1 Evolving Computational Systems

Selim G. Akl
Queen's University

1.1 Introduction

The universe in which we live is in a constant state of evolution. People age, trees grow, the weather varies. From one moment to the next, our world undergoes a myriad of transformations. Many of these changes are obvious to the naked eye, others more subtle. Deceptively, some appear to occur independently of any direct external influences. Others are immediately perceived as the result of actions by other entities.

In the realm of computing, it is generally assumed that the world is static. The vast majority of computations take place in applications where change is thought of, rightly or wrongly, as inexistent or irrelevant. Input data are read, algorithms are applied to them, and results are produced. The possibility that the data, the algorithms, or even the results sought may vary *during* the process of computation is rarely, if ever, contemplated.

In this chapter we explore the concept of *evolving computational systems*. These are systems in which everything in the computational process is subject to change. This includes inputs, algorithms, outputs, and even the computing agents themselves. A simple example of a computational paradigm that meets this definition of an evolving system to a limited extent is that of a computer interacting in real time with a user while processing information. Our focus here is primarily on certain changes that may affect the data required to solve a problem. We also examine changes that affect the complexity of the algorithm used in the solution. Finally, we look at one example of a computer capable of evolving with the computation.

A number of evolving computational paradigms are described for which a parallel computing approach is most appropriate. In Sections 1.3, 1.4, and 1.5, time plays an important role either directly or indirectly in the evolution of the computation. Thus, it is the passage of time that may cause the change in the data. In another context, it may be the order in which a stage of an algorithm is performed that determines the number of operations required by that stage. The effect of time on evolving computations is also used to contrast the capabilities of a parallel computer with those of an unconventional model of computation known as the *accelerating machine*. In Sections 1.6 and 1.7, it is not time but rather external agents acting on the data that are responsible for a variable computation. Thus, the data may be affected by a measurement that perturbs an existing equilibrium, or by a modification in a mathematical structure that violates a required condition. Finally, in Section 1.8 evolving computations allow us to demonstrate the impossibility of achieving universality in computing. Our conclusions are offered in Section 1.9.

1.2 Computational Models

It is appropriate at the outset that we define our models of computation. Two such models are introduced in this section, one sequential and one parallel. (A third model, the accelerating machine, is defined in Section 1.4.3.) We begin by stating clearly our understanding regarding the meaning of time, and our assumptions in connection with the speed of processors.

1.2.1 Time and Speed

In the classical study of algorithms, whether sequential or parallel, the notion of a *time unit* is fundamental to the analysis of an algorithm's running time. A time unit is the smallest discrete measure of time. In other words, time is divided into consecutive time units that are indivisible. All events occur at the beginning of a time unit. Such events include, for example, a variable changing its value, a processor undertaking the execution of a step in its computation, and so on.

It is worth emphasizing that the length of a time unit is not an absolute quantity. Instead, the duration of a time unit is specified in terms of a number of factors. These include the parameters of the computation at hand, such as the rate at which the data are received, or the rate at which the results are to be returned. Alternatively, a time unit may be defined in terms of the speed of the processors available (namely, the single processor on a sequential computer and each processor on a parallel computer). In the latter case, a faster processor implies a smaller time unit.

In what follows, the standard definition of time unit is adopted, namely: A time unit is the length of time required by a processor to perform a *step* of its computation. Specifically, during a time unit, a processor executes a step consisting of

1. A *read* operation in which it receives a constant number of fixed-size data as input
2. 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)
3. A *write* operation in which it returns a constant number of fixed-size data as output

All other occurrences external to the processor (such as the data arrival rate, for example) will be set and measured in these terms. Henceforth, the term *elementary operation* is used to refer to a read, a calculate, or a write operation.

1.2.2 What Does it Mean to Compute?

An important characteristic of the treatment in this chapter is the broad perspective taken to define what it means *to compute*. Specifically, *computation* is a process whereby information is manipulated by, for example, acquiring it (input), transforming it (calculation), and transferring it (output). Any form of information processing (whether occurring spontaneously in nature, or performed on a computer built by humans) is a computation. Instances of computational processes include

1. Measuring a physical quantity
2. Performing an arithmetic or logical operation on a pair of numbers
3. Setting the value of a physical quantity

to cite but a few. These computational processes themselves may be carried out by a variety of means, including, of course, conventional (electronic) computers, and also through physical phenomena [35], chemical reactions [1], and transformations in living biological tissue [42]. By extension, *parallel computation* is defined as the execution of several such processes of the same type simultaneously.

1.2.3 Sequential Model

This is the conventional model of computation used in the design and analysis of sequential (also known as *serial*) algorithms. It consists of a single processor made up of circuitry for executing arithmetic and logical operations and a number of registers that serve as internal memory for storing programs and data. For our purposes, the processor is also equipped with an input unit and an output unit that allow it to receive data from, and send data to, the outside world, respectively.

During each time unit of a computation the processor can perform

1. A read operation, that is, receive a constant number of fixed-size data as input
2. A calculate operation, that is, execute a fixed number of constant-time calculations on its input
3. A write operation, that is, return a constant number of fixed-size results as output

It is important to note here that the read and write operations can be, respectively, from and to the model's internal memory. In addition, both the reading and writing may be, on occasion, from and to an external medium in the environment in which the computation takes place. Several incarnations of this model exist, in theory and in practice [40]. In what follows, we refer to this model of computation as a *sequential computer*.

1.2.4 Parallel Model

Our chosen model of parallel computation consists of n processors, numbered 1 to n, where $n \geq 2$. Each processor is of the type described in Section 1.2.3. The processors are connected in some fashion and are able to communicate with one another in order to exchange data and results. These exchanges may take place through an *interconnection network*, in which case selected pairs of processors are directly connected by two-way communication links. Pairs of processors not directly connected communicate indirectly by creating a route for their messages that goes through other processors. Alternatively, all exchanges may take place via a global *shared memory* that is used as a bulletin board. A processor wishing to send a datum to another processor does so by writing it to the shared memory, from where it is read by the other processor. Several varieties of interconnection networks and modes of shared memory access exist, and any of them would be adequate as far as we are concerned here. The exact nature of the communication medium among the processors is of no consequence to the results described in this chapter. A study of various ways of connecting processors is provided in [2].

During each time unit of a computation a processor can perform

1. A read operation, that is, receive as input a constant number of fixed-size data
2. A calculate operation, that is, execute a fixed number of constant-time calculations on its input
3. A write operation, that is, return as output a constant number of fixed-size results

As with the sequential processor, the input can be received from, and the output returned to, either the internal memory of the processor or the outside world. In addition, a processor in a parallel computer may receive its input from and return its output to the shared memory (if one is available), or another processor (through a direct link or via a shared memory, whichever is available). Henceforth, this model of computation will be referred to as a *parallel computer* [3].

1.2.5 A Fundamental Assumption

The analyses in this chapter assume that all models of computation use the fastest processors possible (within the bounds established by theoretical physics). Specifically, no sequential computer exists that is faster than the one of Section 1.2.3, and similarly no parallel computer exists whose processors are faster than those of Section 1.2.4. Furthermore, no processor on the parallel computer of Section 1.2.4 is faster than the processor of the sequential computer of Section 1.2.3. This is the *fundamental assumption in parallel computation.* It is also customary to suppose that the sequential and parallel computers use identical processors. We adopt this convention throughout this chapter, with a single exception: In Section 1.4.3, we assume that the processor of the sequential computer is in fact capable of increasing its speed at every step (at a pre-established rate, so that the number of operations executable at every consecutive step is known a priori and fixed once and for all).

1.3 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 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 (e.g., x_i may be a true random variable).

All the physical variables exist in their natural environment within which the computation is to take place. They are all available to be operated on at the beginning of the computation. Thus, for each variable $x_i(t)$, it is possible to compute $F_i(x_i(t))$, provided that a computer is available to perform the calculation (and subsequently return the result).

Recall that time is divided into intervals, each of duration one time unit. 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 $t = t_0$. In other words, once all the variables have assumed their respective values at time $t = t_0$, the functions F_i are to be evaluated for all values of i. Specifically,

$$F_0(x_0(t_0)), F_1(x_1(t_0)), \ldots, F_{n-1}(x_{n-1}(t_0))$$

are to be computed. The fact that $x_i(t)$ changes with the passage of time should be emphasized here. Thus, if $x_i(t)$ is not operated on at time $t = t_0$, then after one time unit $x_i(t_0)$ becomes $x_i(t_0 + 1)$, and after two time units it is $x_i(t_0 + 2)$, and so on. Indeed, time exists as a fundamental fact of life. It is real, relentless, and unforgiving. Time cannot be stopped, much less reversed. (For good discussions of these issues, see [28, 45].) Furthermore, 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. We illustrate this behavior through an example from physics.

1.3.1 Quantum Decoherence

A binary variable is a mathematical quantity that takes exactly one of a total of two possible values at any given time. In the base 2 number system, these values are 0 and 1, and are known as *binary digits* or *bits*. Today's conventional computers use electronic devices for storing and manipulating bits. These devices are in either one or the other of two physical states at any given time (e.g., two voltage levels), one representing 0, the other 1. We refer to such a device, as well as the digit it stores, as a *classical bit*.

In *quantum computing*, a bit (aptly called a quantum bit, or *qubit*) is both 0 and 1 at the same time. The qubit is said to be in a *superposition* of the two values. One way to implement a qubit is by encoding the 0 and 1 values using the spin of an electron (e.g., clockwise, or "up" for 1, and counterclockwise, or "down" for 0). Formally, a qubit is a unit vector in a two-dimensional state space, for which a particular orthonormal basis, denoted by $\{|0\rangle, |1\rangle\}$ has been fixed. The two basis vectors $|0\rangle$ and $|1\rangle$ correspond to the possible values a classical bit can take. However, unlike classical bits, a qubit can also take many other values. In general, an arbitrary qubit can be written as a linear combination of the computational basis states, namely, $\alpha|0\rangle + \beta|1\rangle$, where α and β are complex numbers such that $|\alpha|^2 + |\beta|^2 = 1$.

Measuring the value of the qubit (i.e., reading it) returns a 0 with probability $|\alpha|^2$ and a 1 with a probability $|\beta|^2$. Furthermore, the measurement causes the qubit to undergo *decoherence* (literally, to lose its coherence). When decoherence occurs, the superposition is said to collapse: any subsequent measurement returns the same value as the one obtained by the first measurement. The information previously held in the superposition is lost forever. Henceforth, the qubit no longer possesses its quantum properties and behaves as a classical bit [33].

There is a second way, beside measurement, for decoherence to take place. A qubit loses its coherence simply through prolonged exposure to its natural environment. The interaction between the qubit and its physical surroundings may be thought of as an external action by the latter causing the former to behave as a classical bit, that is, to lose all information it previously stored in a superposition. (One can also view decoherence as the act of the qubit making a mark on its environment by adopting a classical value.) Depending on the particular implementation of the qubit, the time needed for this form of decoherence to take place varies. At the time of this writing, it is well below 1 s (more precisely, in the vicinity of a nanosecond). The information lost through decoherence cannot be retrieved. For the purposes of this example, the time required for decoherence to occur is taken as one time unit.

Now suppose that a quantum system consists of n independent qubits, each in a state of superposition. Their respective values at some time t_0, namely, $x_0(t_0)$, $x_1(t_0)$, $\ldots$, $x_{n-1}(t_0)$, are to be used as inputs to the n functions F_0, F_1, $\ldots$, F_{n-1}, in order to perform the computation described at the beginning of Section 1.3, that is, to evaluate $F_i(x_i(t_0))$, for $0 \leq i \leq n-1$.

1.3.2 Sequential Solution

A sequential computer fails to compute all the F_i as desired. Indeed, suppose that $x_0(t_0)$ is initially operated upon. It follows that $F_0(x_0(t_0))$ can be computed correctly. However, when the next variable, x_1, for example, is to be used (as input to F_1), the time variable would have changed from $t = t_0$ to $t = t_0 + 1$, and we obtain $x_1(t_0 + 1)$, instead of the $x_1(t_0)$ that we need. Continuing in this fashion, $x_2(t_0 + 2)$, $x_3(t_0 + 3), \ldots, x_{n-1}(t_0 + n - 1)$, represent the sequence of inputs. In the example of Section 1.3.1, by the time $F_0(x_0(t_0))$ is computed, one time unit would have passed. At this point, the $n - 1$ remaining qubits would have undergone decoherence. 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.

Since the function according to which each x_i changes with time is not known, it is impossible to recover $x_i(t_0)$ from $x_i(t_0 + i)$, for $i = 1, 2, \ldots, n - 1$. Consequently, this approach cannot produce $F_1(x_1(t_0))$, $F_2(x_2(t_0)), \ldots, F_{n-1}(x_{n-1}(t_0))$, as required.

1.3.3 Parallel Solution

For a given n, *any* computer capable of performing n calculate operations per step can easily evaluate the $F_i(x_i(t_0))$, all simultaneously, leading to a successful computation.

Thus, 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 $F_i(x_i(t_0))$. In the example of Section 1.3.1, the n functions are computed in parallel at time $t = t_0$, before decoherence occurs.

1.4 Time-Varying Computational Complexity

In traditional computational complexity theory, the *size* of a problem $\mathcal{P}$ plays an important role. If $\mathcal{P}$ has size n, for example, then the number of operations required in the worst case to solve $\mathcal{P}$ (by any algorithm) is expressed as a function of n. Similarly, the number of operations executed (in the best, average, and worst cases) by a specific algorithm that solves $\mathcal{P}$ is also expressed as a function of n. Thus, for example, the problem of sorting a sequence of n numbers requires $\Omega(n \log n)$ comparisons, and the sorting algorithm Quicksort performs $O(n^2)$ comparisons in the worst case.

In this section, we depart from this model. Here, the size of the problem plays a secondary role. In fact, in most (though not necessarily all) cases, the problem size may be taken as constant. The computational complexity now depends on *time*. Not only science and technology but also everyday life provide many instances demonstrating time-varying complexity. Thus, for example,

1. An illness may get better or worse with time, making it more or less amenable to treatment.
2. Biological and software viruses spread with time making them more difficult to cope with.
3. Spam accumulates with time making it more challenging to identify the legitimate email "needles" in the "haystack" of junk messages.
4. Tracking moving objects becomes harder as they travel away from the observer (e.g., a spaceship racing toward Mars).
5. Security measures grow with time in order to combat crime (e.g., when protecting the privacy, integrity, and authenticity of data, ever stronger cryptographic algorithms are used, i.e., ones that are more computationally demanding to break, thanks to their longer encryption and decryption keys).
6. Algorithms in many applications have complexities that vary with time from one time unit during the computation to the next. Of particular importance here are
   a. *Molecular dynamics* (the study of the dynamic interactions among the atoms of a system, including the calculation of parameters such as forces, energies, and movements) [18, 39].
   b. *Computational fluid dynamics* (the study of the structural and dynamic properties of moving objects, including the calculation of the velocity and pressure at various points) [11].

Suppose that we are given an algorithm for solving a certain computational problem. The algorithm consists of a number of stages, where each stage may represent, for example, the evaluation of a particular arithmetic expression such as

$$c \leftarrow a + b.$$

Further, let us assume that a computational stage executed at time t requires a number $C(t)$ of constant-time operations. As the aforementioned situations show, the behavior of C varies from case to case.

Typically, C may be an increasing, decreasing, unimodal, periodic, random, or chaotic function of t. In what follows, we study the effect on computational complexity of a number of functions $C(t)$ that grow with time.

It is worth noting that we use the term *stage* to refer to a component of an algorithm, hence a variable entity, in order to avoid confusion with a *step*, an intrinsic property of the computer, as defined in Sections 1.2.1 and 1.4.3. In conventional computing, where computational complexity is invariant (i.e., oblivious to external circumstances), a *stage* (as required by an algorithm) is exactly the same thing as a *step* (as executed by a computer). In *unconventional computing* (the subject of this chapter), computational complexity is affected by its environment and is therefore variable. Under such conditions, one or more steps may be needed in order to execute a stage.

1.4.1 Examples of Increasing Functions $C(t)$

Consider the following three cases in which the number of operations required to execute a computational stage increases with time. For notational convenience, we use $S(i)$ to express the number of operations performed in executing stage i, at the time when that stage is in fact executed. Denoting the latter by t_i, it is clear that $S(i) = C(t_i)$.

1. For $t \geq 0$, $C(t) = t + 1$. Table 1.1 illustrates t_i, $C(t_i)$, and $S(i)$, for $1 \leq i \leq 6$.

   It is clear in this case that $S(i) = 2^{i-1}$, for $i \geq 1$. It follows that the total number of operations performed when executing all stages, from stage 1 up to and including stage i, is

   $$\sum_{j=1}^{i} 2^{j-1} = 2^i - 1.$$

   It is interesting to note that while $C(t)$ is a linear function of the time variable t, the quantity $S(i)$ grows exponentially with $i - 1$, where i is the number of stages executed so far. The effect of this behavior on the total number of operations performed is appreciated by considering the following example. When executing a computation requiring $\log n$ stages for a problem of size n,

   $$2^{\log n} - 1 = n - 1$$

   operations are performed.
2. For $t \geq 0$, $C(t) = 2^t$. Table 1.2 illustrates t_i, $C(t_i)$, and $S(i)$, for $1 \leq i \leq 5$.

   In this case, $S(1) = 1$, and for $i > 1$, we have

   $$S(i) = 2^{\sum_{j=1}^{i-1} S(j)}.$$

TABLE 1.1 Number of Operations Required to Complete Stage i When $C(t) = t + 1$

Stage i	t_i	$C(t_i)$	$S(i)$
1	0	$C(0)$	1
2	$0 + 1$	$C(1)$	2
3	$1 + 2$	$C(3)$	4
4	$3 + 4$	$C(7)$	8
5	$7 + 8$	$C(15)$	16
6	$15 + 16$	$C(31)$	32
7	$31 + 32$	$C(63)$	64

TABLE 1.2 Number of Operations Required to Complete Stage i When $C(t) = 2^t$

Stage i	t_i	$C(t_i)$	$S(i)$
1	0	$C(0)$	2^0
2	$0 + 1$	$C(1)$	2^1
3	$1 + 2$	$C(3)$	2^3
4	$3 + 8$	$C(11)$	2^{11}
5	$11 + 2048$	$C(2059)$	2^{2059}

TABLE 1.3 Number of Operations Required to Complete Stage i When $C(t) = 2^{2^t}$

Stage i	t_i	$C(t_i)$	$S(i)$
1	0	$C(0)$	2^{2^0}
2	$0 + 2$	$C(2)$	2^{2^2}
3	$2 + 16$	$C(18)$	$2^{2^{18}}$

Since $S(i) > \sum_{j=1}^{i-1} S(j)$, the total number of operations required by i stages is less than $2S(i)$, that is, $O(S(i))$.
Here we observe again that while $C(t) = 2C(t-1)$, the number of operations required by $S(i)$, for $i > 2$, increases significantly faster than double those required by all previous stages combined.

3. For $t \geq 0$, $C(t) = 2^{2^t}$. Table 1.3 illustrates t_i, $C(t_i)$, and $S(i)$, for $1 \leq i \leq 3$.
   Here, $S(1) = 2$, and for $i > 1$, we have

$$S(i) = 2^{2^{\sum_{j=1}^{i-1} S(j)}}.$$

Again, since $S(i) > \sum_{j=1}^{i-1} S(j)$, the total number of operations required by i stages is less than $2S(i)$, that is, $O(S(i))$.
In this example, the difference between the behavior of $C(t)$ and that of $S(i)$ is even more dramatic. Obviously, $C(t) = C(t-1)^2$, where $t \geq 1$ and $C(0) = 2$, and as such $C(t)$ is a fast growing function ($C(4) = 65,536$, while $C(7)$ is represented with 39 decimal digits). Yet, $S(i)$ grows at a far more dizzying pace: Already $S(3)$ is equal to 2 raised to the power $4 \times 65,536$.

The significance of these examples and their particular relevance in parallel computation are illustrated by the paradigm in the following section.

1.4.2 Computing with Deadlines

Suppose that a certain computation requires that n functions, each of one variable, be computed. Specifically, let $f_0(x_0), f_1(x_1), \ldots, f_{n-1}(x_{n-1})$ be the functions to be computed. This computation has the following characteristics:

1. The n functions are entirely independent. There is no precedence whatsoever among them; they can be computed in any order.
2. Computing $f_i(x_i)$ at time t requires $C(t) = 2^t$ operations, for $0 \leq i \leq n-1$ and $t \geq 0$.
3. There is a deadline for reporting the results of the computations: All n values $f_0(x_0), f_1(x_1), \ldots, f_{n-1}(x_{n-1})$ 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 (i.e., 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 $f_2(x_2), f_3(x_3), \ldots, f_{n-1}(x_{n-1})$ 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.

The example in this section is based on one of the three functions for $C(t)$ presented in Section 1.4.1. Similar analyses can be performed in the same manner for $C(t) = t + 1$ and $C(t) = 2^{2^t}$, as well as other functions describing time-varying computational complexity.

1.4.3 Accelerating Machines

In order to put the result in Section 1.4.2 in perspective, we consider a variant on the sequential model of computation described in Section 1.2.3. An *accelerating machine* is a sequential computer capable of increasing the number of operations it can do at each successive step of a computation. This is primarily a theoretical model with no existing implementation (to date!). It is widely studied in the literature on unconventional computing [10,12,14,43,44,46]. The importance of the accelerating machine lies primarily in its role in questioning some long-held beliefs regarding uncomputability [13] and universality [7].

It is important to note that the rate of acceleration is specified at the time the machine is put in service and remains the same for the lifetime of the machine. Thus, the number of operations that the machine can execute during the ith step is known in advance and fixed permanently, for $i = 1, 2, \ldots$.

Suppose that an accelerating machine that can *double* the number of operations that it can perform at each step is available. Such a machine would be able to perform one operation in the first step, two operations in the second, four operations in the third, and so on. How would such an extraordinary machine fare with the computational problem of Section 1.4.2?

As it turns out, an accelerating machine capable of doubling its speed at each step, is unable to solve the problem for $n \geq 4$. It would compute $f_0(x_0)$, at time $t = 0$ in one time unit. Then it would compute $f_1(x_1)$, which now requires two operations at $t = 1$, also in one time unit. Finally, $f_2(x_2)$, requiring four operations at $t = 2$, is computed in one time unit, by which time $t = 3$. The deadline has been reached and none of $f_3(x_3), f_4(x_4), \ldots, f_{n-1}(x_{n-1})$ has been computed.

In closing this discussion of accelerating machines we note that once an accelerating machine has been defined, a problem can always be devised to expose its limitations. Thus, let the acceleration function be $\Phi(t)$. In other words, $\Phi(t)$ describes the number of operations that the accelerating machine can perform at time t. For example, $\Phi(t) = 2\Phi(t-1)$, with $t \geq 1$ and $\Phi(0) = 1$, as in the case of the accelerating machine in this section. By simply taking $C(t) > \Phi(t)$, the accelerating machine is rendered powerless, *even in the absence of deadlines.*

1.5 Rank-Varying Computational Complexity

Unlike the computations in Section 1.4, the computations with which we are concerned here have a complexity that does not vary with time. Instead, suppose that a computation consists of n stages. There may be a certain precedence among these stages, that is, the order in which the stages are performed matters since some stages may depend on the results produced by other stages. Alternatively, 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. In this section, we focus on computations with the property that the number of operations required to execute a stage whose rank is i is a function of i only. For example, as in Section 1.4, this function may be increasing, decreasing, unimodal, random, or chaotic. Instances of algorithms whose computational complexity varies from one stage to another are described in Reference 15. As we did before, we concentrate here on the case where the computational complexity C is an increasing function of i.

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

1. A penalty may be charged for missing a deadline, such as when a stage s must be completed by a certain time d_s.
2. 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.
3. Biological tissues may have been altered (by previous stages) when stage s is reached.
4. The effect of $s - 1$ quantum operations may have to be reversed to perform stage s.

1.5.1 An Algorithmic Example: Binary Search

Binary search is a well-known (sequential) algorithm in computer science. It searches for an element x in a sorted list L of n elements. In the worst case, binary search executes $O(\log n)$ stages. In what follows, we denote by $B(n)$ the total number of elementary operations performed by binary search (on a sequential computer), and hence its running time, in the worst case.

Conventionally, it is assumed that $C(i) = O(1)$, that is, each stage i requires the same constant number of operations when executed. Thus, $B(n) = O(\log n)$. Let us now consider what happens to the computational complexity of binary search when we assume, unconventionally, that the computational complexity of every stage i increases with i. Table 1.4 shows how $B(n)$ grows for three different values of $C(i)$.

In a parallel environment, where n processors are available, the fact that the sequence L is sorted is of no consequence to the search problem. Here, each processor reads x, compares one of the elements of L to x, and returns the result of the comparison. This requires one time unit. Thus, regardless of $C(i)$, the running time of the parallel approach is always the same.

1.5.2 The Inverse Quantum Fourier Transform

Consider a quantum register consisting of n qubits. There are 2^n computational basis vectors associated with such a register, namely,

$$\begin{aligned} |0\rangle &= |000\cdots00\rangle, \\ |1\rangle &= |000\cdots01\rangle, \\ &\vdots \\ |2^n - 1\rangle &= |111\cdots11\rangle. \end{aligned}$$

TABLE 1.4 Number of Operations Required by Binary Search for Different Functions $C(i)$

$C(i)$	$B(n)$
i	$O(\log^2 n)$
2^i	$O(n)$
2^{2^i}	$O(2^n)$

Let $|j\rangle = |j_1 j_2 j_3 \cdots j_{n-1} j_n\rangle$ be one of these vectors. For $j = 0, 1, \ldots, 2^n - 1$, the quantum Fourier transform of $|j\rangle$ is given by

$$\frac{\left(|0\rangle + e^{2\pi i 0.j_n}|1\rangle\right) \otimes \left(|0\rangle + e^{2\pi i 0.j_{n-1}j_n}|1\rangle\right) \otimes \cdots \otimes \left(|0\rangle + e^{2\pi i 0.j_1 j_2 \cdots j_n}|1\rangle\right)}{2^{n/2}},$$

where

1. Each transformed qubit is a balanced superposition of $|0\rangle$ and $|1\rangle$.
2. For the remainder of this section $i = \sqrt{-1}$.
3. The quantities $0.j_n$, $0.j_{n-1}j_n$, $\ldots$, $0.j_1 j_2 \cdots j_n$, are binary fractions, whose effect on the $|1\rangle$ component is called a *rotation*.
4. The operator $\otimes$ represents a tensor product; for example.

$$(a_1|0\rangle + b_1|1\rangle) \otimes (a_2|0\rangle + b_2|1\rangle) = a_1 a_2|00\rangle + a_1 b_2|01\rangle + b_1 a_2|10\rangle + b_1 b_2|11\rangle.$$

We now examine the inverse operation, namely, obtaining the original vector $|j\rangle$ from its given quantum Fourier transform.

1.5.2.1 Sequential Solution

Since the computation of each of $j_1, j_2, \ldots j_{n-1}$ depends on j_n, we must begin by computing the latter from $|0\rangle + e^{2\pi i 0.j_n}|1\rangle$. This takes one operation. Now j_n is used to compute j_{n-1} from $|0\rangle + e^{2\pi i 0.j_{n-1}j_n}|1\rangle$ in two operations. In general, once j_n is available, j_k requires knowledge of j_{k+1}, $j_{k+2}, \ldots, j_n$, must be computed in $(n-k+1)$st place, and costs $n-k+1$ operations to retrieve from $|0\rangle + e^{2\pi i 0.j_k j_{k+1} \cdots j_n}|1\rangle$, for $k = n-1, n-2, \ldots, 1$. Formally, the sequential algorithm is as follows:

```
for $k = n$ downto 1 do
  $|j_k\rangle \leftarrow \frac{1}{\sqrt{2}} \begin{pmatrix} |0\rangle \\ e^{2\pi i 0.j_k j_{k+1} \cdots j_n}|1\rangle \end{pmatrix}$
  for $m = k+1$ to $n$ do
    if $j_{n+k+1-m} = 1$ then
      $|j_k\rangle \leftarrow |j_k\rangle \begin{pmatrix} 1 & 0 \\ 0 & e^{-2\pi i/2^{n-m+2}} \end{pmatrix}$
    end if
  end for
  $|j_k\rangle \leftarrow |j_k\rangle \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$
end for. ■
```

Note that the inner **for** loop is not executed when $m > n$. It is clear from the above analysis that a sequential computer obtains $j_1, j_2, \ldots, j_n$ in $n(n+1)/2$ time units.

1.5.2.2 Parallel Solution

By contrast, a parallel computer can do much better in two respects. First, for $k = n, n-1, \ldots, 2$, once j_k is known, all operations involving j_k in the computation of $j_1, j_2, \ldots, j_{k-1}$ can be performed simultaneously, each being a rotation. The parallel algorithm is given below:

```
for $k = 1$ to $n$ do in parallel
  $|j_k\rangle \leftarrow \frac{1}{\sqrt{2}} \begin{pmatrix} |0\rangle \\ e^{2\pi i 0.j_k j_{k+1} \cdots j_n}|1\rangle \end{pmatrix}$
end for
```

```
  $|j_n\rangle \leftarrow |j_n\rangle \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$
  for $k = n-1$ downto $1$ do
    if $j_{k+1} = 1$ then
      for $m = 1$ to $k$ do in parallel
        $|j_m\rangle \leftarrow |j_m\rangle \begin{pmatrix} 1 & 0 \\ 0 & e^{-2\pi i/2^{n-m+1}} \end{pmatrix}$
      end for
    end if
    $|j_k\rangle \leftarrow |j_k\rangle \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$
  end for. ■
```

The total number of time units required to obtain $j_1, j_2, \ldots, j_n$ is now $2n-1$.

Second, and more important, if decoherence takes place within δ time units, where $2n-1 < \delta < n(n+1)/2$, the parallel computer succeeds in performing the computation, while the sequential computer fails [34].

1.6 Interacting Variables

So far, in every one of the paradigms that we have examined, the unconventional nature of the computation was due either to the passage of time or to the order in which an algorithmic stage is performed. In this and the next section, we consider evolving computations that occur in computational environments where time and rank play no role whatsoever either in the outcome or the complexity of the computation. Rather, it is the interactions among mutually dependent variables, caused by an interfering agent (performing the computation) that is the origin of the evolution of the system under consideration.

The computational paradigm to be presented in this section does have one feature in common with those discussed in the previous sections, namely, the central place occupied by the physical environment in which the computation is carried out. Thus, in Section 1.3, for example, the passage of time (a physical phenomenon, to the best of our knowledge) was the reason for the variables acquiring new values at each successive time unit. However, the attitude of the physical environment in the present paradigm is a passive one: Nature will not interfere with the computation until it is disturbed.

Let $\mathcal{S}$ be a physical system, such as one studied by biologists (e.g., a living organism), or one maintained by engineers (e.g., a power generator). The system has n variables each of which is to be measured or set to a given value at regular intervals. One property of $\mathcal{S}$ is that measuring or setting one of its variables modifies the values of any number of the system variables unpredictably. We show in this section how, under these conditions, a parallel solution method succeeds in carrying out the required operations on the variables of $\mathcal{S}$, while a sequential method fails. Furthermore, it is principles governing such fields as physics, chemistry, and biology that are responsible for causing the inevitable failure of any sequential method of solving the problem at hand, while at the same time allowing a parallel solution to succeed. A typical example of such principles is the uncertainty involved in measuring several related variables of a physical system. Another principle expresses the way in which the components of a system in equilibrium react when subjected to outside stress.

1.6.1 Disturbing the Equilibrium

A physical system $\mathcal{S}$ possesses the following characteristics:

1. For $n > 1$, the system possesses a set of n variables (or properties), namely, $x_0, x_1, \ldots, x_{n-1}$. Each of these variables is a physical quantity (such as, e.g., temperature, volume, pressure, humidity, density, electric charge, etc.). These quantities can be measured or set independently, each at a

given discrete location (or point) within $\mathcal{S}$. Henceforth, x_i, $0 \leq i \leq n-1$ is used to denote a variable as well as the discrete location at which this variable is measured or set.

2. The system is in a state of *equilibrium*, meaning that the values $x_0, x_1, \ldots, x_{n-1}$ satisfy a certain global condition $\mathcal{G}(x_0, x_1, \ldots, x_{n-1})$.
3. At regular intervals, the state of the physical system is to be recorded and possibly modified. In other words, the values $x_0, x_1, \ldots, x_{n-1}$ are to be measured at a given moment in time where $\mathcal{G}(x_0, x_1, \ldots, x_{n-1})$ is satisfied. New values are then computed for $x_0, x_1, \ldots, x_{n-1}$, and the variables are set to these values. Each interval has a duration of $\mathcal{T}$ time units; that is, the state of the system is measured and possibly updated every $\mathcal{T}$ time units, where $\mathcal{T} > 1$.
4. If the values $x_0, x_1, \ldots, x_{n-1}$ are measured or set *one by one*, each separately and independently of the others, this disturbs the equilibrium of the system. Specifically, suppose, without loss of generality, that all the values are first measured, and later all are set, in the order of their indices, such that x_0 is first and x_{n-1} last in each of the two passes. Thus
   a. When x_i is measured, an arbitrary number of values x_j, $0 \leq j \leq n-1$ will change unpredictably shortly thereafter (within one time unit), such that $\mathcal{G}(x_0, x_1, \ldots, x_{n-1})$ is no longer satisfied. Most importantly, when $i < n-1$, the values of $x_{i+1}, x_{i+2}, \ldots, x_{n-1}$, none of which has yet been registered, may be altered irreparably.
   b. Similarly, when x_i is set to a new value, an arbitrary number of values x_j, $0 \leq j \leq n-1$ will change unpredictably shortly thereafter (within one time unit), such that $\mathcal{G}(x_0, x_1, \ldots, x_{n-1})$ is no longer satisfied. Most importantly, when $i > 0$, the values of $x_0, x_1, \ldots, x_{i-1}$, all of which have already been set, may be altered irreparably.

This last property of $\mathcal{S}$, namely, the way in which the system reacts to a sequential measurement or setting of its variables, is reminiscent of a number of well-known phenomena that manifest themselves in many subfields of the physical and natural sciences and engineering [8]. Examples of these phenomena are grouped into two classes and presented in what follows.

1.6.1.1 Uncertainty in Measurement

The phenomena of interest here occur in systems where measuring one variable of a given system affects, interferes with, or even precludes the subsequent measurement of another variable of the system. It is important to emphasize that the kind of uncertainty of concern in this context is in no way due to any errors that may be introduced by an imprecise or not sufficiently accurate measuring apparatus.

1. In quantum mechanics, *Heisenberg's uncertainty principle* puts a limit on our ability to measure pairs of "complementary" variables. Thus, the *position* and *momentum* of a subatomic particle, or the *energy* of a particle in a certain state and the *time* during which that state existed, cannot be defined at the same time to arbitrary accuracy [9]. In fact, one may interpret this principle as saying that once *one* of the two variables is measured (however accurately, but independently of the other), the act of measuring itself introduces a disturbance that affects the value of the *other* variable. For example, suppose that at a given moment in time t_0 the position p_0 of an electron is measured. Assume further that it is also desired to determine the electron's momentum m_0 at time t_0. When the momentum is measured, however, the value obtained is not m_0, as it would have been changed by the previous act of measuring p_0.
2. In digital signal processing, the *uncertainty principle* is exhibited when conducting a Fourier analysis. Complete resolution of a signal is possible either in the time domain t or the frequency domain w, but not both simultaneously. This is due to the fact that the Fourier transform is computed using e^{iwt}. Since the product wt must remain constant, narrowing a function in one domain causes it to be wider in the other [19, 41]. For example, a pure sinusoidal wave has no time resolution, as it possesses nonzero components over the infinitely long time axis. Its Fourier transform, on the other hand, has excellent frequency resolution: it is an impulse function with a single positive frequency component. By contrast, an impulse (or *delta*) function has only one

value in the time domain, and hence excellent resolution. Its Fourier transform is the constant function with nonzero values for all frequencies and hence no resolution.

Other examples in this class include image processing, sampling theory, spectrum estimation, image coding, and filter design [49]. Each of the phenomena discussed typically involves *two* variables in equilibrium. Measuring one of the variables has an impact on the value of the other variable. The system $\mathcal{S}$, however, involves *several* variables (two or more). In that sense, its properties, as listed at the beginning of this section, are extensions of these phenomena.

1.6.1.2 Reaction to Stress

Phenomena in this class arise in systems where modifying the value of a parameter causes a change in the value of another parameter. In response to stress from the outside, the system automatically reacts so as to relieve the stress. Newton's third law of motion ("For every action there is an equal and opposite reaction") is a good way to characterize these phenomena.

1. In chemistry, *Le Châtelier's principle* states that if a system at equilibrium is subjected to a stress, the system will shift to a new equilibrium in an attempt to reduce the stress. The term *stress* depends on the system under consideration. Typically, stress means a change in pressure, temperature, or concentration [36]. For example, consider a container holding gases in equilibrium. Decreasing (increasing) the volume of the container leads to the pressure inside the container increasing (decreasing); in response to this external stress the system favors the process that produces the sleast (most) molecules of gas. Similarly, when the temperature is increased (decreased), the system responds by favoring the process that uses up (produces) heat energy. Finally, if the concentration of a component on the left (right) side of the equilibrium is decreased (increased), the system's automatic response is to favor the reaction that increases (decreases) the concentration of components on the left (right) side.
2. In biology, the *homeostatic principle* is concerned with the behavior displayed by an organism to which stress has been applied [37,48]. An automatic mechanism known as *homeostasis* counteracts external influences in order to maintain the equilibrium necessary for survival, at all levels of organization in living systems. Thus, at the molecular level, homeostasis regulates the amount of enzymes required in metabolism. At the cellular level, it controls the rate of division in cell populations. Finally, at the organismic level, it helps maintain steady levels of temperature, water, nutrients, energy, and oxygen. Examples of homeostatic mechanisms are the sensations of hunger and thirst. In humans, sweating and flushing are automatic responses to heating, while shivering and reducing blood circulation to the skin are automatic responses to chilling. Homeostasis is also seen as playing a role in maintaining population levels (animals and their prey), as well as steady state conditions in the Earth's environment.

Systems with similar behavior are also found in cybernetics, economics, and the social sciences [25]. Once again, each of the phenomena discussed typically involves *two* variables in equilibrium. Setting one of the variables has an impact on the value of the other variable. The system $\mathcal{S}$, however, involves *several* variables (two or more). In that sense, its properties, as listed at the beginning of this section, are extensions of these phenomena.

1.6.2 Solutions

Two approaches are now described for addressing the problem defined at the beginning of Section 1.6.1, namely, to measure the state of $\mathcal{S}$ while in equilibrium, thus disturbing the latter, then setting it to a new desired state.

1.6.2.1 Simplifying Assumptions

In order to perform a concrete analysis of the different solutions to the computational problem just outlined, we continue to assume in what follows that the time required to perform all three operations below (in the given order) is one time unit:

1. Measuring one variable x_i, $0 \leq i \leq n-1$
2. Computing a new value for a variable x_i, $0 \leq i \leq n-1$
3. Setting one variable x_i, $0 \leq i \leq n-1$

Furthermore, once the new values of the parameters $x_0, x_1, \ldots, x_{n-1}$ have been applied to $\mathcal{S}$, the system requires one additional time unit to reach a new state of equilibrium. It follows that the smallest $\mathcal{T}$ can be is two time units; we therefore assume that $\mathcal{T} = 2$.

1.6.2.2 A Mathematical Model

We now present a mathematical model of the computation in Section 1.6.1. Recall that the physical system has the property that all variables are related to, and depend on, one another. Furthermore, measuring (or setting) one variable disturbs any number of the remaining variables unpredictably (meaning that we cannot tell which variables have changed value, and by how much). Typically, the system evolves until it reaches a state of equilibrium and, in the absence of external perturbations, it can remain in a stable state indefinitely.

Formally, the interdependence among the n variables can be modeled using n functions, $g_0, g_1, \ldots, g_{n-1}$, as follows:

$$\begin{aligned}
x_0(t+1) &= g_0(x_0(t), x_1(t), \ldots, x_{n-1}(t)) \\
x_1(t+1) &= g_1(x_0(t), x_1(t), \ldots, x_{n-1}(t)) \\
&\vdots \\
x_{n-1}(t+1) &= g_{n-1}(x_0(t), x_1(t), \ldots, x_{n-1}(t)).
\end{aligned}$$

These equations describe the evolution of the system from state $(x_0(t), x_1(t), \ldots, x_{n-1}(t))$ at time t to state $(x_0(t+1), x_1(t+1), \ldots, x_{n-1}(t+1))$, one time unit later. While each variable is written as a function of time, there is a crucial difference between the present situation and that in Section 1.3: When the system is in a state of equilibrium, its variables do not change over time. It is also important to emphasize that, in most cases, the dynamics of the system are very complex, so the mathematical descriptions of functions $g_0, g_1, \ldots, g_{n-1}$ are either not known to us or we only have rough approximations for them.

Assuming the system is in an equilibrium state, our task is to measure its variables (in order to compute new values for these variables and set the system to these new values). In other words, we need the values of $x_0(t_0), x_1(t_0), \ldots, x_{n-1}(t_0)$ at moment $t = t_0$, when the system is in a stable state.

We can obtain the value of $x_0(t_0)$, for instance, by measuring that variable at time t_0 (noting that the choice of x_0 here is arbitrary; the argument remains the same regardless of which of the n variables we choose to measure first). Although we can acquire the value of $x_0(t_0)$ easily in this way, the consequences for the entire system can be dramatic. Unfortunately, any measurement is an external perturbation for the system, and in the process, the variable subjected to measurement will be affected unpredictably.

Thus, the measurement operation will change the state of the system from $(x_0(t_0), x_1(t_0), \ldots, x_{n-1}(t_0))$ to $(x_0'(t_0), x_1(t_0), \ldots, x_{n-1}(t_0))$, where $x_0'(t_0)$ denotes the value of variable x_0 after measurement. Since the measurement process has a nondeterministic effect on the variable being measured, we cannot estimate $x_0'(t_0)$ in any way. Note also that the transition from $(x_0(t_0), x_1(t_0), \ldots, x_{n-1}(t_0))$, that is, the state before measurement, to $(x_0'(t_0), x_1(t_0), \ldots, x_{n-1}(t_0))$, that is, the state after measurement, does not correspond to the normal evolution of the system according to its dynamics described by functions g_i, $0 \leq i \leq n-1$.

However, because the equilibrium state was perturbed by the measurement operation, the system will react with a series of state transformations, governed by equations defining the g_i. Thus, at each time unit after t_0, the parameters of the system will evolve either toward a new equilibrium state or perhaps fall into a chaotic behavior. In any case, at time $t_0 + 1$, all n variables have acquired new values, according to the functions g_i:

$$\begin{aligned} x_0(t_0+1) &= g_0(x_0'(t_0), x_1(t_0), \ldots, x_{n-1}(t_0)) \\ x_1(t_0+1) &= g_1(x_0'(t_0), x_1(t_0), \ldots, x_{n-1}(t_0)) \\ &\vdots \\ x_{n-1}(t_0+1) &= g_{n-1}(x_0'(t_0), x_1(t_0), \ldots, x_{n-1}(t_0)). \end{aligned}$$

Consequently, unless we are able to measure all n variables, in parallel, at time t_0, some of the values composing the equilibrium state

$$(x_0(t_0), x_1(t_0), \ldots, x_{n-1}(t_0))$$

will be lost without any possibility of recovery.

1.6.2.3 Sequential Approach

The 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. Any value read afterward will not satisfy $\mathcal{G}(x_0, x_1, \ldots, x_{n-1})$.

Similarly, the sequential approach cannot update the variables of $\mathcal{S}$ properly: once x_0 has received its new value, setting x_1 disturbs x_0 unpredictably.

1.6.2.4 Parallel Approach

A parallel computer with n processors, by contrast, will measure *all* the variables $x_0, x_1, \ldots, x_{n-1}$ simultaneously (one value per processor), and therefore obtain an accurate reading of the state of the system within the given time frame. Consequently,

1. A snapshot of the state of the system that satisfies $\mathcal{G}(x_0, x_1, \ldots, x_{n-1})$ has been obtained.
2. The new variables $x_0, x_1, \ldots, x_{n-1}$ can be computed in parallel (one value per processor).
3. These new values can also be applied to the system simultaneously (one value per processor).

Following the resetting of the variables $x_0, x_1, \ldots, x_{n-1}$, a new equilibrium is reached. The entire process concludes within $\mathcal{T}$ time units successfully.

1.6.3 Distinguishability in Quantum Computing

We conclude our study of interacting variables with an example from quantum computation. In Section 1.3.1 we saw that a single qubit can be in a superposition of two states, namely $|0\rangle$ and $|1\rangle$. In the same way, it is possible to place an entire quantum register, made up of n qubits, in a superposition of two states. The important point here is that, unlike the case in Section 1.3.1, it is not the individual qubits that are in a superposition, but rather the entire register (viewed as a whole).

Thus, for example, the register of n qubits may be put into any one of the following 2^n states:

$$\begin{gathered} \frac{1}{\sqrt{2}}(|000\cdots0\rangle \pm |111\cdots1\rangle) \\ \frac{1}{\sqrt{2}}(|000\cdots1\rangle \pm |111\cdots0\rangle) \\ \vdots \\ \frac{1}{\sqrt{2}}(|011\cdots1\rangle \pm |100\cdots0\rangle). \end{gathered}$$

These vectors form an orthonormal basis for the state space corresponding to the n-qubit system. In such superpositions, the n qubits forming the system are said to be *entangled*: Measuring any one of them causes the superposition to collapse into one of the two basis vectors contributing to the superposition. Any subsequent measurement of the remaining $n - 1$ qubits will agree with that basis vector to which the superposition collapsed. This implies that it is impossible through single measurement to distinguish among the 2^n possible states. Thus, for example, if after one qubit is read the superposition collapses to $|000\cdots0\rangle$, we will have no way of telling which of the two superpositions, $\frac{1}{\sqrt{2}}(|000\cdots0\rangle + |111\cdots1\rangle)$ or $\frac{1}{\sqrt{2}}(|000\cdots0\rangle - |111\cdots1\rangle)$, existed in the register prior to the measurement.

The only chance to differentiate among these 2^n states using quantum measurement(s) is to observe the n qubits simultaneously, that is, perform a single joint measurement of the entire system. In the given context, *joint* is really just a synonym for *parallel*. Indeed, the device in charge of performing the joint measurement must possess the ability to "read" the information stored in each qubit, in parallel, in a perfectly synchronized manner. In this sense, at an abstract level, the measuring apparatus can be viewed as having n probes. With all probes operating in parallel, each probe can "peek" inside the state of one qubit, in a perfectly synchronous operation. The information gathered by the n probes is seen by the measuring device as a single, indivisible chunk of data, which is then interpreted to give one of the 2^n entangled states as the measurement outcome.

It is perhaps worth emphasizing that if such a measurement cannot be applied then the desired distinguishability can no longer be achieved, regardless of how many other measuring operations we are allowed to perform. In other words, even an infinite sequence of measurements touching at most $n - 1$ qubits at the same time cannot equal a single joint measurement involving all n qubits. Furthermore, with respect to the particular distinguishability problem that we have to solve, a single joint measurement capable of observing $n - 1$ qubits simultaneously offers no advantage whatsoever over a sequence of $n - 1$ consecutive *single* qubit measurements [31,32].

1.7 Computations Obeying Mathematical Constraints

In this section, we examine computational problems in which a certain mathematical condition must be satisfied throughout the computation. Such problems are quite common in many subareas of computer science, such as numerical analysis and optimization. Thus, the condition may be a local one; for example, a variable may not be allowed to take a value larger than a given bound. Alternatively, the condition may be global, as when the average of a set of variables must remain within a certain interval. Specifically, for $n > 1$, suppose that some function of the n variables, $x_0, x_1, \ldots, x_i, \ldots, x_{n-1}$, is to be computed. The requirement here is that the variables satisfy a stated condition at each step of the computation. In particular, if the effect of the computation is to change x_i to x_i' at some point, then the condition must remain true, whether it applies to x_i alone or to the entire set of variables, whatever the case may be. If the condition is not satisfied at a given moment of the computation, the latter is considered to have failed.

Our concern in what follows is with computations that fit the broad definition just presented, yet can only be performed successfully in parallel (and not sequentially). All n variables, $x_0, x_1, \ldots, x_i, \ldots, x_{n-1}$, are already stored in memory. However, modifying any *one* of the variables from x_i to x_i', to the exclusion of the others, causes the required condition (whether local or global) to be violated, and hence the computation to fail.

1.7.1 Mathematical Transformations

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. Three examples of such transformations are now described.

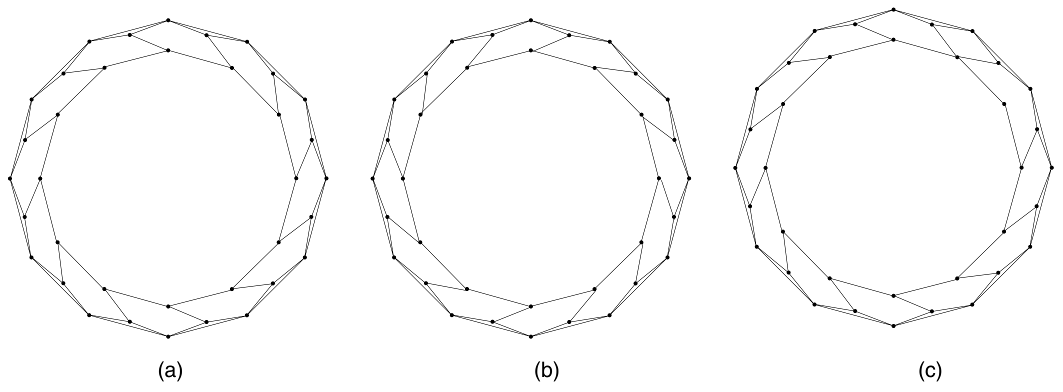

FIGURE 1.1 Subdivision: (a) origin, (b) destination, (c) with a concavity.

1.7.1.1 Geometric Flips

The object shown in Figure 1.1a is called a *convex subdivision*, as each of its faces is a convex polygon. This convex subdivision is to be transformed into that in Figure 1.1b.

The transformation can be effected by removing edges and replacing them with other edges. The condition for a successful transformation is that each intermediate figure (resulting from a replacement) be a convex subdivision as well. There are n edges in Figure 1.1a that can be removed and replaced with another n edges to produce Figure 1.1b, where $n = 12$ for illustration. These are the "spokes" that connect the outside "wheel" to the inside one. However, as Figure 1.1c illustrates, removing any *one* of these edges and replacing it with another creates a concavity, thus violating the condition [6, 29].

1.7.1.2 Map Coloring

A simple map is given consisting of n contiguous regions, where $n > 1$. Each region is a vertical strip going from the top edge to the bottom edge of the map. The regions are colored using two colors, red (R) and blue (B), in alternating fashion, thus

$$\ldots RBRBRBRBRBRBRBRBRB \ldots$$

It is required to recolor this map, such that each region previously colored R is now colored B, and conversely, each region previously colored B is now colored R, thus

$$\ldots BRBRBRBRBRBRBRBRBR \ldots$$

The condition to be satisfied throughout the recoloring is that no two adjacent regions are colored using the same color, and no third color (beside R and B) is ever used. It is clear that changing any *one* color at a time violates this requirement [24].

1.7.1.3 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 $n > 1$. The rewrite rules to be used are

$$\begin{aligned} a &\rightarrow ab \\ b &\rightarrow ab. \end{aligned}$$

Thus, for $n = 3$, the target string is $ababab$. 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 [38]. Here we note that applying any *one* of the two rules at a time causes the computation to fail (e.g., if *ab* is changed to *abb*, by the first rewrite rule, or to *aab* by the second) [24].

1.7.2 Sequential Solution

With all the x_i in its memory, suppose without loss of generality that the sequential computer obtains x'_0. This causes the computation to fail, as the set of variables $x'_0, x_1, x_2, \ldots, x_{n-1}$ does not satisfy the global condition. Thus, in Section 1.7.1.1, only one edge of the subdivision in Figure 1.1(a) can be replaced at a time. Once any one of the n candidate edges is replaced, the global condition of convexity no longer holds. The same is true in Sections 1.7.1.2 and 1.7.1.3, where the sequential computer can change only one color or one symbol at once, respectively, thereby causing the adjacency conditions to be violated.

1.7.3 Parallel Solution

For a given n, a parallel computer with n processors can easily perform a transformation on all the x_i collectively, with processor i computing x'_i. The required property in each case is maintained leading to a successful computation. Thus, in Section 1.7.1.1, n edges are removed from Figure 1.1a and n new edges replace them to obtain Figure 1.1b, all in one step. Similarly in Section 1.7.1.2, all colors can be changed at the same time. Finally, in Section 1.7.1.3, 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 1.5). Indeed, each legal string (i.e., 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).

1.8 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 [17, 20, 30]. 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 [16, 21, 22].

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 1.3 through 1.7. 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 finite computer is endowed with an unlimited memory and is allowed to compute for an indefinite period 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 [26], the random access machine [40], and other idealized models, 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 [5]. It is clear from Section 1.4.3 that even accelerating machines are not universal.

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 the field. Of particular relevance to parallel computing are the myths of the *Speedup Theorem* (speedup is at most equal to the number of processors used in parallel), the *Slowdown Theorem*, also known as *Brent's Theorem* (when q instead of p processors are used, $q < p$, the slowdown is at most p/q), and Amdahl's Law (maximum speedup is inversely proportional to the portion of the calculation that is sequential). Each of these myths can be dispelled using the same computations presented in this chapter. Other computations for dispelling these and other myths are presented in Reference 4.

1.9 Conclusion

An evolving computation is one whose characteristics vary during its execution. In this chapter, we used evolving computations to identify a number of computational paradigms involving problems whose solution necessitates the use of a parallel computer. These include computations with variables whose values change with the passage of time, computations whose computational complexity varies as a function of time, computations in which the complexity of a stage of the computation depends on the order of execution of that stage, computations with variables that interact with one another and hence change each other's values through physical processes occurring in nature, and computations subject to global mathematical constraints that must be respected throughout the problem-solving process. In each case, n computational steps must be performed simultaneously in order for the computation to succeed. A parallel computer with n processors can readily solve each of these problems. No sequential computer is capable of doing so. Interestingly, this demonstrates that one of the fundamental principles in computing, namely, that any computation by one computer can be simulated on another, is invalid. None of the parallel solutions described in this chapter can be simulated on a sequential computer, regardless of how much time and memory are allowed.

Another consequence of our analysis is that the concept of universality in computing is unachievable. For every putative universal computer $\mathcal{U}_1$ capable of $V(t)$ operations at time unit t, it is always possible to define a computation $\mathcal{P}_1$ requiring $W(t)$ operations at time unit t to be completed successfully, where $W(t) > V(t)$, for all t. While $\mathcal{U}_1$ fails, another computer $\mathcal{U}_2$ capable of $W(t)$ operations at time unit t succeeds in performing $\mathcal{P}_1$ (only to be defeated, in turn, by a computation $\mathcal{P}_2$ requiring more than $W(t)$ operations at time unit t). Thus, no finite computer can be universal. That is to say, no machine, defined once and for all, can do all computations possible on other machines. This is true regardless of how $V(t)$ is defined, so long as it is fixed: It may be a constant (as with all of today's computers), or grow with t (as with accelerating machines). The only possible universal computer would be one that is capable of an *infinite* number of operations per step. As pointed out in Reference 5 the Universe satisfies this condition. This observation agrees with recent thinking to the effect that the Universe is a computer [23, 27, 47, 50]. As stated in Reference 17: "[T]hink of all our knowledge-generating processes, our whole culture and civilization, and all the thought processes in the minds of every individual, and indeed the entire evolving biosphere as well, as being a gigantic *computation*. The whole thing is executing a self-motivated, self-generating computer program."

Acknowledgment

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

References

[1] A. Adamatzky, B. DeLacy Costello, and T. Asai, *Reaction-Diffusion Computers*, Elsevier Science, New York, 2005.

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

[3] S.G. Akl, The design of efficient parallel algorithms, in: *Handbook on Parallel and Distributed Processing*, Blazewicz, J., Ecker, K., Plateau, B., and Trystram, D., Eds., Springer-Verlag, Berlin, 2000, pp. 13–91.

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

[5] S.G. Akl, 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, Austria and Jožef Stefan Institute, Ljubljana, Slovenia, 2005, pp. 211–236.

[6] S.G. Akl, Inherently parallel geometric computations, *Parallel Processing Letters*, Vol. 16, No. 1, March 2006, pp. 19–37.

[7] S.G. Akl, Even accelerating machines are not universal, Technical Report No. 2006-508, School of Computing, Queen's University, Kingston, Ontario, Canada, 2006.

[8] S.G. Akl, B. Cordy, and W. Yao, An analysis of the effect of parallelism in the control of dynamical systems, *International Journal of Parallel, Emergent and Distributed Systems*, Vol. 20, No. 2, June 2005, pp. 147–168.

[9] B.H. Bransden and C.J. Joachain, *Quantum Mechanics*, Pearson Education, Harlow (Essex), England, 2000.

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

[11] T.J. Chung, *Computational Fluid Dynamics*, Cambridge University Press, Cambridge, United Kingdom, 2002.

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

[13] B.J. Copeland, Even Turing machines can compute uncomputable functions, in: *Unconventional Models of Computation*, Calude, C.S., Casti, J., and Dinneen, M.J., Eds., Springer-Verlag, Singapore, 1998, pp. 150–164.

[14] B.J. Copeland, Accelerating Turing machines, *Mind and Machines*, Vol. 12, No. 2, 2002, pp. 281–301.

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

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

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

[18] P.L. Freddolino, A.S. Arkhipov, S.B. Larson, A. McPherson, and K. Schulten, Molecular dynamics simulation of the complete satellite tobacco mosaic virus, *Structure*, Vol. 14, No. 3, March 2006, pp. 437–449.

[19] D. Gabor, Theory of communication, *Proceedings of the Institute of Electrical Engineers*, Vol. 93, No. 26, 1946, pp. 420–441.

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

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

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

[23] K. Kelly, God is the machine, *Wired*, Vol. 10, No. 12, December 2002, pp. 170–173.

[24] A. Koves, personal communication, 2005.

[25] R. Lewin, *Complexity*, The University of Chicago Press, Chicago, 1999.

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

[27] S. Lloyd, *Programming the Universe*, Knopf, New York, 2006.

[28] M. Lockwood, *The Labyrinth of Time: Introducing the Universe*, Oxford University Press, Oxford, England, 2005.

[29] H. Meijer and D. Rappaport, Simultaneous Edge Flips for Convex Subdivisions, *16th Canadian Conference on Computational Geometry*, Montreal, August 2004, pp. 57–69.

[30] M.L. Minsky, *Computation: Finite and Infinite Machines*, Prentice-Hall, Englewood Cliffs, NJ, 1967.

[31] M. Nagy and S.G. Akl, 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.

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

[33] M. Nagy and S.G. Akl, Quantum computation and quantum information, *International Journal of Parallel, Emergent and Distributed Systems*, Vol. 21, No. 1, February 2006, pp. 1–59.

[34] M. Nagy and S.G. Akl, Coping with decoherence: Parallelizing the quantum Fourier transform, Technical Report No. 2006-507, School of Computing, Queen's University, Kingston, Ontario, Canada, 2006.

[35] M.A. Nielsen and I.L. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press, Cambridge, United Kingdom, 2000.

[36] L.W. Potts, *Quantitative Analysis*, Harper & Row, New York, 1987.

[37] D.B. Pribor, *Functional Homeostasis: Basis for Understanding Stress*, Kendall Hunt, Dubuque, IA, 1986.

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

[39] D.C. Rapaport, *The Art of Molecular Dynamics Simulation*, Cambridge University Press, Cambridge, United Kingdom, 2004.

[40] J.E. Savage, *Models of Computation*, Addison-Wesley, Reading, MA, 1998.

[41] C.E. Shannon, Communication in the presence of noise, *Proceedings of the IRE*, Vol. 37, 1949, pp. 10–21.

[42] T. Sienko, A. Adamatzky, N.G. Rambidi, M. Conrad, Eds., *Molecular Computing*, MIT Press, Cambridge, MA, 2003.

[43] I. Stewart, Deciding the undecidable, *Nature*, Vol. 352, August 1991, pp. 664–665.

[44] I. Stewart, The dynamics of impossible devices, *Nonlinear Science Today*, Vol. 1, 1991, pp. 8–9.

[45] G. Stix, Real time, *Scientific American Special Edition: A Matter of Time*, Vol. 16, No. 1, February 2006, pp. 2–5.

[46] K. Svozil, The Church-Turing thesis as a guiding principle for physics, in: *Unconventional Models of Computation*, Calude, C.S., Casti, J., and Dinneen, M.J., Eds., Springer-Verlag, Singapore, 1998, pp. 150–164.

[47] F.J. Tipler, *The Physics of Immortality*, Macmillan, London, 1995.

[48] P.A. Trojan, *Ecosystem Homeostasis*, Dr. W. Junk, Dordrecht, The Netherlands, 1984.

[49] R. Wilson and G.H. Granlund, The uncertainty principle in image processing, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-6, No. 6, 1984, pp. 758–767.

[50] S. Wolfram, *A New Kind of Science*, Wolfram Media, Champaign, IL, 2002.