UNCONVENTIONAL WISDOM: SUPERLINEAR SPEEDUP AND INHERENTLY PARALLEL COMPUTATIONS

Selim G. Akl
School of Computing, Queen's University
Kingston, Ontario K7L 3N6, Canada
akl@cs.queensu.ca

February 23, 2018

Abstract

A number of unconventional computational problems are described in which parallelism plays a fundamental role. These problems highlight two recently uncovered aspects of parallel computation:

1. There exist computations for which the running time of a parallel algorithm, compared to that of the best sequential algorithm, shows a speedup that is superlinear in the number of processors used, a feat that was previously believed to be impossible.
2. There exist inherently parallel computations, that is, computations that can be carried successfully in parallel, but not sequentially.

A surprising consequence of these discoveries is that the concept of universality in computation, long held as a basic truth, is in fact false.

Keywords and phrases: models of computation; inherently parallel computations; unconventional computations; superlinear speedup; superlinear slowdown; universality.

1 Introduction

The purpose of this chapter is to show the important role played by parallel processing in a host of out-of-the-ordinary computational problems, also referred to as *unconventional computations*.

For definiteness, we shall use two models of computation. Our *sequential* model of computation is the Random Access Machine (RAM) [1], which consists of a single processor p_1, having access to a memory that holds programs and data. The processor also possesses a number of local storage registers. The RAM implements a (conventional) sequential algorithm. Each step of a RAM algorithm runs in constant time, by definition a *time unit*, and consists of (up to) three phases:

1. A READ phase, in which the processor reads a datum from an arbitrary location in memory into one of its registers,
2. A COMPUTE phase, in which the processor performs an elementary arithmetic or logical operation on the contents of one or two of its registers, and
3. A WRITE phase, in which the processor writes the contents of one register into an arbitrary memory location.

Our *parallel* model of computation is the Parallel Random Access Machine (PRAM) [1], which is endowed with n processors p_1, p_2, ..., p_n, where $n \geq 2$, and implements a parallel algorithm. The processors share a common memory that holds data and to which they have access for reading or writing purposes. The processors act synchronously under the control of a program, a copy of which each processor possesses in its local registers. If needed, the processors may simultaneously access the same memory location in the common memory, for the purpose of reading (Concurrent Read, CR) or writing (Concurrent Write, CW). Write conflicts are resolved in several ways in order to determine what ends up being written in the memory location to which several processors are attempting to write at the same time, as required by the (unconventional) parallel algorithm. Thus, the repertoire of the PRAM includes CW instructions, such as

for example, instructions of the form MIN CW (for selecting the minimum of several values), AND CW (for obtaining the logical AND of several binary values), and SUM CW (for calculating the sum of several values), and so on.

Each step of a PRAM algorithm runs in constant time, again by definition a *time unit*, the same as in the RAM, and consists of (up to) three phases:

1. A READ phase, in which (up to n) processors read simultaneously from (up to n) memory locations. Each processor reads from at most one memory location and stores the value obtained in a local register,

2. A COMPUTE phase, in which (up to n) processors perform elementary arithmetic or logical operations on their local data, and

3. A WRITE phase, in which (up to n) processors write simultaneously into (up to n) memory locations. Each processor writes the value contained in a local register into at most one memory location.

The running time of a parallel algorithm designed for a certain problem is compared to that of the best available sequential algorithm for the same problem, by computing a ratio known as the *speedup*, defined as follows. Let t_1 denote the worst-case running time of the fastest known sequential algorithm for the problem, and let t_n denote the worst-case running time of the parallel algorithm using n processors. Then the speedup provided by the parallel algorithm is the ratio:

$$S(1,n) = \frac{t_1}{t_n}.$$

A good parallel algorithm is one for which this ratio is large. Usually (but not always) the speedup equals (up to a constant factor) the number of processors used. For many computational problems, this is the largest speedup possible; that is, the speedup is at most equal to the number of processors used by the parallel computer. Because this condition is satisfied by so many traditional problems, it has become part of the folklore of parallel computation and is usually formulated as a theorem:

> **Speedup Folklore Theorem:** For a given computational problem, the speedup provided by a parallel algorithm using n processors, over the fastest possible sequential algorithm for the problem, is at most equal to n; that is, $S(1,n) \leq n$.

Another concept that is useful in studying the running time of parallel algorithms is *slowdown* (by contrast with speedup). Slowdown measures the effect on running time of reducing the number of processors on a parallel computer. Naturally, one would expect the running time of an algorithm to increase as the number of processors decreases. The question is, how much slower is a parallel algorithm when solving a problem with fewer processors? The traditional answer to this question has given rise to a second folklore theorem:

> **Slowdown Folklore Theorem:** If a certain computation can be performed with n processors in time t_n and with p processors in time t_p, where $p < n$, then $t_n \leq t_p \leq t_n + nt_n/p$.

The slowdown folklore theorem puts an upper bound on the running time of the machine with fewer processors, essentially that $t_p/t_n < n/p$.

Both folklore theorems have been contradicted by counterexamples. Unconventional problems have been presented which provide parallel speedups greater than predicted by the speedup folklore theorem, as well as slowdowns greater than predicted by the slowdown folklore theorem when fewer than the required processors are available. This chapter surveys the previously presented counterexamples and offers new ones.

The remainder of the chapter is organized as follows. Previous counterexamples to the two folklore theorems are reviewed in Section 2. New unconventional computational problems that contradict these two folklore theorems are presented in Sections 3–8. Consequences of these results are offered in Section 10.

2 Previous work

Several unconventional computational problems were recently described whose purpose was to highlight two hitherto unknown aspects of parallelism [2]–[22], [29], [46]–[51]:

1. There exists computations for which a parallel algorithm permits a superlinear speedup, a feat that was previously believed to be impossible.
2. There exist inherently parallel computations, that is, computations that can be carried successfully in parallel, but not sequentially.

These unconventional problems are reviewed in this section.

2.1 One-way functions

A function f is said to be *one-way* if the function itself takes little time to compute, but (to the best of our knowledge) its inverse f^{-1} is computationally prohibitive. For example, let $x_1, x_2, \ldots, x_n$ be a sequence of integers. It is easy to compute the sum of a given subset of these integers. However, starting from a sum, and given only the sum, no efficient algorithm is known to determine a subset of the integer sequence that add up to this sum.

Consider that in order to solve a certain problem, it is required to compute $g(x_1, x_2, \ldots, x_n)$, where g is some function of n variables. The computation of g requires $\Omega(n)$ operations. For example, $g(x_1, x_2, \ldots, x_n) = x_1^2 + x_2^2 + \cdots + x_n^2$, might be such a function. The inputs $x_1, x_2, \ldots, x_n$ needed to compute g are received as n pairs of the form $\langle x_i, f(x_1, x_2, \ldots, x_n)\rangle$, for $i = 1, 2, \ldots, n$.

The function f possesses the following property: Computing f from $x_1, x_2, \ldots, x_n$ is done in n time units; on the other hand, extracting x_i from $f(x_1, x_2, \ldots, x_n)$ takes 2^n time units.

Because the function g is to be computed in real time, there is a deadline constraint: If a pair is not processed within one time unit of its arrival, it becomes obsolete (it is overwritten by other data in the fixed-size buffer in which it was stored).

Sequential Solution. The n pairs arrive simultaneously and are stored in a buffer, waiting in queue to be processed by the RAM. In the first time unit, the pair $\langle x_1, f(x_1, x_2, \ldots, x_n)\rangle$ is read and x_1^2 is computed. At this point, the other $n-1$ pairs are no longer available. In order to retrieve $x_2, x_3, \ldots, x_n$, the single processor p_1 needs to invert f. This requires $(n-1) \times 2^n$ time units. It then computes $g(x_1, x_2, \ldots, x_n) = x_1^2 + x_2^2 + \cdots + x_n^2$. Consequently, $t_1 = 1 + (n-1) \times 2^n + 2 \times (n-1)$ time units. Clearly, this is optimal for the RAM considering the time required to obtain the data.

Parallel Solution. Once the n pairs are received, they are processed by the n-processor PRAM immediately. Processor p_i reads the pair $\langle x_i,\ f(x_1, x_2, \ldots, x_n)\rangle$ and computes x_i^2, for $i = 1, 2, \ldots, n$. The PRAM processors now compute $g(x_1, x_2, \ldots, x_n)$ using a SUM CW. Consequently, $t_n = 1$.

Speedup and slowdown. The speedup provided by the PRAM over the RAM, namely, $S(1, n) = (n-1) \times 2^n + 2n - 1$, is superlinear in n and thus contradicts the speedup folklore theorem. What if only p processors are available on the PRAM, where $2 \leq p < n$? In this case, only p of the n variables (for example, $x_1, x_2, \ldots, x_p$) are read directly from the input buffer (one by each processor). Meanwhile, the remaining $n - p$ variables vanish and must be extracted from $f(x_1, x_2, \ldots, x_n)$. It follows that

$$t_p = 1 + \lceil (n-p)/p \rceil \times 2^n + \Big(\sum_{i=1}^{\log_p (n-p)} \lceil (n-p)/p^i \rceil \Big) + 1,$$

where the first term is for computing $x_1^2 + x_2^2 + \cdots + x_p^2$, the second for extracting $x_{p+1}, x_{p+2}, \ldots, x_n$, the third for computing $x_{p+1}^2 + x_{p+2}^2 + \cdots + x_n^2$, and the fourth for producing g. Therefore, t_p/t_n is asymptotically larger than $\lceil n/p \rceil$ by a factor that grows exponentially with n, and the slowdown folklore theorem is violated.

Throughout the remainder of this section, both the speedup folklore theorem and the slowdown folklore theorem will fail, and neither the speedup nor the slowdown will be measurable. Indeed, each one of the computations described in Sections 2.2– 2.8 is feasible if and only if an n-processor PRAM is available. No RAM and no PRAM with fewer processors than n can succeed in performing these computations, and as a result their running times are undefined.

2.2 Sorting with a twist

There exists a family of computational problems where, given a mathematical object satisfying a certain property, we are asked to transform this object into another which also satisfies the same property. Furthermore, the property is to be maintained throughout the transformation, and be satisfied by every intermediate

object, if any. More generally, the computations we consider here are such that every step of the computation must obey a certain predefined mathematical constraint. 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.

An example of computations obeying a mathematical constraint is provided by a variant to the problem of sorting a sequence of numbers stored in the memory of a computer. For a positive even integer n, where $n \geq 8$, let n distinct integers be stored in an array A with n locations $A[1], A[2], \ldots, A[n]$, one integer per location. Thus, $A[j]$, for all $1 \leq j \leq n$, 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 $1 \leq j \leq n-2$.

2. When the sort terminates we have:

$$A[1] < A[2] < \cdots < A[n].$$

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 (stable sorting), and so on.

An *oblivious* (that is, input-independent) algorithm for an $n/2$-processor parallel computer solves the aforementioned variant of the sorting problem handily in n steps, by means of predefined 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) [1]. This is illustrated in what follows:

Parallel Sort

for $k = 1$ **to** n **do**
  for $i = 1$ **to** $n-1$ **do in parallel**
    if $i \bmod 2 = k \bmod 2$
    then $A[i]$ and $A[i+1]$ are compared, and swapped if needed
    end if
  end for
end for.

An input-dependent algorithm succeeds on a computer with $(n/2)-1$ processors. However, a 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

$$A[1] > A[2] > \cdots > A[n] \ ,$$

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

2.3 Computational complexity as a function of time

Here, the computational complexity of the problems at hand changes with the passage of *time* (rather than being, as usual, a function of the problem *size*). Thus, for example, in real life, an illness that is undiagnosed for a long period becomes more difficult to treat, and an object lost in the forest is harder to find as darkness falls. Similarly, a digital file to which successive layers of encryption have been applied over time is increasingly more computationally demanding to cryptanalyze.

A certain computation requires that n independent functions, each of one variable, namely,

$$f_1(x_1), f_2(x_2), \ldots, f_n(x_n),$$

be computed. Computing $f_i(x_i)$ at time t requires 2^t algorithmic steps, for $t \geq 0$ and $1 \leq i \leq n$. Further, there is a strict deadline for reporting the results of the computations: All n values $f_1(x_1), f_2(x_2), \ldots, f_n(x_n)$ must be returned by the end of the third time unit, that is, when $t = 3$.

It should be easy to verify that the RAM, which by definition is capable of exactly one algorithmic step per time unit, cannot perform this computation for $n \geq 3$. Indeed, $f_1(x_1)$ takes $2^0 = 1$ time unit, $f_2(x_2)$ takes another $2^1 = 2$ time units, by which time three time units would have elapsed. At this point none of $f_3(x_3), \ldots, f_n(x_n)$ would have been computed. By contrast, an n-processor PRAM solves the problem handily. With all processors operating simultaneously, processor p_i computes $f_i(x_i)$ at time $t = 0$, for $1 \leq i \leq n$. This consumes one time unit, and the deadline is met.

2.4 Computational complexity as a function of rank

A computation consists of n stages. There may be a certain precedence among these stages, or the n stages may be totally independent, in which case the order of execution is of no consequence to the correctness of the computation. Let the *rank* of a stage be the order of execution of that stage. Thus, stage i is the ith stage to be executed. Here we focus on computations with the property that the number of algorithmic steps required to execute stage i is a function of i only.

When does rank-varying computational complexity arise? Clearly, if the computational requirements grow with the rank, this type of complexity manifests itself in those circumstances where it is a disadvantage, whether avoidable or unavoidable, to being ith, for $i \geq 2$. For example, the precision and/or ease of measurement of variables involved in the computation in a stage s may decrease with each stage executed before s.

The same analysis as in Section 2.3 applies by substituting the rank for the time.

2.5 Variables that vary with time

For a positive integer n larger than 1, we are given n functions, each of one variable, namely, $f_1, f_2, \ldots, f_n$, operating on the n physical variables $x_1, x_2, \ldots, x_n$, respectively. Specifically, it is required to compute $f_i(x_i)$, for $i = 1, 2, \ldots, n$. For example, $f_i(x_i)$ may be equal to x_i^2. What is unconventional about this computation, is the fact that the x_i are themselves (unknown) functions $x_1(t), x_2(t), \ldots, x_n(t)$, of the time variable t. It takes one time unit to evaluate $f_i(x_i(t))$. The problem calls for computing $f_i(x_i(t))$, $1 \leq i \leq n$, at time $t = t_0$. Because the function $x_i(t)$ is unknown, it cannot be inverted, and for $k > 0$, $x_i(t_0)$ cannot be recovered from $x_i(t_0 + k)$. Note that the time taken by the value of an input variable $x_i(t)$ to change (that is, become $x_i(t+1)$), is equal to the time taken by a processor to evaluate the function $f_i(x_i(t))$; both occur in one time unit.

The RAM fails to compute all the f_i as desired. Indeed, suppose that $x_1(t_0)$ is initially operated upon. By the time $f_1(x_1(t_0))$ is computed, one time unit would have passed. At this point, the values of the $n-1$ remaining variables would have changed. The same problem occurs if the RAM attempts to first read all the x_i, one by one, and store them before calculating the f_i.

By contrast, a PRAM endowed with n independent processors may perform all the computations at once: For $1 \leq i \leq n$, and all processors working at the same time, processor p_i computes $f_i(x_i(t_0))$, leading to a successful computation.

2.6 Variables that influence one another

A physical system has n variables, $x_1, x_2, \ldots, x_n$, each of which is to be measured or set to a given value at regular intervals. One property of this system is that measuring or setting one of its variables modifies the

values of any number of the system variables uncontrollably, unpredictably, and irreversibly.

The RAM measures *one* of the values (x_1, for example) and by so doing it disturbs an unknowable number of the remaining variables, thus losing all hope of recording the state of the system within the given time interval. Similarly, the RAM approach cannot update the variables of the system properly: Once x_1 has received its new value, setting x_2 may disturb x_1 in an uncertain way.

A PRAM with n processors, by contrast, will measure *all* the variables $x_1, x_2, \ldots, x_n$ simultaneously (one value per processor), and therefore obtain an accurate reading of the state of the system within the given time frame. Consequently, new values $x_1, x_2, \ldots, x_n$ can be computed in parallel and applied to the system simultaneously (one value per processor).

2.7 Deadlines that are uncertain

In this paradigm, we are given a computation consisting of three distinct stages, namely, input, calculation, and output, each of which needs to be completed by a certain deadline. However, unlike the standard situation in conventional computation, the deadlines here are not known at the outset. In fact, to add to the unconventional character of this problem, we do not know at the moment the computation is set to start, *what* needs to be done, and *when* it should be done. Certain physical parameters, from the external environment surrounding the computation, become spontaneously available. The values of these parameters, once received from the outside world, are then used to evaluate two functions, f_1 and f_2, that tell us precisely *what* to do and *when* to do it, respectively.

The difficulty posed by this paradigm is that the evaluation of the two functions f_1 and f_2 is itself quite demanding computationally. Specifically, for a positive integer n, the two functions operate on n variables (the physical parameters). Only a PRAM equipped with n processors can succeed in evaluating the two functions on time to meet the deadlines.

2.8 Working with a global variable

A computation C_1 consists of two distinct and separate processes P_1 and P_2 operating on a global variable x. The variable x is *time-critical* in the sense that its value throughout the computation is intrinsically related to real (external or physical) time. Actions taken throughout the computation, based on the value of x, depend on x having that particular value at that particular time. Here, time is kept internally by a global clock. Specifically, the computer performing C_1 has a clock that is synchronized with real time. Henceforth, real time is synonymous with internal time. In this framework, therefore, resetting x artificially, through simulation, to a value it had at an earlier time is entirely insignificant, as it fails to meet the true timing requirements of C_1. At the beginning of the computation, $x = 0$.

Let the processes of the computation C_1, namely, P_1 and P_2, be as follows:

P_1: **if** $x = 0$ **then** $x \leftarrow x + 1$ **else** loop forever **end if**.

P_2: **if** $x = 0$ **then** read y; $x \leftarrow x + y$; return x **else** loop forever **end if**.

In order to better appreciate this simple example, it is helpful to put it in some familiar context. Think of x as the altitude of an airplane and think of P_1 and P_2 as software controllers actuating safety procedures that must be performed at this altitude. The local nonzero variable y is an integral part of the computation; it helps to distinguish between the two processes and to separate their actions.

The question now is this: on the assumption that C_1 succeeds, that is, that both P_1 and P_2 execute the "**then**" part of their respective "**if**" statements (not the "**else**" part), what is the value of the global variable x at the end of the computation, that is, when both P_1 and P_2 have halted?

We examine two approaches to executing P_1 and P_2:

1. **Using a single processor:** Consider the RAM equipped, by definition, with a single processor p_1. The processor executes one of the two processes first. Assuming it starts with P_1: p_1 computes $x = 1$ and terminates. It then proceeds to execute P_2. Because now $x \neq 0$, p_1 executes the nonterminating computation in the "**else**" part of the "**if**" statement. The process is uncomputable and the computation fails. Note that starting with P_2 and then executing P_1 would lead to a similar outcome, with the difference being that P_2 will return an incorrect value of x, namely y, before switching to P_1, whereby it executes a nonterminating computation, given that now $x \neq 0$.

2. **Using two processors:** Consider a PRAM with two processors, namely, p_1 and p_2. In parallel, p_1 executes P_1 and p_2 executes P_2. Both terminate successfully and return the correct value of x, that is, $x = y + 1$.

Two observations are in order:

1. The first concerns the RAM (that is, the single-processor) solution. Here, no *ex post facto* simulation is possible or even meaningful. This includes legitimate simulations, such as executing one of the processes and then the other, or interleaving their executions, and so on. It also includes illegitimate simulations, such as resetting the value of x to 0 after executing one of the two processes, or (assuming this is feasible) an *ad hoc* rewriting of the code, as for example,

   if $x = 0$ **then** $x \leftarrow x + 1$; read y; $x \leftarrow x + y$; return x
   else loop forever
   end if.

   and so on. To see this, note that for either P_1 or P_2 to terminate, the **then** operations of its **if** statement must be executed *as soon as* the global variable x is found to be equal to 0, and not one time unit later. It is clear that any sequential simulation must be seen to have failed. Indeed:

   - A legitimate simulation will not terminate, because for one of the two processes, x will no longer be equal to 0, while
   - An illegitimate simulation will "terminate" illegally, having executed the "**then**" operations of one or both of P_1 or P_2 too late.

2. The second observation follows directly from the first. It is clear that P_1 and P_2 must be executed simultaneously for a proper outcome of the computation. The PRAM (that is, the two-processor) solution succeeds in accomplishing exactly this.

A word about the role of time. Real time, as mentioned earlier, is kept by a global clock and is equivalent to internal computer time. It is important to stress here that the time variable is never used explicitly by the computation C_1. Time intervenes only in the circumstance where it is needed to signal that C_1 has failed (when the "**else**" part of an "**if**" statement, either in P_1 or in P_2, is executed). In other words, time is noticed solely when the time requirements are neglected.

To generalize the global variable paradigm, we assume the presence of n global variables, namely, $x_1, x_2, \ldots, x_n$, all of which are time critical, and all of which are initialized to 0. There are also n nonzero local variables, namely, $y_1, y_2, \ldots, y_n$, belonging, respectively, to the n processes $P_1, P_2, \ldots, P_n$ that make up C_2. The computation C_2 is as follows:

P_1: **if** $x_1 = 0$ **then** $x_2 \leftarrow y_1$ **else** loop forever **end if**.

P_2: **if** $x_2 = 0$ **then** $x_3 \leftarrow y_2$ **else** loop forever **end if**.

P_3: **if** $x_3 = 0$ **then** $x_4 \leftarrow y_3$ **else** loop forever **end if**.

$\vdots$

P_{n-1}: **if** $x_{n-1} = 0$ **then** $x_n \leftarrow y_{n-1}$ **else** loop forever **end if**.

P_n: **if** $x_n = 0$ **then** $x_1 \leftarrow y_n$ **else** loop forever **end if**.

Assume that the computation C_2 begins when $x_i = 0$, for $i = 1, 2, \ldots, n$. For every i, $1 \leq i \leq n$, if P_i is to be completed successfully, it must be executed *while* x_i is indeed equal to 0, and not at any later time when it is no longer equal to 0, having been modified by p_{i-1} for $i > 1$, or by p_n for $i = 1$. On a PRAM with n processors, namely, $p_1, p_2, \ldots, p_n$, it is possible to test all the x_i, $1 \leq i \leq n$, for equality to 0 in one time unit; this is followed by assigning to all the x_i, $1 \leq i \leq n$, their new values during the next time unit.

Thus, all the processes P_i, $1 \leq i \leq n$, and hence the computation C_2, terminate successfully. The RAM has but a single processor p_1 and, as a consequence, it fails to meet the time-critical requirements of C_2. At best, it can perform no more than $n-1$ of the n processes as required (assuming it executes the processes in the order $P_n, P_{n-1}, \ldots, P_2$, then fails at P_1 since x_1 was modified by P_n), and thus does not terminate. A PRAM with only $n-1$ processors, $p_1, p_2, \ldots, p_{n-1}$, cannot do any better. At best, it too will attempt to execute at least one of the P_i when $x_i \neq 0$ and hence fail to complete at least one of the processes on time.

Finally, and most importantly, even a computer capable of an *infinite* number of algorithmic steps per time unit (like an Accelerating Machine [31] or, more generally, a Supertask Machine [23], [28], [58]) would fail to perform the computations required by the global variable paradigm if it were restricted to execute these algorithmic steps *sequentially*.

3 Data rearrangement

An array $X[1], X[2], \ldots, X[n]$ is given that contains n distinct integers $I_1, I_2, \ldots, I_n$ in the range $(-\infty, n]$ such that $X[i] = I_i$ for $1 \leq i \leq n$. It is required to modify the array X so that for all i, $1 \leq i \leq n$, $X[I_i] = I_i$ if and only if $1 \leq I_i \leq n$; otherwise, $X[i] = I_i$. In what follows we show that the PRAM and RAM solutions to this problem lead to a contradiction with the speedup folklore theorem.

A PRAM with n processors solves the problem in one READ-COMPUTE-WRITE step executed simultaneously by all processors:

```
for i = 1 to n do in parallel
    if X[i] > 0
    then X[X[i]] ← X[i]
    end if
end for.
```

Now consider a RAM. Any algorithm for performing this computation includes (possibly among other steps) READ-COMPUTE-WRITE steps of the form:

```
if I_i > 0
then X[I_i] ← I_i
end if.
```

Consider the first such step executed by the algorithm. Since a positive I_i may take any value from 1 to n, the WRITE operation can occur at any position of array X, thus destroying its old contents. Therefore, the remaining $n-1$ I_j's ($j \neq i$, $1 \leq j \leq n$) must have been "seen" previously by the algorithm in $n-1$ steps involving READ operations and preceding the current step. Since, in addition, there could be n steps involving WRITE operations, any RAM algorithm must require $2n-1$ steps. A RAM algorithm requiring exactly this many READ-COMPUTE-WRITE steps uses an additional array W of $n-1$ locations, and is as follows:

```
for i = 1 to n − 1 do
   W[i] ← X[i]
end for
if X[n] > 0
then X[X[n]] ← X[n]
end if
for i = 1 to n − 1 do
   if W[i] > 0
   then X[W[i]] ← W[i]
   end if
end for.
```

Since $S(1,n) = 2n-1$, the speedup is larger than that predicted by the speedup folklore theorem (i.e., n), albeit by a constant multiplicative factor.

4 Cyclic shift

Given an array $X[1], X[2], \ldots, X[n]$ containing arbitrary data and an integer q that divides n evenly, it is required to shift cyclically the contents of every sequence of q consecutive elements of X by one position to the right.

Two PRAM solutions to this problem, one with n processors and one with p processors, where $2 \leq p < q$, lead to a contradiction with the slowdown folklore theorem, as demonstrated in the following.

A PRAM with n processors clearly solves the problem in one step. Each group of q processors performs a cyclic shift on a different group of q consecutive elements of X.

```
for j = 1 to n/q do in parallel
  for i = (j − 1)q + 1 to jq do in parallel
    X[[i + 1] mod jq] ← X[i]
  end for
end for.
```

By contrast, on a PRAM with p processors, $2 \leq p < q$, the number of necessary and sufficient steps is $\lceil (n/p) + n/(pq) \rceil$. We show this as follows.

(a) Assume that fewer than $\lceil (n/p) + n/(pq) \rceil$ steps are sufficient. Since during each step, at most p memory accesses can be performed, the total number of memory accesses is smaller than $n + (n/q)$. However, because there are fewer processors than elements to be shifted, one supplementary memory access is necessary for each cyclic shift in order that no element be lost. Hence, any solution to the problem necessitates at least $n + (n/q)$ memory accesses, which contradicts the assumption.

(b) An algorithm requiring $\lceil (n/p) + n/(pq) \rceil$ steps is obtained in the following way. For each group of q consecutive elements to be shifted: Store the last element in an additional memory location, shift every element (except the last) by one position to the right (starting from the end of the array and proceeding to the beginning, in groups of p elements, with the last group to be shifted possibly containing fewer that p elements), and finally copy the content of the additional memory location into the location of the first element.

The slowdown folklore theorem predicts a running time of at most $1 + (n/p)$. For $pq < n$, the time required by the PRAM algorithm with p processors exceeds this bound.

5 Time stamps

Consider a PRAM variant that allows several processors to gain access to the memory simultaneously for different purposes. Thus, some processors may be reading, while others may be writing. If two processors gain access to the same location at the same time, one for reading and one for writing, then the reading takes place before the writing. An array X of n elements is stored in the shared memory. Each element $X[i]$, $1 \leq i \leq n$, is associated with a *time stamp*, giving the time when $X[i]$ was last overwritten. This time stamp is modified every time a processor gains access to $X[i]$ for the purpose of writing. Let r be the probability that a given element of X is *not* overwritten during a given time unit. The task to be executed is as follows: Select a time D, and return the value of $X[i]$, $1 \leq i \leq n$, at D.

There are two sets of processors: One set is executing some algorithm that causes entries of X to change, while the second set is in charge of reading and reporting these values. In what follows we focus on the second of these two sets.

We first observe that n processors can perform the task in one time unit. With n processors, all $X[i]$, $1 \leq i \leq n$, are read simultaneously at time D and produced as output. We now show that if fewer than n processors are used, both the speedup and slowdown folklore theorems are violated.

Assume that $p = n/a$ processors are used, where $n \geq a > 1$. We derive the probability that the task is completed successfully (i.e., that all locations of X are read in a time units, without any of them being modified). With n/a processors, the time required to read all $X[i]$, $1 \leq i \leq n$, is a time units. Each processor reads a entries of X. The probability that a processor reads the a entries $X[j], X[j+1], \ldots, X[j+a-1]$, without some location $X[j+i]$, $1 \leq i \leq (a-1)$, being modified after $X[j]$ and before $X[j+a-1]$ is read is $r^{a(a-1)/2}$. The probability of this occurring for all n/a processors is $r^{n(a-1)/2}$. Since $a > 1$, a linear decrease in the number of processors has resulted in an exponential decrease in the probability of success.

Let us now assume that if n/a processors fail to execute the task, they must restart. The expected time required by n/a processors to complete the task successfully is our main result in this section. With n/a processors, the expected number of attempts before success is $1/(r^{n(a-1)/2})$. The expected running time of an attempt is

$$(\sum_{x=1}^{a-1} xr^{x-1}(1-r)) + ar^{a-1} = \frac{1-r^a}{1-r}.$$

To see this, note that one cannot fail on the first read. If after that the second entry (in a group of a entries) has changed, then the current attempt would have taken one time unit. This explains the first term of the summation, namely, $1 \times r^0(1-r)$. In general, the exponent of r, i.e., $x-1$, is one less than the number of values read successfully in a group of a values (because one cannot fail on the first attempt), while the random variable x is the number of time units spent reading successfully x values, and the factor $(1-r)$ is the probability that the $(x+1)$st value has changed. If all a values in a group are read successfully, this attempt is guaranteed to succeed, and last a time units. We therefore have the term ar^{a-1} (without the factor $(1-r)$).

Thus, the expected time before success is:

$$\frac{1}{r^{n(a-1)/2}} \times \frac{1-r^a}{1-r}.$$

It should be noted that it is not necessary to check the time stamp of the first value, and consequently checking the $(x+1)$st time stamp is included in the time taken to read the xth value.

6 Data stream

In a certain application, a set of n data is received every k time units and stored in a computer's memory. Here $2 < k < n$; for example, let $k = 5$. The ith data set received is stored in the ith row of a two-dimensional array A. In other words, the elements of the ith set occupy locations $A[i,1]$, $A[i,2], \ldots, A[i,n]$. At most 2^n such sets may be received. Thus, A has 2^n rows and n columns. Initially, A is empty. The n data forming a set are received and stored simultaneously: One time unit elapses from the moment the data are received from the outside world to the moment they settle in a row of A. Once a datum has been stored in $A[i,j]$, it requires one time unit to be processed; that is, a certain operation must be performed on it which takes one time unit. This operation depends on the application. For example, the operation may simply be

$$A[i,j] \leftarrow (A[i,j])^2.$$

The computation terminates once all data currently in A have been processed, *regardless of whether more data arrive later*.

In what follows:

1. We compare the performance of a PRAM with n processors to that of a RAM in solving this problem, and contrast the result with that predicted by the speedup folklore theorem.

2. We compare the performance of a PRAM with $p < n$ processors to that of a PRAM using n processors in solving this problem, and contrast the result with that predicted by the slowdown folklore theorem.

6.1 PRAM with n processors

A PRAM with n processors receives the first data set, stores it in:

$$A[1,1],\ A[1,2],\ \ldots,\ A[1,n],$$

and updates it to:

$$(A[1,1])^2,\ (A[1,2])^2,\ \ldots,\ (A[1,n])^2,$$

all in two time units. Since all data currently in A have been processed and no new data have been received, the computation terminates.

A RAM receives the first set of n data in one time unit. It then proceeds to update it. This requires n time units. Meanwhile, $n/5$ additional data sets would have arrived in A, and must be processed. The RAM does not catch up with the arriving data until they cease to arrive. Therefore, the RAM must process $2^n \times n$ values. This requires $2^n \times n$ time units. The speedup is $2^n \times n/2$, which is significantly larger than the maximum speedup of n predicted by the speedup folklore theorem.

6.2 PRAM with $p < n$ processors

Let a PRAM with p processors be used, where $p < n$, and assume that $(n/p) > 5$. The first set of data is processed in n/p time units. Meanwhile, $(n/p)/5$ new data sets would have been received. This way, the PRAM never catches up with the arriving data until the data cease to arrive. Therefore, $2^n \times n$ data must be processed, and this requires $(2^n \times n)/p$ time units. This running time is asymptotically larger than the $2 \times (1 + (n/p))$ time predicted by the slowdown folklore theorem.

7 Unpredictable data

Let n data on which a certain computation is to be performed be stored in the memory of a computer. For example, it may be required to compute the sum of the n data currently in memory. Every $n/2$ time units, the values of k of the data (not known ahead of time) change. There are at most n such updates (each involving k values). If the result of the computation is reported after D time units, then it must be obtained using the n values in memory at the end of D time units.

As in the previous section, we shall:

1. Compare the performance of a PRAM with n processors to that of a RAM in executing this computation, and contrast the result we obtain with that predicted by the speedup folklore theorem.

2. Assume that a PRAM with $p < n$ processors is used to perform the computation and compare this PRAM's performance to that of a PRAM using n processors, and contrast our result with that predicted by the slowdown folklore theorem.

7.1 PRAM with n processors

For definiteness, let $k = n/2$ and $D = n/4$. The sum of the values currently in memory must be reported by time unit 1. If the sum is not ready, then the sum of the new values is reported at time D; if not, then at time $2D$, and so on. A PRAM with n processors computes the sum using one **SUM CW** instruction in one time unit, delivers the sum by the first deadline, and terminates.

A RAM is not ready to deliver the sum at time D, since it would have only added $n/4$ numbers. It is still not ready at time $2D$. Now a change occurs, and if all $n/2$ values added up so far have changed, a new sum must be computed. This continues with the RAM never able to catch up while changes occur. After the n changes have taken place, that is, at time $n^2/2$, the RAM uses at most n additional time units to deliver the sum at time $(2n + 4)D$.

The speedup is $O(n^2)$, which is asymptotically larger than the speedup of $O(n)$ predicted by the speedup folklore theorem.

7.2 PRAM with $p < n$ processors

Recall that n processors compute the sum in one time unit and meet the first deadline.

Let $D = (n/2) + (1/n^2)$ and $k > n/2 > p$. In order to compute the sum with p PRAM processors, $O(n/p)$ time is required. This means that when the data change, the p processors will not be ready to deliver the sum by the next deadline. Therefore, at the end of $n^2/2$ time units, the p processors will take another $O(n/p)$ time units to compute the final sum, for a total time of $O(n^2 + (n/p))$. This time is asymptotically larger than the $1 \times (1 + (n/p))$ time units predicted by the slowdown folklore theorem.

8 Setting the elements of an array

An array A of size n, such that $A[i] = 0$ for $1 \leq i \leq n$, is given. It is required to set $A[i] \leftarrow T$, for all i, $1 \leq i \leq n$, where $T = n^x$, for some positive integer constant $x > 1$, provided that at no time during the update two elements of A differ by more than a certain constant w. A PRAM with n processors solves the problem in constant time, that is, $t_n = O(1)$. A RAM, on the other hand, updates each element of A by w units at a time, thus requiring $t_1 = n \times (T/w) = O(n^{x+1})$ time to complete the task. The speedup t_1/t_n is $O(n^{x+1})$. The speedup predicted by the speedup folklore theorem is n.

Now assume that p processors are available, where $p < n$. The PRAM now updates the elements of A in groups of p elements by w units at a time. The total time required is $t_p = (n/p) \times (T/w) = O(n^{x+1}/p)$. Thus, $t_p/t_n = O(n^{x+1}/p)$. The ratio predicted by the slowdown folklore theorem is $1 + (n/p)$.

9 Several data streams

Consider n independent streams of data arriving as input at a computer. Each stream contains a distinct cyclic permutation of the values in a sequence $S = \{s_1, s_2, \ldots, s_n\}$. Thus, for $n = 4$, the four input streams may be $< s_1, s_2, s_3, s_4 >$, $< s_2, s_3, s_4, s_1 >$, $< s_3, s_4, s_1, s_2 >$ and $< s_4, s_1, s_2, s_3 >$. In addition, the ith value in a stream is separated from the $(i+1)$st value by 2^i time units. Furthermore, a stream remains active if and only if its first value has been read and stored by a processor.

A single processor can monitor the values in only one stream: By the time it reads and stores the first value of a selected stream, it is too late to turn and process the remaining $n - 1$ values from the other streams, which arrived at the same time.

Suppose that we need to compute the smallest value in S. A RAM selects a stream and reads the consecutive values it receives, keeping track of the smallest encountered so far. In one time unit the RAM processor can read a value, compare it to the smallest so far, and update the latter if necessary. It therefore takes n time units to process the n inputs, plus $(2^1 + 2^2 + \cdots + 2^{n-1}) = 2^n - 2$ time units of waiting time in between consecutive inputs. Therefore, after exactly $n + 2^n - 2$ time units, the minimum value is known.

On the other hand, let the computer be an n-processor PRAM. In one parallel READ operation, each processor reads one value from a distinct stream. This is followed by a MIN CW operation, the result of which is to store the minimum value of S in a location in the shared memory. This requires one time unit. The speedup is therefore $(n + 2^n - 2)/1 = O(2^n)$, which is asymptotically larger than n, the number of processors used on the parallel computer. A PRAM with fewer processors has the same performance as the RAM.

10 Conclusion

For each of the computational problems described in Sections 2–9 we have the following:

1. Either the computational problem can be readily solved on a computer capable of executing n algorithmic steps per time unit, but fails to be executed on a computer capable of fewer than n algorithmic steps per time unit,

2. Or a computer capable of executing n algorithmic steps per time unit is superior in performance to any computer capable of executing p algorithmic steps per time unit, where $1 \leq p < n$, by a factor larger than n/p.

Furthermore, the problem size n itself is a variable that changes with each problem instance. As a result, *no* computer, regardless of how many algorithmic steps it can perform in one time unit, can cope with a growing problem size, as long as it obeys the "finiteness condition", that is, as long as the number of algorithmic steps it can perform per time unit is finite and fixed. This observation leads to a theorem that there does not exist a *finite* computational device that can be called a Universal Computer. The proof of this theorem proceeds as follows. Let us assume that there exists a Universal Computer capable of n algorithmic steps per time unit, where n is a finite and fixed integer. This computer will fail to perform a computation *requiring* n' algorithmic steps per time unit, for any $n' > n$, and consequently lose its claim of universality. Naturally, for each $n' > n$, another computer capable of n' algorithmic steps per time unit will succeed in performing the aforementioned computation. However, this new computer will in turn be defeated by a

problem requiring $n'' > n'$ algorithmic steps per time unit. This holds even if the computer purporting to be universal is endowed with an unlimited memory and is allowed to compute for an indefinite amount of time [7]–[16].

The only constraint that is placed on the computer (or model of computation) that aspires to be universal is the aforementioned finiteness condition, namely, that the number of operations of which the computer is capable per time unit be finite and fixed once and for all. In this regard, it is important to note that:

1. The requirement that the number of operations per time unit, or step, be *finite* is necessary for any "reasonable" model of computation; see, for example, [57], p. 141.

2. The requirement that this number be *fixed* once and for all is necessary for any model of computation that claims to be "universal"; see, for example, [26], p. 210.

The condition that the number of operations per time unit be finite and fixed is fundamental and of utmost importance in computer science. Without it, the relevance of the theory of computation, in general, and of the design and analysis of algorithms, in particular, would be severely diminished. The absence of a bound on the number of operations per time unit, would make it possible for all algorithms to run in constant time. A case in point is the celebrated question of whether P is equal to NP. Here, P stands for the class of problems solvable in polynomial time on a deterministic Turing Machine, while NP is the class of problems solvable in polynomial time on a nondeterministic Turing Machine. In the preceding definitions of the classes P and NP, the phrase "polynomial time" means that there exists an algorithm for solving every problem of size n in either one of the two classes, whose running time is a polynomial function of n. Note that both complexity classes P and NP are defined in terms of the *time* required to solve a problem, not in terms of the number of operations (as they technically should). This means that time has been equated with the number of operations. In other words, the number of operations per time unit must be finite and fixed. Failing this, the question "$P = NP$?" is nonsensical, for it is clear that $P = NP$ when the number of operations per time unit is neither finite nor fixed.

It should be noted that computers obeying the finiteness condition include all "reasonable" models of computation, both theoretical and practical, such as the Turing Machine, the Random Access Machine, and other idealized models [54], as well as all of today's general-purpose computers, including existing conventional computers (both sequential and parallel), and contemplated unconventional ones such as biological and quantum computers [7]. It is true for computers that interact with the outside world in order to read input and return output (unlike the Turing Machine, but like every realistic general-purpose computer). It is also valid for computers that are given unbounded amounts of time and space in order to perform their computations (like the Turing Machine, but unlike realistic computers). Even Accelerating Machines that increase their speed at every step at a rate of acceleration that is defined in advance, once and for all, and in no way is a function of input characteristics, cannot be universal.

As a result, it is possible to conclude that the only possible universal computer would be one capable of an infinite number of algorithmic steps per time unit *executed in parallel*.

In fact, this work has led to the discovery of computations that can be performed on a quantum computer but that cannot, even in principle, be performed on any classical computer (even one with infinite resources), thus showing for the first time that the class of problems solvable by classical means is a true subset of the class of problems solvable by quantum means [48]. Consequently, the only possible universal computer would have to be quantum (as well as being capable of an infinite number of algorithmic steps per time unit *executed in parallel*).

References

[1] Akl, S.G.: Parallel Computation: Models and Methods. Prentice Hall, Upper Saddle River (1997)

[2] Akl, S.G.: Superlinear performance in real-time parallel computation. J. of Supercomputing. **29**, 89–111 (2004)

[3] Akl, S.G.: Non-Universality in Computation: The Myth of the Universal Computer. School of Computing, Queen's University.
`http://research.cs.queensu.ca/Parallel/projects.html`

[4] Akl, S.G.: A computational challenge. School of Computing, Queen's University.
http://www.cs.queensu.ca/home/akl/CHALLENGE/A_Computational_Challenge.htm

[5] Akl, S.G.: The myth of universal computation. In: Trobec, R., Zinterhof, P., Vajteršic, M., Uhl, A. (eds.) Parallel Numerics, pp. 211-236. University of Salzburg, Salzburg and Jozef Stefan Institute, Ljubljana (2005)

[6] Akl, S.G.: Universality in computation: Some quotes of interest. Technical Report No. 2006-511, School of Computing, Queen's University.
http://www.cs.queensu.ca/home/akl/techreports/quotes.pdf

[7] Akl, S.G.: Three counterexamples to dispel the myth of the universal computer. Parallel Proc. Lett. **16**, 381–403 (2006)

[8] Akl, S.G.: Conventional or unconventional: is any computer universal? In: Adamatzky, A., Teuscher, C. (eds.) From Utopian to Genuine Unconventional Computers, pp. 101-136. Luniver Press, Frome (2006)

[9] Akl, S.G.: Gödel's incompleteness theorem and nonuniversality in computing. In: Nagy, M., Nagy, N. (eds.) Proceedings of the Workshop on Unconventional Computational Problems, pp. 1-23. Sixth International Conference on Unconventional Computation, Kingston (2007)

[10] Akl, S.G.: Even accelerating machines are not universal. Int. J. of Unconventional Comp. **3**, 105–121 (2007)

[11] Akl, S.G.: Unconventional computational problems with consequences to universality. Int. J. of Unconventional Comp. **4**, 89–98 (2008)

[12] Akl, S.G.: Evolving computational systems. In: Rajasekaran, S., Reif, J.H. (eds.) Parallel Computing: Models, Algorithms, and Applications, pp. 1-22. Taylor and Francis, Boca Raton (2008)

[13] Akl, S.G.: Ubiquity and simultaneity: the science and philosophy of space and time in unconventional computation. Keynote address, Conference on the Science and Philosophy of Unconventional Computing, The University of Cambridge, Cambridge (2009)

[14] Akl, S.G.: Time travel: A new hypercomputational paradigm. Int. J. of Unconventional Comp. **6**, 329–351 (2010)

[15] Akl, S.G.: What is computation? Int. J. of Parallel, Emergent and Distributed Syst. **29**, 337–345 (2014)

[16] Akl, S.G.: Unconventional computational problems. In: Meyers, R.A. (ed) Encyclopedia of Complexity and Systems Science. Springer, New York (2017)

[17] Akl, S.G., Cosnard, M., Ferreira, A.G.: Data-movement-intensive problems: Two folk theorems in parallel computation revisited. Theoretical Computer Science. **95**, 323–337 (1992)

[18] Akl, S.G., Fava Lindon, L.: Paradigms for superunitary behavior in parallel computations. Journal of Parallel Algorithms and Applications. **11**, 129–153 (1997)

[19] Akl, S.G., Nagy, M.: Introduction to parallel computation. In: Trobec, R., Vajteršic, M., Zinterhof, P. (eds.) Parallel Computing: Numerics, Applications, and Trends, pp. 43-80. Springer-Verlag, London (2009)

[20] Akl, S.G., Nagy: M.: The future of parallel computation. In: Trobec, R., Vajteršic, M., Zinterhof, P. (eds.) Parallel Computing: Numerics, Applications, and Trends, pp. 471-510. Springer-Verlag, London (2009)

[21] Akl, S.G., Salay, N.: On computable numbers, nonuniversality, and the genuine power of parallelism. International Journal of Unconventional Computing. **11**, 283–297 (2015)

[22] Akl, S.G., Yao, W.: Parallel computation and measurement uncertainty in nonlinear dynamical systems. J. of Math. Model. and Alg. **4**, 5–15 (2005)

[23] Davies, E.B.: Building infinite machines. Br. J. for Phil. of Sc. **52**, 671–682 (2001)

[24] Davis, M.: The Universal Computer. W.W. Norton, New York (2000)

[25] Denning, P.J., Dennis, J.B., Qualitz, J.E.: Machines, Languages, and Computation. Prentice-Hall, Englewood Cliffs (1978)

[26] Deutsch, D.: The Fabric of Reality. Penguin Books, London (1997)

[27] Durand-Lose, J.: Abstract geometrical computation for black hole computation. Research Report No. 2004-15, Laboratoire de l'Informatique du Parallélisme, École Normale Supérieure de Lyon, Lyon (2004)

[28] Earman, J., Norton, J.D.: Infinite pains: The trouble with supertasks. In: Morton, A., Stich, S.P. (eds.) Benacerraf and his Critics, pp. 231–261. Blackwell, Cambridge (1996)

[29] Fava Lindon, L.: Synergy in Parallel Computation. Ph.D. Thesis, Department of Computing and Information Science, Queen's University, Kingston, Ontario (1996)

[30] Fortnow, L.: The enduring legacy of the Turing machine.
`http://ubiquity.acm.org/article.cfm?id=1921573`

[31] Fraser, R., Akl, S.G.: Accelerating machines: a review. Int. J. of Parallel, Emergent and Distributed Syst. **23**, 81–104 (2008)

[32] Gleick, J.: The Information: A History, a Theory, a Flood. HarperCollins, London (2011)

[33] Harel, D.: Algorithmics: The Spirit of Computing. Addison-Wesley, Reading (1992)

[34] Hillis, D.: The Pattern on the Stone. Basic Books, New York (1998)

[35] Hopcroft, J.E. Turing Machines. Sci. Amer. **250**, 86–98 (1984)

[36] Hopcroft, J., Tarjan R.: Efficient planarity testing. J. of the ACM **21**, 549–568 (1974)

[37] Hopcroft, J.E., Ullman, J.D.: Formal Languages and their Relations to Automata. Addison-Wesley, Reading (1969)

[38] Hypercomputation.
`http://en.wikipedia.org/wiki/Hypercomputation`

[39] Kelly, K.: God is the machine. Wired **10** (2002)

[40] Kleene, S.C.: Introduction to Metamathematics. North Holland, Amsterdam (1952)

[41] Lewis, H.R., Papadimitriou, C.H.: Elements of the Theory of Computation. Prentice Hall, Englewood Cliffs (1981)

[42] Lloyd, S.: Programming the Universe. Knopf, New York (2006)

[43] Lloyd, S., Ng, Y.J.: Black hole computers. Sci. Amer. **291**, 53–61 (2004)

[44] Mandrioli, D., Ghezzi, C.: Theoretical Foundations of Computer Science. John Wiley, New York (1987)

[45] Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice-Hall, Englewood Cliffs (1967).

[46] Nagy, M., Akl, S.G.: On the importance of parallelism for quantum computation and the concept of a universal computer. In: Calude, C.S., Dinneen, M.J., Paun, G., Pérez-Jiménez, M. de J., Rozenberg, G. (eds.) Unconventional Computation, pp. 176-190. Springer, Heildelberg (2005).

[47] Nagy, M., Akl, S.G.: Quantum measurements and universal computation. Int. J. of Unconventional Comp. **2**, 73–88 (2006)

[48] Nagy, M., Akl, S.G.: Quantum computing: Beyond the limits of conventional computation. Int. J. of Parallel, Emergent and Distributed Syst. **22**, 123–135 (2007)

[49] Nagy, M., Akl, S.G.: Parallelism in quantum information processing defeats the Universal Computer. Par. Proc. Lett. **17**, 233–262 (2007)

[50] Nagy, N., Akl, S.G., Computations with uncertain time constraints: effects on parallelism and universality. In: Calude, C.S., Kari, J., Petre, I., Rozenberg, G. (eds.) Unconventional Computation, pp. 152-163. Springer, Heidelberg (2011)

[51] Nagy, N., Akl, S.G.: Computing with uncertainty and its implications to universality. Int. J. of Parallel, Emergent and Distributed Syst. **27**, 169–192 (2012)

[52] Prusinkiewicz, P., Lindenmayer, A.: The Algorithmic Beauty of Plants. Springer, New York (1990)

[53] Rucker, R.: The Lifebox, the Seashell, and the Soul. Thunder's Mouth Press, New York (2005)

[54] Savage, J.E.: Models of Computation. Addison-Wesley, Reading (1998)

[55] Seife, C.: Decoding the Universe. Viking Penguin, New York (2006)

[56] Siegfried, T.: The Bit and the Pendulum. John Wiley & Sons, New York (2000)

[57] Sipser, M., Introduction to the Theory of Computation. PWS Publishing Company, Boston (1997)

[58] Steinhart, E.: Infinitely complex machines. In: Schuster, A. (ed.) Intelligent Computing Everywhere, pp. 25–43. Springer, New York (2007)

[59] Stepney, S.: Journeys in non-classical computation. In: Hoare, T., Milner, R. (eds.) Grand Challenges in Computing Research, pp. 29-32. BCS, Swindon (2004)

[60] Stepney, S.: The neglected pillar of material computation. Physica D **237**, 1157–1164 (2004)

[61] Tipler, F.J.: The Physics of Immortality: Modern Cosmology, God and the Resurrection of the Dead. Macmillan, London (1995)

[62] Toffoli, T.: Physics and Computation. Int. J. of Th. Phys. **21**, 165–175 (1982)

[63] Turing, A.M.: Systems of logic based on ordinals. Proc. of the London Math. Soc. 2 **45**, 161–228 (1939)

[64] Vedral, V.: Decoding Reality. Oxford University Press, Oxford (2010)

[65] Wegner, P., Goldin, D., Computation beyond Turing Machines. Comm. of the ACM **46**, 100–102 (1997)

[66] Wheeler, J.A.: Information, physics, quanta: The search for links. In: Proc. of the Third Int. Symp. on Foundations of Quantum Mechanics in Light of New Technology, pp. 354-368. Tokyo (1989)

[67] Wheeler, J.A.: Information, physics, quantum: the search for links. In: Zurek, W. (ed.) Complexity, Entropy, and the Physics of Information. Addison-Wesley, Redwood City (1990)

[68] Wheeler, J.A.: At Home in the Universe. American Institute of Physics Press, Woodbury (1994)

[69] Wolfram, S.: A New Kind of Science. Wolfram Media, Champaign (2002)

[70] Zuse, K.: Calculating space. MIT Technical Translation AZT-70-164-GEMIT, Massachusetts Institute of Technology (Project MAC). Cambridge (1970)