Computing and Informatics, Vol. 21, 2002, 455–487

PARALLEL REAL-TIME COMPUTATION: SOMETIMES QUANTITY MEANS QUALITY[*]

Selim G. Akl

School of Computing, Queen's University
Kingston, Ontario K7L 3N6, Canada
e-mail: akl@cs.queensu.ca

Manuscript received 10 December 2001; revised 28 November 2002
Communicated by Peter Ružička

Abstract. The primary purpose of parallel computation is the fast execution of computational tasks that require an inordinate amount of time to perform sequentially. As a consequence, interest in parallel computation to date has naturally focused on the speedup provided by parallel algorithms over their sequential counterparts. The thesis of this paper is that a second equally important motivation for using parallel computers exists. Specifically, the following question is posed: Can parallel computers, thanks to their multiple processors, do more than simply speed up the solution to a problem? We show that within the paradigm of real-time computation, some classes of problems have the property that a solution to a problem in the class, when computed in parallel, is far superior in quality than the best one obtained on a sequential computer. What constitutes a better solution depends on the problem under consideration. Thus, 'better' means 'closer to optimal' for optimization problems, 'more secure' for cryptographic problems, and 'more accurate' for numerical problems. Examples from these classes are presented. In each case, the solution obtained in parallel is significantly, provably, and consistently better than a sequential one.

It is important to note that the purpose of this paper is not to demonstrate merely that a parallel computer can obtain a better solution to a computational problem than one derived sequentially. The latter is an interesting (and often surprising) observation in its own right, but we wish to go further. It is shown here that the improvement in quality can be *arbitrarily high* (and certainly *superlinear* in the number of processors used by the parallel computer). This result is akin

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

to superlinear speedup — a phenomenon itself originally thought to be impossible.

Keywords: Parallelism, real-time computation, optimization, cryptography, numerical analysis

1 INTRODUCTION

The range and scope of computer applications in today's society are breathtaking. From business to medicine, from communication to education and entertainment, there is an ever increasing demand for computers that can perform complex tasks quickly and precisely. It is also becoming apparent that present and foreseeable computers, largely conventional in design, will fall short of the expectations. Indeed, on a *sequential computer* there is but *one* processor. An algorithm for solving a problem on such a computer is executed *one* step at a time. Due to physical limitations, single-processor computers are steadily approaching a point beyond which they will be unable to provide answers to certain computational problems within the required time limits.

One way out of this impasse is to abandon the sequential model of computation. A *parallel computer*, by contrast with a sequential one, possesses *several* processors. A problem to be solved is broken into smaller parts that are solved simultaneously. The processors, working in parallel, execute *several* steps of an algorithm *at the same time*. This way, we hope to achieve a significant reduction in the time required to solve the problem at hand.

Ever since parallel computation appeared on the computer science scene as an alternative to conventional computing, questions were raised regarding the capabilities of parallel computers. The vast majority of these questions have to do with the *speedup*, if any, provided by parallel computers: Can parallel computers solve computational problems faster than sequential computers, and if so, how much faster? In this paper, we ask a different type of question: Can parallel computers, thanks to their multiple processors, do more than simply speed up the solution to a problem? In particular, can a parallel computer provide a solution to a computational problem that is *better* than the best-possible solution that can be obtained sequentially?

We begin by reviewing some of the issues surrounding the notion of speedup, leading up to the central question of this paper. In what follows, the speedup provided by a parallel algorithm when solving a problem is defined as follows: Worst-case running time of the best sequential algorithm for the problem divided by the worst-case running time of the parallel algorithm. Throughout the paper we adopt the standard definition of *time unit*, that is, the unit traditionally used to measure the running time of an algorithm [1, 13, 25, 37]: A time unit is the length of time required by a processor to read a datum from memory, perform a constant-time operation (such as adding two numbers), and write a datum to memory.

1.1 Speedup

As the raison-d'être of parallel computers is the speeding up of computations performed sequentially (that is, using one processor), the first question to be asked was: *Is speedup possible at all?* More specifically, and perhaps more precisely, if a computation requires T_1 time units on a one-processor computer, can it be performed on a parallel computer with n processors, $n > 1$, in time $T_n = T_1/f(n)$, where $f(n)$ is $\omega(1)$ and $O(n)$, that is, $f(n)$ is asymptotically larger than any constant and asymptotically no larger than n?

It is now widely known that this question is answered in the affirmative, at least in theory. There is ample and well documented evidence of parallel algorithms whose running time satisfies the aforementioned condition. For instance, parallel algorithms using n processors and running in $O(\log n)$ time exist to

1. sort n numbers in non-decreasing order using comparisons [57],
2. find the convex hull of n planar points [37],
3. compute the discrete Fourier transform of n inputs [61],

to name just a few examples of problems for which the best sequential algorithms run in $O(n \log n)$ time.

1.2 Superlinear Speedup

Having established that a speedup by a factor of $f(n)$ is indeed possible, it was natural to ask whether further speedup could be achieved through parallelism. The second question, therefore, was: *Is superlinear speedup possible?* In other words, can a computation requiring T_1 time units sequentially be performed on a parallel computer with n processors in time $T_n = T_1/g(n)$, where $g(n)$ is $\Omega(n)$, that is, $g(n)$ is asymptotically larger than or equal to cn, for some positive constant c? Once again, several examples of computations satisfying this condition have been published. These examples, are less well known as they concern nonstandard, yet realistic, paradigms, including, for example, problems where

1. all the data are not available at the outset of the computation, but instead arrive over time; the computation is considered complete when all the data arrived so far have been handled regardless of whether more data arrive later [14, 15, 17, 18, 45, 46, 47],
2. the values of the data change as the algorithm proceeds; the computation is considered complete when all the corrections arrived so far have been handled regardless of whether more corrections arrive later [16, 45, 46, 47],
3. a computation is involved that is not efficiently invertible: with a sufficient degree of parallelism, the inverse computation is not required; with an insufficient number of processors, the inverse computation becomes necessary [2, 9].

1.3 Infinite Speedup

Taking the line of reasoning expressed in Section 1.2 to its logical limit, the ultimate question was: *Are there computations for which parallelism makes the difference between success and failure?* In other words, are there computational problems whose solution can be obtained only on a parallel computer with sufficiently many processors (while any computer with fewer processors is guaranteed to fail in computing the solution)? Recent work has demonstrated that the answer here is also positive. Examples include computations with deadlines, computations involving several streams of input, and computations where data arrive in real time but each new input depends on the previous output [1, 4].

1.4 Beyond Speedup

The purpose of this paper is to begin the exploration of other capabilities of parallel computers beyond speedup (as originally hinted to in [1]). To this end, we wish to ask whether parallel computers can, in some circumstances, do more than just speed up the computation. We show that for real-time problems, a parallel computer can obtain a solution that is better than that obtained by a sequential one. Consider the following example.

Example 1.1. Suppose that a computer is programmed to play a two-player board game of strategy (such as Checkers, Chess, or Go), against a human or another computer. The computer program typically involves searching a tree data structure. In this tree, each node represents a board configuration and each edge represents a move. The root R of the tree represents the current configuration from which the computer is to make a move. The children of the root represent all possible board configurations reached by computer moves. The children of these represent configurations reached by the opponent's moves, and so on. In what follows we assume that each node has B children. Associated with each node is an evaluation function which assigns a value to that board configuration indicating its goodness from the computer's point of view.

In order to make a move, the computer searches the tree, up to a certain depth, and determines (among all leaves at that depth or less) which leaf (call it L) is best for it (assuming that each of the two players has chosen the best move from its viewpoint at each level). The edge leaving the root (on the unique path leading to L) is the selected move. Usually, such a move must be found by the computer within a fixed amount of time, for example $\mathcal{T}$ time units.

Assume that it is the computer's turn to move. If the computer is a sequential one, it can search the game tree up to a depth of d_1, that is, examine B^{d_1} leaves in $\mathcal{T}$ time units. A parallel computer, with p processors, on the other hand, can search the game tree up to a depth of d_p, where $d_p > d_1$, that is examine B^{d_p} leaves in $\mathcal{T}$ time units. The parallel computer, therefore, makes a more informed decision when choosing its move, having looked farther ahead in the game tree [5].

The situation described in the preceding example has worked fairly well for some games, such as Chess. For instance, the world Chess champion today is a parallel computer [50]. There is, however, no proof that this strategy works in all cases, for at least two reasons:

1. The game tree is not searched in full: Nodes at depth d_p do not necessarily represent end-game configurations (the latter may occur at a depth $D > d_p$).
2. The evaluation function may not measure the goodness of a position as accurately as one would want.

As a result, the parallel computer may on occasion arrive at a move that is *worse* than that obtained sequentially.

Yet, for many computational problems, parallel computation provides solutions that are not only *faster*, but also *better*, than any solution obtained sequentially. We offer three examples here:

1. *optimization problems* in which a solution is *better* if it is *closer to optimal*,
2. *cryptographic problems* in which a solution is *better* if it is *more secure*,
3. *numerical problems* in which a solution is *better* if it is *more accurate*.

The computations we describe fall within the *real-time paradigm*. Here, the data needed to solve a problem are received *on-line* and the results of the computation are to be delivered by a certain *deadline*. In each case we present, the parallel solution is significantly better than the best solution obtained sequentially. Furthermore, the improvement is provable and consistent. This point deserves to be stressed: The purpose of this paper is not to demonstrate merely that a parallel computer can obtain a better solution to a computational problem than one derived sequentially. The latter is an interesting (and often surprising) observation in its own right, but we wish to go further. It is shown in what follows that the improvement in quality can be *arbitrarily high* (and certainly superlinear in the number of processors used by the parallel computer). This result is akin to superlinear speedup — a phenomenon itself originally thought to be impossible.

We coin a new term to express the improvement in the quality of a solution to a problem afforded by parallel computation. Recall that the *speedup* is equal to the ratio of the running time of a sequential algorithm to that of the running time of a parallel algorithm, when both algorithms are used to solve a given computational problem. In the same way, we define *quality-up* as the ratio of the quality of a solution to a certain problem obtained in parallel to the quality of a solution to the same problem computed sequentially.

The remainder of this paper is organized as follows. Some background material relative to *real-time computation* and *models of computation* is presented in Section 2. The three application areas illustrating the ability of parallel computers to obtain solutions of higher quality than possible sequentially are the subject of Sections 3, 4, and 5. Some concluding thoughts are offered in Section 6.

We close this section with the following important observation. There exists an assumption underlying most theoretical results in parallel computation that leads some readers (who are unfamiliar with the field, and hence unaware of the implicit assumption) to ask: Why can't we use a faster sequential computer and achieve the same results obtained with the parallel one? In most cases, one can easily respond to this question by showing how simple it is to defeat the 'faster' sequential machine. For example, in a real-time environment (as in the present paper), it suffices to make the data-arrival rate *faster* than the new and improved sequential machine can handle! However, in order to avoid any such (perhaps confusing) arguments, we make the standard assumption explicitly at the outset (at the risk of stating the obvious): The analyses in this paper assume that all models of computation are the fastest possible (within the bounds established by theoretical physics). Specifically, no machine exists that is faster than the sequential computer of Section 2.2.1, and similarly no parallel computer exists whose processors are faster than the processors of the parallel computer of Section 2.2.2. This is the fundamental assumption in parallel computation. One should also keep in mind here that the length of a time unit is not an absolute quantity. Instead, the duration of a time unit is defined in terms of the speed of the processors available (namely, the single processor on the sequential computer and each processor on the parallel machine).

2 BACKGROUND

In this section we introduce the real-time paradigm as well as the models of computation used in this paper.

2.1 Real-Time Computation

The prevalent paradigm of computation, to which everyone who uses computers is accustomed, is one in which all the data required by an algorithm are available when the computer starts working on the problem to be solved. A different paradigm is *real-time* computation. Here, not all inputs are given at the outset. Rather, the algorithm receives its data (one or several at a time) *during* the computation, and must incorporate the newly arrived inputs in the solution obtained so far. Often, the data-arrival rate is constant; specifically, $\mathcal{N}$ data are received every $\mathcal{T}$ time units, where both $\mathcal{N}$ and $\mathcal{T}$ are fixed in advance.

A fundamental property of real-time computation is that certain operations must be performed by specified deadlines. Thus, one or more of the following conditions may be imposed:

1. Each received input (or set of inputs) must be processed within a certain time after its arrival.
2. Each output (or set of outputs) must be returned within a certain time after the arrival of the corresponding input (or set of inputs).

Thus, for example, it may be crucial for an application that each input be operated on as soon as it is received. Similarly, each partial solution (as well as the final one) may need to be returned as soon as it is available [32, 40, 56]. It is helpful to note here that, when no deadlines are imposed, computations for which inputs arrive while the algorithm is in progress are referred to as *on-line* [27, 33, 35, 36], *incremental* [21, 22, 49, 59], *dynamic* [11, 12, 67], and *updating* [20, 23, 28, 38, 53, 54, 62, 66]. It is also important to note that our definition, while striving to be as general as possible, is particularly suitable for our purposes in this paper. Many other definitions exist; see, for example, the various interpretations of the notion of real time provided in [10, 42, 65].

2.2 Models of Computation

Two models, one sequential and one parallel, are applied to the solution of the various real-time computational problems studied in this paper.

2.2.1 Sequential Model

This is the conventional model of computation used in the design and analysis of sequential (or *serial*) algorithms. It consists of a single processor equipped with a random-access memory to which the processor can gain access for the purpose of reading and writing. The processor has some local registers for intermediate results, a control memory to store its program, and some circuitry to perform arithmetic and logical operations. It also has input and output devices for communication with the outside world. During each cycle of the computation, the processor executes one instruction from its program: It fetches a datum from memory, performs an operation on it, and stores the result back in memory.

2.2.2 Parallel Model

Our chosen parallel model is the *pipeline* computer, shown in Fig. 1 [1]. In this model, n processors, denoted by P_1, P_2, $\ldots$, P_n, are connected to one another by (one-way) communication links such that:

1. P_1 receives its input from (and only from) the outside world.
2. P_i receives its input from (and only from) P_{i-1}, $2 \leq i \leq n$.
3. P_i sends its output to (and only to) P_{i+1}, $1 \leq i \leq n-1$.
4. P_n sends its output to (and only to) a memory or a communications channel.

Data travel from P_1 to P_n, with P_i beginning to operate only when it receives input, $1 \leq i \leq n$. Each processor is of the type described in Section 2.2.1.

It can be argued that the pipeline computer is the weakest of all models of parallel computation in which the processors have some means of communicating among themselves. Nonetheless this model, with its rudimentary communication

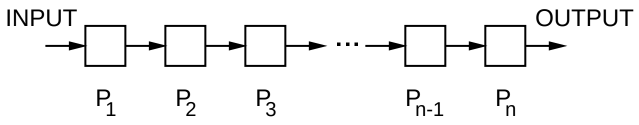


Fig. 1. Parallel computer

paths, is perfectly suitable when solving the real-time computational problems of this paper. This is demonstrated in Sections 3, 4, and 5, where it is shown that the pipeline computer affords a parallel algorithm that is significantly better than a sequential one. One should also recall here that the *weaker* the computational model used in an algorithmic analysis, the *stronger* the result obtained. This is true because any algorithm designed for a certain model can be executed (through simulation or otherwise) on a more powerful model *without any loss in performance.* It follows therefore that the results of this paper, which are derived for the weakest of all models of parallel computation (namely, the pipeline computer), hold in general (that is, for *all* parallel models).

3 OPTIMIZATION

Let f be a function of some real (or complex) variables. The field of *mathematical optimization* is concerned with finding an optimal value of f, subject to a number of conditions. Here, f is called the *objective function* and the conditions are known as the *constraints.* The optimal value is typically a *maximum* or a *minimum* of f satisfying the constraints. Often, when the exact maximum (or minimum) is difficult to obtain, an *approximation* of the optimal value is computed.

Mathematical optimization is a rich field of knowledge with many algorithms and a diversity of practical applications. The field is usually divided into several subfields, most notably: dynamic, linear, nonlinear, integer, stochastic, and geometric programming, control theory, combinatorial and discrete optimization, and so on. In combinatorial optimization, for example, typical functions to be minimized are defined in terms of weighted graphs; these include the problems of computing minimum-weight spanning trees, matchings, and paths [41, 52].

Evidently, we are interested here in mathematical optimization in real time [3]. Our purpose is to demonstrate the ability of a parallel algorithm to do better than the best sequential algorithm when solving an optimization problem in real time. In this context, a *better* solution is one that is *closer to optimal.*

The computational paradigm used in this section is presented in Section 3.1 along with a definition of the specific problem to be solved. Sequential and parallel solutions and their analyses are developed in Sections 3.2, 3.3, and 3.4, respectively.

3.1 Real-Time Optimization

We begin by describing the specific computation chosen to illustrate our point. For ease of exposition, the objective function to be optimized, as well as the constraints, are kept as simple as possible. Thus, the optimization problem to be solved calls for finding the maximum of a function of a single real variable, in a given range.

A few examples of the function to be maximized are first presented. The real-time computational environment and the conditions under which the solution is to be obtained are then introduced.

3.1.1 Examples of Functions to be Maximized

Consider the following functions whose maximum is to be found in a given range.

Example 3.1. Let f be a function of a real variable x. Given a real x_0, and a positive integer n, it is required to find an *integer* z^* in the range $R = [x_0, x_0 + n]$ for which f reaches its maximum value f^* over all integers in R. (If f is not defined for *any* integer in R, then 'undefined' is to be returned.)

It should be noted that nothing is known a priori regarding the exact form of f or its behavior in the range R. In particular, it is unknown whether f is continuous, or whether it is differentiable, or whether its first derivative has discontinuities in R. All these considerations are in fact irrelevant to the problem at hand since we are concerned with the maximum achieved by f *at an integer* inside the range R. Thus, for instance, a function defined over the range [0.5, 10.5], may achieve an overall maximum value at $x = 2.73$, while its maximum for an integer is achieved at $z^* = 8$.

Note also that f may be defined in terms of simple arithmetic and logical operators such as absolute value, mod, integer division, ceiling, floor, min, max, or, and, not, exclusive-or, and so on.

An example of a function here may be $f(x) = |ax - b\lfloor x + 1 \rfloor \bmod c|$, for integer constants a, b, and c.

Since nothing about f is known in advance, the only general procedure for solving this problem is exhaustive enumeration (i.e., try all integers in the given range and find one—among perhaps several—for which the function attains its largest value).

Example 3.2. Alternatively, the objective function may be defined recursively. Let x_0 be a real and n a positive integer. Thus, a sequence $x_1, x_2, \ldots, x_n$ of real numbers is obtained from the relation:

$$x_{i+1} = f({x_i}^b, i, n), \qquad \text{for } b > 1 \text{ and all } i \geq 0.$$

Here, f combines ${x_i}^b$, i, and the constant n with various multiplicative and additive terms, as well as other simple arithmetic functions. In this case, given f, x_0, and n, it is required to find the largest of $x_1, x_2, \ldots, x_n$.

One example of such a function, which will prove particularly useful to our subsequent analysis, is:

$$x_{i+1} = \left[\left(\lfloor x_i \rfloor + (-1)^{i+1}(i+1) \right)^{2u} \bmod (n + (-1)^{i+1})^v \right]^{w/v} \tag{1}$$

for $i \geq 0$, and positive integers u, v, and w, with $w > 1$.

Suppose for illustration that $u = 1$, $v = 2$ and $w = 3$. We have:

$$x_{i+1} = \left[\left(\lfloor x_i \rfloor + (-1)^{i+1}(i+1) \right)^{2} \bmod (n + (-1)^{i+1})^2 \right]^{3/2}$$

for $i \geq 0$.

Taking, for instance, $x_0 = 14.0$ and $n = 10$, we get: $x_1 = 18.520259$, $x_2 = 225.06221$, $x_3 = 216.0$, $x_4 = 0.0$, $x_5 = 125.0$, $x_6 = 1000.0$, $x_7 = 216.0$, $x_8 = 742.54158$, $x_9 = 64.0$, $x_{10} = 172.60069$.

In other words, the x_i values oscillate unpredictably, and that particular x_i achieving the maximum cannot be guessed in advance. The only way to find the largest x_i is to compute $x_1, x_2, \ldots, x_n$.

In some contexts the functions described in this example are called *nonlinear feedback functions* (and sometimes *aperiodic*, *chaotic*, and *complex functions*) [19, 24, 26, 30, 34, 43, 44, 68].

Example 3.3. In fact, the function to be maximized may not even have a closed form mathematically. The function may be defined by a program that takes a value for x and returns $f(x)$, presumably after performing certain tests on x. These tests may include arithmetic, as well as logical operations, or even table lookup. Given a range of x values, each value is fed to the program and the one achieving the maximum is found.

For the purposes of this paper we make the following assumptions:

1. For definiteness, we assume henceforth that the objective function is of the form given in Example 3.2, that is, the function f to be maximized is defined recursively as: $x_{i+1} = f({x_i}^b, i, n)$, where $b > 1$.
2. The function f to be maximized consists of a constant number of terms (i.e., f can be expressed using no more than a certain number of symbols fixed in advance). Similarly, each of x_0 and n fits in a constant number of words in memory. It is to be noted, as a consequence, that the optimization problem to be solved, being defined by f, x_0, and n, has a constant size formulation.
3. Each of $x_1, x_2, \ldots, x_n$ (and, consequently, the maximum value of f) also fits in a constant number of words. This assumption and the previous one together imply that the size of f, n, x_i, $i \geq 0$, and the current maximum is a constant multiple of the word size in bits. Therefore, this quadruple can be transmitted and received in a constant number of time units.

3.1.2 Computing the Maximum in Real Time

The specific problem to be solved in this section is defined as follows:

1. A computer system receives a stream of input in real time. These inputs represent the data of an optimization problem.
2. Time is divided into *intervals*. Each interval is $\mathcal{T}$ time units long, where $\mathcal{T}$ is a constant.
3. At the beginning of the j-th time interval, $j > 0$, an objective function f^j is received, together with a pair of constraints $C^j = (x_0^j, n^j)$.
4. It is required that the pair (f^j, C^j) be processed as soon as it is received and that the maximum value of $x_1^j, x_2^j, \ldots, x_n^j$ (or an approximation of it) subject to C^j be produced as output as soon as it is computed. Furthermore, one output must be produced at the end of each time interval (with possibly an initial delay before the first output is produced).
5. **Computational Assumption.** In one time interval a processor can
   (a) read f^j, n^j, x_i^j, and the current maximum,
   (b) compute x_{i+1}^j and the new maximum, and
   (c) output f^j, n^j, x_{i+1}^j, and the new maximum.

We now provide sequential and parallel solutions to this problem. This is followed by a comparative analysis.

3.2 Sequential Solution

A function f^j and a pair of constraints C^j are received at the beginning of the j-th time interval. These must be processed and the required maximum (or an approximation thereof) must be produced before the new function f^{j+1} and the new pair of constraints C^{j+1} are received (and demand to be processed) at the beginning of the $(j + 1)$st time interval. A sequential computer, by definition, has only one processor. Conforming to the computational assumption, in one time interval, the processor

1. receives f^j, x_0^j, and n^j,
2. computes x_1^j using x_0^j, n^j, and the definition of f^j, and
3. returns x_1^j as the required maximum.

Note here that x_1^j is not guaranteed to be the maximum of $x_1^j, x_2^j, \ldots, x_n^j$, as specified by the problem definition. Since the sequential computer cannot compute $x_2^j, x_3^j, \ldots, x_n^j$ before the pair (f^{j+1}, C^{j+1}) is received, it returns the *only* approximation of the maximum that it can obtain.

3.3 Parallel Solution

On a pipeline computer with n processors, P_1, P_2, …, P_n, processor P_1 is in charge of reading new inputs, while P_n is designated to produce the output. Therefore, at the beginning of the j-th interval, P_1 receives (f^j, C^j). It computes x_1^j and (for lack of another value with which to compare it) calls it the current maximum. It then sends the quadruple $(f^j, x_1^j, n^j$, current maximum) to P_2. The jth time interval has now ended and the $(j+1)$st commences. While P_1 is reading a new input, P_2 receives the quadruple sent by P_1. It computes x_2^j, compares it with current maximum, updates the latter if necessary, and sends the new quadruple $(f^j, x_2^j, n^j$, current maximum) to P_3. This continues, with processor P_k computing x_k^j during time interval $j+k-1$, $j>0$, $k \geq 1$. The maximum of $x_1^j, x_2^j, \ldots, x_n^j$ is produced by P_n at the end of the $(j+n-1)$st time interval. One time interval later, that is, at the end of the $(j+n)$th time interval, P_n produces as output the maximum of $x_1^{j+1}, x_2^{j+1}, \ldots, x_n^{j+1}$.

3.4 Analysis

For concrete analysis, suppose that the function f^j is of the form given by Equation (1). It is clear that, for this function, the ratio of x_1^j to the maximum of x_1^j, x_2^j, …, x_n^j could be $O(1/n^w)$, in the worst case. Since the sequential computer returns x_1^j as the maximum, while the parallel computer obtains the exact maximum, using n processors instead of one yields an $O(n^w)$ improvement in the quality of the solution.

4 CRYPTOGRAPHY

We present a problem from real-time cryptography for which a parallel solution is consistently better than a sequential solution. In Section 3, the notion of 'better' meant 'closer to optimal'. In this section, 'better' is interpreted as meaning 'more secure'. Specifically, the problem to be solved is one in which blocks of data are received by a computer system from the outside world at regular intervals and must be encrypted. No input block can be stored unencrypted, and thus must be processed as soon as it arrives. The encrypted blocks are to be produced as output, also at regular intervals. If the computer system operates sequentially, it can apply only one iteration of an encryption function on each block within the time available. By contrast, if n processors are used, n iterations of the encryption function are possible. This results in a significantly higher degree of security. In fact, we show that the parallel implementation is infinitely better than the sequential one [7].

We first provide a brief description of some basic notions from the field of cryptography. The problem to be solved is defined in Section 4.1. Sequential and parallel solutions and their analyses are presented in Sections 4.2, 4.3, and 4.4, respectively.

Modern Cryptography. The purpose of contemporary cryptography is the protection of digital information. The information may be, for example, personal, commercial, financial, or military. It may be stored in the memory of a device (such as a bank card or a computer), or it may be traveling on an insecure communications channel (such as a telephone cable or the electromagnetic waves of a wireless transmission). What is to be protected is the *secrecy* of the information, its *integrity*, its *authenticity*, and so on.

In order to accomplish these goals, modern cryptography uses a mathematical transformation known as a *cryptosystem.* Let M be a meaningful piece of information, called the *plaintext.* Thus, M may contain, for example, alphabetical, numerical, sound, or image data. An encryption function E transforms M, using a key K, into another piece of information C, referred to as the *ciphertext.* This function E typically works in a number n of iterations as follows:

$$C_i = E(K, C_{i-1}),$$

where $C_0 = M$ and $C_n = C$. Usually, M is replaced with C (in memory or on the communications channel) and the information contained in M is thus protected against various forms of attack by an opponent (such as eavesdropping, for example). When the plaintext is to be recovered by a legitimate party, a *decryption* function D, using cryptographic key K', operates on C (in a manner similar to the way E operated on M) and allows M to be obtained from the ciphertext.

What constitutes an iteration in the definition of E (and D) depends on the cryptosystem being used.

Symmetric Cryptosystem. If the cryptosystem is a *symmetric* one, meaning that $K = K'$, then an iteration of E consists of a constant number r of substitution-transposition rounds, numbered 1 to r. Here, the text to be encrypted is viewed as a string of bits. This string is divided into *blocks*, where each block M is b bits long. The function E is now applied to M. The first round receives M as input. Subsequently, the output of round i is the input to round $i + 1$, $1 \leq i \leq r - 1$. Within each round, a substitution followed by a transposition are performed under the control of the key K:

1. *Substitution*: Each bit of the binary string received as input to this round is replaced by another bit (for example, assuming that $b = 6$, the block 101101 may become 011100 under a substitution transformation).
2. *Transposition*: The bits of the binary string resulting from the substitution phase are permuted (for example, the block 011100 may become 100110 under a transposition transformation).

The same description applies to an iteration of D. An example of a symmetric cryptosystem is the Data Encryption Standard (DES) [48, 58, 60, 64]. Here, M and C are each 64 bits long, K has 56 bits, and one iteration consisting of 16 rounds is performed. Usually, the number of iterations (and hence the total number of rounds) depends on the length of the key. The longer is the key used, the larger is the number of iterations possible. For example, a 112-bit key for DES would allow two iterations (that is, 32 rounds). For simplicity, we assume in what follows that a k-bit key allows k encryption rounds.

Asymmetric Cryptosystem. In an *asymmetric* cryptosystem $K \neq K'$. An iteration of E usually performs an operation in modulo arithmetic, such as raising an integer to some exponent, followed by modular reduction. In this case, the text to be encrypted is broken into blocks (of alphabetic characters, for example), and each block is mapped to an integer M, where $0 \leq M \leq m-1$, and m is a large positive integer called the *modulus*. For $1 \leq i \leq n$, an iteration takes the form

$$C_i = C_{i-1}^{e_i} \bmod m,$$

where e_i is a positive integer. In particular,

$$C_n = M^{e_1 e_2 \cdots e_n} \bmod m.$$

The pair $(e_1 e_2 \cdots e_n, m)$ represents the encryption key. Typically, the exponent $e_1 e_2 \cdots e_n$ of M depends on m: A larger modulus allows for a larger exponent, and hence for more iterations of the encryption function E. A similar transformation is used to describe an iteration of D.

The preceding description of an iteration of E is representative of asymmetric encryption and is inspired by the Rivest-Shamir-Adleman (RSA) cryptosystem, named after its inventors [48, 58, 60, 64].

Modern cryptography is founded on the principle that it should be *computationally hard* to obtain the plaintext from the ciphertext without knowledge of the decryption key. For most cryptosystems (symmetric and asymmetric) a necessary and often sufficient condition for achieving this goal is to use *keys that are large in size.* Of course, a large key size makes it impractical for an opponent to launch an exhaustive attack based on key enumeration. Of more importance to our purpose in this paper, however, is the fact that a large key contributes to making the function E computationally hard to invert. One reason for this is that a large key allows for a large number n of iterations of E when computing C from M. In the remainder of this paper we assume that a cryptosystem implemented using a long key is more secure than the same cryptosystem implemented using a shorter key.

The results in this paper apply to both symmetric and asymmetric cryptosystems. However, we do not specify exactly which functions are used for encryption and decryption. Indeed, any function fitting the broad description in this section will be adequate. For definiteness, we do present in our subsequent treatment examples of general functions encompassing each of the two families. We also make specific assumptions about the computational requirements of these functions and their level of security. Detailed introductions to the field of cryptography are provided in [48, 58, 60, 64].

4.1 Real-Time Cryptography

The problem to be solved is defined as follows:

1. A computer system receives a stream of plaintext data in real time. These data are to be encrypted.
2. Time is divided into intervals. Each interval is $\mathcal{T}$ time units long, where $\mathcal{T}$ is a positive constant.
3. At the beginning of each time interval a block of data is received. Depending on the cryptosystem being used, this block may be regarded as:
   (a) A string of bits of constant length, in case of a symmetric cryptosystem.
   (b) A nonnegative integer smaller than some given modulus, in case of an asymmetric cryptosystem.
4. No block received can be stored in plaintext form. Therefore, each input block must be processed as soon as it arrives. Each output block is then stored in some memory or transmitted over an insecure channel.
5. An encrypted block is to be produced as output at the end of each time interval (with possibly an initial delay before the first output is produced).
6. **Computational Assumptions**. The operation of reading a block and that of storing (or transmitting) it require one time unit each. One iteration of the encryption function E requires $\mathcal{T} - 2$ time units. Depending on whether a symmetric or asymmetric cryptosystem is used, this assumption has the following implications:
   (a) *Symmetric cryptosystem.* Recall that an iteration consists of a constant number r of rounds. Since an iteration requires $\mathcal{T} - 2$ time units, only r rounds are performed in one interval.
   (b) *Asymmetric cryptosystem.* Computing $C_{i-1}^{e_i} \bmod m$ requires on the order of $\log_2 e_i$ time units, $1 \leq i \leq n$, since exponentiation can be performed through squaring and multiplication. Again, since one iteration requires $\mathcal{T} - 2$ time units, the value of e_i that can be used within an interval is bounded from above by a constant.
7. **Cryptographic Assumptions**. One iteration of the cryptographic function E (whether symmetric or asymmetric) is breakable without knowledge of any cryptographic key used. Specifically, an opponent can with reasonable computational effort recover M from C_1 by inverting E, that is, by computing

$$M = E^{-1}(C_1).$$

On the other hand, without knowledge of the encryption/decryption keys, n iterations of the encryption function E (whether symmetric or asymmetric) are unbreakable with current mathematical knowledge and present (and foreseeable) computers. Specifically, given C_n, an opponent cannot feasibly recover M.

We now present two implementations of this computation, the first sequential and the second parallel.

4.2 Sequential Solution

Suppose that the computer system receiving the real-time input is a sequential one, that is, there is a single processor in charge of reading each successive block, encrypting it, and finally storing (or transmitting) it. Because a block needs to be processed as soon as it is received, the computer must have finished processing a block by the time the following block arrives. Also, since an interval of $\mathcal{T}$ time units separates consecutive blocks, only one iteration of the encryption function E can be performed on a block before the latter is stored or transmitted. Specifically,

1. If a symmetric cryptosystem is used, then the sequential computer performs r substitution-transposition rounds on a block within an interval.
2. If an asymmetric cryptosystem is used, then the sequential computer performs

$$C_1 = M^e \bmod m_s$$

   where m_s is the sequential modulus and $e \leq 2^{\mathcal{T}-2}$, since $\log_2 e$ cannot exceed $\mathcal{T} - 2$.

In either case, if the plaintext consists of w blocks, the sequential computer requires $w\mathcal{T}$ time units to encrypt all blocks.

4.3 Parallel Solution

In this section we consider the case in which the computer system receiving the real-time input is a parallel one. Naturally, when a pipeline computer (with n processors) is used to implement real-time encryption, processor P_1 is in charge of reading each successive input block, while processor P_n is responsible for storing (or transmitting) the corresponding (encrypted) output block. As observed in the sequential implementation, because a new input block needs to be processed as soon as it is received, the computer must have finished processing a block when the next block arrives. Therefore, again as in the sequential implementation, since a new input block is received every $\mathcal{T}$ time units, processor P_1 can perform only one iteration of the encryption function E on each block it receives. However, unlike the sequential implementation, the parallel implementation allows further iterations to be performed. Thus, when P_1 has executed one encryption iteration on some block M, it sends the resulting encrypted block C_1 to P_2, and turns its attention to the next incoming plaintext block. Now P_2 can execute a second encryption iteration on C_1, before sending the resulting block C_2 to P_3. This continues until C_n emerges from P_n. Meanwhile, $n-1$ other blocks reside in the pipeline (one in each of the other processors) at various stages of encryption. One time interval after P_n has produced its first encrypted block, it produces a second, and so on, so that an encrypted block is stored or transmitted every $\mathcal{T}$ time units. If there are w blocks in all, the final encrypted block is stored or transmitted by P_n

$$n\mathcal{T} + (w-1)\mathcal{T}$$

time units after the first plaintext block arrives at P_1.

Each input plaintext block is therefore subjected to n encryption iterations. Specifically,

1. If a symmetric cryptosystem is used, then the parallel computer performs rn substitution-transposition rounds on a block.
2. If an asymmetric cryptosystem is used, then processor P_i of the parallel computer performs

$$C_i = C_{i-1}^e \bmod m_p,$$

for $1 \leq i \leq n$, where $C_0 = M$, $C_n = C$, $e \leq 2^{\mathcal{T}-2}$, and m_p is the parallel modulus, with $m_p > m_s$. In other words, $C = M^{e^n} \bmod m_p$.

Note that, in the absence of real-time deadlines, the same computation would require $wn\mathcal{T}$ time units sequentially.

4.4 Analysis

By our initial assumptions, the sequential implementation provides a level of encryption that is effectively breakable, while the parallel implementation provides a level of encryption that is unbreakable for all practical purposes. It is therefore possible to say that the parallel solution to the real-time encryption problem is *infinitely* better than the sequential one.

For a quantitative analysis, we introduce the following parameters. We define the *insecurity value* V, $0 \leq V \leq 1$, to be a measure of the likelihood that a cryptosystem can be broken. Similarly, let the *security value*, $1-V$ be a quantity that expresses the level of security offered by a cryptosystem. For an unconditionally secure cryptosystem, $V = 0$, and the security value is 1. At the other extreme, a cryptosystem that is guaranteed to be breakable has $V = 1$ and a security value of 0. The majority of cryptosystems have a security value between 0 and 1.

Suppose that two implementations of a cryptosystem have insecurity values V_1 and V_2, respectively, where $V_1 > V_2$. The *improvement in security* provided by the second implementation is given as $(1-V_2)/(1-V_1)$.

In the context of our discussion, we define V as follows. Let x be the number of iterations of the encryption function E performed by a certain implementation of a given cryptosystem. Then, for this implementation, $V = 1/x$. The sequential implementation of Section 4.2 executes one iteration of E, and consequently its insecurity value V_s is 1. For the parallel implementation of Section 4.3, the number of iterations is n, resulting in an insecurity value V_p of $1/n$. For large n, V_p approaches 0. Hence, the improvement in security provided by the parallel implementation over the sequential one is $(1 - V_p)/(1 - V_s)$. For $n > 1$ this improvement is unbounded.

5 NUMERICAL COMPUTATION

A further class of computational problems is now identified in which parallelism provides better (in addition to faster) solutions. Specifically, we study the class of *numerical computations.* In this context, a solution is 'better' if it is 'more accurate'. In order to convey the idea in the most straightforward way, two simple problems in numerical computation are chosen for illustration, namely, computing definite integrals and finding roots of nonlinear equations. Specifically, we show that when these problems are to be solved in a real-time environment, a solution obtained by a parallel computer is significantly more accurate than one derived sequentially [8].

We begin by defining numerical computation along with the notion of *error* and its application to the two problems chosen for illustration. Real-time numerical computation is the subject of Section 5.1. Sequential and parallel solutions and their analyses are presented in Sections 5.2, 5.3, and 5.4, respectively.

One of the oldest and most important uses of computers is to perform numerical calculations, primarily in scientific and engineering applications. In what follows, the characteristics of numerical computation are first outlined. A definition of numerical error is then provided. Finally, two examples of numerical problems are used for illustration.

Characteristics. Numerical problems, whether they occur in weather prediction or the design of a high-speed train, share a number of common properties that distinguish them from other types of computations:

1. Because they typically involve physical quantities, their data are represented using *floating-point* numbers.
2. Their solutions are obtained using *mathematical* algorithms.
3. Their algorithms often consist of a number of *iterations*: Each iteration is based on the result of the previous one and is supposed, theoretically, to improve on it. Sometimes, the algorithm performs a *discretization*: A computation on a continuous function is transformed into a discrete operation.
4. Generally, the results produced by numerical algorithms are *approximations* of exact answers that may or may not be possible to obtain.
5. There is an almost inevitable element of *error* involved in numerical computations: *Roundoff errors* (which arise when infinite precision real numbers are stored in a memory location of fixed size), *truncation errors* (which arise when

an infinite computation is approximated by a finite one), and *discretization errors* (when operations on discrete values replace computations on continuous functions).

Examples of numerical problems include solving systems of equations, computing eigenvalues, and performing polynomial interpolations [29, 39, 51, 55, 63].

Numerical Error. By properties 4 and 5, a numerical algorithm only computes an approximation of the true answer to a problem, and this answer therefore contains a certain amount of error. Let the exact answer to a problem be A_{exact} and the approximate answer obtained numerically be $A_{\text{approximate}}$. Then, the *absolute numerical error* E_{absolute} in $A_{\text{approximate}}$ is defined as

$$E_{\text{absolute}} = A_{\text{exact}} - A_{\text{approximate}},$$

while the *relative numerical error* E_{relative} is

$$E_{\text{relative}} = \frac{E_{\text{absolute}}}{A_{\text{exact}}}.$$

When analyzing a numerical algorithm it is customary to derive an estimate of the error (absolute or relative). Usually, this estimate is in the form of an upper bound on the absolute value of the numerical error. Quite often, this bound for an absolute error takes the form

$$|A_{\text{exact}} - A_{\text{approximate}}| \leq \frac{K}{g(N)},$$

where K is a constant that depends on the problem at hand, N is a parameter of the algorithm (such as, for example, the number of iterations or the number of discretization steps), and $g(N)$ is an increasing function of N.

Numerical Integration. Given a function f of a real variable x, defined over an interval $[a, b]$ of values of x, it is required to compute the definite integral

$$I_{\text{exact}} = \int_a^b f(x)dx.$$

For example, consider the function $f(x) = e^{-x^{\sin x}}$. In this case, and for most non-trivial values of f, computing I_{exact} is very difficult analytically. Instead, numerical algorithms are used to compute an approximation. One such algorithm is the *trapezoidal method.* In it, the function f is replaced with a piecewise linear function that approximates it over $[a, b]$. Let $h = (b - a)/N$, for some $N \geq 1$. The interval $[a, b]$ on the x-axis is divided into N subintervals, such that $x_1 = a$, $x_{N+1} = b$, and $x_i = a + (i - 1)h$, for $i = 2, 3, \ldots, N$. Thus,

$$I_{\text{approximate}} = \frac{h}{2}\left(f(a) + 2\sum_{i=2}^{N} f(x_i) + f(b)\right).$$

Now, assuming that f'', the second derivative of f with respect to x, is continuous over $[a, b]$, it can be shown that

$$|I_{\text{exact}} - I_{\text{approximate}}| \leq \frac{(b-a)^3 D}{12N^2},$$

where $D > 0$ and $|f''(x)| \leq D$, for all x in $[a, b]$.

Finding Roots of Nonlinear Equations. It is often required to find the *root* of an equation of one variable, such as $e^x - cosx = 0$. This is usually impossible to do analytically, and one must resort to a numerical algorithm in order to obtain an approximate solution. One such algorithm is the *bisection method.*

Suppose that $f(x)$ is a continuous function, with a and b two values of the variable x such that $f(a)f(b) < 0$. A *zero* of f, that is, a value x_{exact} for which $f(x_{\text{exact}}) = 0$, is guaranteed to exist in the interval $[a, b]$. Let $a_1 = a$ and $b_1 = b$. Now the interval $[a_1, b_1]$ is *bisected*, that is, its middle point $m_1 = (a_1 + b_1)/2$ is computed. If $f(a_1)f(m_1) < 0$, then x_{exact} must lie in the interval $[a_2, b_2] = [a_1, m_1]$; otherwise, it lies in the interval $[a_2, b_2] = [m_1, b_1]$. The process is now repeated on the interval $[a_2, b_2]$. This continues until an acceptable approximation $x_{\text{approximate}}$ of x_{exact} is obtained, that is, until for some $N \geq 1$, $|b_N - a_N| < \alpha$, where α is a small positive number chosen such that the desired accuracy is obtained. When the latter condition is satisfied, $x_{\text{approximate}} = (a_N + b_N)/2$. Because

$$|x_{\text{exact}} - x_{\text{approximate}}| \leq \frac{|b_N - a_N|}{2} \leq \frac{|b_{N-1} - a_{N-1}|}{2^2} \leq \cdots \leq \frac{|b_1 - a_1|}{2^N},$$

the absolute error bound is

$$|x_{\text{exact}} - x_{\text{approximate}}| \leq \frac{|b-a|}{2^N}.$$

5.1 Real-Time Numerical Computation

In this section we define a general numerical problem that needs to be solved in a real-time setting.

A computational environment subject to the following conditions is assumed:

1. A computer system receives a stream of inputs in real time. These inputs represent the data of a numerical computation.
2. Time is divided into intervals. Each interval is $n + 2$ time units long, where n is a positive integer.
3. At the beginning of each time interval, a set S of data is received by the computer system. The set S represents the data to some numerical computation whose output is $A_{\text{approximate}}$. For example, for the problems of numerical integration and finding roots of nonlinear equations, the set S contains the specific function $f(x)$ and the values a and b.

4. It is required that S be processed as soon as it is received and that $A_{\text{approximate}}$ be produced as output as soon as it is computed. Furthermore, one output must be produced at the end of each time interval (with possibly an initial delay before the first output is produced).
5. **Computational Assumptions**. We assume that:
   (a) The operation of reading S, and that of producing $A_{\text{approximate}}$ as output once it has been computed, require one time unit each.
   (b) In computing $A_{\text{approximate}}$, the numerical algorithm performs n iterations (if it is an iterative method) or n discretization steps (if it is a discretization method) in n time units. Hereafter, we refer to iterations and discretization steps as *passes*.
6. **Error Bound Assumption**. Let A_{exact} be the exact answer to the problem at hand. For this problem and the algorithm used to solve it

$$|A_{\text{exact}} - A_{\text{approximate}}| \leq \frac{K}{g(N)},$$

   where K is a constant that depends on the problem, N is a parameter of the algorithm (that is, N is the number of passes), and $g(N)$ is an increasing function of N.

5.2 Sequential Solution

We begin by presenting a solution which assumes that the computer system receiving the real-time data is a sequential one. Here, there is a single processor whose task is to read each incoming S, to compute $A_{\text{approximate}}$, and to produce the latter as output. Recall that the computational environment we assumed dictates that a new input set be received at the beginning of each time interval, and that such a set be processed immediately upon arrival. Therefore, the processor must have finished processing a set before the next one arrives. Since one interval is $n + 2$ time units long, it follows that the algorithm can perform no more than n passes on each input S.

5.3 Parallel Solution

Our second solution to the real-time numerical computation assumes that the computer system is a parallel one. When solving the problem of Section 5.1 on the n-processor computer of Section 2.2.2, it is evident that processor P_1 must be designated to receive the successive input sets S, while it is the responsibility of P_n to produce $A_{\text{approximate}}$ as output. As pointed out in Section 5.2, the fact that each set needs to be processed as soon as it is received implies that the processor must be finished processing a set before the next one arrives. Since a new set is received every $n + 2$ time units, processor P_1 can perform only n passes on each set it receives. Unlike the sequential solution of Section 5.2, however, the present algorithm can perform additional passes. This is done as follows. Once P_1 has executed its n passes on S, it sends the intermediate results, along with S, to P_2, and turns its attention to the next input set. Now P_2 can execute n passes before sending the results (along with S) to P_3. This continues until $A_{\text{approximate}}$ is produced as output by P_n. Meanwhile, $n - 1$ other input sets co-exist in the pipeline (one set in each of P_1, P_2, …, P_{n-1}), at various stages of processing. One time interval after P_n has produced its first $A_{\text{approximate}}$, it produces a second, and so on, so that an output emerges from the pipeline every $n+2$ time units. Note that each output $A_{\text{approximate}}$ is the result of applying n^2 passes to the input set S, since there are n processors and each executes n passes.

5.4 Analysis

Our purpose is to show that a parallel computer can obtain a solution that is *better*, in other words, *more accurate*, than one obtained by a sequential computer. Therefore, this analysis focuses, not on the reduction in the running time, but rather on the reduction in the size of the error, achieved through parallelism. In what follows we derive a bound on the size of the error in $A_{\text{approximate}}$ for the sequential and parallel solutions. For definiteness, we use the two numerical computations introduced at the beginning of this section.

5.4.1 The Trapezoidal Method

Here, the numerical problem to be solved is to compute the definite integral of a given function $f(x)$ from $x = a$ to $x = b$. We have $g(N) = N^2$ (and $K = (b - a)^3 D/12$). The sequential computer performs n passes, that is, it divides the interval $[a, b]$ into n subintervals to compute $I_{\text{approximate}}$, and hence $N = n$. Consequently,

$$|I_{\text{exact}} - I_{\text{approximate}}| \leq \frac{K}{n^2}.$$

By contrast, the parallel computer performs n^2 passes, that is, it divides the interval $[a, b]$ into n^2 subintervals. Each processor in the pipeline computes the definite integral over n consecutive subintervals. Specifically, with $h = (b - a)/n^2$, P_1 computes

$$I_1 = \frac{h}{2}\left(f(a) + 2\sum_{i=2}^{n} f(x_i) + f(x_{n+1})\right)$$

and sends it to P_2 along with f, a, and b. Now P_2 computes

$$I_2 = I_1 + \frac{h}{2}\left(f(x_{n+1}) + 2\sum_{i=n+2}^{2n} f(x_i) + f(x_{2n+1})\right)$$

and sends it to P_3 along with f, a, and b. This continues until P_n computes

$$I_n = \sum_{i=1}^{n-1} I_i + \frac{h}{2}\left(f(x_{(n-1)n+1}) + 2\sum_{i=(n-1)n+2}^{n^2} f(x_i) + f(b)\right)$$

and produces it as $I_{\text{approximate}}$. Therefore, with $N = n^2$,

$$|I_{\text{exact}} - I_{\text{approximate}}| \leq \frac{K}{n^4}.$$

It follows that, in the worst case, the error in the solution obtained in parallel with n processors is n^2 times smaller than the error in the solution obtained sequentially.

5.4.2 The Bisection Method

The numerical problem to be solved here is to find a zero for a continuous function $f(x)$ that falls between $x = a$ and $x = b$. In this case, $g(N) = 2^N$ (and $K = |b-a|$). Sequentially, n passes of the bisection method are performed to obtain $x_{\text{approximate}}$, that is, $N = n$, and

$$|x_{\text{exact}} - x_{\text{approximate}}| \leq \frac{K}{2^n}.$$

In parallel, each processor in the pipeline performs n passes of the bisection method. Specifically, P_1 performs n passes and sends (a_n, b_n) to P_2 along with f. The latter performs n additional passes and sends (a_{2n}, b_{2n}) to P_3 along with f. Eventually, P_n performs the final n passes and obtains $x_{\text{approximate}}$ as $(a_{n^2} + b_{n^2})/2$. Therefore, $N = n^2$, and

$$|x_{\text{exact}} - x_{\text{approximate}}| \leq \frac{K}{2^{n^2}}.$$

The ratio of the sequential error to the parallel error in this case is $2^{n(n-1)}$. In other words, increasing the number of processors by a factor of n leads to a reduction in the size of the error by a factor on the order of 2^{n^2}.

6 CONCLUSION

Parallelism was invented in order to speed up computations. Today, the principal purpose for using parallel computers remains the execution of computations that require an inacceptably long time when performed sequentially. The overwhelming majority of theoretical and empirical analyses of parallel algorithms use the speedup provided by these algorithms as a measure of their goodness. Speedup is usually defined as the ratio of the time required by the best sequential algorithm solving the problem at hand to the time required by the parallel algorithm being evaluated. Here, *time* refers to *worst-case time* and is typically a function of the size of the problem. It is also customary to express the number of processors used by a parallel algorithm as a function of the size of the problem. For these reasons, speedup has been traditionally evaluated in terms of its relation to the number of processors. Thus, a speedup may be sublinear, linear, or superlinear in the number of processors.

Another justification for using parallel computers, however, and one that is important in its own right, turns out to be a by-product of their speed. In this paper we articulated the thesis that other measures of the goodness of parallel algorithms may be employed. In particular, one such measure proposed here is the *quality* of the solution obtained by a parallel algorithm. Thus, we set out to address the following question originally asked in [1]: Can a parallel computer not only reduce the amount of time required to solve a problem sequentially, but also improve the quality of the solution obtained by a sequential computer? It was shown in this paper that for certain computational problems in a real-time environment, the answer is definitely affirmative. Parallel computers can often solve computational problems faster, while at the same time delivering solutions that are better than is possible sequentially. For many of these computations, the ratio of the quality of the solution obtained by the parallel algorithm to the quality of the solution obtained by the best possible sequential algorithm grows arbitrarily large in the worst case. In particular, there exist problems for which an n-fold increase in the number of processors typically results in a solution that improves the one computed sequentially by a factor exponential in n.

These results suggest that, while on the surface the main purpose of parallelism is to speed up computation, a closer look reveals that there is more to it than meets the eye. Clearly, in each case where a superlinear improvement in the quality of the solution is observed, the underlying cause for the phenomenon is *not* the fact that the parallel algorithm is *faster* than the sequential one. Rather, it is the existence of several processors working in parallel. Furthermore, the net effect (that is, the observed phenomenon itself) is entirely distinct from speedup. Indeed, for the examples studied in this paper, the speedup achieved by the parallel approach is far from spectacular. It should also be pointed out that the parallel algorithms of this paper would not have succeeded in obtaining solutions of such quality to the problems they tackled had the time between data arrivals been smaller or had fewer than the required number of processors been available.

Additional examples of real-time optimization problems and their parallel solutions can be easily developed along the same lines outlined in this paper. These include, for example, problems that call for the computation of shortest paths, maximum-sum subsequences, and minimum-weight matchings and spanning trees [6]. Furthermore, for the class of NP-hard problems, several real-time approximation algorithms have been proposed (for a survey, see [36]). However, all of these algorithms are sequential, and none of the problems states either the rate at which data are to be received or the rate at which results are to be produced. Developing parallel real-time approximation algorithms for NP-hard problems, that also take into account the input and output rates, appears to be a worthwhile prospect.

In Section 4, a real-time cryptographic problem is presented for which the parallel solution is significantly better than one computed sequentially. The following remarks are in order regarding the problem and its two solutions:

1. The cryptographic assumption made in Section 4.1 is that one iteration of the encryption function E is readily breakable, while n iterations are effectively unbreakable. Clearly, this is an abstraction that allows for a more general treatment, while simplifying the presentation and subsequent analysis. It is important to note here that, in practice, one iteration of E is likely to provide some form of security. Furthermore, this level of security increases with subsequent iterations. These considerations can be easily incorporated in the analysis, essentially without changing the results of this paper.
2. A related observation concerns the quantitative analysis of Section 4.4. There, we define the *security value* of a cryptographic implementation as $1-V$, where V is the inverse of the number of iterations x of the encryption function. This measure has the advantage of being intuitive, while at the same time leading to a more comprehensive coverage of cases. Evidently, many other measures (more closely tied to specific cryptosystems) are possible. For example, some cryptosystems show a threshold behavior: If fewer than n iterations of encryption are applied to the input data, the system is breakable, whereas n iterations or more render it practically secure. In this case, $V = 1$ for $x < n$, and $V = 0$ for $x \geq n$. Here, the improvement in security is infinite. As another example, note that the complexity of brute-force attacks on cryptosystems typically grows exponentially with the number of iterations, if we assume that every iteration increases the key length by a fixed amount. In this case we may take, for instance, $V = 1/2^x$. Further, let us define the security value here as $1/V$. It follows that $(1/V_p)/(1/V_s) = 2^{n-1}$, implying that the improvement grows exponentially as n increases. It is apparent that each of these measures strengthens the results of Section 4.4.
3. The computational problem studied in this paper can be generalized in the following way. Consider a communication system in which a certain transformation operation is to be applied on the data before transmission. The various forms of coding, such as source coding, error correction coding, and of course encryption, are examples of such a transformation operation. Moreover, the "goodness" of the transformation can be measured. Thus, for example,

   (a) A source coding algorithm is "better" if it yields a higher compression rate,
   (b) An error correction code is "better" if it provides a superior error correction capability, and
   (c) A cryptosystem is "better" if it affords more security.

   In addition, the following characteristic holds: The quality of the transformation increases with more processing applied to the data. For any of these computational problems, therefore, a parallel approach will lead to a "better" solution *in a real-time environment* than one obtained sequentially.

It is shown in Section 5 that the accuracy of a solution to a numerical problem can be increased through the use of parallelism. Numerical computations that present themselves naturally here are numerical integration and finding roots of nonlinear equations. These computations were used to show that the ratio of the numerical error in the solution obtained sequentially to the numerical error in the solution computed in parallel is superlinear in the number of processors used on the parallel computer. In the same vein, it would be interesting to discover other instances of numerical problems within the real-time paradigm for which solutions obtained in parallel are of better quality than those computed sequentially. Candidate numerical computations for this purpose are polynomial interpolation and power series manipulation, in which the data arrive in real time.

The real-time computations studied in this paper differ from the one described in [6] in the following way. In the real-time optimization problem of [6], the purpose is to compute at each time interval the spanning tree of least possible weight for the current graph. At each time interval a new vertex and its associated edges are added to the graph, and must be incorporated in the solution obtained so far. Thus, each output depends on all previous inputs. By contrast, in the problems of Sections 3.1, 4.1, and 5.1, each time interval procures an entirely new problem to be solved. It follows that each output is entirely independent of any previous input. It may be useful to find real-time problems for the application areas exemplified in this paper where each output depends on previous inputs in a nontrivial way. It is especially interesting to note in this regard that for the nonlinear feedback functions of Example 3.2 the ratio of the exact solution obtained in parallel to the approximate solution obtained sequentially is arbitrarily large in the worst case. This effect would certainly be magnified if the output of each computation were to be used as the input to the next computation (namely, if the maximum of $x_1^j, x_2^j, \ldots, x_n^j$ were to serve as the initial value x_0^{j+1}).

Another paradigm of real-time computation occurs when *corrections* to the existing data arrive on line and must be incorporated in the solution to the problem at hand [16, 46]. Within this framework, an interesting case arises in connection with the minimum-weight spanning tree (MST) problem when corrections to the weights of the edges currently in the MST are received in real time and must be taken into consideration. Sequential and parallel algorithms for this problem are described in [21, 28, 54]. However, while these algorithms update the MST as required, their analyses (much like those of the algorithms in [36]) do not allow for the corrections to arrive, or for the results to be produced, at a certain specified rate. Here too an open avenue for research suggests itself quite naturally.

As pointed out in Section 1.3, many computational problems are *inherently parallel*: If the *available* number of processors is smaller than the number of processors *required* to solve one of these problems (even if the difference is *one* processor), then the running time of the parallel algorithm is no better than that of the best sequential algorithm for the same problem [1]. Some problems, by contrast, are believed to be *inherently sequential*: No efficient parallel algorithm is known for solving any of these problems [31]. Real-time computation allows a different look at (apparently) inherently sequential problems. Suppose that a problem can be solved optimally in n (consecutive) time units. Further, let a new such problem be received by some computer system every time unit. The computer system is to process each new problem as soon as it arrives and produce its solution no later than n time units after receiving the problem. (These conditions are not unlike those established in Sections 3, 4, and 5.) The parallel pipeline computer of Section 2.2.2 uses n processors to solve m such problems in $(m-1)+n$ time units. After an initial delay of n time units, an answer is produced every time unit. The parallel computer, therefore, meets the requirements of the problem. Furthermore, these computations (supposed to be inherently sequential) now seem to require constant time. On the other hand, it is clear (and paradoxical) that a sequential computer is hopelessly inadequate to solve these problems.

To date, only one computational paradigm, namely, real-time computation, has been identified, in which parallel computers obtain better solutions faster. Within this paradigm, three problem areas manifesting this phenomenon have, so far, been recognized, namely, *optimization*, *cryptography*, and *numerical computation*. As noted in this paper, there are many ways to measure the quality of a solution. Thus, one solution is 'better' than another if it is 'closer to optimal' (in optimization), 'more secure' (in cryptography), and 'more accurate' (in numerical computation). Other areas of computation bring different meanings to the word 'better', and real-time parallel computation may have a role to play therein. Two such areas mentioned earlier in this section are source coding and error correction. A third example that comes to mind is *statistics*. Here, a better statistical measure may be one based on a larger sample size. Consider, for example, a computer that receives data in real time and must keep track of the average of all inputs received so far, and report such average. A sequential computer can only incorporate a subset of the data received at each time interval when computing the new average. By contrast, a parallel computer may be able to include most, if not all, of the received data. The average reported by the parallel computer at the end of each time interval is better than that obtained sequentially.

Other areas beside real-time computation need to be explored for further measures to evaluate parallel algorithms. As mentioned in the previous paragraph, every example of a computation where a parallel computer provides a better solution than a sequential one, has occurred within the real-time paradigm. Clearly, it would appear especially relevant to determine whether other paradigms of computation exist in which this phenomenon manifests itself. A candidate paradigm of this sort is one in which the data needed by an algorithm can be acquired from one of several sources. Each source holds a set of inputs sufficient by itself to solve the problem at hand. The inputs held by one particular source lead to a solution that is 'better' than any solution reached by using data from another source. At any given time, a single processor can acquire data from exactly one source. Furthermore, a source that is not selected for providing input to the algorithm ceases to exist (and its data can no longer be retrieved). In this paradigm, a sequential computer can find the best solution with probability $1/n$, where $n \geq 1$ is the number of sources. A parallel computer with n processors, on the other hand, assigns one processor to each source, and is therefore guaranteed to arrive at the best solution.

A variant to the paradigm described in the previous paragraph is one in which *all* sources need to be monitored simultaneously in order to obtain the best solution. Here, using a parallel computer with as many processors as there are sources (namely, n) is the only guarantee of success. This remains true even if — contrary to the standard assumption articulated at the end of Section 1 — we allowed the sequential computer to use a processor that is n times faster than each of the processors on the parallel computer. When $n = 2$, a colorful illustration of the paradigm is the *pursuit and evasion on a ring* example presented in [1]. In this version, an entity A is in pursuit of another entity B on the circumference of a circle, such that A and B move at the same speed; clearly, A *never* catches B. Now, suppose that two entities C and D are in pursuit of entity B on the circumference of a circle. Each of C and D moves at $1/k$ the speed of A (and B), where k is a positive integer larger than 1. In this case, C and D *always* catch B. The present paradigm is another instance of inherently parallel problems in which it is the *parallelism* offered by the parallel computer that matters, rather than its speed [18]. Do other computational paradigms exist in which it is possible for parallel computers to obtain better solutions to computational problems than sequential ones?

It may be interesting to conclude by going back full circle and returning to the starting point of this discussion, namely, *speedup*. Suppose that, for a given problem, the best (possible, or known) sequential algorithm runs in time T_1. Further, let some parallel algorithm using p processors run in time T_p when solving the same problem. Then, in this case, speedup is defined as T_1/T_p. In every one of the examples discovered so far, in which a parallel computer with n processors provides a better solution than one obtained sequentially, the ratio of the sequential running time to the parallel running time has been at best linear in n. Thus,

1. In Section 3, an n-processor parallel computer obtains the exact maximum of the sequence x_1^j, x_2^j, ..., x_n^j, and requires on the order of n time units. Had no real-time deadlines been imposed, the same computation would have required on the order of n time units sequentially.
2. Similarly, in Section 4, an n-processor parallel computer encrypts each of w data blocks using n iterations of an encryption function. This requires on the order of $w + n$ time units. The same computation (assuming no real-time deadlines are imposed) would have required on the order of wn time units sequentially.
3. The algorithm of Section 5 solves real-time numerical problem using an n-processor parallel computer. If w input sets S are received, the number of time units required is $n(n + 2) + (w - 1)(n + 2)$. In the absence of real-time deadlines, the same computation requires on the order of wn^2 time units.

By contrast, the improvement in the quality of the solution in each case is superlinear in n. Recent work, however, has demonstrated that *superlinear speedups* are indeed possible, particularly in the real-time environment [1, 2, 9, 14, 15, 16, 45, 46, 47]. It is therefore tempting to ask: Can a superlinear speedup and a superlinear improvement in quality be achieved simultaneously? Would a model of parallel computation more powerful than the one used in this paper be required?

REFERENCES

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

[2] Akl, S. G.: Secure File Transfer: A Computational Analog to the Furniture Moving Paradigm. Proceedings of the Conference on Parallel and Distributed Computing and Systems, Cambridge, Massachusetts, November 1999, pp. 227–233.

[3] Akl, S. G.: Nonlinearity, Maximization, and Parallel Real-Time Computation. Proceedings of the Twelfth Conference on Parallel and Distributed Computing and Systems, Las Vegas, Nevada, November 2000, pp. 31–36.

[4] Akl, S. G.: The Design of Efficient Parallel Algorithms. In Handbook on Parallel and Distributed Processing, J. Blazewicz, K. Ecker, B. Plateau, and D. Trystram, Eds., Springer Verlag, Berlin, 2000, pp. 13–91.

[5] Akl, S. G.—Barnard, D. T.—Doran, R. J.: Design, Analysis and Implementation of a Parallel Tree Search Algorithm. IEEE Transactions on Machine Analysis and Artificial Intelligence, Vol. 4, 1982, No. 2, 1982, pp. 192–203.

[6] Akl, S. G.—Bruda, S. D.: Parallel Real-Time Optimization: Beyond Speedup. Parallel Processing Letters, Vol. 9, 1999, No. 4, pp. 499–509.

[7] Akl, S. G.—Bruda, S. D.: Parallel Real-Time Cryptography: Beyond Speedup II. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, June 2000, pp. 1283–1289.

[8] Akl, S. G.—Bruda, S. D.: Parallel Real-Time Numerical Computation: Beyond Speedup III. International Journal of Computers and their Applications, Special Issue on High Performance Computing Systems, Vol. 7, 2000, No. 1, pp. 31–38.

[9] Akl, S. G.—Fava Lindon, L.: Paradigms Admitting Superunitary Behaviour in Parallel Computation. Parallel Algorithms and Applications, Vol. 11, 1997, pp. 129–153.

[10] Bestavros, A.—Fay-Wolfe, V. Eds.: Real-Time Database and Information Systems. Kluwer Academic Publishers, Boston, 1997.

[11] Boxer, L.—Miller, R.: Dynamic Computational Geometry on Meshes and Hypercubes. Journal of Supercomputing, Vol. 3, 1989, pp. 161–191.

[12] Boxer, L.—Miller, R.: Parallel Dynamic Computational Geometry. Journal of New Generation Computer Systems, Vol. 2, 1989, No. 3, pp. 227–246.

[13] Brassard, G.—Bratley, P.: Algorithmics: Theory and Practice. Prentice Hall, Englewood Cliffs, New Jersey, 1998.

[14] Bruda, S. D.—Akl, S. G.: On the Data-Accumulating Paradigm. Proceedings of the Fourth International Conference on Computer Science and Informatics, Research Triangle Park, North Carolina, October 1998, pp. 150–153.

[15] Bruda, S. D.—Akl, S. G.: The Characterization of Data-Accumulating Algorithms. Proceedings of the International Parallel Processing Symposium, San Juan, Puerto Rico, April 1999, pp. 2–6.

[16] Bruda, S. D.—Akl, S. G.: A Case Study in Real-Time Parallel Computation: Correcting Algorithms, to Appear. In Journal of Parallel and Distributed Computing.

[17] Bruda, S. D.—Akl, S. G.: Towards a Meaningful Formal Definition of Real-Time Computations. Proceedings of the Fifteenth International Conference on Computers and Their Applications, New Orleans, Louisiana, March 2000, pp. 274–279.

[18] Bruda, S. D.—Akl, S. G.: On the Necessity of Formal Models for Real-Time Parallel Computations. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, June 2000, pp. 1291–1297.

[19] Casti, J.: Complexification: Explaining a Paradoxical World through the Science of Surprise, HarperCollins, New York, 1994.

[20] Chaudhuri, P.: Finding and Updating Depth First Spanning Trees of Acyclic Digraphs in Parallel. The Computer Journal, Vol. 33, 1990, pp. 247–251.

[21] Chaudhuri, P.: Parallel Algorithms: Design and Analysis. Prentice Hall, Sydney, Australia, 1992.

[22] Chaudhuri, P.: Parallel Incremental Algorithms for Analyzing Activity Networks. Parallel Algorithms and Applications, Vol. 13, 1998, No. 2, pp. 153–165.

[23] Chin, F. Y.—Houck, D.: Algorithms for Updating Minimum Spanning Trees. Journal of Computer and System Sciences, Vol. 16, 1978, pp. 333–344.

[24] Cohen, J.—Stewart, I.: The Collapse of Chaos: Discovering Simplicity in a Complex World. Viking, New York, 1994.

[25] Cormen, T. H.—Leiserson, C. E.—Rivest, R. L.: Introduction to Algorithms. McGraw-Hill, New York, 1990.

[26] Coveny, P.—Highfield, R.: Frontiers of Complexity: The Search for Order in a Chaotic World. Fawcett Columbine, New York, 1995.

[27] Even, S.—Shiloach, Y.: An On-Line Edge Deletion Problem. Journal of the ACM, Vol. 28, 1982, pp. 1–4.

[28] Frederickson, G.: Data Structures for On-Line Updating of Minimum Spanning Trees. Proceedings of the ACM Symposium on Theory of Computing, Boston, Massachusetts, April 1983, pp. 252–257.

[29] Gerald, C. F.: Applied Numerical Analysis. Addison Wesley, Reading, Massachusetts, 1978.

[30] Gleick, J.: Chaos: Making a New Science. Penguin Books, New York, 1987.

[31] Greenlaw, R.—Hoover, H. J.—Ruzzo, W. L.: Limits to Parallel Computation. Oxford University Press, New York, 1995.

[32] Harel, D.: Algorithmics: The Spirit of Computing. Addison Wesley, Reading, Massachusetts, 1987.

[33] Havill, J. T.—Mao, W.: On-Line Algorithms for Hybrid Flow Shop Scheduling. Proceedings of the Fourth International Conference on Computer Science and Informatics, Research Triangle Park, North Carolina, October 1998, pp. 134–137.

[34] Horgan, J.: The End of Science. Broadway Books, New York, 1996.

[35] Ibaraki, T.—Katoh, N.: On-Line Computation of Transitive Closure Graphs. Information Processing Letters, Vol. 16, 1983, pp. 95–97.

[36] Irani, S.—Karlin, A. R.: Online Computation. In: D. S. Hochbaum, Ed., Approximation Algorithms for NP-Hard Problem, International Thomson Publishing, Boston, Massachusetts, 1997, pp. 521–564.

[37] JáJá, J.: An Introduction to Parallel Algorithms. Addison Wesley, Reading, Massachusetts, 1992.

[38] Jung, H.—Mehlhorn, K.: Parallel Algorithms for Computing Maximal Independent Sets in Trees and for Updating Minimum Spanning Trees. Information Processing Letters, Vol. 27, 1988, pp. 227–236.

[39] King, J. T.: Introduction to Numerical Computation. McGraw-Hill, New York, 1984.

[40] Knuth, D. E.: The Art of Computer Programming, Vol. 1, Fundamental Algorithms. Addison-Wesley, Reading, Massachusetts, 1975.

[41] Lawler, E. L.: Combinatorial Optimization: Networks and Matroids. Holt, Rinehart & Winston, New York, 1976.

[42] Lawson, H. W.: Parallel Processing in Industrial Real-Time Applications. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[43] Levy, S.: Artificial Life: A Report from the Frontier Where Computers Meet Biology. Vintage, New York, 1992.

[44] Lewin, R.: Complexity: Life at the Edge of Chaos. Macmillan, New York, 1992.

[45] Luccio, F.—Pagli, L.: The p-Shovelers Problem (Computing with Time-Varying Data). Proceedings of the Fourth Symposium on Parallel and Distributed Computing, Arlington, Texas, December 1992, pp. 188–193.

[46] Luccio, F.—Pagli, L.: Computing with Time-Varying Data: Sequential Complexity and Parallel Speed-Up. Theory of Computing Systems, Vol. 31, 1998, No. 1, pp. 5–26.

[47] Luccio, F.—Pagli, L.—Pucci, G.: Three Non Conventional Paradigms of Parallel Computation. Lecture Notes in Computer Science, 678, 1992, pp. 166–175.

[48] Menezes, A. J.—van Oorschot, P. C.—Vanstone, S. A.: Handbook of Applied Cryptography. CRC Press, Boca Raton, Florida, 1996.

[49] Miltersen, P. B.—Subramanian, S.—Vitter, J. S.—Tamassia, R.: Complexity Models for Incremental Computation. Theoretical computer Science, 130, 1994, pp. 203–236.

[50] Newborn, M. M.: Kasparov Versus Deep Blue: Computer Chess Comes of Age. Springer-Verlag, New York, 1996.

[51] Ortega, J. M.: Numerical Analysis. Academic Press, New York, 1972.

[52] Papadimitriou, C. H.—Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Prentice Hall, Englewood Cliffs, New Jersey, 1982.

[53] Pawagi, S.: A Parallel Algorithm for Multiple Updates of Minimum Spanning Trees. Proceedings of the International Conference on Parallel Processing, St. Charles, Illinois, August 1989, Vol. III, pp. 9–15.

[54] Pawagi, S.—Ramakrishnan, I. V.: An $O(\log n)$ Algorithm for Parallel Update of Minimum Spanning Trees. Information Processing Letters, Vol. 22, 1986, pp. 223–229.

[55] Ralston, A.—Rabinowitz, P.: A First Course in Numerical Analysis. McGraw-Hill, New York, 1978.

[56] Rawlins, G. J. E.: Compared to What? An Introduction to the Analysis of Algorithms. W. H. Freeman, New York, 1992.

[57] Reif, J. H.: Synthesis of Parallel Algorithms. Morgan Kaufmann, San Mateo, California, 1993.

[58] Schneier, B.: Applied Cryptography: Protocols, Algorithms, and Source Code in C. John Wiley & Sons, New York, 1995.

[59] Sherlekar, D. D.—Pawagi, S.—Ramakrishnan, I. V.: $O(1)$ Parallel Time Incremental Graph Algorithms. Lecture Notes in Computer Science, 206, 1985, pp. 477–493.

[60] Simmons, G. J.: Contemporary Cryptology: The Science of Information Integrity. IEEE Press, Piscataway, New Jersey, 1992.

[61] Smith, J. R.: The Design and Analysis of Parallel Algorithms. Oxford University Press, New York, 1993.

[62] Spira, P. M.—Pan, A.: On Finding and Updating Spanning Trees and Shortest Paths. SIAM Journal on Computing, Vol. 4, 1975, No. 3, pp. 375–380.

[63] Stewart, G. W.: Introduction to Matrix Computations. Academic Press, New York, 1973.

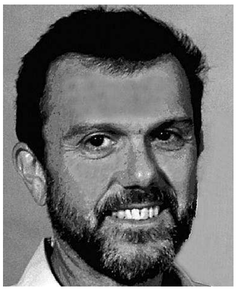
[64] Stinson, D. R.: Cryptography: Theory and Practice. CRC Press, Boca Raton, Florida, 1996.

[65] Thorin, M.: Real-Time Transaction Processing. Macmillan, London, 1992.

[66] Tsin, Y. H.: On Handling Vertex Deletion in Updating Minimum Spanning Trees. Information Processing Letters, 27, 1988, pp. 167–168.

[67] Varman, P.—Doshi, K.: A Parallel Vertex Insertion Algorithm for Minimum Spanning Trees. Lecture Notes in Computer Science, 226, 1986, pp. 424–433.

[68] Waldrop, M. M.: Complexity: The Emerging Science at the Edge of Order and Chaos. Simon and Schuster, New York, 1992.



Selim G. Akl is a professor of computing at Queen's University, Kingston, Ontario, Canada. His research interests are in parallel computation. He is author of Parallel Sorting Algorithms (Academic Press, 1985), The Design and Analysis of Parallel Algorithms (Prentice Hall, 1989), and Parallel Computation: Models and Methods (Prentice Hall, 1997). He is also a co-author of Parallel Computational Geometry (Prentice Hall, 1992), and an editor of Computational Geometry (Elsevier), Parallel Processing Letters (World Scientific Publishing), and Parallel Algorithms and Applications (Gordon and Breach).