Technical Report No. 2015-625
ON COMPUTABLE NUMBERS, NONUNIVERSALITY, AND THE GENUINE POWER OF PARALLELISM

Selim G. Akl* Nancy Salay†

July 29, 2015

Abstract

We present a simple example that disproves the universality principle. Unlike previous counter-examples to computational universality, it does not rely on extraneous phenomena, such as the availability of input variables that are time varying, computational complexity that changes with time or order of execution, physical variables that interact with each other, uncertain deadlines, or mathematical conditions among the variables that must be obeyed throughout the computation. In the most basic case of the new example, all that is used is a single pre-existing global variable whose value is modified by the computation itself. In addition, our example offers a new dimension for separating the computable from the uncomputable, while illustrating the power of parallelism in computation.

Keywords and phrases: nonuniversality in computation; universal computer; simulation; models of computation; time unit; time-varying variables; time-varying computational complexity; rank-varying computational complexity; interacting variables; uncertain time constraints; mathematical constraints; finite and fixed number of operations per time unit, general-purpose computer; finiteness condition; unbounded space; unbounded time; communication with the outside world; Turing Machine; Random Access Machine; Parallel Random Access Machine.

1 Introduction

The universality principle is the cornerstone of computing and the reason for the rapid ascendancy of the discipline as the most influential science of our time. According to this principle, any general-purpose computer A can execute, through simulation, and more or less efficiently, any computation that is possible on any other general-purpose computer B [42]. In essence, the principle expresses a deep and important insight into the relationship between computability and universality. Perspicuously stated, it says that a function is

*School of Computing and Department of Mathematics and Statistics, Queen's University, Kingston, Ontario, Canada K7L 3N6, akl@cs.queensu.ca

†Department of Philosophy and School of Computing, Queen's University, Kingston, Ontario, Canada K7L 3N6, salay@queensu.ca

computable if and only if its value can be obtained by simulation on any general-purpose computer [1, 26, 29, 36, 37, 38, 42, 44, 45, 56]. Here, general-purpose computers, our domain of discourse, are to be understood as ones that are defined and fixed once and for all; the capabilities of a general-purpose computer are never modified in order to fit the computational problem to be solved. In theoretical computing, general-purpose computers are represented using computational models such as the Turing Machine, the Random Access Machine (RAM), the Cellular Automaton, and the like [54]. In practice, they are the processors in our tablets, our mobile phones, our cars, and so on. It follows from the universality principle that any function that can be evaluated on any general-purpose computer is a computable function. In other words, being universally simulatable is a sufficient condition of computability. It also follows from this principle that a function that is not universally simulatable must not be computable. In other words, being universally simulatable is a necessary condition of computability. A function that *could not* be evaluated on some general-purpose computer, but *could*, nevertheless, be computed by another, would be a counter-example to the necessity clause of the universality principle and would show us that the connection between computability and universal simulatability is weaker than is generally assumed: simulatability is sufficient for computability, but it is not necessary; that is, a function can be computable in some contexts, but not in all contexts.

The universality principle does in fact hold for *conventional* computations, such as, for example, sorting into non-decreasing order a list of numbers that are given in arbitrary order, searching a list for a given datum, numerical computation, text processing, and so on. To illustrate, consider the following parallel algorithm for sorting a sequence $S = g_0, g_1, \ldots, g_{n-1}$ of n distinct integers on a linear array of processors $p_0, p_1, \ldots, p_{n-1}$. Processor p_i contains g_i and can communicate with its two neighbours p_{i-1} and p_{i+1} (except for p_0 and p_{n-1} which have only one neighbour each, namely, p_1 and p_{n-2}, respectively). In the "compare and swap if needed" operation of the algorithm, processors p_i and p_{i+1} compare their integers, placing the smaller in p_i and the larger in p_{i+1}.

Parallel Sort

for $k = 0$ **to** $n - 1$ **do**
  for $i = 0$ **to** $n - 2$ **do in parallel**
    if $i \bmod 2 = k \bmod 2$
    then p_i and p_{i+1} compare and swap if needed
    end if
  end for
end for. ■

Algorithm Parallel Sort completes the sort in $O(n)$ time [3]. If it so happens that only one processor, namely p_0, is available, then the parallel algorithm can be easily simulated by having the single processor methodically imitate the operations of the n processors. The sequential solution uses an array S to store the sequence to be sorted. Initially, $S[i]$ contains g_i, for $i = 0, 1, \ldots, n - 1$. The algorithm is given in what follows. In it, the operation "compare $S[i]$ and $S[i + 1]$ and swap if needed" compares the two integers currently in $S[i]$ and $S[i + 1]$, placing, as a result, the smaller in $S[i]$ and the larger in $S[i + 1]$.

Sequential Sort

for $k = 0$ **to** $n - 1$ **do**
  for $i = 0$ **to** $n - 2$ **do**
    if $i \bmod 2 = k \bmod 2$
    then compare $S[i]$ and $S[i + 1]$ and swap if needed
    end if
  end for
end for. ■

Algorithm Sequential Sort completes the sort in $O(n^2)$ time (clearly not the best sorting algorithm sequentially, but a sufficient illustration of the idea of simulation for our purposes).

But here's the rub: simulation is always feasible *only for conventional computations*. Several classes of *unconventional computations* have been uncovered recently for which simulation is not always possible and, consequently, for which universality does not hold. These classes include computations that involve time-varying variables, time-varying computational complexity, rank-varying computational complexity, interacting variables, uncertain time constraints, mathematical constraints, and so on [11, 12, 13, 14, 15, 16, 17, 18]. While these unconventional computations can be executed successfully on certain computers, they cannot be simulated on a unique fixed computer. Because simulation is not always possible, the universality principle as currently understood is false. This conclusion is referred to as nonuniversality in computation [6, 7, 8, 9, 10].

Let time be divided into discrete time units. The nonuniversality result is usually stated as follows: no computer U can be universal if it is capable of only a finite and fixed number of basic arithmetic and logical operations, such as addition, comparison, exclusive-or, and so on, per time unit.

> **Nonuniversality Proof:** Assume that computer U can perform $D(i)$ operations during time unit i of a computation, for $i = 0, 1, \ldots$ For any computation C requiring $E(i)$ operations during time unit i, where $E(i) > D(i)$ for at least one i, U will fail to successfully complete C. Therefore, U cannot be universal. Note that C is computable on another computer U' capable of $E(i)$ operations during time unit i. However, U', in turn, will be defeated by another computation C' requiring $F(i)$ operations during time unit i, where $F(i) > E(i)$, for at least one i, and consequently U' cannot be universal either.

One example of such a computation C calls for sorting an input sequence of elements, while imposing an extra condition to be satisfied by any candidate algorithm. Thus, in this unconventional version of the standard sorting problem presented earlier in this section, a sequence $S = g_0, g_1, \ldots, g_{n-1}$ of n distinct integers is given. It is required to transform the sequence S *in situ* into a sequence $S' = g'_0, g'_1, \ldots, g'_{n-1}$ whose elements are the same as those of S, with the difference that, at the end of the computation, $g'_0 < g'_1 < g'_2 < \cdots < g'_{n-2} < g'_{n-1}$. So far, this is the classic sorting problem. The unconventional variant adds a new requirement: at no time, once the sorting process has begun, should there be three consecutive elements of an intermediate sequence $S'' = g''_0, g''_1, \ldots, g''_{n-1}$, such that $g''_i > g''_{i+1} > g''_{i+2}$, for $i = 0, 1, \ldots, n - 3$. A complete description of this example, and

its implications can be found in [14]. It suffices to note here that algorithm Parallel Sort succeeds in carrying out this computation for all input sequences S of size n, while algorithm Sequential Sort fails when presented with a sequence $S = g_0, g_1, \ldots, g_{n-1}$ in which $g_0 > g_1 > g_2 > \cdots > g_{n-2} > g_{n-1}$. A parallel computer with fewer than $n-2$ processors also fails to solve this problem.

Every reasonable model of computation is, by definition, capable of only a finite and fixed number of operations per time unit. The same is obviously true for any *practical* computer which is built once and for all; it too can only perform a finite (and fixed) number of operations per time unit. Given the nonuniversality proof, it follows that unless the unreasonable assumption is made that a computer is capable at the outset of an infinite number of operations per time unit, universality cannot be achieved by any computer [6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 46, 47, 48, 49, 50, 51].

In this paper we present an even stronger result: there are computable functions that are not computable universally, even on systems capable of an infinite number of operations per time unit, so long as the general purpose computers in question are constrained to perform operations *sequentially*. This is supported by the simplest counter-example to the universality principle of which we are aware. As a bonus, the counter-example that we propose illustrates the true, often unappreciated, power of the idea of parallelism in computation: parallelism does not just speed up sequential computations; it makes certain computations *possible*. An example of the lack of appreciation of what parallelism brings to computing is the *Speedup Theorem* and specifically its 'proof' [2, 35, 39, 40, 41]. This theorem states that the best sequential (that is, single-processor computer) solution to a given problem P can be sped up, at most, by a factor of n if an n-processor parallel computer is used instead. The proof goes as follows.

> Let t_1 be the running time of the best sequential algorithm for P, and let t_n be the running time of a parallel algorithm. Assume that the speedup t_1/t_n is larger than n. In that case we can simply simulate the parallel algorithm on a single processor, resulting in a running time of $n \times t_n < t_1$, which is impossible, since t_1 is already, by definition, the best possible sequential running time. The assumption is therefore false and t_1/t_n cannot be larger than n.

A demonstration of the fallacy of this 'theorem', through several counter-examples that achieve speedups *exponential* in n, is provided in [3, 4], where a number of additional references can also be found. Even in popular science writing, claims can be found to the effect that parallel computing can do no more, in principle, than sequential computing [52].

The remainder of this paper is organized as follows. Our counter-example is described in Section 2. Some consequences of our result are derived in Section 3. In Section 4 we generalize our counter-example using two different models of parallel computation. Conclusions are offered in Section 5.

2 The global variable paradigm

Our computation, call it 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 perhaps 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, based, for example, on the RAM model of computation [24], 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, based, for example, on the Concurrent-Read Exclusive-Write Parallel Random Access Machine (CREW PRAM) model of computation [3]. In this model, two or more processors can read from, but not write to, the same memory location simultaneously. 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.

Finally, 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.

3 Consequences

The two-process computation C_0 of Section 2 shows that no sequential (that is, uniprocessor) computer can ever be universal. Even if it is given an unbounded amount of memory and an unlimited amount of time (like a Turing Machine, for example), processor p_0 fails to solve the problem. Even if it is permitted interaction with the outside world (unlike a Turing Machine), p_0 fails. Finally, and most importantly for our purposes in this paper, even if p_0 is capable of an infinite number of *sequential operations* per time unit (like an Accelerating Machine [33] or, more generally, a Supertask Machine [27, 30, 58]), it still fails to meet the requirements of the computation C_0.

Notice that the parallel (that is, multiprocessor) computer succeeded in performing C_0 satisfactorily. This demonstrates an important and often overlooked feature of parallelism: far from being simply a faster alternative to sequential computing, it is essential for the success of certain inherently parallel computations [18, 19, 46, 47, 48, 49, 50, 51]. The two-process problem is *uncomputable* by a sequential computer and *computable* by a parallel one. Thus, the example not only serves to make the nonuniversality result more general and therefore stronger, it also offers a new way to distinguish computability from uncomputability via sequential and parallel computing.

Does this mean that the parallel computer is universal? Certainly not, for it is possible to construct a computation with three processes, namely, P_0, P_1, and P_2, for which a two-processor computer fails. A three-processor computer may succeed, but it will then be thwarted by a four-process computation. Such reasoning continues indefinitely. Taking this argument to its logical conclusion, only a computer capable of an infinite number of *parallel* operations per time unit can be universal.

4 Generalizations

Various options are available to generalize our result. In this section, we describe two such generalizations. Recall that in Section 2 we used the CREW PRAM as the parallel model of computation. In this model, several processors can read simultaneously from the same shared memory location, but no simultaneous write is allowed. Two alternative shared memory parallel models are the Exclusive Read Exclusive Write Parallel Random Access Machine (EREW PRAM) and the Concurrent-Read Concurrent-Write Parallel Random Access Machine (CRCW PRAM) [3]. In the EREW PRAM, at most one processor can gain access to a shared memory location during a time unit, either for reading or for writing. In the CRCW PRAM, a shared memory location can be accessed simultaneously by several processors during a time unit, either for reading by all of them (when executing a concurrent-read instruction) or for writing by all of them (when executing a concurrent-write instruction). In the latter case, memory conflicts are resolved in a variety of ways, including the *priority* concurrent-write instruction, where the processor with the highest writing priority succeeds in writing and all others fail, the *common* concurrent-write instruction, where the write operation succeeds if and only if all processors are attempting to write the same value, and the *combining* concurrent-write instruction, where all the values being written are combined into one (using, for example, the *arithmetic sum*, the *logical and*, the *maximum*, and so on) [3]. For our purposes in this paper, we shall use the *combining with arithmetic sum* as our write instruction.

Let C_1 and C_2 be the two generalizations of the computation C_0, to be proposed in Sections 4.1 and 4.2, respectively. Both C_1 and C_2 use the idea hinted to in Section 3, whereby several processes are part of the computation to be carried out. We will show that C_1 is possible if an n-processor EREW PRAM is available, while an n-processor CRCW PRAM is needed to execute C_2. Furthermore, both C_1 and C_2 cannot be performed successfully, neither by a RAM nor by a PRAM, of any type, equipped with fewer than n processors.

4.1 Using several global and local variables

In our first generalization of the example in Section 2, 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**.

$\vdots$

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 an EREW PRAM 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 RAM 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. An EREW PRAM 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.

4.2 Using a single global variable and no local variable

Our second generalization of the example in Section 2 requires the presence of only a single, time-critical, global variable x. Let $x = 0$ initially. With n processes, $P_0, P_1, \ldots, P_{n-1}$, the computation C_2 looks as follows:

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

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

P_2: **if** $x = 0$ **then** $x \leftarrow 1$ **else** loop forever **end if**.

$\vdots$

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

If the computation C_2 starts when $x = 0$, it is required that the "**then** $x \leftarrow 1$" operation be performed *as soon as* it is determined that x is indeed equal to 0.

The n processors of a CRCW PRAM, namely p_0, p_1, $\ldots$, p_{n-1}, read x in parallel, find it equal to 0, and simultaneously increment x, by 1 each, resulting in $x = n$. Now all the processes, and hence the computation C_2, halt gracefully. A single-processor computer is hopeless to perform this computation, but so also is the n-processor CRCW PRAM if presented with an $n+1$ process version of C_2; they will both run forever.

5 Conclusion

Despite considerable evidence to the contrary [15, 16, 21, 22, 23, 28, 31, 34, 55, 57, 59, 60, 62, 63], belief in the universality principle, particularly (but not exclusively) in connection with the Turing Machine, remains one of the most enduring myths in computer science (see, for example, [5, 32, 38]). In this paper we presented a new counter-example to it, the simplest such counter-example of which we are aware. Unlike previous counter-examples to computational universality, it does not rely on extraneous phenomena, such as the availability of

input variables that are time varying, computational complexity that changes with time or order of execution, physical variables that interact with each other, uncertain deadlines, or mathematical conditions among the variables that must be obeyed throughout the computation. In the most basic case of the new example, all that is used is a single pre-existing global variable whose value is modified by the computation itself.

Further to its extreme simplicity, this new nonuniversality result is more powerful than earlier ones. It was previously thought, based on past counter-examples, that only a computer capable of an infinite number of basic operations per time unit could be universal [6]. We have shown in this paper that even a computer capable of an infinite number of basic *sequential* operations cannot be universal. Thus, computational universality requires an infinite number of basic *parallel* operations per time unit.

In his classic, discipline-creating paper [61], Alan Turing defined what it means for a number to be computable or uncomputable. The distinction is made by fixing a model of computation and determining whether or not that model is capable of producing a desired number. Thus, the mathematical constants π and e, for example, are computable (to a desired precision). By contrast, there are uncomputable numbers, namely, those that are the outcome of unsolvable problems such as, for example, the *Halting Problem* [25]. This conventional distinction between the computable and the uncomputable has hitherto been adopted, almost universally, with respect to the Turing Machine, as the 'ultimate' model of computation, the baseline. The present paper provides an alternative but complementary way to distinguish between computable and uncomputable numbers. While the background of Turing's distinction is a fixed model of computation, our examples exploit the fact that there exist multiple possible general-purpose computer models, not all equivalent. A number that may be uncomputable on some models, may be computable on others. For example, the number x in C_0 is computable on a parallel computer with the proper number of processors, but uncomputable otherwise (whether sequentially or in parallel). The same is true for the numbers in C_1 and C_2. Our examples, therefore, offer up parallelism as a new baseline model for computation, acknowledging that other models yet to be dreamed up will eventually replace it [46].

Our counter-example to universality also serves to illustrate the importance of parallelism in computing. Virtually the entire body of literature on parallel computation suggests that the *raison d'être* of parallel computers is to speed up sequential computations [20, 43, 53, 64]. We have shown here that parallel computing is considerably more valuable since it can make the difference between computability and uncomputability. Specifically, we have identified a problem that a parallel computer can solve while a sequential computer cannot. In other words, the set of problems solvable in parallel is a strict superset of the set of problems solved sequentially. Therefore, on the hierarchy of computational models, in which models are ranked by their power [42, 54], a parallel computer is strictly more powerful than a sequential one.

In summary, our paper offers three contributions. It strengthens the notion of nonuniversality in computation by extending the domain in which it holds to all sequential machines, even those capable of an infinite number of operations per time unit; it offers a new unconventional way to distinguish between what is computable and what is uncomputable; and it puts in sharp focus an important difference between sequential and parallel computing.

References

[1] Abramsky, S. et al, *Handbook of Logic in Computer Science*, Clarendon Press, Oxford, 1992.

[2] Akl, S.G., *The Design and Analysis of Parallel Algorithms*, Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[3] Akl, S.G., *Parallel Computation: Models and Methods*, Prentice Hall, Upper Saddle River, New Jersey, 1997.

[4] Akl, S.G., Superlinear performance in real-time parallel computation, *The Journal of Supercomputing*, Vol. 29, No. 1, 2004, pp. 89–111

[5] Akl, S.G., Universality in computation: Some quotes of interest, Technical Report No. 2006-511, School of Computing, Queen's University, Kingston, Ontario, April 2006, 13 pages. `http://www.cs.queensu.ca/home/akl/techreports/quotes.pdf`

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

[7] Akl, S.G., Conventional or Unconventional: Is Any Computer Universal?, Chapter 6 in: *From Utopian to Genuine Unconventional Computers*, A. Adamatzky and C. Teuscher, Eds., Luniver Press, Frome, United Kingdom, 2006, pp. 101–136.

[8] Akl, S.G., Gödel's incompleteness theorem and nonuniversality in computing, *Proceedings of the Workshop on Unconventional Computational Problems*, Sixth International Conference on Unconventional Computation, Kingston, Canada, August 2007, pp. 1–23.

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

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

[11] Akl, S.G., Evolving Computational Systems, Chapter 1 in: *Parallel Computing: Models, Algorithms, and Applications*, S. Rajasekaran and J.H. Reif, Eds., Taylor and Francis, CRC Press, Boca Raton, Florida, 2008, pp. 1–22.

[12] 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, United Kingdom, 2009.

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

[14] Akl, S.G., What is computation?, *International Journal of Parallel, Emergent and Distributed Systems*, Vol. 29, No. 4, 2014, pp. 337–345.

[15] Akl, S.G., Nonuniversality in computation: Fifteen misconceptions rectified, Chapter in: *Advances in Unconventional Computing*, A. Adamatzky, Ed., Springer, 2015.

[16] Akl, S.G., Nonuniversality explained, to appear in *International Journal of Parallel, Emergent and Distributed Systems*.

[17] Akl, S.G. and Nagy, M., Introduction to Parallel Computation, Chapter 2 in: *Parallel Computing: Numerics, Applications, and Trends*, R. Trobec, M. Vajteršic, and P. Zinterhof, Eds., Springer-Verlag, London, United Kingdom, 2009, pp. 43–80.

[18] Akl, S.G. and Nagy, M., The Future of Parallel Computation, Chapter 15 in: *Parallel Computing: Numerics, Applications, and Trends*, R. Trobec, M. Vajteršic, and P. Zinterhof, Eds., Springer-Verlag, London, United Kingdom, 2009, pp. 471–510.

[19] Akl, S.G. and Yao, W., Parallel computation and measurement uncertainty in nonlinear dynamical systems, *Journal of Mathematical Modelling and Algorithms*, Special Issue on Parallel and Scientific Computations with Applications, Vol. 4, 2005, pp. 5–15.

[20] Blazewicz, J., Ecker, K., Plateau, B., and Trystram, D., Eds., *Handbook on Parallel and Distributed Processing*, Springer Verlag, Berlin, 2000.

[21] Burgin, M., *Super-Recursive Algorithms*, Springer, New York, 2005.

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

[23] Copeland, B.J., Super Turing-machines, *Complexity*, Vol. 4, 1998, pp. 30–32.

[24] Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C., *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, 2009.

[25] Davis, M., *Computability and Unsolvability*, McGraw-Hill, New York, 1958.

[26] Davis, M., *The Universal Computer*, W.W. Norton, 2000.

[27] Davies, E.B., Building infinite machines, *British Journal for Philosophy of Science*, Vol. 52, 2001, pp. 671–682.

[28] Denning, P.J., Reflections on a Symposium on Computation, *The Computer Journal*, Vol. 55, No. 7, 2012, pp. 799–802.

[29] Deutsch, D., *The Fabric of Reality*, Penguin Books, London, United Kingdom, 1997.

[30] Earman, J. and Norton, J.D., Infinite pains: The trouble with supertasks, in: *Benacerraf and his Critics*, A. Morton and S.P. Stich, Eds., Blackwell, Cambridge, Massachusetts, 1996, pp. 231–261.

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

[32] Fortnow, L., The enduring legacy of the Turing machine, *The Computer Journal*, Vol. 55, No. 7, 2012, pp. 830–831.

[33] Fraser, R. and Akl, S.G., Accelerating machines: a review, *International Journal of Parallel Emergent and Distributed Systems*, Vol. 23, No. 1, February 2008, pp. 81–104.

[34] Goldin, D. and Wegner, P., The Church-Turing thesis: Breaking the myth, *Proceedings of the First international conference on Computability in Europe: New Computational Paradigms*, Springer-Verlag, Berlin, 2005, pp. 152–168.

[35] Greenlaw, R., Hoover, H.J., and Ruzzo, W.L., *Limits to Parallel Computation*, Oxford University Press, New York, 1995.

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

[37] Hillis, D., *The Pattern on the Stone*, Basic Books, New York, New York, 1998.

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

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

[40] Kronsjö, L., *Computational Complexity of Sequential and Parallel Algorithms*, John Wiley & Sons, New York, 1985.

[41] Leighton, F.T., *Introduction to Parallel Algorithms and Architectures*, Morgan Kaufmann, San Mateo, California, 1992.

[42] Lewis, H.R. and Papadimitriou, C.H., *Elements of the Theory of Computation*, Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[43] Lewis, T.G. and El-Rewini, H., *Introduction to Parallel Computing*, Prentice Hall, Englewood Cliffs, New Jersey, 1992.

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

[45] Minsky, M.L., *Computation: Finite and Infinite Machines*, Prentice-Hall, 1967.

[46] Nagy, M. and Akl, S.G., On the importance of parallelism for quantum computation and the concept of a universal computer, *Proceedings of the Fourth International Conference on Unconventional Computation*, Sevilla, Spain, October 2005, LNCS 3699, pp. 176-190.

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

[48] Nagy, M. and Akl, S.G., Quantum computing: Beyond the limits of conventional computation, *International Journal of Parallel, Emergent and Distributed Systems*, Special Issue on Emergent Computation, Vol. 22, No. 2, April 2007, pp. 123–135.

[49] Nagy, M. and Akl, S.G., Parallelism in quantum information processing defeats the Universal Computer, *Proceedings of the Workshop on Unconventional Computational Problems*, Sixth International Conference on Unconventional Computation, Kingston, Canada, August 2007, pp. 25–52; also in: *Parallel Processing Letters*, Special Issue on Unconventional Computational Problems, Vol. 17, No. 3, September 2007, pp. 233 - 262.

[50] Nagy, N. and Akl, S.G., Computations with uncertain time constraints: Effects on parallelism and universality, *Proceedings of the Tenth International Conference on Unconventional Computation*, Turku, Finland, June 2011, LNCS 6714, pp. 152–163.

[51] Nagy, N. and Akl, S.G., Computing with uncertainty and its implications to universality, *International Journal of Parallel, Emergent and Distributed Systems*, Vol. 27, Issue 2, April 2012, pp. 169–192.

[52] Penrose, R., *The Emperor's New Mind*, Oxford University Press, New York, 1989.

[53] Rajasekaran, S. and Reif, J.H., Eds., *Parallel Computing: Models, Algorithms, and Applications*, Taylor and Francis, CRC Press, Boca Raton, Florida, 2008.

[54] Savage, J.E., *Models of Computation*, Addison-Wesley, 1998.

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

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

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

[58] Steinhart, E., Infinitely complex machines, in: *Intelligent Computing Everywhere*, A. Schuster, Ed., Springer, New York, 2007, pp. 25–43.

[59] Stepney, S., Non-classical hypercomputation, *International Journal of Unconventional Computing*, Vol. 5, Nos. 3-4, 2009, pp. 267–276.

[60] Syropoulos, A., *Hypercomputation*, Springer, New York, 2008.

[61] A.M. Turing, On computable numbers with an application to the Entscheidungsproblem, *Proceedings of the London mathematical Society*, Ser. 2, Vol. 42, 1936, pp. 230–265; Vol. 43, 1937, pp. 544–546.

[62] Van Leeuwen, J. and Wiedermann J., The Turing machine paradigm in contemporary computing, in: *Mathematics Unlimited - 2001 and Beyond*, B. Engquist and W. Schmidt, Eds., Springer-Verlag, Berlin, 2000, pp. 1139–1156.

[63] Wegner, P., Why interaction is more powerful than algorithms, *Communications of the ACM*, Vol. 40, No. 5, May 1997, pp. 80–91.

[64] Zomaya, A.Y., Ed., *Parallel and Distributed Computing Handbook*, McGraw-Hill, New York, 1996.