Technical Report No. 99-423

Parallel Real-Time Cryptography: Beyond Speedup II *

Selim G. Akl and Stefan D. Bruda
Department of Computing and Information Science
Queen's University
Kingston, Ontario K7L 3N6
Canada
Email: {akl,bruda}@cs.queensu.ca

August 10, 1999

Abstract

The primary purpose of parallel computation is the fast execution of computational tasks that are too slow to perform sequentially. However, it was shown recently that a second equally important motivation for using parallel computers exists: Within the paradigm of real-time computation, some classes of problems have the property that a solution to a problem in the class computed in parallel is better than the one obtained on a sequential computer. What constitutes a better solution depends on the problem under consideration. Thus, for optimization problems, 'better' means 'closer to optimal'. The present paper continues this line of inquiry by exploring another class enjoying the aforementioned property, namely, cryptographic problems in a real-time setting. In this class, 'better' means 'more secure'. A real-time cryptographic problem is presented for which the parallel solution is significantly better than a sequential one.

Key words and phrases: Parallelism, real-time computation, cryptography.

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

1 Introduction

Parallelism was invented in order to speed up time-consuming computations. 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. A *parallel computer*, by contrast, possesses *several* processors. These processors, working in parallel, execute *several* steps of an algorithm *simultaneously*. Usually, this results in a significant reduction in the time required to solve the problem at hand [1, 2, 6, 9, 10, 11, 23, 26, 27, 33, 37].

There exists, however, a second equally important motivation for using parallel computers. A parallel computer can in some circumstances obtain a solution to a problem that is *better* than that obtained by a sequential computer. A computational paradigm in which this phenomenon is manifested is real-time computation. Consider the following example.

Example. 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. (The standard definition of *time unit* is used throughout the paper. Specifically, 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. This is the unit used to measure the running time of an algorithm [1, 8, 16, 23].)

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 [3]. ■

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 [29]. 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.

Real-time optimization provided the first example of a problem for which a solution obtained in parallel is *always* better than one obtained sequentially [4]. The real-time weighted spanning tree problem is described in [4] as follows:

1. The minimum-weight spanning tree (MST) of an undirected, connected, and weighted graph is given; it consists of n vertices and $n-1$ edges.

2. Time is divided into intervals of cn^{ϵ} time units, where $c > 0$ and $0 < \epsilon < 1$.

3. A new vertex and its associated edges are received at the beginning of every time interval. A new MST, or a best approximation possible to it, incorporating the new data must now be computed; the new tree is produced as output at the beginning of the next time interval.

It is shown in [4] that for this minimization problem the ratio of the weight of a solution obtained sequentially to the weight of a solution obtained in parallel can be arbitrarily large.

The purpose of this paper is to continue the study begun in [4]. We present a problem from real-time cryptography for which a parallel solution is consistently better than a sequential solution. In [4], the notion of 'better' meant 'closer to optimal'. In this paper, '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.

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 analysis in this paper assumes that all models of computation are the fastest possible (within the bounds of theoretical physics). Specifically, no machine exists that is faster than the sequential computer of Section 4.1, and similarly no parallel computer exists whose processors are faster than the processors of the parallel computer of Section 4.2.1. 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).

The remainder of the paper is organized as follows. The real-time computational paradigm is introduced in Section 2. Section 3 provides a brief description of some basic notions from the field of cryptography. The problem to be solved is defined in Section 4, along with sequential and parallel solutions. Section 5 offers some concluding remarks.

2 The Real Time Paradigm

In the conventional paradigm of computation, an algorithm solves a problem by operating on a set of inputs all of which are available at the outset. By contrast, in the real time computational paradigm the data needed by an algorithm are not all given initially. Instead, inputs to a problem arrive, typically at regular intervals, while the algorithm is in the process of computing a solution. The newly arrived data must be incorporated in the solution at hand. The final solution is to be returned by a certain deadline.

Real-time computations form a subclass of a larger class of problems known variably as *on-line*, *incremental*, *dynamic*, and *updating* computations [7, 12, 13, 14, 17, 18, 19, 20, 21, 22, 24, 25, 30, 31, 32, 36, 38, 40, 41]. What distinguishes a real-time problem from problems in the larger class is the presence of *deadlines*—by which the input is to be processed, by which the output is to be produced, and so on.

3 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.

3.1 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) [28, 34, 35, 39]. 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.

3.2 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 [28, 34, 35, 39].

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 [28, 34, 35, 39].

4 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 $\mathcal{T}+2$ 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 $\mathcal{T}$ 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 $\mathcal{T}$ 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 $\mathcal{T}$ 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.

4.1 Sequential implementation

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 be finished processing a block by the time the following block arrives. Also, since an interval of $\mathcal{T}+2$ 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 $e \leq 2^{\mathcal{T}}$, since $\log_2 e$ cannot exceed $\mathcal{T}$.

In either case, if the plaintext consists of w blocks, the sequential computer requires $w(\mathcal{T}+2)$ time units to encrypt all blocks.

4.2 Parallel real-time cryptography

In this section we consider the case in which the computer system receiving the real-time input is a parallel one. Our chosen model of parallel computation is first presented, followed by the parallel implementation.

4.2.1 Parallel model

The parallel computer consists of n processors, denoted by P_1, P_2, ..., P_n, as illustrated in Fig. 1. The processors 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.

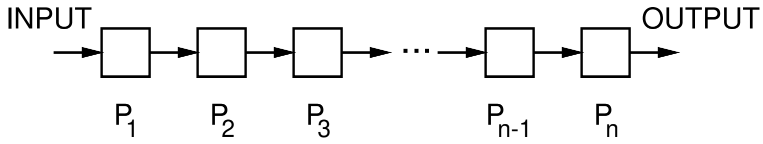

Figure 1: Parallel computer.

This parallel computer operates in a *pipeline* fashion: Data travel from P_1 to P_n, with P_i beginning to operate only when it receives input, $1 \leq i \leq n$. Among all models of parallel computation that provide some form of communication among their processors, the pipeline model is arguably the weakest. Yet, despite the severely limited communication paths it affords, this model is entirely adequate for our purposes. Indeed, we show in the following section that the pipeline model allows an implementation of real-time cryptography that is significantly better than a sequential one.

4.2.2 Parallel implementation

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 $\mathcal{T}+2$ 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 $\mathcal{T}+2$ time units. If there are w blocks in all, the final encrypted block is stored or transmitted by P_n

$$n(\mathcal{T}+2) + (w-1)(\mathcal{T}+2)$$

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 $C_0 = M$, $C_n = C$, $e \leq 2^{\mathcal{T}}$, 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 $w(n\mathcal{T}+2)$ time units sequentially.

4.3 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 V_1/V_2.

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 4.1 executes one iteration of E, and consequently its insecurity value V_s is 1. For the parallel implementation of Section 4.2, 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 V_s/V_p. This improvement grows without bound as n increases.

5 Conclusion

The principal purpose for using parallel computers remains the speeding up of computations that are too slow when performed sequentially. 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. As it is beginning to become apparent, parallel computers can often solve computational problems faster, while at the same time delivering solutions that are better than is possible sequentially.

To date, only one computational paradigm, namely, real-time computation, has been identified, in which parallel computers obtain better solutions faster. Within this paradigm, two problem areas manifesting this phenomenon have, so far, been recognized. These two areas are *optimization* (as shown in [4]) and *cryptography* (the subject of this paper). A third area that holds promise is *numerical computation*. Here the parallel computer can obtain a solution to a numerical problem with a higher degree of accuracy than a sequential computer [5].

We conclude with an open problem: Do other computational paradigms exist, besides real-time computation, in which it is possible for parallel computers to obtain better solutions to computational problems than sequential ones?

References

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

[2] S. G. Akl, Secure File Transfer: A Computational Analog to the Furniture Moving Paradigm, Technical Report No. 99-422, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, March 1999.

[3] S.G. Akl, D.T. Barnard, and R.J. Doran, Design, analysis and implementation of a parallel tree search algorithm, *IEEE Transactions on Machine Analysis and Artificial Intelligence*, 4, 1982, 192–203.

[4] S. G. Akl and S. D. Bruda, Parallel real-time optimization: Beyond speedup, Technical Report No. 99-421, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, January 1999, to appear in *Parallel Processing Letters*.

[5] S. G. Akl and S. D. Bruda, Parallel real-time numerical computation: Beyond speedup III, Technical Report No. 99-424, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, May 1999.

[6] S. G. Akl and L. Fava Lindon, Paradigms admitting superunitary behaviour in parallel computation, *Parallel Algorithms and Applications*, 11, 1997, 129–153.

[7] L. Boxer and R. Miller, Dynamic computational geometry on meshes and hypercubes, *Journal of Supercomputing*, 3, 1989, 161–191.

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

[9] S. D. Bruda and S. G. Akl, On the data-accumulating paradigm, *Proceedings of the Fourth International Conference on Computer Science and Informatics*, Research Triangle Park, North Carolina, October 1998, 150–153.

[10] S. D. Bruda and S. G. Akl, The characterization of data-accumulating algorithms, *Proceedings of the International Parallel Processing Symposium*, San Juan, Puerto Rico, April 1999, 2–6.

[11] S. D. Bruda and S. G. Akl, A case study in real-time parallel computation: Correcting algorithms, Technical Report No. 98-420, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, December 1998.

[12] P. Chaudhuri, Finding and updating depth first spanning trees of acyclic digraphs in parallel, *The Computer Journal*, 33, 1990, 247–251.

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

[14] P. Chaudhuri, Parallel incremental algorithms for analyzing activity networks, *Parallel Algorithms and Applications*, 13(2), 1998, 153–165.

[15] F.Y. Chin and D. Houck, Algorithms for updating minimum spanning trees, *Journal of Computer and System Sciences*, 16, 1978, 333–344.

[16] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, McGraw-Hill, New York, 1990.

[17] S. Even and Y. Shiloach, An on-line edge deletion problem, *Journal of the ACM*, 28, 1982, 1–4.

[18] G. Frederickson, Data structures for on-line updating of minimum spanning trees, *Proceedings of the ACM Symposium on Theory of Computing*, Boston, Massachusetts, April 1983, 252–257.

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

[20] J.T. Havill and W.Mao, 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, 134–137.

[21] T. Ibaraki and N. Katoh, On-line computation of transitive closure graphs, *Information Processing Letters*, 16, 1983, 95–97.

[22] S. Irani and A.R. Karlin, Online computation, in: D.S. Hochbaum, Ed., *Approximation Algorithms for NP-Hard Problems*, International Thomson Publishing, Boston, Massachusetts, 1997, 521–564.

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

[24] H. Jung and K. Mehlhorn, Parallel algorithms for computing maximal independent sets in trees and for updating minimum spanning trees, *Information Processing Letters*, 27, 1988, 227–236.

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

[26] F. Luccio and L. Pagli, Computing with time-varying data: Sequential complexity and parallel speed-up, *Theory of Computing Systems*, 31(1), 1998, 5–26.

[27] F. Luccio, L. Pagli, and G. Pucci, Three non conventional paradigms of parallel computation, *Lecture Notes in Computer Science*, 678, 1992, 166–175.

[28] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, Boca Raton, Florida, 1996.

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

[30] S. Pawagi, 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, 9–15.

[31] S. Pawagi and I.V. Ramakrishnan, An $O(\log n)$ algorithm for parallel update of minimum spanning trees, *Information Processing Letters*, 22, 1986, 223–229.

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

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

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

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

[36] D.D. Sherlekar, S. Pawagi, and I.V. Ramakrishnan, $O(1)$ parallel time incremental graph algorithms, *Lecture Notes in Computer Science*, 206, 1985, 477–493.

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

[38] P.M. Spira and A. Pan, On finding and updating spanning trees and shortest paths, *SIAM Journal on Computing*, 4(3), 1975, 375–380.

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

[40] Y.H. Tsin, On handling vertex deletion in updating minimum spanning trees, *Information Processing Letters*, 27, 1988, 167–168.

[41] P. Varman and K. Doshi, A parallel vertex insertion algorithm for minimum spanning trees, *Lecture Notes in Computer Science*, 226, 1986, 424–433.