Technical Report 2011-580*
Time Indeterminacy, Non-Universality in Computation, and the Demise of the Church-Turing Thesis†

Naya Nagy and Selim G. Akl

School of Computing, Queen's University

Kingston, Ontario K7L 3N6

Canada

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

August 19, 2011

Abstract

It is known that there exist computational problems that can be solved on a parallel computer, yet are impossible to be solved sequentially. Specifically, no general purpose sequential model of computation, such as the Turing Machine or the Random Access Machine, can simulate a large family of computations (for example, solutions to certain real-time problems), each of which is capable of being carried out readily by a particular parallel computer. We extend 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 the input data available. A second type of time constraints refers to uncertain *deadlines* on when outputs are to be produced.

Our main objective 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 here, prove once more that it is impossible to define a 'universal computer', that is, a computer able to perform (through simulation or otherwise) all computations that are executable on other computers.

Finally, one of the contributions of this paper is to promote the study of a topic, conspicuously absent to date from theoretical computer science, namely, the role of physical time and physical space in computation. The focus of our work is to analyze the effect of external natural phenomena on the various components of a computational process, namely, the input phase, the calculation phase (including the *algorithm* and the computing *agents* themselves), and the output phase.

Keywords: real-time computation, unconventional computation, Turing Machine, universal computer, parallel computing, physical time, physical space, external phenomena.

*This is an updated and extended version of Technical Report 2008-553.

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

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 have 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* [11, 12] when computations are performed under time constraints.

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

Phase 1. *Input:* Data are read from the outside world.

Phase 2. *Calculation:* Some operations are performed with these data.

Phase 3. *Output:* The result data are written out.

Most often, computational time constraints refer to phase 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 [11]) or less valuable (the deadline is soft [12]). The implications of a time deadline on the capacity of Turing-type computational models to simulate each other are studied in [4], where it is proven that a parallel model is theoretically superior to a sequential model.

Phase 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 data [4],

2. Interdependent data [4], and

3. Accumulating data [10].

Phase 2 is performed internally by the computer and is less likely to be directly under time constraints. To our knowledge, computations in which the calculation phase is subject to time constraint have received scant attention to date from the research community. Nevertheless, it should be noted that time can affect the internal calculations of a computer, if the latter is considered as part of its environment, rather than operating as a closed system. An obvious example in this category is that computers are able to work as long as they have a power supply. If the power supply is time dependent, then internal calculations are also constrained by this dependency. Other examples include calculations whose running time complexity is a function of the physical time at which they are executed, and calculational agents whose behavior changes with physical time [6]. Time constraints on phase 2 will be the topic of a future study.

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 is significantly limited in comparison with the parallel machine and therefore fails where the latter succeeds.

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.

One of the consequences of our work is to demonstrate, yet again, that the concept of a universal computer cannot be realized. No computer, fixed a priori, can successfully perform the computations described in this paper, even if given an unbounded amount of memory and time, as well as the ability to interact with the outside world. This result on the non-universality in computation, aside from exposing a fundamental principle, opens a new area of investigation into the role of time and space in computing. We note here that in conventional computing, both time and space are:

1. Internal to the computer (time is *running time*, space is *memory space*, neither is related to the outside world, and both are valid only when performing a computation);

2. Static (time depends on the algorithm while nothing depends on time, and placement of data in memory does not affect the results);

3. Unbounded (at least in theory, it is assumed that the computer has as much memory, and can take as much time, as it will ever need in order to complete its computations);

4. Entirely under the control of the user (a better algorithm improves time and/or space requirements, and often there is a time-space tradeoff); and

5. Solely used in performance analysis.

In unconventional computing, by contrast, time and space acquire a new significance. As the examples in this paper (among many others [4, 5, 6, 7]) show, physical time and physical space in the environment external to the computer play a crucial role in the definition and requirements of the computational problems, and ultimately in the success or failure of the computation. The present paper proposes to begin an exploration of the role of physical time and physical space in computing and the properties of computational problems that depend on them. This will introduce new paradigms to the theory of computation and promote the study of problems whose importance can only grow with the widespread and diverse uses of computers. Preliminary results on this topic were presented in [8]. There it was shown that "computational ubiquity" is required in several important computations.

It is important to emphasize here that there exist many computational paradigms that have the appearance of being in some way or another related to *time*. These include, for example, on-line computing, π-calculus, trace theory, streaming algorithms, stochastic scheduling, neural networks, complex systems, time-relativistic computation, hyperbolic cellular automata, adaptive algorithms, learning algorithms, genetic algorithms, evolutionary algorithms, emergent algorithms, reconfigurable circuits, and of course, real-time algorithms. The crucial observation here is that 'time' in *all* of these paradigms is *computer time*, that is, time internal to the machine currently performing the computation. In *none* of these paradigms does the computer have any awareness of true physical time in its environment, or of the physical space surrounding it, and therefore *none* of the external physical phenomena taking place during the computation has (or is supposed to have) any direct influence on the variables, the algorithm, the machine itself, or its output.

The remainder of this 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. The models of computation used by our algorithms are defined in Section 3. In Section 4 we propose 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 offers a discussion of those issues common to the problems presented in this paper. Finally, we conclude in Section 8 with suggestions for future work.

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 a program, consisting of input, calculation, and output. 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. By contrast, *dynamic tasks* arrive to the scheduler during the computation in an unpredictable way.

The time requirements of a task are defined either as the time constraints on the input of data, or as the deadlines for the output of results, or in general as any time constraints 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 [2, 11, 12, 28], the time constraints are specified 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 on 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, as 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 on, or vary over time, nor are they influenced by the execution of a task or by the scheduler itself. Descriptions of how to schedule static tasks are given in [11, 28].

		Tasks	
		Static	Dynamic
Time requirements	Static	[11, 28]	[2, 12]
	Dynamic	Studied in this paper	Not yet studied

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

The **second scenario** in Table 1 is defined by dynamic tasks and static time requirements. Dynamic systems exhibiting dynamic scheduling problems are treated in [2, 12]. 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 for producing results 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 for producing 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, thus creating one of the following situations:

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

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

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

4. The deadline has 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.

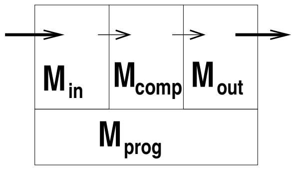

Figure 1: The four divisions of the memory.

3 Models of Computation

In this section we define the models of computation, as well as the fundamental notion of a *time unit*, to be used in our analyses. In addition, we review a recent result on non-universality in computation for which further evidence is provided in this paper.

3.1 The Sequential Machine

The sequential computational model used in this paper is the standard Random Access Machine (RAM) [15]. 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 moment the input data are available, that is, the time at which the input is written from the outside world into the input memory M_{in}, is not under the control of the RAM. Likewise, special time requirements apply to M_{out}, namely, a specification of the time at which the output data are supposed to be written into M_{out}, and thus 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. Read an input datum from the outside world and store it into the input memory M_{in}.

2. Perform an operation on one or two values of the memory M and write 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. We elaborate on this point in Section 3.3. The standard RAM is extended in Section

4 to take into consideration time constraints imposed on the computation by the external environment. We refer to the extended model as a time-aware RAM.

3.2 The Parallel Machine

The parallel algorithms described in the next sections are meant to run on a Parallel Random Access Machine (PRAM) [24]. 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 units M_{in}, M_{comp}, M_{out}, and M_{prog}. Several processors can read from the same memory location, an operation referred to as a *concurrent read*. Also, several processors may write into the same memory location, thereby executing a *concurrent write*. The convention adopted in this paper for the concurrent write, is that the values, simultaneously written in a memory location, are added together and their sum is stored in that memory location.

For the PRAM, all processors simultaneously execute one computational step in each time unit. Since the PRAM is viewed as a collection of RAMs, the time-aware RAM leads to the time-aware PRAM.

3.3 The Time Unit

All analyses in this paper are based on using a time unit to measure the running time of an algorithm. We define a time unit as the time taken by the RAM to execute a step in a computation. As mentioned earlier but worth repeating, an algorithmic step, in turn, is defined as consisting of three phases: reading a fixed-size input, performing a basic calculation (that is, an elementary arithmetic or logical operation, such as addition or logical AND) on one or two data elements, and finally producing a fixed-size output.

Furthermore, in order to avoid any possible confusion, we make two explicit assumptions:

1. The RAM operates at the speed of light in vacuum; in other words, there does not exist a sequential model of computation faster than the RAM.

2. Each processor on the PRAM is a RAM operating at the speed of light in vacuum; in other words, there does not exist a model of computation faster than the PRAM.

These two assumptions allow us to avoid comments of the type: "Why not use a faster sequential computer that outperforms the parallel computer?".

3.4 Non-Universality

Almost all the other fellows
do not look from the facts to the theory
but from the theory to the facts;
they cannot get out of the network of already accepted concepts;
instead, comically, they only wriggle about inside.
Albert Einstein in a letter to Erwin Schrödinger, August 8, 1935,
quoted by Louisa Gilder in: The Age of Entanglement, Vintage Books, 2009, p. 170.

One of the purposes of this paper is to provide further examples in support of the previously established theoretical result whereby a universal model of computation does not exist [4]–[8]. So, what is a 'universal computer'? The simplest and most appropriate definition is that a 'universal computer' is one capable of performing in finite time, through simulation, any computation that is possible on any other computer. The 'universal computer' is such that once it has been defined, it is fixed once and for all; one cannot go back at a later time and change its specifications. The following quotes from well respected sources elaborate on the above definition, without caveat, exception, or qualification (other similar quotes abound in the literature, and a selection is provided in [3]):

> "It can also be shown that any computation that can be performed on a modern-day digital computer can be described by means of a Turing machine. Thus if one ever found a procedure that fitted the intuitive notions, but could not be described by means of a Turing machine, it would indeed be of an unusual nature since it could not possibly be programmed for any existing computer." [23]

> "[...] as primitive as Turing machines seem to be, attempts to strengthen them seem not to have any effect.
>
> [...] Thus any computation that can be carried out on the fancier type of machine can actually be carried out on a Turing machine of the standard variety. [...] *any* way of formalizing the idea of a 'computational procedure' or an 'algorithm' is equivalent to the idea of a Turing Machine. [...]
>
> It is theoretically possible, however, that Church's Thesis could be overthrown at some future date, if someone were to propose an alternative model of computation that was publicly acceptable as fulfilling the requirement of 'finite labor at each step' and yet was provably capable of carrying out computations that cannot be carried out by any Turing machine. No one considers this likely. " [25]

> "[...] if we have shown that a problem can (or cannot) be solved by any TM, we can deduce that the same problem can (or cannot) be solved by existing mathematical computation models nor by any conceivable computing mechanism. The lesson is: Do not try to solve mechanically what cannot be solved by TMs!" [26]

> "The interesting thing about a universal Turing machine is that, for any well-defined computational procedure whatever, a universal Turing machine is capable of simulating a machine that will execute those procedures. It does this by reproducing exactly the input/output behaviour of the machine being simulated." [14]

> "[...] any algorithmic problem for which we can find an algorithm that can be programmed in some programming language, *any* language, running on some computer, *any* computer, even one that has not been built yet but *can* be built, and even one that will require unbounded amounts of time and memory space for ever-larger inputs, is also solvable by a Turing machine." [20]

> "A Turing machine can do everything that a real computer can do." [32]

> "It is possible to build a universal computer: a machine that can be programmed to perform any computation that any other physical object can perform." [17]

> "[...] any computation that can be performed by any physical computing device can be performed by any universal computer, as long as the latter has sufficient time and memory." [21]

> "What is important is that on the basis of Turing's analysis of the notion of computation, it is possible to conclude that anything computable by any algorithmic process can be computed by a Turing machine. So if we can prove that some particular task cannot be accomplished by a Turing machine, we can conclude that no algorithmic process can accomplish that task." [16]

> "But theoretically, anything one computer is capable of doing, another computer is also capable of doing. Given the right instructions, and sufficient memory, the computer found in your fridge could, for example, simulate Microsoft Windows. The fact that it might be ridiculous to waste time using the embedded computer in your fridge to do anything other than what it was designed for is irrelevant - the point is that it obeys the same model of computation as every other computer and can therefore - by hook or by crook - eventually achieve the same result.
>
> This notion is based on what is now called the Church-Turing thesis (dating back to 1936), a hypothesis about calculation devices, such as electronic computers. Alan Turing and Alonzo Church introduced the formalization of an algorithm and a 'purely mechanical' model (which assumes that sufficient time and storage space are available). This leads us to the notion of a universal computer, on which all modern computers are based." [34]

Several models of computation have been claimed to be 'universal', such as the Turing Machine, the Random Access Machine, the Cellular Automaton, and so on. In fact, none is. Contrary to the above quotes, no universal model is possible, even if it is provided with an unlimited supply of time and memory, and the ability to interact with the outside world. Indeed, for any model of computation X that purports to be universal, there exists a computational problem Y that X fails to solve, yet Y is capable of being solved on another model of computation X'. For example, if X (being a putative 'universal computer' and thus fixed once and for all) is capable of z operations per time unit, and Y requires for its solution that z' operations be executed per time unit, where $z' > z$, then X fails to solve Y. If X', on the other hand, is capable, by definition, of performing z' operations per time unit, then X' succeeds to solve Y. However X', in turn, fails when presented with a computation necessitating z'' operations per time unit, where $z'' > z'$. The process continues indefinitely.

3.4.1 An Example

The journey is the reward, not the destination.
Proverb

The following problem illustrates non-universality in computation. For a positive even integer n, where $n \geq 8$, let n distinct integers be stored in an array A with n locations $A[0]$,

$A[1]$, ..., $A[n-1]$, one integer per location. Thus $A[j]$, for all $0 \leq j \leq n-1$, represents the integer currently stored in the jth location of A. It is required to sort the n integers in place into increasing order, such that:

1. After step i of the sorting algorithm, for all $i \geq 1$, no three consecutive integers satisfy:

$$A[j] > A[j+1] > A[j+2]$$

for all $0 \leq j \leq n-3$.

2. When the sort terminates we have:

$$A[0] < A[1] < \cdots < A[n-1].$$

This is the standard sorting problem in computer science, but with a twist. In it, the journey is more important than the destination. While it is true that we are interested in the outcome of the computation (namely, the sorted array, this being the *destination*), in this particular variant we are more concerned with *how* the result is obtained (namely, there is a condition that must be satisfied throughout all steps of the algorithm, this being the *journey*). It is worth emphasizing here that the condition to be satisfied is germane to the problem itself; specifically, there are no restrictions whatsoever on the model of computation or the algorithm to be used. Our task is to find an algorithm for a chosen model of computation that solves the problem exactly as posed. One should also observe that computer science is replete with problems with an inherent condition on how the solution is to be obtained. Examples of such problems include: inverting a nonsingular matrix without ever dividing by zero, finding a shortest path in a graph without examining an edge more than once, sorting a sequence of numbers without reversing the order of equal inputs, and so on.

An algorithm for an $n/2$-processor PRAM solves the aforementioned variant of the sorting problem handily by means of pairwise swaps applied to the input array A, during each of which $A[j]$ and $A[k]$ exchange positions (using an additional memory location for temporary storage). The RAM, and a PRAM with fewer than $(n/2) - 1$ processors, both fail to solve the problem consistently, that is, they fail to sort all possible $n!$ permutations of the input while satisfying, at every step, the condition that no three consecutive integers are such that $A[j] > A[j+1] > A[j+2]$ for all j. In the particularly nasty case where the input is of the form

$$A[0] > A[1] > \cdots > A[n-1],$$

any RAM algorithm and any algorithm for a PRAM with fewer than $(n/2) - 1$ processors fail after the first swap.

It is interesting to note here that a Turing Machine with $n/2$ heads succeeds in solving the problem, yet its simulation by a standard (single-head) Turing Machine fails to satisfy the requirements of the problem. Indeed, suppose that the standard Turing Machine is presented with the input sequence $A[0] > A[1] > \cdots > A[n-1]$:

1. It will either use the given representation of the input, and proceed to perform an operation (a swap, for example), in which case it would fail after one step of the algorithm,

2. Or it will transform the given representation into a different encoding (perhaps one intended to capture the behavior of the Turing Machine with $n/2$ heads) in preparation for the sort, in which case it would again fail since the transformation itself constitutes an algorithmic step.

This is a surprising result as it goes against the common belief that any computation by a variant of the Turing Machine can be effectively simulated by the standard model [25].

The variant of sorting described here belongs to a general class of problems on which a mathematical constraint is imposed that needs to be satisfied throughout the computation. (Analogies from popular culture include picking up sticks from a heap one by one without moving the other sticks, drawing a geometric figure without lifting the pencil, and so on.) It is useful to point out that no machine that claims to be 'universal' can solve problems of this class, even if such a machine were endowed with the formidable power of time travel [9].

Though abundantly clear from the preceding analysis, we state the following conclusion explicitly: The sorting variant problem definitively disproves the famous Church-Turing Thesis ("everything computable is computable by a Turing machine" [3, 32, 37].) In fact, this counterexample, not only proves that the Turing machine is not universal, but in fact that no computer that performs a finite, and a priori fixed, number of operations per time unit is universal. This non-universality result applies to all models of computation, sequential as well as parallel, theoretical as well as practical, conventional as well as unconventional [4]-[7].

4 A Time-Aware Model of Computation

Computation is physical;
it is necessarily embodied in a device
whose behaviour is guided by the laws of physics
and cannot be completely captured by a closed mathematical model.
This fact of embodiment is becoming ever more apparent
as we push the bounds of those physical laws.
Susan Stepney, Journeys in non-classical computation,
in: T. Hoare and R. Milner, Eds.,
Grand Challenges in Computing Research,
Swinson, BCS, 2004, pp. 29–32.

We describe a new model of computation that takes into consideration the time at which an operation is executed. This model is principally a formalization of the ideas presented in Section 3.1. It will allow us to state computational problems where external physical phenomena impose time constraints on the input and on the output. Algorithmic solutions to such problems will be expressed in the rest of the paper using the formalism introduced in this section.

The new time-aware model of computation is essentially a RAM 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 of unbounded 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 $i \bmod 4 \neq 0$ then no input is read. For the purposes of this paper, we assume for simplicity that x may hold a simple arithmetic and/or logical expression to determine, when necessary, which of two input values is to be 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}. Otherwise, k, l, or O_r is ignored.

   (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}. Otherwise, O_r is ignored.

3. (CP, M, k, l) means that a value from $M(k)$ is copied to $M(l)$, where $k \bmod 4 = 0$ or 1, and $l \bmod 4 = 2$. This operation will be used to transfer data from M_{in} or M_{comp} to M_{out}. For the purposes of this paper, we assume for simplicity that k may hold a simple arithmetic and/or logical expression to compute, if necessary, the value of k.

4. (W, M_{out}, s) means that the value of $M_{\text{out}}(s)$ is sent to the outside world, provided that $s \bmod 4 = 2$.

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. When several feasible schedules exist, one is chosen arbitrarily. 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 computations, 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?

Never having been very fond of gurus ...
I could see that [they] commanded nature
to behave in accordance with the models.
Erwin Chargaff, Building the tower of babble,
Nature 248, 1972, p. 778.

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 [6]. 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 (in the sense of physical) time into consideration. The same limitation applies to certain 'timed' concurrent calculi that appear to use some kind of clocked synchronization, when in fact their clocks are internal artifacts, unrelated to the outside physical world. 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 [19], 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 [4, 5, 7], 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 physical time as an explicit and necessary parameter of the computation can also be formalized on this model, but not on the Turing Machine, which by definition has no access to the outside world. 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 [4] in which input data are unstable and vary with time.

2. Problems with time dependent complexity [10] 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 that occur *during* the execution of a task. Henceforth we use the new model in one of its two incarnations mentioned earlier, namely, the time-aware RAM and the time-aware PRAM.

5 Dynamic Time Requirements on the Acquisition of Data

If faced with a task to be executed, we are accustomed to knowing *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 happens *during* the task's computation.

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 does not give any clues about who caused the damage to 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 cars, a collision occurs. The problem is to find the cause of the accident, that is, the cars responsible for the unexpected 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 [36]. 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 devastating natural phenomenon, such as an earthquake, the problem is to best describe the conditions and parameters that allow us to predict the disaster.
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 computer equipped with a sensor has to monitor the environment in which it is located. The environment is defined by two different parameters, namely, temperature T and pressure

p. These parameters vary unpredictably with time t; thus, T and p are expressed as functions of t, that is, $T = T(t)$ and $p = p(t)$.

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

5.1.1 Requirements

The computer 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 computation, that is, at time t_0, the computer does not yet know 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, specifically,

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

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. In addition, it is not known at the beginning of the computation which of the two parameters is to be measured. This information is represented by a binary value *which* and will also be obtained later on during the computation. If $which = 0$ then the sensor has to measure the temperature and if $which = 1$ then the sensor 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}, suddenly several variables will make the computation of t_{input} and *which* possible. Three variables denoted x_0, x_1, and x_2 are now available in the environment for the computer to read in. In this example, the number of variables has been set to three, in order for it to be smaller than the six time units left for computation, from t_{middle} to t_f, inclusive. This will give the sequential computer some small chance to successfully complete the task. If there are more variables, the sequential computer will never be successful.

The time t_{input} and the binary *which* will be computed from the sum of x_0, x_1, and x_2 according to the following formulas:

$$t_{\text{input}} = (x_0 + x_1 + x_2) \bmod (length + 1)$$

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

where, in this example, $length + 1 = 11$. Clearly, 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 show in the following subsections that the type of the computer employed to solve the problem will make the difference between success and failure.

5.1.2 Idle Sequential Solution

The first algorithmic solution runs on a sequential computer, that is, on a time-aware RAM. The latter does not start monitoring its environment unless it fully knows what to do. In this case, the computer is working "on request", or according to the principle: *Don't do anything unless you absolutely have to ...*

As specified earlier, the RAM is able to read a fixed-size input, perform a fixed-size calculation, and produce a fixed-size output, 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 5 time units to compute the values for t_{input} and *which*. The following situations can happen:

1. $t_{\text{input}} > t_{\text{middle}} + 5 - 1$. 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. $t_{\text{middle}} \leq t_{\text{input}} < t_{\text{middle}} + 5$. 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 computational steps performed using the time aware model:

$$\begin{aligned}
I_0 &= ((R, M_{\text{in}}, 0, x_0), -, -, (5,5)) \\
I_1 &= ((R, M_{\text{in}}, 4, x_1), (+, M, 0, 4, 1), -, (6,6)) \\
I_2 &= ((R, M_{\text{in}}, 8, x_2), (+, M, 1, 8, 1), -, (7,7)) \\
I_3 &= (-, (\text{mod } 2, M, 1, 10, 5), -, (8,8)) \\
I_4 &= (-, (\text{mod } 11, M, 1, 10, 9), -, (9,9)) \\
I_5 &= ((R, M_{\text{in}}, 12, (1 - M(5)) \times T + M(5) \times p), \\
&\qquad (CP, M, 12, 2), (W, M_{\text{out}}, 2), (10,10)).
\end{aligned}$$

This computer has little chance of success, as it will try to get input only after all information concerning the input, namely, t_{input} and *which*, is computed. This leaves only $t_f - t_{\text{middle}} - 5 + 1$ time units at the end, in which the environment is actually listened to. If t_{input} falls in this range, the computer is successful, otherwise it fails.

The overall success rate of this computer is:

$$\textit{success} = (t_f - t_{\text{middle}} - 5 + 1)/(t_f - t_0) = 1/10 = 10\%.$$

5.1.3 Active (Smart) Sequential Solution

In this case, the computer will try to do better by anticipating the request. It 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 computer is again a time-aware RAM as in the previous case. Instead of being idle while waiting for the request, it busily measures the environment. This is done from the moment the computation starts at t_0, until the request is received at t_{middle}. Because the parameter of interest is unknown, the best the computer can do is to choose just one parameter arbitrarily, 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. $t_{\text{input}} > t_{\text{middle}} + 5 - 1$. The computer is able to measure the required environment parameter and output the value of interest after it received the request.

2. $t_{\text{middle}} \leq t_{\text{input}} < t_{\text{middle}} + 3$. 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 computational steps performed by the time aware model are given below:

$$
\begin{aligned}
I_0 &= ((R, M_{\text{in}}, 0, T), -, -, (0,0)) \\
I_1 &= ((R, M_{\text{in}}, 4, T), -, -, (1,1)) \\
I_2 &= ((R, M_{\text{in}}, 8, T), -, -, (2,2)) \\
I_3 &= ((R, M_{\text{in}}, 12, T), -, -, (3,3)) \\
I_4 &= ((R, M_{\text{in}}, 16, T), -, -, (4,4)) \\
I_5 &= ((R, M_{\text{in}}, 20, x_0), -, -, (5,5)) \\
I_6 &= ((R, M_{\text{in}}, 24, x_1), (+, M, 0, 4, 1), -, (6,6)) \\
I_7 &= ((R, M_{\text{in}}, 28, x_2), (+, M, 1, 8, 1), -, (7,7)) \\
I_8 &= ((R, M_{\text{in}}, 32, T), (\bmod 2, M, 1, 10, 5), -, (8,8)) \\
I_9 &= ((R, M_{\text{in}}, 36, T), (\bmod 11, M, 1, 10, 9), -, (9,9)) \\
I_{10} &= ((R, M_{\text{in}}, 40, (1 - M(5)) \times T + M(5) \times p), \\
&\qquad (CP, M, M(9) \times 4, 2), (W, M_{\text{out}}, 2), (10, 10)).
\end{aligned}
$$

This computer has a better chance of being successful. If the temperature is indeed required, the computer's chances are high. The exact chance of success is $(t_f - t_0 - 3)/(t_f - t_0) = 7/10$. If the value of the pressure is required, the success rate is the same as for the idle solution $(t_f - t_{\text{middle}} - 5 + 1)/(t_f - t_0) = 1/10$.

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

$$
success = \frac{1}{2} \times \left(\frac{t_f - t_0 - 3}{t_f - t_0} + \frac{t_f - t_{\text{middle}} - 5 + 1}{t_f - t_0}\right) = \frac{1}{2}\left(\frac{7}{10} + \frac{1}{10}\right) = 40\%.
$$

5.1.4 Parallel Solution

The last and most successful algorithmic solution is offered by a parallel computer. The latter 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 parallel computer is a time-aware PRAM, that is, a collection of time-aware RAMs operating synchronously. It can perform several steps (each executed by a distinct processor, and each consisting of a measurement, a computation, and an output) simultaneously, in one time unit. For our example, it needs to have three processors, denoted P_0, P_1, and P_2.

The PRAM can measure both the temperature and the pressure of the environment, at the same time. It 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. $t_{input} > t_{\text{middle}} + 4 - 1$. The computer will measure the requested parameter at t_{input} and write it out at the end of the computation.

2. $t_{input} < t_{\text{middle}} + 4$. The computer inspects its recorded history of the environment and outputs the desired parameter.

The computational steps performed using the time aware model are shown in Table 2. This monitor always answers the request successfully. Therefore, the success rate is:

$$success = 100\%.$$

It should be noted here that, in general for n monitored parameters, the number of processors on the parallel computer needs to be at least $n + 1$.

6 Dynamic Time Requirements on the Output of Data

This section explores the situation where the deadline for a task to produce its output 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.

6.1 Requirements

A computer equipped with a sensor is required to measure some parameter of the environment. For definiteness, consider the parameter to be the temperature T. The computer 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 computer 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,

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

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

Time t_{output} is not given at the outset, but has to be calculated by the computer according to the following rules. The n input variables x_0, x_1, …, x_{n-1} are available in the environment throughout the computation. The output time is defined as:

$$t_{\text{output}} = t_0 + \delta + [(x_0 + x_1 + \cdots + x_{n-1}) \bmod \ length] + 1$$

The problem will be solved on a sequential computer and on a parallel computer. This will show again that the size of the machine matters. A sequential machine, or a parallel

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	$P_0 : ((R, M_{\text{in}}, 24, T), -, -, (6, 6))$ $P_1 : ((R, M_{\text{in}}, 124, p), -, -, (6, 6))$ $P_2 : ((R, M_{\text{in}}, 204, x_1), (+, M, 200, 204, 1), -, (6, 6))$
I_7	$P_0 : ((R, M_{\text{in}}, 28, T), -, -, (7, 7))$ $P_1 : ((R, M_{\text{in}}, 128, p), -, -, (7, 7))$ $P_2 : ((R, M_{\text{in}}, 208, x_2), (+, M, 1, 208, 1), -, (7, 7))$
I_8	$P_0 : ((R, M_{\text{in}}, 32, T), (\bmod 2, M, 1, 10, 5), -, (8, 8))$ $P_1 : ((R, M_{\text{in}}, 132, p), (\bmod 11, M, 1, 10, 9), -, (8, 8))$ $P_2 : (-, -, -, (8, 8))$
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}	$P_0 : ((R, M_{\text{in}}, 40, T), -, -, (10, 10))$ $P_1 : ((R, M_{\text{in}}, 140, p), -, -, (10, 10))$ $P_2 : (-, (CP, M, ((1 - M(5)) \times (M(9) \times 4)) + M(5) \times (100 + M(9) \times 4), 2), (W, M_{\text{out}}, 2), (10, 10))$

Table 2: Computational steps of the time aware model.

machine without adequate 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 $n = 10$.

6.2 Sequential Solution

The sequential computer is a time-aware RAM. It measures the temperature at t_0 and then proceeds to read the input variables x_0, x_1, …, x_9. By the time the computer has managed to calculate the deadline t_{output}, that moment has already passed. Therefore, the sequential computer will not be able to output the result, for no other reason than the fact that it did not know the deadline in time.

The RAM follows the steps below (all feasible schedules are equivalent in this case):

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

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 before 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.

6.3 Parallel Solution

The parallel computer is a time-aware PRAM. It has $n + 1 = 11$ processors, P_0, P_1, P_2, …, P_{10}, in order to be able to perform all measurements at the very beginning, that is, at t_0. The 10 inputs x_0, x_1, …, x_9 are read simultaneously and concurrently written in $M_{\text{in}}(0)$. As a result, $M_{\text{in}}(0)$ now contains the sum $x_0 + x_1 + \cdots + x_9$ (as specified in Section 3.2). 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	$P_0 : ((R, M_{\text{in}}, 0, x_0), -, -, (0,0))$ $P_1 : ((R, M_{\text{in}}, 0, x_1), -, -, (0,0))$ $\ldots$ $P_9 : ((R, M_{\text{in}}, 0, x_9), -, -, (0,0))$ $P_{10} : ((R, M_{\text{in}}, 4, T), -, -, (0,0))$
I_1	$P_0 : (-, (\bmod 8, M, 0, 10, 1), -, (1,1))$ $P_1 : -$ $P_2 : -$ $\ldots$ $P_{10} : -$
I_2	$P_0 : (-, (CP, M, 4, 2), (W, M_{\text{out}}, 2), (M(1)+3, M(1)+3))$ $P_1 : -$ $\ldots$ $P_{10} : -$

It can be seen that the parallel computer can meet the deadline because it is able to compute t_{output} 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 $\lfloor (n+1)/2 \rfloor = 5$, 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

Facts do not go away when scientists
debate rival theories to explain them.
Stephen Jay Gould,
Evolution as fact and theory,
Discover, Vol. 2, 1981, pp. 34–37.

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 these time requirements. This 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

Although estimates differ, it is clear that the end of Moore's Law is in sight;
there are physical limits to the density of binary logic devices

and to their speed of operation. ...
Traditionally, a sort of Cartesian dualism has reigned in computer science;
programs and algorithms have been conceived as idealized mathematical objects;
software has been developed, explained, and analyzed independently of hardware;
the focus has been on the formal rather than the material.
Post-Moore's Law computing, in contrast,
because of its greater assimilation to physics,
will be less idealized, less independent of its physical realization.
Bruce J. MacLennan, Bodies –both informed and transformed:
Embodied computation and information processing,
in: G. Dodig-Crnkovic and M. Burgin, Eds.,
Information and Computation, World Scientific, 2011.

All founders of the field of computer science were mathematicians. One thinks of Alan Turing, Alonzo Church, Stephen Kleene, Kurt Gödel, Emil Post, John von Neumann, and others. It was natural for them, therefore, to define 'computation' as 'function evaluation', that is, 'Given x, calculate $f(x)$', where x is a number, usually an integer, and $f()$ is a 'function' in the strict mathematical sense of the word. This is exactly what the Turing Machine does, namely, evaluate functions over the integers in the same way as a mathematician would. For three quarters of a century, theoretical computer science has remained fixated on this definition of 'computation'.

There are two problems with equating computation with the evaluation of a mathematical function of a given integer. First, this definition reduces the celebrated Church-Turing Thesis to a vacuous tautology: If 'to compute' is what the Turing Machine does, then obviously the Turing Machine can compute everything that is computable! Second, while the function evaluation model is simple, elegant, and precise, it is also practically useless in providing a complete description of what it really means 'to compute' in the real world. The conventional definition of computation as 'what the Turing Machine does' is too narrow, too limiting, and too shortsighted to be able to capture the complexity and richness of the vast universe in which we live. Computation is far too important, too fundamental, and too pervasive a concept to be restricted to such a weak definition.

There are signs, however, that this view is beginning to change. A glance at the literature [1, 4, 13, 17, 18, 27, 30, 31, 33, 35] reveals a growing realization that there is a lot more to 'computation' than simply 'function evaluation'. Today, a new understanding of what it means 'to compute' is emerging. Computation is no longer regarded as just a process that takes place in a closed box impervious to the phenomena that surround it. Rather, it is being suggested that computation is any form of information processing conducted in the physical world, whether prompted by humans or occurring naturally.

Using mathematical notation, computation is a transformation $\mathcal{T}$ applied to n spatially and temporally related physical variables $x_0, x_1, \ldots, x_{n-1}$. These variables are not necessarily "given" to $\mathcal{T}$ at the outset, and the latter, far from being oblivious to its environment, operates under the control of surrounding physical phenomena in space and time. Thus, for example, the x_i themselves may vary with physical time, the x_i themselves may interact in physical space, the complexity of evaluating $\mathcal{T}(x_i)$ may vary with physical time, the complexity of evaluating $\mathcal{T}(x_i)$ may depend on the order in which it is evaluated, $\mathcal{T}(x_i)$ may have to satisfy a global mathematical constraint in physical space and time, and so on. None of these computations fits in the narrow notion of mathematical function evaluation.

This definition of 'to compute' as 'the process of transforming information' applies to all three phases of computation:

1. The input phase, where data such as keystrokes, mouse clicks, or temperatures are reduced, for example, to binary strings;

2. The calculation phase, where data are manipulated through arithmetic and logic operations, as well as operations on strings, and so on; and

3. The output phase, where data are produced as numbers on a screen, or rendered as images and sounds, for instance.

Each of the three phases is an integral part of every computation, and no computation worthy of that name can exclude any of them. In particular, input and output, the subject of this paper, are fundamental in the design and analysis of algorithms. Input-sensitive and output-sensitive computations often play an especially important role in deriving lower and upper bounds on algorithmic complexity. One of the most dramatic illustrations of the interweaving of input and output with calculation is the well-known linear-time planarity testing algorithm [22]. In order to determine whether a graph with n vertices is planar, Step 1 of that algorithm avoids a potentially $\Omega(n^2)$-time computation, by reading in the edges of the graph *one at a time* from the outside world; if there is one more edge than $3n-6$, the graph is declared non-planar.

Within this context, the examples in this paper describe computations that are unconventional in the sense that their time constraints are themselves difficult to compute. These examples show that a parallel computer of an appropriate size is essential for the successful completion of some computational task. In particular, a standard Turing Machine is not able to perform such tasks successfully. Furthermore, the Turing Machine cannot simulate a more powerful machine capable of carrying these tasks to completion. This confirms, once again, the previously established result [13, 18, 29, 31, 33, 35] 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 identified 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. Specifically, 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 [4], 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. In what follows, we propose some directions.

1. An obvious feature of computing systems is that if a system needs to perform another N operations to compute a value, and can perform at most n operations per time unit, where $n \leq N$, then it can complete the computation only if its deadline is at least $\lceil N/n \rceil$ time units away. For basic computations this is already enough to explain why parallel computers do better than sequential ones, and why more processors are better than fewer: When the N operations are independent, and more processors are available, the value of n is greater. As we show in this paper, however, the question is more subtle than this, since we consider 'self-referential' situations in which the value of N includes the number of steps needed to compute N itself. This suggests a wide range of subsidiary questions that might be tackled. For example, suppose that a computation consists of two parts, namely, computing a non-negative integer N, then performing N elementary operations. The overall complexity is the sum of the complexities of the two parts. In some cases, like the ones described in the present paper, performing the first part efficiently leads to a successful computation. In general: When and how does the overall complexity of a computation depend on the complexity of 'self-referentially' computing the number of operations it needs to perform?

2. A formalization of schedulers of dynamic tasks with dynamic time requirements would lead to settings that are more general than the one described here.

3. As demonstrated previously in [4], it is not always the case that observations are independent of one another. For example, measuring temperature may cause pressure to change. What will be the effect of such behavior on our parallel implementation?

4. 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, that is, the calculation phase, which may be subject to constraints, perhaps affected by the computation itself.

These questions are prime candidates for future investigations.

References

[1] A. Adamatzky and S.G. Akl, Trans-Canada Slimeways: Slime mould imitates the Canadian transport network, to appear in *International Journal of Natural Computing Research.*

[2] L. Abeni and G. Buttazzo, Resource reservation in dynamic real-time systems, *Real-Time Systems*, Vol. 27, No. 2, 2004, pp.123–167.

[3] S.G. Akl, Universality in computation: Some quotes of interest, Technical Report No. 2006-511, School of Computing, Queen's University, Kingston, Ontario, April 2006, 12 pages.

[4] S.G. Akl, Three counterexamples to dispel the myth of the universal computer, *Parallel Processing Letters*, Vol 16, No. 3, 2006, pp. 381–403.

[5] S.G. Akl, Even accelerating machines are not universal, *International Journal of Unconventional Computing*, Vol. 3, No. 2, 2007, pp. 105–121.

[6] S.G. Akl, Evolving computational systems, Chapter One in: S. Rajasekaran and J.H. Reif, Eds., *Handbook of Parallel Computing: Models, Algorithms, and Applications*, Taylor and Francis, CRC Press, Boca Raton, Florida, 2008, pp. 1–22.

[7] S.G. Akl, Unconventional computational problems with consequences to universality, *International Journal of Unconventional Computing*, Vol. 4, No. 1, 2008, pp. 89–98.

[8] S.G. Akl, Ubiquity and simultaneity: The science and philosophy of space and time in unconventional computation, Invited talk, *Conference on the Science and Philosophy of Unconventional Computing*, The University of Cambridge, Cambridge, United Kingdom, 2009.

[9] S.G. Akl, Time travel: A new hypercomputational paradigm, *International Journal of Unconventional Computing*, Vol. 6, No. 5, 2010, pp. 329 - 351.

[10] S.D. Bruda and S.G. Akl, The characterization of data-accumulating algorithms, *Theory of Computing Systems*, Vol. 33, January 2000, pp. 85–96.

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

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

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

[14] P.M. Churchland, *Matter and Consciousness*, MIT Press, Cambridge, Massachusetts, 1988, p. 105.

[15] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms, second edition*, MIT Press, Cambridge, Massachusetts, 2001.

[16] M. Davis, *The Universal Computer*, W.W. Norton, New York, New York, 2000, p. 151.

[17] D. Deutsch, *The Fabric of Reality*, Penguin Books, London, England, 1997, p. 134.

[18] G. Etesi and I. Németi, Non-Turing computations via Malament-Hogarth space-times, *International Journal of Theoretical Physics*, Vol. 41, No. 2, 2002, pp. 341–370.

[19] A. Gut, L. Miclea, Sz. Enyedi, M. Abrudean, and I. Hoka, Database globalization in enterprise applications, *Proceedings of the IEEE International Conference on Automation, Quality and Testing, Robotics*, Cluj-Napoca, Romania, 2006, pp. 356–359.

[20] D. Harel, *Algorithmics: The Spirit of Computing*, Addison-Wesley, Reading, Massachusetts, 1992, p. 233.

[21] D. Hillis, *The Pattern on the Stone*, Basic Books, New York, New York, 1998, pp. 63–64.

[22] J. Hopcroft and R. Tarjan, Efficient planarity testing, *Journal of the ACM*, Vol. 21, No. 4, 1974, pp. 549–568.

[23] J.E. Hopcroft and J.D. Ullman, *Formal Languages and their Relations to Automata*, Addison-Wesley, Reading, Massachusetts, 1969, p. 80.

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

[25] H.R. Lewis and C.H. Papadimitriou, *Elements of the Theory of Computation*, Prentice Hall, Englewood Cliffs, New Jersey, 1981, p. 168-169; p. 223.

[26] D. Mandrioli and C. Ghezzi, *Theoretical Foundations of Computer Science*, John Wiley, New York, New York, 1987, p. 152.

[27] C.D. McKay, J.G. Affleck, N. Nagy, S.G. Akl, and V.K. Walker, Molecular codebreaking and double encoding - Laboratory experiments, *International Journal of Unconventional Computing*, Vol. 5, No. 6, 2009, pp. 547–564.

[28] A. Moitra, Scheduling of hard real-time systems, *Proceedings of the Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*, New Delhi, India, 1986, pp. 362–381.

[29] M. Nagy and S.G. Akl, Quantum measurements and universal computation, *International Journal of Unconventional Computing*, Vol. 2, No.1, 2006, pp. 73–88.

[30] N. Nagy and S.G. Akl, Aspects of biomolecular computing, *Parallel Processing Letters*, Vol. 17, No. 2, 2007, pp. 185–211.

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

[32] M. Sipser, *Introduction to the Theory of Computation*, PWS Publishing Company, Boston, Massachusetts, 1997, p. 125.

[33] M. Stannet, X-machines and the halting problem: Building a super-Turing machine, *Formal Aspects of Computing*, Vol. 2, No. 4, 1990, pp. 331–341.

[34] V. Vedral, *Decoding Reality*, Oxford University Press, Oxford, United Kingdom, 2010, p. 135.

[35] P. Wegner and D. Goldin, Computation beyond Turing Machines, *Communications of the ACM*, Vol. 46, No. 4, 1997, pp. 100–102.

[36] C. Wehrenberg and T. Mulhall-Wehrenberg, *The Best-Kept Secret to Raising a Healthy Child ... and the Possible Prevention of Sudden Infant Death Syndrome (SIDS).* Specific Chiropractic, Clifton Park, New York, 2000.

[37] http://en.wikipedia.org/wiki/Church-Turing_thesis