Unconventional Computational Problems

Selim G. Akl
School of Computing and Department of Mathematics and Statistics, Queen's University, Kingston, ON, Canada

Article Outline

Glossary

The *processor* of a computer is that component in charge of executing the operations of an algorithm.

A *time unit* is the length of time required by a processor to perform a *step* of its computation, consisting of three elementary operations: a *read* operation in which it receives a constant number of fixed-size data as input, a *calculate* operation in which it performs a fixed number of constant-time *arithmetic* and *logical* calculations (such as adding two numbers, comparing two numbers, and so on), and a *write* operation in which it returns a constant number of fixed-size data as output.

A *sequential computer,* consists of a single processor. A *parallel computer* has n processors, numbered 1 to n, where $n \geq 2$. Both computers use the same type of processor, and that processor is the fastest possible (Akl 1997). The assumption that the computers on hand, whether sequential or parallel, use the fastest conceivable processor is an important one. This is because the speed of the processor used is what specifies the duration of a time unit, as defined in the previous paragraph; the faster the processor, the smaller the time unit.

The *principle of simulation* in computer science states that any computation that can be performed successfully on a general-purpose computer A can be performed more or less efficiently but still successfully on another general-purpose computer B (Harel 1992; Lewis and Papadimitriou 1981; Mandrioli and Ghezzi 1987; Minsky 1967).

The *principle of universality,* which follows directly from the principle of simulation, states that there exists a *universal computer U,* with fixed specifications, capable of performing successfully any computation that can be performed on any other computer (Abramsky et al. 1992; Davis 2000; Denning et al. 1978; Hillis 1998; Hopcroft and Ullman 1969).

An *unconventional computational problem* is a computation requiring n algorithmic steps per time unit, where n is larger than 1. Such a computation cannot be performed successfully on a computer that can only perform a finite and fixed number m of algorithmic steps per time unit, where m is smaller than n (Akl 2005; Akl and Salay 2015; Burgin 2005; Calude and Paun 2005; Copeland 1998; Denning 2012; Deutsch 1997; Etesi and Németi 2002; Goldin and Wegner 2005; Nagy and Akl 2005, 2012; Siegelmann 1999; Stepney 2004; Toffoli 1982).

Introduction

The importance of unconventional computations stems from their ability to provide exceptions to the principles of simulation and universality:

1. There exist unconventional computational problems that are demonstrably solvable on a computer capable of n algorithmic steps per time unit, such as, for example, a parallel computer with n processors. Simulation of the latter solution on a computer capable of fewer than n algorithmic steps per time unit is demonstrably impossible.

A. Adamatzky (ed.), *Unconventional Computing*,
https://doi.org/10.1007/978-1-4939-6883-1_680

Originally published in
R. A. Meyers (ed.), *Encyclopedia of Complexity and Systems Science*, © Springer Science+Business Media LLC 2017
https://doi.org/10.1007/978-3-642-27737-5_680-1

2. No computer capable of a finite and fixed number of algorithmic steps per time unit can be universal.

Examples of unconventional computational problems are presented in the following section.

Unconventional Computations

Seven computational paradigms are described in what follows to illustrate the class of unconventional computational problems. Implications of these paradigms are then presented.

Unconventional Computational Paradigms

Each of the computations described in what follows can be characterized as being *inherently parallel*, due to the fact that it is executed successfully only on a computer capable of n algorithmic steps per time unit, a feature enjoyed by a parallel computer with n processors.

Computations Obeying Mathematical Constraints

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 $A[1]$, $A[2]$, ... , $A[n]$, 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

$$A[j] > A[j+1] > A[j+2], \tag{1}$$

for all $1 \leq j \leq n-2$.

2. When the sort terminates we have

$$A[1] < A[2] < \cdots < A[n]. \tag{2}$$

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) (Akl 1997). An input-dependent algorithm succeeds on a computer with $(n/2) - 1$ processors. However, a sequential computer, and a parallel computer 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

$$A[1] > A[2] > \cdots > A[n], \tag{3}$$

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

Time-Varying Computational Complexity

Here, the computational complexity of the problems at hand depends on *time* (rather than being, as usual, a function of the problem *size*). Thus, for example, tracking a moving object (such as a spaceship racing toward Mars) becomes harder as it travels away from the observer.

Suppose that 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 $C(t) = 2^t$ 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 no sequential computer, capable of exactly one algorithmic step per time unit, can perform this computation for $n \geq 3$. Indeed, $f_1(x_1)$ takes $C(0) = 2^0 = 1$ time unit, $f_2(x_2)$ takes another $C(1) = 2^1 = 2$ 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 parallel computer solves the problem handily. With all processors operating simultaneously, processor 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.

Rank-Varying Computational Complexity

Suppose that 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 $C(i)$, that 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 "Time-Varying Computational Complexity" applies by substituting the rank for the time.

Time-Varying Variables

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 value of an input variable $x_i(t)$ changes at the same speed as the processor in charge of evaluating the function $f_i(x_i(t))$.

A sequential computer 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 sequential computer attempts to first read all the x_i, one by one, and store them before calculating the f_i.

By contrast, a parallel computer consisting of 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 i computes $f_i(x_i(t_0))$, leading to a successful computation.

Interacting Variables

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.

A sequential computer 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 sequential 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 parallel computer 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).

Uncertain Time Constraints

In this paradigm, we are given a computation consisting of three distinct phases, 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, and this is what makes this paradigm truly unconventional, 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, which 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 parallel computer equipped with n processors can succeed in evaluating the two functions on time to meet the deadlines.

The Global Variable Paradigm

A computation C_0 consists of two distinct and separate processes P_0 and P_1 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_0 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_0. At the beginning of the computation, $x = 0$.

Let the processes of the computation C_0, namely, P_0 and P_1, be as follows:

P_0: **if** $x = 0$ **then** $x \leftarrow x + 1$ **else** loop forever **end if.**

P_1: **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_0 and P_1 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_0 succeeds, that is, that both P_0 and P_1 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_0 and P_1 have halted?

We examine two approaches to executing P_0 and P_1:

1. **Using a single processor:** Consider a sequential computer equipped, by definition, with a single processor p_0. The processor executes one of the two processes first. Suppose it starts with P_0: p_0 computes $x = 1$ and terminates. It then proceeds to execute P_1. Because now $x \neq 0$, p_0 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_1 and then executing P_0 would lead to a similar outcome, with the difference being that P_1 will return an incorrect value of *x,* namely, *y,* before switching to P_0, whereby it executes a nonterminating computation, given that now $x \neq 0$.
2. **Using two processors:** The two processors, namely, p_0 and p_1, are part of a shared-memory parallel computer in which two or more processors can read from, but not write to, the same memory location simultaneously (Akl 1997). In parallel, p_0 executes P_0 and p_1 executes P_1. 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 sequential (that is, 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_0 or P_1 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_0 or P_1 too late.

2. The second observation follows directly from the first. It is clear that P_0 and P_1 must be executed simultaneously for a proper outcome of the computation. The parallel (that is, 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_0. Time intervenes only in the circumstance where it is needed to signal that C_0 has failed (when the **"else"** part of an **"if"** statement, either in P_0 or in P_1, is executed). In other words, time is noticed solely when time requirements are neglected.

To generalize the global variable paradigm, we assume the presence of n global variables, namely, $x_0, x_1, \ldots, x_{n-1}$, all of which are time critical, and all of which are initialized to 0. There are also n nonzero local variables, namely, $y_0, y_1, \ldots, y_{n-1}$, belonging, respectively, to the n processes $P_0, P_1, \ldots, P_{n-1}$ that make up C_1. The computation C_1 is as follows:

P_0: **if** $x_0 = 0$ **then** $x_1 \leftarrow y_0$ **else** loop forever **end if.**

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_{n-2}: **if** $x_{n-2} = 0$ **then** $x_{n-1} \leftarrow y_{n-2}$ **else** loop forever **end if.**

P_{n-1}: **if** $x_{n-1} = 0$ **then** $x_0 \leftarrow y_{n-1}$ **else** loop forever **end if**.

Suppose that the computation C_1 begins when $x_i = 0$, for $i = 0, 1, \ldots, n-1$. For every i, $0 \leq i \leq n-1$, 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 x_i has been modified by $P_{(i-1) \bmod n}$ and is no longer equal to 0. On a parallel computer with n processors, namely, p_0, $p_1, \ldots, p_{n-1}$, it is possible to test all the x_i, $0 \leq i \leq n-1$, for equality to 0 in one time unit; this is followed by assigning to all the x_i, $0 \leq i \leq n-1$, their new values during the next time unit. Thus all the processes P_i, $0 \leq i \leq n-1$, and hence the computation C_1, terminate successfully. A sequential computer has but a single processor p_0 and, as a consequence, it fails to meet the time-critical requirements of C_1. At best, it can perform no more than $n-1$ of the n processes as required (assuming it executes the processes in the order $P_{n-1}, P_{n-2}, \ldots, P_1$, then fails at P_0 since x_0 was modified by P_{n-1}), and thus does not terminate. A parallel computer with only $n-1$ processors, $p_0, p_1, \ldots, p_{n-2}$, 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 (Fraser and Akl 2008) or, more generally, a Supertask Machine (Davies 2001; Earman and Norton 1996; Steinhart 2007)) would fail to perform the computations required by the global variable paradigm if it were restricted to execute these algorithmic steps *sequentially.*

Implications

Each of the computational problems described in section "Unconventional Computational Paradigms" 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. 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. Suppose 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.

The only constraint that is placed on the computer (or model of computation) that aspires to be universal is that the number of operations of which it 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, (Sipser 1997), 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, (Deutsch 1997), p. 210.

These two requirements are fundamental to the theory of computation in general, and to the theory of algorithms, in particular.

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 (Savage 1998), 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 (Akl 2006a). 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 (such as doubling it, or squaring it, or any such fixed acceleration) at a rate that is set in advance, 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 (Nagy and Akl 2007a). 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*).

Future Directions

Some approaches have been proposed in an attempt to meet the challenges to universality posed by unconventional computational problems. For example, mathematical logic has been used to address the time-varying variables paradigm (Bringsjord 2017). On another front (Akl 2010), closed timelike curves seem to overcome the difficulties of some of the problems in section "Unconventional Computational Paradigms", but not all. In particular, computations that are subject to mathematical constraints appear to have withstood all attacks. It is important to note that in order to salvage the principle of universality, all unconventional computational problems listed in section "Unconventional Computational Paradigms" (not just some) need to be shown solvable by a computer that obeys the finiteness condition. Alternatively, it appears that the only way to proceed may be to work on developing a universal computer such as the one described in the closing sentence of the previous section (Davies 2001).

Bibliography

(A) Primary Literature

Abramsky S et al (1992) Handbook of logic in computer science. Clarendon Press, Oxford

Akl SG (1997) Parallel computation: models and methods. Prentice Hall, Upper Saddle River

Akl SG (2005) The myth of universal computation. In: Trobec R, Zinterhof P, Vajtersic M, Uhl A (eds) Parallel numerics. University of Salzburg/Jozef Stefan Institute, Salzburg/Ljubljana, pp 211–236

Akl SG (2006a) Three counterexamples to dispel the myth of the universal computer. Parallel Proc Lett 16:381–403

Akl SG (2006b) Conventional or unconventional: is any computer universal? In: Adamatzky A, Teuscher C (eds) From utopian to genuine unconventional computers. Luniver Press, Frome, pp 101–136

Akl SG (2007a) Godel's incompleteness theorem and nonuniversality in computing. In: Nagy M, Nagy N (eds) Proceedings of the workshop on unconventional computational problems. Sixth International Conference on Unconventional Computation, Kingston, pp 1–23

Akl SG (2007b) Even accelerating machines are not universal. Int J Unconv Comput 3:105–121

Akl SG (2008a) Unconventional computational problems with consequences to universality. Int J Unconv Comput 4:89–98

Akl SG (2008b) Evolving computational systems. In: Rajasekaran S, Reif JH (eds) Parallel computing: models, algorithms, and applications. Taylor and Francis, Boca Raton, pp 1–22

Akl SG (2009) 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

Akl SG (2010) Time travel: a new hypercomputational paradigm. Int J Unconv Comput 6:329–351
Akl SG (2014) What is computation? Int J Parallel Emergent Distrib Syst 29:337–345
Akl SG (2016) Nonuniversality explained. Int J Parallel Emergent Distrib Syst 31:201–219
Akl SG (2017) Nonuniversality in computation: fifteen misconceptions rectified. In: Adamatzky A (ed) Advances in unconventional computing. Springer, Cham, pp 1–31
Akl SG 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
Akl SG, Nagy M (2009a) Introduction to parallel computation. In: Trobec R, Vajtersic M, Zinterhof P (eds) Parallel computing: numerics, applications, and trends. Springer, London, pp 43–80
Akl SG, Nagy M (2009b) The future of parallel computation. In: Trobec R, Vajtersic M, Zinterhof P (eds) Parallel computing: numerics, applications, and trends. Springer, London, pp 471–510
Akl SG, Salay N (2015) On computable numbers, nonuniversality, and the genuine power of parallelism. Int J Unconv Comput 11:283–297
Bringsjord S (2017) Is universal computation a myth? In: Adamatzky A (ed) Emergent computation: a festschrift for Selim G. Akl. Springer, Cham, pp 19–37
Burgin M (2005) Super-recursive algorithms. Springer, New York
Calude CS, Paun G (2004) Bio-steps beyond Turing. Biosystems 77:175–194
Copeland BJ (1998) Super Turing-machines. Complexity 4:30–32
Davies EB (2001) Building infinite machines. Br J Philos Sci 52:671–682
Davis M (2000) The universal computer. W.W. Norton, New York
Denning PJ (2012) Reflections on a symposium on computation. Comput J 55:799–802
Denning PJ, Dennis JB, Qualitz JE (1978) Machines, languages, and computation. Prentice-Hall, Englewood Cliffs
Deutsch D (1997) The fabric of reality. Penguin Books, London
Earman J, Norton JD (1996) Infinite pains: the trouble with supertasks. In: Morton A, Stich SP (eds) Benacerraf and his critics. Blackwell, Cambridge, pp 231–261
Etesi G, Nemeti I (2002) Non-Turing computations via Malament-Hogarth space-times. Int J Theor Phys 41:341–370
Fraser R, Akl SG (2008) Accelerating machines: a review. Int J Parallel Emergent Distrib Syst 23:81–104
Goldin D, Wegner P (2005) The church-Turing thesis: breaking the myth. In: Cooper BS, Lowe B (eds) New computational paradigms. Springer, Berlin, pp 152–168
Harel D (1992) Algorithmics: the spirit of computing. Addison-Wesley, Reading
Hillis D (1998) The pattern on the stone. Basic Books, New York
Hopcroft JE, Ullman JD (1969) Formal languages and their relations to automata. Addison-Wesley, Reading
Lewis HR, Papadimitriou CH (1981) Elements of the theory of computation. Prentice Hall, Englewood Cliffs
Mandrioli D, Ghezzi C (1987) Theoretical foundations of computer science. Wiley, New York
Minsky ML (1967) Computation: finite and infinite machines. Prentice-Hall, Englewood Cliffs
Nagy M, Akl SG (2005) On the importance of parallelism for quantum computation and the concept of a universal computer. In: Calude CS, Dinneen MJ, Paun G, de Perez-Jimenez, M. J, Rozenberg G (eds) Unconventional computation. Springer, Heildelberg, pp 176–190
Nagy M, Akl SG (2006) Quantum measurements and universal computation. Int J Unconv Comput 2:73–88
Nagy M, Akl SG (2007a) Quantum computing: beyond the limits of conventional computation. Int J Parallel Emergent Distrib Syst 22:123–135
Nagy M, Akl SG (2007b) Parallelism in quantum information processing defeats the universal computer. Par Proc Lett 17:233–262
Nagy N, Akl SG (2011) Computations with uncertain time constraints: effects on parallelism and universality. In: Calude CS, Kari J, Petre I, Rozenberg G (eds) Unconventional computation. Springer, Heidelberg, pp 152–163
Nagy N, Akl SG (2012) Computing with uncertainty and its implications to universality. Int J Parallel Emergent Distrib Syst 27:169–192
Savage JE (1998) Models of computation. Addison-Wesley, Reading
Siegelmann HT (1999) Neural networks and analog computation: beyond the Turing limit. Birkhauser, Boston
Sipser M (1997) Introduction to the theory of computation. PWS Publishing Company, Boston
Steinhart E (2007) Infinitely complex machines. In: Schuster A (ed) Intelligent computing everywhere. Springer, New York, pp 25–43
Stepney S (2004) The neglected pillar of material computation. Physica D 237:1157–1164
Toffoli T (1982) Physics and computation. Int J Theor Phys 21:165–175

(B) Books and Reviews

Akl SG (2004) Superlinear performance in real-time parallel computation. J Supercomput 29:89–111
Akl SG Non-universality in computation: the myth of the universal computer. School of Computing, Queen's University. http://research.cs.queensu.ca/Parallel/projects.html
Akl SG A computational challenge. School of computing, Queen's University http://www.cs.queensu.ca/home/akl/CHALLENGE/A_Computational_Challenge.htm
Akl SG, Yao W (2005) Parallel computation and measurement uncertainty in nonlinear dynamical systems. J Math Model Alg 4:5–15
Durand-Lose J (2004) Abstract geometrical computation for black hole computation. Research report no

2004–15, Laboratoire de l'lnformatique du Parallelisme, Ecole Nor-male Superieure de Lyon, Lyon
Einstein A (2009) Letter to Erwin Schrodinger. In: Gilder L (ed) The age of entanglement. Vintage Books, New York, p 170
Fortnow L The enduring legacy of the Turing machine. http://ubiquity.acm.org/article.cfm?id=1921573
Gleick J (2011) The information: a history, a theory, a flood. HarperCollins, London
Hypercomputation. http://en.wikipedia.org/wiki/Hypercomputation
Kelly K (2002) God is the machine. Wired 10. https://www.wired.com/2002/12/holytech/
Kleene SC (1952) Introduction to metamathematics. North Holland, Amsterdam
Lloyd S (2006) Programming the universe. Knopf, New York
Lloyd S, Ng YJ (2004) Black hole computers. Sci Am 291:53–61
Rucker R (2005) The lifebox, the seashell, and the soul. Thunder's Mouth Press, New York
Seife C (2006) Decoding the universe. Viking Penguin, New York
Siegfried T (2000) The bit and the pendulum. Wiley, New York
Stepney S (2004) Journeys in non-classical computation. In: Hoare T, Milner R (eds) Grand challenges in computing research. BCS, Swindon, pp 29–32
Tipler FJ (1995) The physics of immortality: modern cosmology, God and the resurrection of the dead. Macmillan, London
Turing AM (1939) Systems of logic based on ordinals. Proc London Math Soc 2 45:161–228
Vedral V (2010) Decoding reality. Oxford University Press, Oxford
Wegner P, Goldin D (1997) Computation beyond Turing machines. Comm ACM 46:100–102
Wheeler JA (1989) Information, physics, quanta: The search for links. In: Proceedings of the third international symposium on foundations of quantum mechanics in light of new technology,Tokyo, pp 354–368
Wheeler JA (1990) Information, physics, quantum: the search for links. In: Zurek W (ed) Complexity, entropy, and the physics of information. Addison-Wesley, Redwood City
Wheeler JA (1994) At home in the universe. American Institute of Physics Press, Wood-bury
Wolfram S (2002) A new kind of science. Wolfram Media, Champaign
Zuse K (1970) Calculating space. MIT Technical Translation AZT-70-164-GEMIT, Massachusetts Institute of Technology (Project MAC), Cambridge