INHERENTLY PARALLEL COMPUTATIONS

Selim G. Akl
School of Computing
Queen's University
Kingston, Ontario, Canada K7L 3N6

The term *inherently parallel computation* refers to a computation that can be performed successfully only on a parallel computer with an appropriate number of processors. Examples are computations that involve: time-varying variables, time-varying complexity, rank-varying complexity, interacting variables, uncertain time constraints, and variables obeying mathematical constraints. These computations also provide counterexamples to the existence of a universal computer [2]-[9], [13], [18]-[21].

1 Computational Models

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 $n \geq 2$.

2 Time-Varying Variables

For a positive integer n larger than 1, we are given n functions, each of one variable, namely, F_0, F_1, ..., F_{n-1}, operating on the n physical variables x_0, x_1, ..., x_{n-1}, respectively. Specifically, it is required to compute $F_i(x_i)$, for $i = 0, 1, \ldots, n-1$. 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 functions that vary with time. It is therefore appropriate to write the n variables as $x_0(t), x_1(t), \ldots, x_{n-1}(t)$, that is, as functions of the time variable t. It is important to note here that, while it is known that the x_i change with time, the actual functions that effect these changes are not known (for example, x_i may be a true random variable). It takes one time unit to evaluate $F_i(x_i(t))$. The problem calls for computing $F_i(x_i(t))$, $0 \leq i \leq n-1$, at time $t = t_0$. The fact that $x_i(t)$ changes with the passage of time means that, for $k > 0$, not only is each value $x_i(t_0 + k)$ different from $x_i(t_0)$, but also the latter cannot be obtained from the former.

A sequential computer fails to compute all the F_i as desired. Indeed, suppose that $x_0(t_0)$ is initially operated upon. By the time $F_0(x_0(t_0))$ 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 $0 \leq i \leq n-1$, and all processors working at the same time, processor i computes $F_i(x_i(t_0))$, leading to a successful computation.

3 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 towards 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, $f_0(x_0)$, $f_1(x_1)$, ..., $f_{n-1}(x_{n-1})$, be computed. Computing $f_i(x_i)$ at time t requires $C(t) = 2^t$ operations, for $t \geq 0$ and $0 \leq i \leq n-1$. Further, there is a deadline for reporting the results of the computations: All n values $f_0(x_0)$, $f_1(x_1)$, ..., $f_{n-1}(x_{n-1})$ 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 constant-time operation per step (that is, per time unit), can perform this computation for $n \geq 3$. Indeed, $f_0(x_0)$ takes $C(0) = 2^0 = 1$ time unit, $f_1(x_1)$ takes another $C(1) = 2^1 = 2$ time units, by which time three time units would have elapsed. At this point none of $f_2(x_2), \ldots, f_{n-1}(x_{n-1})$ 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 $0 \leq i \leq n-1$. This consumes one time unit, and the deadline is met.

4 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 operations 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 the previous section applies by substituting the rank for the time.

5 Interacting Variables

A physical system has n variables, x_0, x_1, ..., x_{n-1}, 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 unpredictably.

A sequential computer measures *one* of the values (x_0, for example) and by so doing it disturbs the equilibrium, 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_0 has received its new value, setting x_1 disturbs x_0 unpredictably.

A parallel computer with n processors, by contrast, will measure *all* the variables x_0, x_1, ..., x_{n-1} 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 $x_0, x_1, \ldots, x_{n-1}$ can be computed in parallel and applied to the system simultaneously (one value per processor).

6 Uncertain Time Constraints

In this paradigm, we are given a computation each of whose components, namely, the input phase, the calculation phase, and the output phase, needs to be computed 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, call them 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 they 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.

7 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 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.)

7.1 Rewriting Systems

An example of such transformations is provided by *rewriting systems*. From an initial string ab, in some formal language consisting of the two symbols a and b, it is required to generate the string $(ab)^n$, for $n > 1$. Thus, for $n = 3$, the target string is $ababab$. The rewrite rules to be used are: $a \rightarrow ab$ and $b \rightarrow ab$. Throughout the computation, no intermediate string should have two adjacent identical characters. Such rewrite systems (also known as $\mathcal{L}$-systems) are used to draw fractals and model plant growth [22]. Here we note that applying any *one* of the two rules at a time causes the computation to fail (for example, if ab is changed to abb, by the first rewrite rule, or to aab by the second).

A sequential computer can change only one symbol at once, thereby causing the adjacency condition to be violated. By contrast, for a given n, a parallel computer with n processors can easily perform a transformation on all the inputs collectively. The required property is maintained leading to a successful computation. Thus, the string $(ab)^n$ is obtained in $\log n$ steps, with the two rewrite rules being applied simultaneously to all symbols in the current intermediate string, in the following manner: ab, $abab$, $abababab$, and so on. It is interesting to observe that a successful generation of $(ab)^n$ also provides an example of a rank-varying computational complexity (as described in Section 4). Indeed, each legal string (that is, each string generated by the rules and obeying the adjacency property) is twice as long as its predecessor (and hence requires twice as many operations to be generated).

7.2 Sorting Variant

A second example of computations obeying a mathematical constraint is provided by a variant to the problem of sorting. For a positive even integer n, where $n \geq 8$, let n distinct integers be stored in an array A with n locations $A[0]$, $A[1]$, ..., $A[n-1]$, one integer per location. Thus $A[j]$, for all $0 \leq j \leq n-1$, represents the integer currently stored in the jth location of A. It is required to sort the n integers in place into increasing order, such that:

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

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

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

2. When the sort terminates we have:

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

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

An algorithm for an $n/2$-processor parallel computer solves the aforementioned variant of the sorting problem handily by means of pairwise swaps applied to the input array A, during each of which $A[j]$ and $A[k]$ exchange positions (using an additional memory location for temporary storage). 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

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

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

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

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

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

This is a surprising result as it goes against the common belief that any computation by a variant of the Turing Machine can be effectively simulated by the standard model [15].

8 The Universal Computer Is A Myth

The Principle of Simulation is the cornerstone of computer science. It is at the heart of most theoretical results and practical implements of the field such as programming languages, operating systems, and so on. The principle states that *any* computation that can be performed on *any* one general-purpose computer can be equally carried out through simulation on *any* other general-purpose computer [12, 14, 17]. At times, the imitated computation, running on the second computer, may be faster or slower depending on the computers involved. In order to avoid having to refer to different computers when conducting theoretical analyses, it is a generally accepted approach to define a model of computation that can simulate *all* computations by other computers. This model would be known as a Universal Computer $\mathcal{U}$. Thus, Universal Computation, which clearly rests on the Principle of Simulation, is also one of the foundational concepts in the field [11].

Our purpose here is to prove the following general statement: There does not exist a *finite* computational device that can be called a Universal Computer. Our reasoning proceeds as follows. Suppose there exists a Universal Computer capable of n elementary operations per step, where n is a finite and fixed integer. This

computer will fail to perform a computation *requiring* n' operations per step, for any $n' > n$, and consequently lose its claim of universality. Naturally, for each $n' > n$, another computer capable of n' operations per step will succeed in performing the aforementioned computation. However, this new computer will in turn be defeated by a problem requiring $n'' > n'$ operations per step.

This reasoning is supported by each of the computational problems presented in Sections 2–7. As we have seen, these problems *can* easily be solved by a computer capable of executing n operations at every step. Specifically, an n-processor parallel computer led to a successful computation in each case. However, *none* of these problems is solvable by any computer capable of at most $n-1$ operations per step, for any integer $n > 1$. Furthermore, the problem size n itself is a variable that changes with each problem instance. As a result, *no* parallel computer, regardless of how many processors it has available, can cope with a growing problem size, as long as the number of processors is finite and fixed. 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 preceding reasoning applies to any computer that obeys the *finiteness condition*, that is, a computer capable of only a finite and fixed number of operations per step. 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 [23], as well as all of today's general-purpose computers, including existing conventional computers (both sequential and parallel), as well as contemplated unconventional ones such as biological and quantum computers [3].

Therefore, the Universal Computer $\mathcal{U}$ is clearly a myth. As a consequence, the Principle of Simulation itself (though it applies to most *conventional* computations) is, in general, a fallacy. In fact, the latter principle is responsible for many other myths in computing, such as the *Speedup Theorem*, the *Slowdown Theorem*, and *Amdahl's Law*. Counterexamples for dispelling these and other myths are presented in [1, 10].

References

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

[2] Akl, S.G., The myth of universal computation, in: *Parallel Numerics*, Trobec, R., Zinterhof, P., Vajteršic, M., and Uhl, A., Eds., Part 2, Systems and Simulation, University of Salzburg, Salzburg, Austria and Jožef Stefan Institute, Ljubljana, Slovenia, 2005, pp. 211–236.

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

[4] Akl, S.G., Conventional or unconventional: Is any computer universal?, in: *From Utopian to Genuine Unconventional Computers*, Adamatzky, A. and Teuscher, C., Eds., Luniver Press, Frome, United Kingdom, 2006, pp. 101–136.

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

[6] 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.

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

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

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

[10] 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.

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

[12] Deutsch, D., *The Fabric of Reality*, Penguin Books, 1997.

[13] 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.

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

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

[16] Lloyd, S., *Programming the Universe*, Knopf, 2006.

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

[18] 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, pp. 176–190.

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

[20] Nagy, M. and Akl, S.G., Parallelism in quantum information processing defeats the Universal Computer, *Parallel Processing Letters*, Special Issue on Unconventional Computational Problems, Vol. 17, No. 3, September 2007, pp. 233–262.

[21] Nagy, N. and Akl, S.G., Computing with uncertainty and its implications to universality, to appear in the *International Journal of Parallel, Emergent and Distributed Systems.*

[22] Prusinkiewicz, P. and Lindenmayer, A. *The Algorithmic Beauty of Plants*, Springer Verlag, 1990.

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