Technical Report No. 99-424

Parallel Real-Time Numerical Computation: Beyond Speedup III *

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

Parallel computers can do more than simply speed up sequential computations. They are capable of finding solutions that are far better in quality than those obtained by sequential computers. This fact is demonstrated by analyzing sequential and parallel solutions to numerical problems in a real-time paradigm. In this setting, numerical data required to solve a problem are received as input by a computer system, at regular intervals. The computer must process its inputs as soon as they arrive. It must also produce its outputs at regular intervals, as soon as they are available. We show that for some real-time numerical problems a parallel computer can deliver a solution that is significantly more accurate than when computed by a sequential computer. Similar results were derived recently in the areas of real-time optimization and real-time cryptography.

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

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

1 Introduction

The range and scope of computer applications in today's society are breathtaking. From business to medicine, from communication to 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, *sequential* (that is 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 use computers with several processors. A *parallel* computer breaks the problem to be solved into smaller parts that are solved simultaneously on its many processors. As a result, the solution time is dramatically reduced [1, 25, 38, 40].

It has been recently observed that in some circumstances parallel computers can solve computational problems, not only *faster*, but also *better*. One such circumstance is 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*. Within this paradigm, what constitutes a better solution depends on the class of computational problems under consideration. For example, for *optimization* problems, 'better' means 'closer to optimal'. Similarly, for *cryptographic* problems, an implementation is 'better' than another if it is 'more secure'. Examples of problems in optimization and cryptography, in which parallel solutions are always better than sequential ones, are described in [4] and [5], respectively.

The purpose of this paper is to identify a further class of computational problems 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.

The remainder of this paper is organized as follows. Previous work is summarized in Section 2. This includes an introduction to real-time computation and a brief review of earlier results achieved when applying parallelism within this paradigm to problems in optimization and cryptography. In Section 3, numerical computation is defined 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 4; here, the main results of the paper are presented. Some final thoughts are given in Section 5.

Throughout the paper, we use the standard definition of *time unit*, that is, the unit used to measure the running time of an algorithm: 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, 9, 17, 25].

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 4.2, and similarly no parallel computer exists whose processors are faster than the processors of the parallel computer of Section 4.3.1. 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).

2 Previous Work

The fact that parallel computers can do more than just speed up computations had been suspected for some time [1, 3]. However, it was one particular paradigm, namely, real-time computation, that provided the appropriate environment for this phenomenon to manifest itself for the first time. It was shown that for two computations, each from a different class of problems, a solution obtained in parallel is arbitrarily better than a sequential one.

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. Usually, the inter-arrival rate is constant.

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 its computation.

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 [21, 28, 37]. (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* [18, 22, 23, 24], *incremental* [14, 15, 32, 39], *dynamic* [7, 8, 44], and *updating* [13, 16, 19, 26, 34, 35, 41, 43].)

2.2 Real-time optimization

The first example of a computation for which a parallel solution is consistently better than a sequential one was provided by real-time optimization. The *real-time weighted spanning tree problem* is defined 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.

2.3 Real-time cryptography

An input source produces blocks of data in real time. Each block arrives at a computer system which needs to encrypt it immediately for security purposes. Once encrypted, the block is sent to an output destination, also in real time. Specifically,

1. Time is divided into intervals of constant duration.
2. An input block is produced by the source at the beginning of each time interval.
3. An encrypted block must be transmitted to the destination at the end of each time interval.
4. The three operations of reading an input block, performing one iteration of an encryption function, and producing the output block, require together one time interval.

5. For an integer n larger than 1, n iterations of the encryption function offer unconditional security, while any number of iterations smaller than n is effectively breakable by an opponent without knowledge of the cryptographic keys used.

A parallel implementation of this scheme is shown in [5] to provide a level of security that is infinitely better than a sequential one.

3 Numerical Computation

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

3.1 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 [20, 27, 33, 36, 42].

3.2 Numerical error

By properties 4 and 5 of Section 3.1, 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.

3.3 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 nontrivial 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 $[a, b]$.

3.4 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

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

4 Real-Time Numerical Computation

In this section we define a general numerical problem that needs to be solved in a real-time setting. Sequential and parallel solutions are described, followed by an analysis.

4.1 Problem definition

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 Sections 3.3 and 3.4 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**.

   (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.

4.2 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.

4.3 Parallel solution

Our second solution to the real-time numerical computation assumes that the computer system is a parallel one. In what follows, we first describe the model of parallel computation used, then present the parallel algorithm.

4.3.1 Model of parallel computation

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

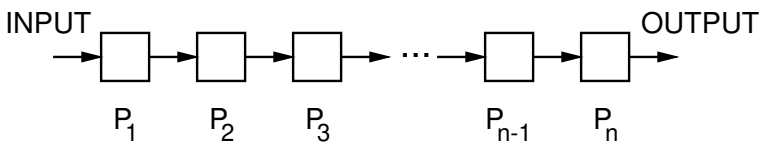

Figure 1: Parallel computer.

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, $1 \leq i \leq n$. 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 paths, is perfectly suitable when solving the real-time numerical problem of Section 4.1. This is demonstrated in the next section where it is shown that the pipeline computer affords a parallel algorithm that is significantly better than a sequential one.

4.3.2 Parallel algorithm

When solving the real-time numerical problem of Section 4.1 on the n-processor parallel computer of Section 4.3.1, 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 4.2, 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 4.2, 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.

4.4 Analysis

As mentioned earlier, the purpose of this paper 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, our 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 of Sections 3.3 and 3.4.

4.4.1 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,

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

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

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

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

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

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

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

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

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

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.

4.4.2 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}.

5 Conclusion

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 numerical problems in a real-time computational environment, the answer is definitely affirmative. In particular, there exists numerical problems for which an n-fold increase in the number of processors results in a solution that is more accurate than one computed sequentially by a factor exponential in n. Similar examples were previously found in optimization [4] and cryptography [5]. 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.

Several lines of investigation present themselves naturally for future work:

1. As noted in this paper and its predecessors [4, 5], 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 parallel computation may have a role to play in such areas. An 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. A parallel computer, on the other hand, 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.

2. The real-time computation of Section 4.1, and to some extent that of [5], differ from the one described in [4] in the following way. In the real-time optimization problem of [4], the purpose is to compute at each time interval the minimum 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 Section 4.1 and [5], 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 worthwhile to find real-time problems in numerical computation and cryptography where each output depends on previous inputs in a non-trivial way.

3. In this paper and previous ones [4, 5], 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.

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 [4], an n-processor parallel computer obtains the exact MST of a graph with n vertices at each time interval, requiring on the order of $\log n$ 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 [5], 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 4.3.2 in this paper, solves the problem of Section 4.1 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, 6, 10, 11, 12, 29, 30, 31]. It is therefore tempting to ask: Can a superlinear speedup and a superlinear improvement in quality be achieved simultaneously?

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 cryptography: Beyond speedup II, Technical Report No. 99-423, Department of Computing and Information Science, Queen's University, Kingston, Ontario, 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] L. Boxer and R. Miller, Parallel dynamic computational geometry, *Journal of New generation Computer Systems*, 2(3), 1989, 227–246.

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

[10] 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.

[11] 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.

[12] 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.

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

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

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

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

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

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

[19] 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.

[20] C. F. Gerald, *Applied Numerical Analysis*, Addison Wesley, Reading, Massachusetts, 1978.

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

[22] 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.

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

[24] 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.

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

[26] 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.

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

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

[29] F. Luccio and L. Pagli, The p-shovelers problem (computing with time-varying data), *Proceedings of the Fourth Symposium on Parallel and Distributed Computing*, Arlington, Texas, December 1992, 188–193.

[30] 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.

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

[32] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia, Complexity models for incremental computation, *Theoretical computer Science*, 130, 1994, 203-236.

[33] J. M. Ortega, *Numerical Analysis*, Academic Press, New York, 1972.

[34] 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.

[35] 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.

[36] A. Ralston and P. Rabinowitz, *A First Course in Numerical Analysis*, McGraw-Hill, New York, 1978.

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

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

[39] 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.

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

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

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

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

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