the environment. The task fails.

3. $t_{\text{input}} < t_{\text{middle}}$ and the value of the temperature is required. The computer has recorded the history of the temperature and can output the desired recorded value.

4. $t_{\text{input}} < t_{\text{middle}}$ and the value of the pressure is required. The computer has not recorded the history of the pressure and is unable to meet the request.

The computation steps perform by the time aware computation model are given below:

$I_0 = ((R, M_{\text{in}}, 0, T), -, -, (0, 0))$
$I_1 = ((R, M_{\text{in}}, 4, T), -, -, (1, 1))$

$I_2 = ((R, M_{\text{in}}, 8, T), -, -, (2,2))$
$I_3 = ((R, M_{\text{in}}, 12, T), -, -, (3,3))$
$I_4 = ((R, M_{\text{in}}, 16, T), -, -, (4,4))$
$I_5 = ((R, M_{\text{in}}, 20, x_0), -, -, (5,5))$
$I_6 = ((R, M_{\text{in}}, 24, x_1), (+, M, 0, 4, 1), -, (6,6))$
$I_7 = ((R, M_{\text{in}}, 28, x_2), (+, M, 1, 8, 1), -, (7,7))$
$I_8 = ((R, M_{\text{in}}, 32, T), (\text{mod2}, M, 1, value(10), 5), -, (8,8))$
$I_9 = ((R, M_{\text{in}}, 36, T), (\text{mod10}, M, 1, value(10), 9), -, (9,9))$
$I_{10} = ((R, M_{\text{in}}, 40, T \; or \; p = (1 - M(5)) \times T + M(5) \times p),$
$\quad (CP, M, t_{\text{input}} \times 4, 2), (W, M_{\text{out}}, 2)$

This monitor has a better chance of being successful. If the temperature is indeed required, the computer's chances are almost 100%. The exact chance of success is $(t_f - t_0 - \delta)/(t_f - t_0) = 7/10$. If the value of the pressure is required, the success rate is the same as for the idle solution $(t_f - t_{\text{middle}} - \delta)/(t_f - t_0) = 1/10$.

The overall success rate is the average of the two rates computed above:

$$success = \frac{1}{2} \times (\frac{t_f - t_0 - \delta}{t_f - t_0} + \frac{t_f - t_{\text{middle}} - \delta}{t_f - t_0}) = \frac{1}{2}(\frac{7}{10} + \frac{1}{10}) = 40\%.$$

5.1.3 Parallel Solution

The last and most successful algorithmic solution is offered by a parallel computer. The parallel computer monitors the environment exhaustively in order to answer the request no matter what is asked or when it is to be performed. The principle of this computer is: *Do absolutely everything and be prepared for the worst ...*

The monitor is a parallel computer and can perform several measurements and computations in one time unit. For our example, it needs to have three processors, denoted P_0, P_1, and P_2.

The parallel computer can measure both the temperature and pressure of the environment at the same time. The parallel computer is able to record the full history of the environment. When t_{input} is computed during $t_{\text{middle}} = t_5$, the computer will still be able to fully monitor the environment and also compute t_{input} and *which*. The following situations can happen:

1. $t_{input} \geq t_{\text{middle}} + \delta$. The computer will measure the requested parameter at t_{input} and write it out at the end of the computation.

2. $t_{input} < t_{\text{middle}} + \delta$. The computer inspects its recorded history of the environment and outputs the desired parameter.

The computation steps performed using the time aware model are shown in table 2.

This monitor always answers the request successfully. Therefore, the success rate is:

$$success = 100\%.$$

6 Dynamic Time Requirements on the Output of Data

This section explores the situation where the deadline of a task is not defined at the outset, but will be computed *during* the execution of the task. Thus it is of the utmost importance that the computer executing the task be able to compute the deadline before it has passed.

I_i	Processor Instruction
I_0	$P_0 : ((R, M_{\text{in}}, 0, T), -, -, (0, 0)$ $P_1 : ((R, M_{\text{in}}, 100, p), -, -, (0, 0)$ $P_2 : (-, -, -, (0, 0)$
I_1	$P_0 : ((R, M_{\text{in}}, 4, T), -, -, (1, 1)$ $P_1 : ((R, M_{\text{in}}, 104, p), -, -, (1, 1)$ $P_2 : (-, -, -, (1, 1)$
I_2	$P_0 : ((R, M_{\text{in}}, 8, T), -, -, (2, 2)$ $P_1 : ((R, M_{\text{in}}, 108, p), -, -, (2, 2)$ $P_2 : (-, -, -, (2, 2)$
I_3	$P_0 : ((R, M_{\text{in}}, 12, T), -, -, (3, 3)$ $P_1 : ((R, M_{\text{in}}, 112, p), -, -, (3, 3)$ $P_2 : (-, -, -, (3, 3)$
I_4	$P_0 : ((R, M_{\text{in}}, 16, T), -, -, (4, 4)$ $P_1 : ((R, M_{\text{in}}, 116, p), -, -, (4, 4)$ $P_2 : (-, -, -, (4, 4)$
I_5	$P_0 : ((R, M_{\text{in}}, 20, T), -, -, (5, 5)$ $P_1 : ((R, M_{\text{in}}, 120, p), -, -, (5, 5)$ $P_2 : ((R, M_{\text{in}}, 200, x_0), -, -, (5, 5)$
I_6	$P_0 : ((R, M_{\text{in}}, 24, T), -, -, (6, 6)$ $P_1 : ((R, M_{\text{in}}, 124, p), -, -, (6, 6)$ $P_2 : ((R, M_{\text{in}}, 204, x_1), (+, M, x_0, x_1, sum), -, (6, 6)$
I_7	$P_0 : ((R, M_{\text{in}}, 28, T), -, -, (7, 7)$ $P_1 : ((R, M_{\text{in}}, 128, p), -, -, (7, 7)$ $P_2 : ((R, M_{\text{in}}, 208, x_2), (+, M, sum, x_2, sum), -, (7, 7)$
I_8	$P_0 : ((R, M_{\text{in}}, 32, T), (/10, M, sum, 10, m), -, (8, 8)$ $P_1 : ((R, M_{\text{in}}, 132, p), (\text{mod}10, M, sum, 10, n), -, (8, 8)$ $P_2 : (-, -, -, (8, 8)$
I_9	$P_0 : ((R, M_{\text{in}}, 36, T), -, -, (9, 9)$ $P_1 : ((R, M_{\text{in}}, 136, p), -, -, (9, 9)$ $P_2 : (-, -, -, (9, 9)$
I_{10}	$P_0 : ((R, M_{\text{in}}, 40, T), (CP, M, (t_{\text{input}} \times 4)\ or\ 100 + (t_{\text{input}} \times 4)),$ $(W, M_{\text{out}}, T\ or\ p(n)), (10, 10)$ $P_1 : ((R, M_{\text{in}}, 140, p), -, -, (10, 10)$ $P_2 : (-, -, -, (10, 10)$

Table 2: Computation steps of the time aware model.

Requirements: A monitor is required to measure some parameter of the environment. For definiteness, consider the parameter to be the temperature T. The monitor is active for a certain length of time $\delta + length$. This time is divided into two intervals δ and $length$, that will be defined later. It starts its activity at time t_0 and consequently finishes at time $t_f = t_0 + \delta + length$. The task of the monitor is to measure the temperature T at the beginning, that is, at time t_0. Then, after some delay, the measured temperature value is to be output at time t_{output}. The delay is not allowed to be null, it has to be larger than δ. Thus,

$$t_0 + \delta \leq t_{\text{output}} \leq t_f$$

With the output of the measured temperature, the task finishes.

Time t_{output} is not given at the outset, but has to be computed by the monitor according to the following rules. The n input variables x_0, x_1, ..., x_n are available in the environment all trough the computation. The output time is defined by the sum modulo the length of its activity,

$$t_{\text{output}} = t_0 + \delta + (x_0 + x_1 + ... + x_n) \bmod length$$

This problem will be solved on a sequential and a parallel machine. This will show again that the size of the machine matters. A sequential machine, or a parallel machine without enough resources, fails to perform the task, whereas a sufficiently powerful parallel machine successfully completes the task.

In our examples, we consider the following values: $t_0 = 0$, $\delta = 2$, $length = 8$, and $n = 10$.

6.1 Sequential Solution

The sequential computer has the same definition as the example of section 5. It is a RAM described by the time-aware model.

The sequential monitor measures the temperature at t_0 and then proceeds to read the input variables x_0, x_1, ..., x_{10}. By the time the computer has managed to calculate the deadline t_{output}, that moment has already passed. Therefore, the sequential monitor will not be able to output the result, *just* because it did not know the deadline in time.

The RAM follows the steps below:

$I_0 = ((R, M_{\text{in}}, 0, T), -, -, (0, 0))$
$I_1 = ((R, M_{\text{in}}, 4, x_0), (+, M, 1, 4, 1), -, (1, 1))$
$I_2 = ((R, M_{\text{in}}, 4, x_1), (+, M, 1, 4, 1), -, (2, 2))$
$I_3 = ((R, M_{\text{in}}, 4, x_2), (+, M, 1, 4, 1), -, (3, 3))$
$I_4 = ((R, M_{\text{in}}, 4, x_3), (+, M, 1, 4, 1), -, (4, 4))$
$I_5 = ((R, M_{\text{in}}, 4, x_4), (+, M, 1, 4, 1), -, (5, 5))$
$I_6 = ((R, M_{\text{in}}, 4, x_5), (+, M, 1, 4, 1), -, (6, 6))$
$I_7 = ((R, M_{\text{in}}, 4, x_6), (+, M, 1, 4, 1), -, (7, 7))$
$I_8 = ((R, M_{\text{in}}, 4, x_7), (+, M, 1, 4, 1), -, (8, 8))$
$I_9 = ((R, M_{\text{in}}, 4, x_8), (+, M, 1, 4, 1), -, (9, 9))$
$I_{10} = ((R, M_{\text{in}}, 4, x_9), (+, M, 1, 4, 1), -, (10, 10))$

As can be seen from the program above, the monitor can barely compute the sum of the input variables x_0, x_1, ..., x_9 until the total time of its activity expires. This happens at $t_f = 10$. Thus the sequential machine fails to output the value at the right time.

6.2 Parallel Solution

The parallel computer is a PRAM. It has $n+1=11$ processors, P_0, P_1, P_2, ..., P_{10}, to be able to perform all measurements at the very beginning, that is, at t_0. Then during the delay δ, the deadline t_{output} is computed. After this, the computer only has to wait for the right time to output the value of the temperature. The PRAM follows the steps below:

I_i	Instruction
I_0	$P_0 : ((R, M_{\text{in}}, 0, x_0), -, -, (0,0))$ $P_1 : ((R, M_{\text{in}}, 0, x_1), -, -, (0,0))$ ... $P_9 : ((R, M_{\text{in}}, 0, x_9), -, -, (0,0))$ $P_{10} : ((R, M_{\text{in}}, 4, T), -, -, (0,0))$
I_1	$P_0 : (-, (\text{mod}10, M, 0, 0), -, (1,1))$ $P_1 : -$ $P_2 : -$ ... $P_{10} : -$
I_2	$P_0 : (-, (CP, M, 4, 2), (W, M_{\text{out}}, 2), (M(0), M(0)))$ $P_1 : -$ ... $P_{10} : -$

It can be seen that the parallel computer is able to meet the deadline because it is able to compute the deadline on time. A parallel computer with enough processors can compute the deadline in δ time units. On the other hand, if the PRAM has fewer processors, for example $(n+1)/2=5$ then this computer is not guaranteed to succeed. If t_{output} is very close to $t_0+\delta$ then the computer will miss the deadline.

7 Discussion

The examples of the previous sections have some common features. Both present very easy tasks: to monitor certain environment variables and to output the measured values. The difficulty in both cases arises from the fact that the time requirements of the task to be executed are not well defined at the outset. It takes considerable effort to compute the time requirements of the tasks. It is an unconventional paradigm in which defining the parameters of the task takes a large percentage of the overall computational effort required to complete the task.

Thus, if time is an intrinsic characteristic of the problem, then the effort to define these time characteristics has to be considered in the abstract definition of the algorithm or program. The new time aware model exhibits just this feature. Time is a parameter of each instruction.

In this case, a computer with enough computational power may be able to successfully complete a task which cannot be computed by a weaker model. In particular, we have shown in our examples that a parallel computer performs better than a sequential one. This happens to the point that the sequential computer cannot simulate the parallel one.

8 Conclusion

Behind me time gathers ... and I darken!
Mihai Eminescu, Years have trailed past ...

The examples in this paper describe problems whose time constraint are difficult to compute. They show that a parallel computer of an appropriate size is essential for the successful completion of some computational task. In particular, a simple Turing machine, classically called "universal", is not able to perform such tasks successfully. This Turing machine is not able to simulate a more powerful machine capable of carrying these tasks to completion. This confirms, once again, the previously established result [9, 11, 15, 16, 17, 18] that the Turing machine is in fact *not* universal, as it cannot simulate a task computable on a particular parallel machine.

Furthermore, any parallel computer can face a problem in which computing the time constraints of a task is above its computational capacity, while a more powerful parallel computer can be defined to solve the given problem. This means that a parallel computer of fixed size cannot be specified that is able to successfully complete any problem of the type described in this paper. For any parallel computer, a problem can be defined such that this computer is not able to compute its time constraints on time. Therefore, even a parallel computer is not universal. This also confirms the more general result, first given in [2], that no computer capable of a finite number of operations per time unit can be universal. This result holds even if the computer has access to the outside world for input and output, is endowed with an unbounded memory and is allowed all the time it needs to simulate a successful computation by another computer.

This study is open to continuation. A formalization of schedulers of dynamic tasks with dynamic time requirements would lead to more general settings. Also, it may be useful to consider other components of a task that allow for a dynamic definition. Thus, components that vary along the computation, according to results of the computation often occur in realistic applications. One example of such components mentioned in this paper is the inner computation of a task which may be subject to constraints, perhaps affected by the computation itself. Such computations would constitute a prime candidate for future investigations.

References

[1] Luca Abeni and Giorgio Buttazzo. Resource reservation in dynamic real-time systems. *Real-Time Syst.*, 27(2):123–167, 2004.

[2] Selim G. Akl. Three counterexamples to dispel the myth of the universal computer. *Parallel Processing Letters*, 16(3):381–403, September 2006.

[3] Selim G. Akl. Even accelerating machines are not universal. *International Journal of Unconventional Computing*, 3(2):105–121, 2007.

[4] Selim G. Akl. Evolving computational systems. In Sanguthevar Rajasekaran and John H. Reif, editors, *Handbook of Parallel Computing: Models, Algorithms, and Applications*, pages 1 – 22. Taylor and Francis, CRC Press, Boca Raton, Florida, 2008.

[5] Selim G. Akl. Unconventional computational problems with consequences to universality. *International Journal of Unconventional Computing*, 4(1):89–98, 2008.

[6] Stefan D. Bruda and Selim G. Akl. On the data-accumulating paradigm. In *Proceedings of the Fourth International Conference on Computer Science and Informatics , Research Triangle*, pages 150–153, 1998.

[7] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications (Real-Time Systems Series).* Kluwer Academic Publishers, Boston, 1997.

[8] Giorgio C. Buttazzo. *Soft Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications (Real-Time Systems Series).* Springer, New York, NY, 2005.

[9] C.S. Calude and Gh. Păun. Bio-steps beyond Turing. *BioSystems*, 77:175–194, 2004.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, second edition.* MIT, Cambridge, Massachusetts, 2001.

[11] G. Etesi and I. Németi. Non-Turing computations via Malament-Hogarth space-times. *International Journal of Theoretical Physics*, 41(2):341–370, February 2002.

[12] A. Gut, L. Miclea, Sz. Enyedi, M. Abrudean, and I. Hoka. Database globalization in enterprise applications. In *IEEE International Conference on Automation, Quality and Testing, Robotics*, pages 356–359, 2006.

[13] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, Reading, Massachusetts, 1992.

[14] Abha Moitra. Scheduling of hard real-time systems. In *Proceedings of the Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 362–381, London, UK, 1986. Springer-Verlag.

[15] Marius Nagy and Selim G. Akl. Quantum measurements and universal computation. *International Journal of Unconvantional Computing*, 2(1):73–88, 2006.

[16] H. T. Siegelmann. *Neural Networks and Analog Computation: Beyond the Turing limit.* Birkhäuser, Boston, 1999.

[17] M. Stannet. X-machines and the halting problem: Building a super-Turing machine. *Formal Aspects of Computing*, 2(4):331–341, 1990.

[18] P. Wegner and D. Goldin. Computation beyond Turing Machines. *Communications of the ACM*, 46(4):100–102, May 1997.

[19] Craig Wehrenberg and Tracey Mulhall-Wehrenberg. *The Best-Kept Secret to Raising a Healthy Child... and the Possible Prevention of Sudden Infant Death Sindrome (SIDS).* Specific Chiropractic, 2000.