Technical Report No. 99-422

# Secure File Transfer: A Computational Analog to the Furniture Moving Paradigm *

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

March 10, 1999

**Abstract**

One of the most compelling illustrations of the power of parallelism is the furniture-moving paradigm. In it, a large item of furniture needs to be moved from one place to another. A single mover, working alone, must take the item apart, move each piece separately, and then reassemble the item at the new location, taking a long time to complete the job. By contrast, four movers can simply lift the item and quickly move it to its new location. Thus, the time required to accomplish the task is reduced by a factor significantly larger than four.

This paper describes a computational analog to the furniture-moving paradigm. The computation in question is concerned with transferring a computer file from one computer system to another over an insecure communications channel. The file contains private or sensitive information whose secrecy and integrity need to be maintained. Cryptography is used to obtain a digital signature of the file, thereby protecting its integrity, and then encrypting it to ensure its secrecy. If the file transfer is performed sequentially, the file and its digital signature need to be broken into blocks, each of which is signed and encrypted individually then transmitted. At the receiving end, each block is checked for authenticity, then the original file is reassembled and its digital signature verified. On the other hand, performing the file transfer in parallel allows the entire file and its digital signature to be sent as a whole in one step. Consequently, the parallel solution speeds up the sequential one by a factor that is superlinear in the number of processors used.

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

# 1 Introduction

The *furniture-moving paradigm* is described in [1] as follows:

> "[A] large piece of furniture [..] needs to be moved from one place to another. One mover working alone is unable to lift, push, or drag the item and, in order to move it, must take it apart, transport each of the parts individually, and then put them back together at the indicated spot. The job requires one hour. On the other hand, four movers working together can simply lift the piece of furniture and put it in its new location in 15 seconds."

The furniture-moving paradigm clearly illustrates the power of parallelism in everyday life: Certain tasks are faster to accomplish if done by more than one person. However, the point of the paradigm, as presented in [1] (and even earlier, in a slightly different form, in [12]), goes beyond this obvious observation. Indeed, it is generally believed that four people should finish a job in *at best* 1/4 of the time required by one person. Yet in the furniture-moving example, the four workers complete the task much faster than the 15 minutes predicted by common sense!

In the theory of parallel computation there is also a belief that mirrors the conventional wisdom of everyday experience. It goes as follows: If $p$ processors are put to a computation, they can complete it in *at best* $1/p$ of the time required by one processor. This is known as the 'speedup theorem'. The motivation in [1] behind the furniture-moving paradigm was to suggest that it may be possible to contradict the speedup theorem of parallel computation. Certain computations are described in [1] that achieve this. Thanks to a phenomenon called *parallel synergy*, these computations are performed by $p$ processors in much less than $1/p$ of the time they take using one processor. However, none of the computations described in [1] and displaying parallel synergy is a true analog to the furniture-moving paradigm.

The purpose of the present paper is to propose a computation that captures the essence of the furniture-moving paradigm. It concerns transferring a file securely from one computer system to another. The file contains some sensitive (or private) information. It is required to safeguard the *secrecy* of the information while in transit: A wiretapper must face a difficult job when attempting to read it. Furthermore, the *integrity* of the file is to be protected: Any tampering with its contents should be hard to conceal. Sequential and parallel solutions to this problem are presented. We show that the parallel solution has a running time significantly smaller than that prescribed by the speedup theorem. It should be emphasized here that the proposed computation is an analog to the furniture-moving paradigm not because of the apparent similarity between the two tasks suggested by the fact that they both involve 'displacing' an object. Instead, the analogy stems from their common property of being able to be carried out quickly if and only if performed *simultaneously* by several *agents* (that is, movers in one case and processors in the other).

The remainder of this paper is organized as follows. Some background material is presented in Section 2. This includes a definition of the speedup theorem and a brief introduction to the field of cryptography whose techniques are used by the algorithms in this paper. Section 3 states the file transfer problem to be solved. Sequential and parallel solutions are described in Sections 4 and 5, respectively, along with their analyses. A discussion of the superlinear speedup achieved by the parallel solution over the sequential one is provided in Section 6. Some concluding remarks are offered in Section 7.

# 2 Background

This section gives some background to the two main ideas used in this paper, namely, speedup and cryptography.

## 2.1 Speedup

Speeding up the sequential solutions to computational problems is the principal motivation behind parallel processing. In order to determine the goodness of a parallel algorithm that solves a certain problem, a measure known as *speedup* is used. Speedup is defined as the ratio of the time $T_1$ required by the best sequential algorithm for solving the problem at hand, to the time $T_p$ required by the $p$-processor parallel algorithm being evaluated, where $p > 1$. Denoting the speedup by $\mathit{speedup}(1,p)$, we have:

$$\mathit{speedup}(1,p) = \frac{T_1}{T_p}.$$

It is widely believed that the speedup achieved by a parallel algorithm using $p$ processors over a sequential algorithm is at most equal to $p$ [5, 10, 14, 19, 28]. This belief is usually called the 'speedup theorem' and is stated as:

$$\mathit{speedup}(1,p) \leq p.$$

One can view the above inequality as 'bad news', since it puts an upper bound on the amount of speedup possible with $p$ processors. Most traditional computations (such as sorting, searching, operating on matrices, and so on) when executed in parallel using $p$ processors exhibit a speedup of at most $p$ (or some linear function of $p$), thus obeying (the spirit if not the letter of) the 'speedup theorem'.

Another largely accepted concept in parallel computation is the so-called 'slowdown theorem' (also known as *Brent's principle*) [6, 11, 13, 17, 25]. Let a computation be performed with $p$ processors in time $T_p$ and with $q$ processors, $1 \leq q < p$, using the same algorithm, in time $T_q$, where $T_p < T_q$. The slowdown experienced is defined as:

$$\mathit{slowdown}(q,p) = \frac{T_q}{T_p}.$$

The 'slowdown theorem' states that slowdown is at most the ratio of $p$ to $q$; thus:

$$\mathit{slowdown}(q,p) \leq \frac{p}{q}.$$

The above inequality is in some sense 'good news', as it puts an upper bound on how much slower a computation runs when only $q$ instead of $p$ processors are available. The 'slowdown theorem' is clearly a general form of the 'speedup theorem'. Most traditional computations satisfy the 'slowdown theorem'.

Over the last few years, however, a number of unconventional, yet realistic, paradigms have been advanced which contradict one or both of the 'speedup theorem' and the 'slowdown theorem'. Specifically, these computations have at least one of the following properties:

1. $\mathit{speedup}(1,p)$ is superlinear in $p$; thus, for example, the speedup is on the order of $p^r$, or even $r^p$, for some $r > 1$.

2. $\mathit{slowdown}(q,p)$ is superlinear in $p/q$; thus, for example, the slowdown is on the order of $(p/q)^r$, or even $r^{p/q}$, for some $r > 1$.

These results are described in [1, 4, 7, 8, 9, 20, 21]. They suggest that for some computations it is possible to obtain a speedup that is asymptotically larger than the number of processors used (in other words, the previous bad news are now replaced with good news). Furthermore, if the necessary number of processors is not available then a slowdown is incurred that is asymptotically larger than the processor ratio (in other words, the previous good news are now replaced with bad news). In a nutshell, these results imply that certain computations are *inherently parallel*. One such computation, not previously described, is proposed in this paper. It is based on cryptography to which we now turn.

## 2.2 Cryptography

Modern *cryptography* is a branch of knowledge that combines the methods of computer science, mathematics, and electrical engineering. Its purpose is the protection of the secrecy and/or integrity of digital information that is stored in the memory of a device or is traveling on an insecure communications channel. A cryptographic scheme (also known as a *cryptosystem*) works as follows. Suppose that $M$ is a meaningful string of bits; $M$ is called the *plaintext*. With the help of a cryptographic key $k$, an *encryption* function transforms $M$ into another string $C$, which for all practical purposes now appears totally meaningless; $C$ is called the *ciphertext*. Using a cryptographic key $k'$, a *decryption* function allows $M$ to be recovered from $C$. Note that if $k = k'$, the cryptographic scheme is said to be *symmetric*; otherwise, it is *asymmetric*. Usually, $C$ is substituted for $M$, thus concealing the information that $M$ contained. In other circumstances, $C$ is obtained by encrypting a compressed version of $M$; now $C$ accompanies $M$, thus serving as its *digital signature* and protecting its integrity (an opponent cannot modify $M$ without affecting $C$). Most often, $M$ is both encrypted and signed.

The crucial property here is that $C$ should be computationally hard to obtain from $M$ without knowledge of $k$, and that $M$ should be computationally hard to obtain from $C$ without knowledge of $k'$. This ensures that private or sensitive information is kept secret and/or that a digital signature cannot be forged. For clear introductions to the techniques of modern cryptography, see [22, 26, 27, 29].

Our subsequent treatment makes use of both symmetric and asymmetric cryptographic schemes. It should be noted, however, that we do not specify exactly which encryption and decryption functions are used. Such a specification would give the false impression that our results are tied to particular functions. On the other hand, we do specify the computational requirements of such functions since our analysis is concerned with speedup. Therefore, any choice of encryption and decryption functions satisfying the stated conditions would be acceptable.

# 3 Secure File Transfer

This section presents the computational problem proposed as an analog to the furniture-moving paradigm.

## 3.1 Preliminaries

In the memory of a computer system is stored a block $Q$ of $2nb$ bits. Here, both $n$ and $b$ are positive integers and $b$ is a constant. The block $Q$ consists of the following components:

1. A file $F$ of $nb$ bits containing some sensitive information (this could be text, data, or programs).
2. A cryptographic encryption key $k_1$ of $(n-1)b$ bits used to compute a digital signature.
3. A cryptographic decryption key $k_2$ of $b/2$ bits used to verify a digital signature.
4. A digital signature $S$ of $b/2$ bits.

The signature $S$ is used to verify the authenticity of the file $F$. It is computed as follows:

1. The $(2n-1)b$ bits formed by the concatenation of $F$ and $k_1$ are first compressed into a block $H$ of $b/2$ bits. This is obtained by computing the Exclusive-OR of the $2(2n-1)$ $(b/2)$-bit blocks of $F$ and $k_1$. Note here that the $i$th bit of $H$ is the Exclusive-OR of all $i$th bits of the $2(2n-1)$ blocks, $1 \leq i \leq b/2$.

2. An asymmetric cryptographic scheme is now used to obtain $S$ from $H$. Recall that such as scheme uses distinct keys for encryption and decryption, respectively. Let the encryption function be denoted by $E^a$, where $E$ stands for *encryption* and the superscript $a$ refers to *asymmetric*. The encryption key $k_1$ is used in conjunction with $H$ to obtain $S$ using $E^a$; thus:

$$E^a(k_1, H) = S.$$

When the authenticity of $F$ and $k_1$ is to be verified, this is done as follows:

1. The $(2n-1)b$ bits of $F$ and $k_1$ are compressed into a block of $b/2$ bits (by computing the Exclusive-OR of the $2(2n-1)$ $(b/2)$-bit blocks of $F$ and $k_1$). Let the resulting $(b/2)$-bit block be denoted by $H'$.

2. The decryption component of the asymmetric cryptographic scheme is now used. Let the decryption function be denoted by $D^a$, where $D$ stands for *decryption* and the superscript $a$ refers to *asymmetric*. The decryption key $k_2$ is used in conjunction with $S$ to obtain a $(b/2)$-bit block $H''$ using $D^a$; thus:

$$D^a(k_2, S) = H''.$$

3. The file $F$ and the key $k_1$ are recognized as being authentic if and only if $H' = H''$.

Some relevant points are worth noting here:

1. The two keys $k_1$ (used for signature) and $k_2$ (used for verification) are unique keys generated by the creator and owner of the file $F$. These keys, and only these keys, can be used whenever $F$ (or any part thereof) is to be signed digitally or authenticated.

2. The quadruple $Q = (F, k_1, k_2, S)$ needs to be stored in a fashion that protects its secrecy and integrity. One option is to store $Q$ in a secure location. Another is to store it in encrypted form. For simplicity, we adopt the first option in the present and the next two sections. The second option is discussed in Section 7.

3. The number of bits in a file, key, or signature is chosen to simplify the presentation and has no particular meaning in itself. However, the analysis does require $b$ to be a constant and $n$ a variable. It is also important that $k_1$ be significantly longer than $k_2$. This underscores the fact that computing $S$ from $H$ with $k_1$ should be computationally demanding. As a consequence, obtaining $S$ from $H$ *without* $k_1$, and similarly $H$ from $S$ *without* $k_2$, are computationally hard tasks, making it difficult to commit fraud. By contrast, computing $H$ from $S$ with $k_2$ should be extremely easy computationally, since it is used for signature verification.

4. During the authentication process, to discover that $H' \neq H''$ indicates that $Q$ has been somehow tampered with and should be rejected.

## 3.2 The File Transfer Problem

Throughout its useful lifetime the file $F$ needs to be moved across several computer systems. The communications channel that carries $F$ from one computer to another is considered insecure. Such a channel may be vulnerable to various attacks, in particular:

1. Passive wiretapping, where an enemy may compromise the secrecy of the information through eavesdropping.

2. Active wiretapping, where an enemy may compromise the integrity of the information by modifying it while in transit.

Therefore, when $F$ travels from one computer to another, it is required to safeguard its secrecy and integrity. This is done as follows:

1. Secrecy is protected through encryption before transmission and decryption upon receipt. Here, a symmetric cryptographic scheme is used. Recall that such a scheme uses the same key for encryption and decryption. Let $k_3$ be a $b$-bit secret key shared by the two communicating computers. In other words, $k_3$ is known to both computer systems before the transmission. We denote the encryption function by $E^s$, where $E$ stands for *encryption* and the superscript $s$ refers to *symmetric*. The sending computer encrypts an $mb$-bit message $M$, $m \geq 1$, by applying $E^s$ to it using $k_3$; thus:
$$E^s(k_3, M) = C.$$
The $mb$-bit encrypted message $C$ is now transmitted. The receiving computer decrypts $C$ using a decryption function $D^s$, where $D$ stands for *decryption* and the superscript $s$ refers to *symmetric*. It applies $D^s$ to $C$ using $k_3$ to recover $M$; thus:
$$D^s(k_3, C) = M.$$

2. Integrity is preserved using the functions $E^a$ and $D^a$ and the keys $k_1$ and $k_2$, as explained in Section 3.1.

In what follows we study the computational requirements when moving the file $F$ from the memory of a computer system $A$ to that of a computer system $B$. In doing so, we present two analyses. In the first analysis, both systems $A$ and $B$ are *sequential* (that is, single-processor) computers. In the second analysis, both systems $A$ and $B$ are *parallel* (that is, multi-processor) computers.

### 3.3 Computational Assumptions

We define a *time unit* as the time required to perform a constant-time operation such as addition or comparison of two fixed-size operands, or reading/writing a fixed-size operand from/to memory. Thus, for example, comparing two $(b/2)$-bit blocks for equality requires one time unit. In addition, our analysis makes the following assumptions:

1. Compressing a $w$-bit block into an $x$-bit block, where $1 \leq x < w$ and $w$ is a multiple of $x$, requires $w$ time units.
2. Encrypting or decrypting a $y$-bit block using a $z$-bit key requires $yz$ time units.
3. Moving a $2b$-bit block from one computer to the other requires one time unit.

## 4 Sequential Solution

As mentioned in the previous section, computer systems $A$ and $B$ each have one processor. Computer system $A$ can transmit at most $2b$ bits at a time to computer system $B$. Because each message transmitted must be signed and encrypted, the sender and receiver perform the steps described in what follows.

### 4.1 Computer System A

The quadruple $Q$ is viewed as consisting of $2n$ $b$-bit blocks $Q_i$, $1 \leq i \leq 2n$. Each of these is treated separately; thus:

1. Block $Q_i$ is compressed into a $(b/2)$-bit block $H_i$; this is achieved by computing the Exclusive-OR of the two $(b/2)$-bit blocks of $Q_i$.
2. The $(b/2)$-bit block $H_i$ is signed using $k_1$ to produce a $(b/2)$-bit signature $S_i$ by computing
$$E^a(k_1, H_i) = S_i.$$
3. The $2b$-bit block $M_i$ consisting of $Q_i$, $k_2$, and $S_i$ is now encrypted using $k_3$ to obtain $C_i$ by computing
$$E^s(k_3, M_i) = C_i.$$
4. The $2b$-bit block $C_i$ is now transmitted to computer system $B$.

The previous four steps are performed $2n$ times (once for each $b$-bit block $Q_i$). The time required is therefore

$$2n(b + b^2(n-1)/2 + 2b^2 + 1)$$

time units. It is important to observe here that each message sent to computer system $B$ is transmitted in encrypted form in order to protect its secrecy. Thus, when it leaves computer system $A$, the $2b$-bit ciphertext $C_i$ contains (in encrypted form) a message $Q_i$, the signature $S_i$ required to authenticate $Q_i$, and the key $k_2$ needed to verify $S_i$.

## 4.2 Computer System B

At the receiving end, the following steps are performed on each $2b$-bit block $C_i$ received:

1. Block $C_i$ is decrypted using $k_3$; thus:

$$D^s(k_3, C_i) = M_i.$$

   (This reveals $Q_i$, $k_2$, and $S_i$.)

2. Block $Q_i$ is compressed into a $(b/2)$-bit block $H_i'$.

3. Block $S_i$ is decrypted using $k_2$; thus:

$$D^a(k_2, S_i) = H_i''.$$

4. If $H_i' = H_i''$, then the signature is valid; otherwise, it is not.

Since the preceding four steps are repeated $2n$ times, the time required for this phase is

$$2n(2b^2 + b + b^2/4 + 1)$$

time units.

When all $2nb$ bits of $Q = (F, k_1, k_2, S)$ have been received, the following three steps are performed by computer system $B$:

1. The file $F$ and the key $k_1$ are compressed into a $(b/2)$-bit block $H'$.

2. The signature $S$ is decrypted using $k_2$; thus:

$$D^a(k_2, S) = H''.$$

3. If $H' = H''$, then $F$ and $k_1$ are accepted as authentic; otherwise, they are not.

These three steps run in

$$(2n-1)b + b^2/4 + 1$$

time units.

### 4.3 Sequential Running Time

The sequential solution therefore requires

$$2n(b + b^2(n-1)/2 + 2b^2 + 1) + 2n(2b^2 + b + b^2/4 + 1) + (2n-1)b + b^2/4 + 1,$$

that is, on the order of $\alpha_1 b^2 n^2$ time units, for some positive constant $\alpha_1$.

## 5 Parallel Solution

In this section we present a solution to the file transfer problem that uses several processors operating simultaneously. However, unlike the case with the sequential approach to computation, there is more than one way to organize the processors when addressing a problem from a parallel computing point of view. We therefore begin by defining our chosen model of computation.

### 5.1 Model of Computation

For definiteness we assume in what follows that the model of computation is the Parallel Random Access Machine (PRAM). In this model, a given number of processors execute an algorithm synchronously while sharing a common memory from which they can read and to which they can write. This model is described in detail in [1]. We mention one feature briefly for its relevance to our subsequent treatment. Simultaneous writing by several processors to the same shared memory location, using a Concurrent Write (CW) instruction is allowed by the model. Thus, when several processors write simultaneously to the same memory location $U$, the model requires that the CW instruction indicate what ends up stored in $U$. For example, given several values each sent by one processor, the CW instruction may select one of the values for writing in $U$, or it may select the sum of the values, or their logical AND, and so on. The choice depends on the algorithm being executed and is specified by the algorithm designer. The CW instruction is executed in one time unit.

Let each of computer systems $A$ and $B$ consist of a PRAM with $n$ processors denoted by $P_1$, $P_2$, ..., $P_n$. The quadruple $Q = (F, k_1, k_2, S)$ is viewed as consisting of $n$ $2b$-bit blocks $Q_j$, $1 \leq j \leq n$. Each processor $P_j$ of computer system $A$ can transmit at most $2b$ bits at a time to computer system $B$. However, because all processors can operate in parallel, *all* $n$ $2b$-bit blocks $Q_j$ of $Q$ can be transmitted simultaneously. In order to satisfy the secrecy requirement, each processor $P_j$ encrypts its block $Q_j$ before transmission. The integrity requirement, on the other hand, is satisfied automatically. This is because $Q$ is sent from computer $A$ to computer $B$ as *one* message. Hence, as required, the pair $(F, k_1)$ is accompanied by its signature $S$ as well as the key $k_2$ needed for authentication. The sender and receiver perform the algorithms described below.

### 5.2 Computer System A

With all processors operating in parallel, processor $P_j$, $1 \leq j \leq n$, executes the following two steps:

1. Encrypts the $2b$-bit block $Q_i$ of $Q$ using $k_3$; thus:

$$E^s(k_3, Q_j) = C_j.$$

2. Sends the $2b$-bit block $C_j$ to computer system $B$.

This requires $2b^2 + 1$ time units.

### 5.3 Computer System B

The $n$ $2b$-bit blocks $C_j$, $1 \leq j \leq n$, are received simultaneously. Now, with all processors operating in parallel, each processor $P_j$, $1 \leq j \leq n$, executes the following steps:

1. Decrypts the $2b$-bit block $C_j$ using $k_3$; thus:

$$D^s(k_3, C_j) = Q_j.$$

   (This reveals $Q = (F, k_1, k_2, S)$.)

2. Compresses one $2b$-bit block of the pair $(F, k_1)$ into a $(b/2)$-bit block $q_j$. (Processor $P_n$ compresses a $b$-bit block into a $(b/2)$-bit block, since there $n$ processors and only $(2n-1)b$ bits to compress.)

3. Writes $q_j$ into a memory location $U$ using the instruction Exclusive-OR CW. (Because all processors write in $U$ simultaneously, $U$ now holds a $(b/2)$-bit block $H'$ representing the compressed version of $(F, k_1)$.)

Finally, one processor (for example, $P_1$) executes the following two steps:

1. Decrypts $S$ using $k_2$; thus:

$$D^a(k_2, S) = H''.$$

2. Compares the resulting $(b/2)$-bit block $H''$ to the block $H'$; then $F$ and $k_1$ are accepted as authentic if and only if $H' = H''$.

The previous five steps run in

$$(2b^2 + 2b + 1) + (b^2/4 + 1)$$

time units.

### 5.4 Parallel Running Time

The parallel solution therefore requires

$$(2b^2 + 1) + (2b^2 + 2b + 1) + (b^2/4 + 1)$$

time units. This time is on the order of $\alpha_2 b^2$ time units, for some positive constant $\alpha_2$.

## 6 Superlinear Speedup

The speedup achieved by the parallel solution over the sequential one is therefore:

$$\mathit{speedup}(1, n) = \frac{\alpha_1 b^2 n^2}{\alpha_2 b^2} = \alpha_3 n^2,$$

for some positive constant $\alpha_3$. This speedup is superlinear in $n$, the number of processors used by the parallel solution. Specifically, the speedup in this case is *quadratic* in $n$. This contradicts the 'speedup theorem'. It is also interesting to observe that had the number of processors been any smaller than $n$, the parallel solution of Section 5 would have been impossible. Even $n - 1$ processors would lead to an algorithm with no better running time than the sequential solution described in Section 4. This contradicts the 'slowdown theorem'.

## 7 Conclusion

In this paper we have presented a computational analog to the furniture-moving paradigm. The problem involves a file that needs to be transferred from one computer to another such that its secrecy and integrity are to be preserved. A parallel computer system with multiple processors can transport the file in its entirety in one step and hence meet all the security requirements. A sequential computer system, on the other hand, has but one processor and hence needs to break the file into smaller parts, transmit each part securely, and then reassemble it at the other end. The speedup achieved by the parallel solution over the sequential one is superlinear in the number of processors used.

We conclude with the following remarks:

1. Throughout the paper we have tacitly assumed that all cryptographic functions are safe against *cryptanalytic attack* (that is, the actions of an opponent who wishes to discover the contents of $F$ or compromise its integrity). There is, of course, no such guarantee. Any choice for the encryption and decryption functions $E^s$, $D^s$, $E^a$, and $D^a$ has its weaknesses of which the user must be aware. Similarly, the Exclusive-OR function for computing $H$, selected here for its simplicity, can be replaced with cryptographically stronger compression schemes, still without any guarantee. It follows that when an encrypted and signed message is received, we cannot be sure that its secrecy or integrity have not been compromised. An

eavesdropper may have been able to recover the plaintext, without affecting the transfer. Also, when $H'$ is found to be equal to $H''$, there is no *proof* that they are both equal to the original $H$ computed by the creator of $F$: An enemy may have succeeded in modifying the pair $(F, k_1)$ and its signature $S$ in such a way that the resulting $H'$ and $H''$ remain equal. Only when $H' \neq H''$ are we certain that the integrity of the message has been compromised. All of these considerations, however, are peripheral to the main focus of this paper.

2. Normally, the quadruple $Q = (F, k_1, k_2, S)$ would be initially stored in the memory of computer system $A$ in *encrypted* form. Encryption would have been carried out (at the time file $F$ is created and stored) using a function similar to $E^s$ and a key $k_0$ (similar to $k_3$). Thus, when it is time to send $Q$ to computer system $B$, the quadruple would need to be *decrypted* in preparation for transmission using a function similar to $D^s$ and key $k_0$. Upon receipt in computer system $B$, and following signature verification, the quadruple $Q$ would have to be re-encrypted. (Note that in the sequential solution of Section 4, the key $k_1$ is needed to sign each block $H_i$. When all $2n$ blocks have been transmitted as ciphertext, the plaintext version of $k_1$ should always be erased from the memory of computer system $A$, regardless of whether $Q$ was initially stored in encrypted form or not.) All of these steps are omitted in the solutions presented in Sections 4 and 5 for simplicity of exposition. However, their inclusion would not affect the speedup result of Section 6 in any significant way.

3. In Section 3.1 the encryption key $k_1$ is chosen to be $(n-1)b$ bits in length, while the decryption key $k_2$ is only $b/2$ bits long. The reason given for this choice in Section 3.1 is to make it computationally hard to obtain $S$ from $H$ without $k_1$ and $H$ from $S$ without $k_2$, whereas obtaining $H$ from $S$ with $k_2$ is easy. We note here that the same goal can be reached by taking the two keys $k_1$ and $k_2$ to be of the same length, while making the encryption function $E^a$ significantly more computationally demanding than the decryption function $D^a$. For example, let $F$ be a file of $(2n - 3/2)b$ bits, and suppose that $k_1$, $k_2$, and $H$ are each $b/2$ bits long. These particular lengths are selected for convenience, in order to maintain the number of bits of the quadruple $Q = (F, k_1, k_2, S)$ equal to $2nb$ as previously. If $V$ and $W$ are $x$-bit blocks derived from $F$, then we can take $E^a(k_1, V)$ and $D^a(k_2, W)$ to require $(xb/2)^n$ and $xb/2$ operations, respectively. With this choice, the speedup result of this paper becomes even stronger. Indeed, the sequential and parallel solutions of Sections 4 and 5 now have running times on the order of $\alpha_4 n b^{2n}$ and $\alpha_5 b^2$ time units, respectively, for positive constants $\alpha_4$ and $\alpha_5$. This leads to a speedup on the order of $\alpha_6 n b^{2n-2}$, for some positive constant $\alpha_6$. Such speedup is *exponential* in $n$, the number of processors used in the parallel solution.

4. It may be argued that the CW instruction used in the parallel solution of Section 5 is too powerful, and that perhaps its ability to compute the

Exclusive-OR of $n$ $(b/2)$-bit blocks in one time unit is the reason for the superlinear speedup. Suppose then that the CW instruction is not used. Instead, we assume that the only instruction allowed for writing to memory is Exclusive Write (EW). With the latter, no two processors can write to the same memory location simultaneously: When several processors write to memory, each must write to a distinct memory location. However, it is well known that a CW instruction executed by $n$ processors in one time unit can be simulated by the same number of processors, using the EW instruction only, in $\log n$ time units [17]. It follows that the running time of the parallel solution of Section 5 (with EW replacing CW) is now

$$(2b^2 + 2b + \log n) + (b^2/4 + 1)$$

time units. As a result, the speedup is on the order of $\alpha n^2/\log n$, for some positive constant $\alpha$, which is still superlinear in $n$.

5. It is mentioned in Section 2.1 that computations such as the one described in this paper are said to be inherently parallel. This term is preferred to the expression 'embarrassingly parallel' sometimes used to refer to computations that lend themselves easily to parallelization [30]. There are two reasons for preferring the term *inherently parallel*:

   (a) The expression 'embarrassingly parallel' has a negative connotation that is inappropriate and in fact totally unnecessary (not only for the purposes of this paper, but also in general). If a computation can be executed efficiently in parallel, then there is every reason to celebrate, and nothing to be embarrassed about!

   (b) The computation in this paper is more than just effectively parallelizable. It has two important properties:

      i. The parallel solution leads to a superlinear speedup when the optimal number of processors is used.

      ii. If fewer than the required number of processors are available, then no speedup whatsoever is possible.

6. Another computational paradigm which leads to superlinear speedup is *on-line* (or *real-time*) computation. When a problem is solved on line, not all of its input data are available initially. Data arrive one at a time or in bundles, according to a given data-arrival law, while the solution to the problem is being computed. Whenever a new datum is received, it must be taken into account in updating the solution [15, 16, 18, 24]. The on-line computational paradigm has been used to exhibit superlinear speedup in a variety of contexts [7, 8, 9, 20, 21]. Recently, it was used to demonstrate that parallelism can do more than just speed up computation. It is shown in [3] that for real-time optimization problems, a solution obtained in parallel can be closer to optimal than any solution computed sequentially. Another example is provided by game-playing programs [2, 23]. Consider,

for instance, a program for playing chess in real time (against a human or another program). Suppose that the program runs on a parallel computer and that it is its turn to make a move. Given a fixed amount of time to move, the program can search a game tree much larger than is possible sequentially and hence make a more informed decision.

Today, in many applications of cryptography, both encryption and decryption occur in real time. On-line cryptography, therefore, appears to be an area where parallel computation may prove most profitable. Exploring this possibility promises to be an exciting avenue for future research.

# References


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

[2] 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.

[3] 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.

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

[5] G.S. Almasi and G. Gottlieb, *Highly Parallel Computing*, Benjamin-Cummings, Redwood City, California, 1989.

[6] D.P. Bertsekas and J.N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*, Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] D. Fischer, On superlinear speedups, *Parallel Computing*, 17, 1991, 695–697.

[13] A. Gibbons and W. Rytter, *Efficient Parallel Algorithms*, Cambridge University Press, Cambridge, England, 1988.

[14] G. Golub and J.M. Ortega, *Scientific Computing: An Introduction with Parallel Computing*, Academic Press, San Diego, 1993.

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

[16] 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.

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

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

[19] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing*, Benjamin-Cummings, Menlo Park, California, 1994.

[20] 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.

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

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

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

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

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

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

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

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

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

[30] B. Wilkinson and M. Allen, *Parallel Programming*, Prentice Hall, Upper Saddle River, New Jersey, 1999.