Technical Report No. 2001-443

Superlinear Performance In Real-Time Parallel Computation*

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

March 24, 2001

Abstract

Can a parallel computer with n processors solve a computational problem more than n times faster than a sequential computer? Can it solve it more than n times better? New computational paradigms offer an affirmative answer to the above questions through concrete examples in which the improvement in speed or quality is superlinear in the number of processors used by the parallel computer. Furthermore, the improvement is consistent and provable. All examples are characterized by the presence of one or several real-time input streams. In one of the examples, an exponential improvement in speed is achieved despite the fact that the processors of the parallel computer are significantly slower than their sequential counterpart. In another example, the improvement in quality is unbounded. A metaphor from everyday life motivates each computational paradigm in which a superlinear improvement in performance is exhibited.

Key words and phrases: Parallelism, superlinear speedup, superlinear quality-up, real-time computation, optimization, cryptography, numerical analysis.

1 INTRODUCTION

Suppose that you are given two computers, call them $\mathcal{S}$ and $\mathcal{P}$. Computer $\mathcal{S}$ is *sequential* and therefore has one processor. Computer $\mathcal{P}$ is *parallel* and is equipped with n processors, where n is an integer larger than 1. For a certain computational problem $\mathcal{O}$ that may be solved on either $\mathcal{S}$ or $\mathcal{P}$, consider the following two questions:

Question 1: Can computer $\mathcal{P}$ solve $\mathcal{O}$ *more than n times faster* than is possible on computer $\mathcal{S}$?

Question 2: Can computer $\mathcal{P}$ solve $\mathcal{O}$ *more than n times better* than is possible on computer $\mathcal{S}$?

Evidently, Question 1 has to do with the *speed* of computation; for example if $\mathcal{S}$ solves a problem in 100 seconds, can $\mathcal{P}$ solve the same problem with 50 processors *in less than* 2 seconds? On the other hand, Question 2 is concerned with the *quality* of the computed solution; for example, if $\mathcal{S}$ compresses a file by a factor of 5% (with no loss of information), can a 4-processor $\mathcal{P}$ compress the same file *by more than* 20% (again with no loss of information)?

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

The purpose of this paper is to address these two questions. We present a number of examples (some old [2, 4, 13, 14, 15], some recent [1], and some new) that allow these two questions to be answered in the affirmative. In each case, the improvement in speed or quality is superlinear in the number of processors used by the parallel computer. Furthermore, the improvement is consistent and provable. By this it is meant that the superlinear reduction in speed or the superlinear increase in quality occurs in every instance of the computational problem under consideration. In addition, this improvement is independent of any discrepancies between the sequential and parallel computers used, as we assume explicitly that the two computers use the same basic processor, have the same amount of total memory, and run the same algorithm. A unifying thread in the treatment is that all examples are characterized by the presence of one or several real-time input streams.

It must be said that, in some sense, the results described here may be viewed as controversial. Indeed, it is conventional wisdom in the parallel computation literature to answer Question 1 in the negative. Question 2, on the other hand, is hardly ever posed: When it comes to the *quality* of a solution, *no* improvement (let alone one that causes the solution to be more than n times better) should be possible through parallel computation. The root of these beliefs is the well-known and well-understood principle of *simulation*. According to this important concept (dating back to the work of Alan Turing [16]), a single-processor computer $\mathcal{S}$ can execute exactly the same computations performed by an n-processor computer $\mathcal{P}$. The idea is to program $\mathcal{S}$ to imitate the actions of each processor of $\mathcal{P}$ *sequentially*. As a result, when solving problem $\mathcal{O}$, computer $\mathcal{S}$ obtains *the same solution* computed by computer $\mathcal{P}$. Furthermore, the time required by $\mathcal{S}$ to arrive at the solution is *no more than n times* the running time of $\mathcal{P}$. Specifically, suppose that the n processors of $\mathcal{P}$ operate synchronously, with each processor executing t steps (in other words, if each step lasts one time unit, then the computation performed by $\mathcal{P}$ requires t time units). Let s_{ij} denote the jth step executed by the ith processor of $\mathcal{P}$, $1 \leq i \leq n$, $1 \leq j \leq t$. In order to simulate this computation, $\mathcal{S}$ proceeds as follows: For each j, $j = 1, 2, \ldots, t$, computer $\mathcal{S}$ executes s_{ij} for all i, $1 \leq i \leq n$. Thus, the computation performed by $\mathcal{S}$ consists of $n \times t$ steps, requiring nt time units.

Recently, a number of computational paradigms have emerged where the very notion of simulation no longer makes sense. Within these paradigms, a parallel computer $\mathcal{P}$ is in fact capable of executing a computation more than n times faster than a sequential computer $\mathcal{S}$. Also, in some circumstances, $\mathcal{P}$ obtains a solution that is more than n times better than that obtained by $\mathcal{S}$. What is taking place in these cases is a manifestation of a phenomenon referred to as *synergy*; we say that the parallel computer, thanks to its many processors, is exhibiting synergistic behavior. The purpose of this paper is to present instances of such computations. The exposition is deliberately informal with the purpose of conveying the main ideas to the non-expert in the most accessible way possible.

The remainder of this paper is organized as follows. Section 2 gives some definitions required to make the subsequent material more precise. The notion of speedup is presented in Section 3, together with computational problems for which parallel computation exhibits superlinear speedup. A new measure of the performance of a parallel computer is introduced in Section 4, namely, the quality-up, and examples of a superlinear improvement in the quality of the solution achieved through parallel computation are described. In order to develop some intuition, all computational paradigms studied in Sections 3 and 4 are motivated by metaphors from everyday life. Section 5 offers some conclusions and suggestions for future research.

2 BACKGROUND

This section provides some background on computational models, superlinear improvement in performance, and real-time computation. Thus, although our results hold for most reasonable models of computation (sequential and parallel), for definiteness we specify in Section 2.1 the models of computation intended for $\mathcal{S}$ and $\mathcal{P}$. Also, we note that our objective here is not merely to show that a parallel computer provides a solution more than n times faster or more than n times better than a sequential one. Instead, we prove that in some circumstances the improvement is a superlinear function of the number of processors used by the parallel computer; therefore, we define the notion of superlinear improvement in Section 2.2. Finally, in Section 2.3, the real-time mode of computation is introduced; this is particularly important as all the computations described in this paper are performed in real time.

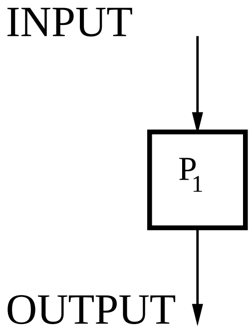

Figure 1: Sequential model of computation.

2.1 Computational Models

We define two models of computation, one sequential and one parallel, to represent $\mathcal{S}$ and $\mathcal{P}$, respectively. In what follows the standard definition of *time unit* is adopted, that is, the unit traditionally used to measure the running time of an algorithm [2]: A time unit is the length of time required by a processor to read a constant number of fixed-size data from memory, perform a fixed-number of constant-time operations (such as adding two numbers, comparing two numbers, and so on), and write a constant number of fixed-size data to memory. It is important to keep in mind 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).

2.1.1 Sequential Model

This is the conventional model of computation used in the design and analysis of sequential (or *serial*) algorithms. It consists of a single processor P_1 made up of circuitry for executing arithmetic and logical operations and a number of registers that serve as memory for storing programs and data. For our purposes, the processor is also equipped with an input unit and an output unit that allow it to receive data from, and send data to, the outside world, respectively. A stylized representation of the model is shown in Fig. 2.1.1.

During each time unit of a computation the processor can:

1. Receive a constant number of fixed-size data as input
2. Perform a fixed number of constant-time operations on its input
3. Produce a constant number of fixed-size data as output.

2.1.2 Parallel Model

Our chosen parallel model is the *linear array of processors* [2]. In this model, n processors, denoted by P_1, P_2, ..., P_n, where $n \geq 2$, are connected to one another by (two-way) communication links to form a one-dimensional array. Each processor is of the type described in Section 2.1.1. Here P_1 is connected only to P_2, and P_n is connected only to P_{n-1}, while P_i, for $2 \leq i \leq n-1$, is connected to both P_{i-1} and P_{i+1}. A pair of processors connected by a link can communicate directly with one another. A stylized representation of the model is shown in Fig. 2.1.2.

During each time unit of a computation the processor can:

1. Receive as input a constant number of fixed-size data from the outside world or from another processor to which it is directly connected
2. Perform a fixed number of constant-time operations on its input

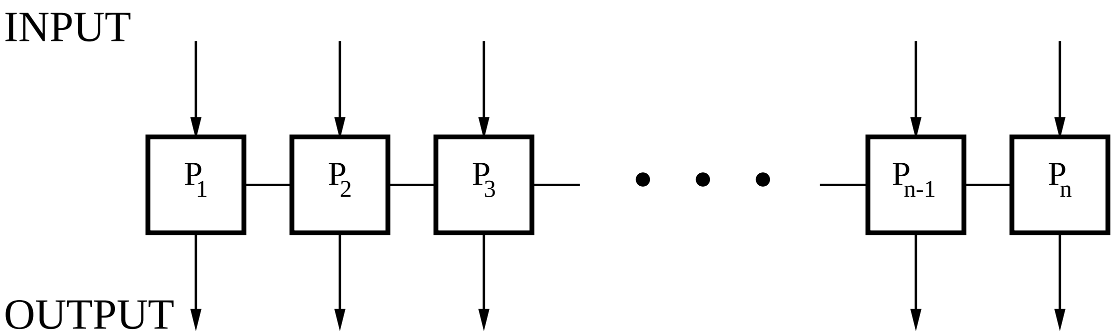

Figure 2: Parallel model of computation.

3. Produce as output a constant number of fixed-size data to the outside world or to another processor to which it is directly connected.

The linear array of processors is considered to be the simplest and most fundamental of all models of parallel computation in which the processors have some means of communicating among themselves. It can similarly be argued that it is the weakest of all such computers. Nonetheless this model, with its rudimentary communication paths, is perfectly suitable when solving the real-time computational problems of this paper. This is demonstrated in Sections 3 and 4, where it is shown that the linear array of processors affords parallel algorithms that are either significantly faster or significantly better than their sequential counterparts. In this regard, it is also important to recall that the *weaker* the computational model used in an algorithmic analysis, the *stronger* the result obtained. This is true because any algorithm designed for a certain model can be executed (through simulation or otherwise) on a more powerful model *without any loss in performance*. It follows therefore that the results of this paper, which are derived for the weakest of all models of parallel computation (namely, the linear array), hold in general (that is, for *all* parallel models).

In closing this section on computational models we note that there exists an assumption underlying most theoretical results in parallel computation that leads some readers (who are unfamiliar with the field, and hence unaware of the implicit assumption) to ask: Why can't we use a faster sequential computer and achieve the same results obtained with the parallel one? In most cases, one can easily respond to this question by showing how simple it is to defeat the 'faster' sequential machine. For example, in a real-time environment (as in the present paper), it suffices to make the data-arrival rate *faster* than the new and improved sequential machine can handle! However, in order to avoid any such (perhaps confusing) arguments, we make the standard assumption explicitly at the outset (at the risk of stating the obvious): The analyses in this paper assume that all models of computation use the fastest processors possible (within the bounds established by theoretical physics). Specifically, no sequential computer exists that is faster than the one of Section 2.1.1, and similarly no parallel computer exists whose processors are faster than those of Section 2.1.2. Furthermore, no processor on the parallel computer of Section 2.1.2 is faster than the processor of the sequential computer of Section 2.1.1. This is the *fundamental assumption in parallel computation*. It is also customary to suppose that the sequential and parallel computers use identical processors. We adopt this convention throughout the paper, with a single exception: In Section 3.2.3 we assume that the processors of the parallel computer are in fact *slower* than their sequential counterpart.

2.2 Superlinear Performance

We need to be specific about what we mean by "more than n" in Questions 1 and 2 of Section 1. For example, the quantities $2n$, $n+5$, and $3n+6$ are, strictly speaking, larger than n. Thus, a computation that is performed by the parallel computer $2n$ times faster than is possible on the sequential computer would provide a positive answer to Question 1. Similarly, a solution to a problem obtained by $\mathcal{P}$ that is $n+5$ times better than that obtained by $\mathcal{S}$ would provide a positive answer to Question 2. In fact, in many practical settings such an improvement in performance would be regarded as a considerable achievement. However, we wish to go further. While the functions $2n$, $n+5$, and $3n+6$ are *linear in n*, the examples presented in this paper exhibit *superlinear performance.*

Definition 2.1 A function f of a positive integer n is said to be *superlinear in n* if there exists a positive integer n_0 such that for all $n > n_0$ and any positive constant c, it holds that $f(n) > cn$.

According to this definition, the functions n^x and x^n, where $x > 1$, are both superlinear in n.

2.3 Real-Time Computation

The prevalent mode 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 mode which is certain to play an increasingly important role in the future is *real-time* computation. Here, not all inputs are given at the outset. Rather, the algorithm receives its data (one or several at a time) *during* the computation, and must incorporate the newly arrived inputs in the solution obtained so far. Often, the data-arrival rate is constant; specifically, $\mathcal{N}$ data are received every $\mathcal{T}$ time units, where both $\mathcal{N}$ and $\mathcal{T}$ are fixed in advance.

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

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

In some applications these deadlines may be crucial, particularly when human lives are at stake. We assume in this paper that all deadlines are *tight*, that is, they are measured in terms of a few time units (as defined in Section 2.1), and that they are *firm*, meaning that missing a deadline causes the computation to fail. It is helpful to note here that, when no time constraints are imposed, computations for which inputs arrive while the algorithm is in progress are referred to in the literature as *on-line*, *incremental*, *dynamic*, and *updating*. We also wish to emphasize that our definition, while striving to be as general as possible, is particularly suitable for our purposes in this paper. Many other more or less specialized definitions exist; see, for example, the various interpretations of the notion of real time provided in [6, 12, 18].

3 SUPERLINEAR 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 (in the worst case) by the best sequential algorithm for solving the problem at hand, to the time T_n required (also in the worst case) by the n-processor parallel algorithm being evaluated, where $n > 1$. Denoting the speedup by $speedup(1,n)$, we have:

$$speedup(1,n) = \frac{T_1}{T_n}.$$

It is widely believed that the speedup achieved by a parallel algorithm using n processors over a sequential algorithm is at most equal to n. This belief is usually called the 'speedup theorem' and is stated as:

$$speedup(1,n) \leq n.$$

One can view the above inequality as 'bad news', since it puts an upper bound on the amount of speedup possible with n processors. Most traditional computations (such as sorting, searching, operating on matrices, and so on) when executed in parallel using n processors exhibit a speedup of at most n (or some linear function of n), 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*). Let a computation be performed with n processors in time T_n and with q processors, $2 \leq q < n$, using the same algorithm, in time T_q, where $T_n < T_q$. The slowdown experienced is defined as:

$$\mathit{slowdown}(q,n) = \frac{T_q}{T_n}.$$

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

$$\mathit{slowdown}(q,n) \leq \lceil \frac{n}{q} \rceil.$$

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 n processors are available. Most traditional computations satisfy the 'slowdown theorem'.

Recently, 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. $\mathit{speedup}(1,n)$ is superlinear in n; thus, for example, the speedup is on the order of n^x, or even x^n, for some $x > 1$.
2. $\mathit{slowdown}(q,n)$ is superlinear in $\lceil n/q \rceil$; thus, for example, the slowdown is on the order of $(\lceil n/q \rceil)^x$, or even $x^{\lceil n/q \rceil}$, for some $x > 1$.

This suggests 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*.

In this section we present three examples of computations in which a superlinear improvement in speed is achieved through parallel computation. All computations occur in real time. In each example, the n-processor solution is more than n times faster than the one-processor solution because the latter performs more computations (or, alternatively, remains idle for long periods). In one of the examples, superlinear speedup is achieved despite the fact that the processors of the parallel computer are significantly slower than their sequential counterpart. As well, in each of the examples, a q-processor solution is more than $\lceil n/q \rceil$ times slower than an n-processor solution.

It is important to emphasize that in each example the superlinear speedup is consistent and provable, as stated in Section 1. These characteristics distinguish the results presented here from those in [5, 10, 11], for example, where a superlinear speedup occurs only occasionally, or is achieved either because the sequential algorithm used is inefficient, or because the size of the memory on the sequential computer is restricted.

3.1 Snow Storms And Data Accumulation

In this section we describe a computational problem in which the sequential solution is more than n times slower than an n-processor parallel solution because it must process *more data*. The problem is aptly illustrated by the following metaphor originally described in [13, 14, 15].

3.1.1 The Snow Shoveling Metaphor

It is winter and the snow has fallen all night. In the morning, as it continues to snow, a group of five people wish to get their car out of the garage. Together, they clear their driveway more than five times faster than their neighbor. This is because the latter works alone, hence longer, and therefore allows more snow to accumulate. Before being able to drive away, the single shoveler ends up removing more snow than the five neighbors do, despite the fact that the two driveways are exactly the same size.

3.1.2 The Data Accumulating Paradigm

The five shovelers are able to drive away before more snow piles up in their driveway. The single shoveler, by contrast, may have to keep at it until the snow stops falling before getting such an opportunity. A computational problem, equivalent to the snow shoveling metaphor is now presented. Let k be a small constant positive integer greater than 1 (for example, $k = 4$). In a certain application, n new data are received every k time units and require to be processed. Each received datum must undergo a constant-time operation (such as multiplying it by a constant, for example), after which its processing is considered complete. If *all data currently in store* have been processed, then the computation terminates, regardless of whether more data arrive later. On the other hand, if the processing of previously received data is not completed before a new set arrives, then it is imperative that this new set also be processed. Therefore, the deadline for early termination is $k - 1$ time units after the arrival of an input set. However, in order to allow for termination to be possible at all, there is an upper bound of 2^n on the total number of sets of n data to be received during a given computation.

Sequential Solution. When the first batch of n data is received, the single processor on computer $\mathcal{S}$ needs one time unit to read each datum, process it, and then produce it as output. This takes n time units. Meanwhile, $\lfloor n/k \rfloor$ additional data sets would have arrived, and are now in some buffer waiting in queue to be read, processed, and produced as output by $\mathcal{S}$. The latter does not catch up with the incoming data until they cease to arrive. Therefore, $\mathcal{S}$ must process $2^n \times n$ values, a computation requiring $2^n \times n$ time units. This is optimal (in the sense that no faster sequential solution is possible), in view of the obvious lower bound on the amount of time required to read the input.

Parallel Solution. At the beginning of the computation, each of the n processors of $\mathcal{P}$ receives one datum from the first data set, processes it, and produces it as output, all in one time unit. Since by the end of the first time unit (that is, at the beginning of the second time unit), all currently available data have been processed, and no new data have been received that demand attention, the parallel computation can be considered terminated. The fact that new data arrive at the beginning of the third time unit (when $k = 2$), or later (when $k > 2$), is of no concern to the parallel computer.

Speedup. By the above analysis, $T_1 = 2^n \times n$ and $T_n = 1$. Therefore, $\mathit{speedup}(1, n) = 2^n \times n$, which is significantly larger than the maximum speedup of n predicted by the speedup theorem.

It is interesting to note that the same example also leads to a contradiction of the slowdown theorem for $\mathcal{P}$. Suppose that, instead of n, the parallel computer had only q processors, where $2 \leq q < n$, and let $(n/q) > k$. The first set of data is processed in $\lceil n/q \rceil$ time units, with each of the q processors handling at most $\lceil n/q \rceil$ data. Meanwhile, $\lfloor (\lceil (n/q) \rceil)/k \rfloor$ new data sets would have been received. This way, $\mathcal{P}$ does not catch up with the input until it ceases to arrive. Therefore, $2^n \times n$ data must be processed, and this requires $T_q = \lceil (2^n \times n)/q \rceil$ time units. As a result, $\mathit{slowdown}(q, n) = \lceil (2^n \times n)/q \rceil$, which is asymptotically larger than the $\lceil n/q \rceil$ slowdown predicted by the slowdown theorem.

3.2 Running In Circles And The Elusive Input

For the computational problem in this section, the solution computed by $\mathcal{S}$ is more than n times slower than that computed by $\mathcal{P}$ because, when presented with a choice of data sets, the sequential machine makes the incorrect selection in the worst case. Consider the following metaphor [2].

3.2.1 The Pursuit And Evasion On A Ring Metaphor

An entity A is in pursuit of another entity B on the circumference of a circle, such that A and B move at the same speed; clearly, A *never* seizes B. Now, suppose that two entities X and Y are in pursuit of entity B on the circumference of a circle. In this case, X and Y, traveling in opposite directions, *always* seize B.

3.2.2 The Proper Data Set Selection Paradigm

In the metaphor just described, entity A can be thought of as a single processor having to contend with two streams of data, namely, the clockwise and counterclockwise motion of entity B. The computational paradigm presented in this section generalizes this idea to multiple streams. In this paradigm, there are n sources that provide data to solve a given computational problem. With all sources operating simultaneously, each source provides a stream of n data, one datum per time unit. The n data contained in any one of the n streams are necessary and sufficient to solve the problem at hand. However, one particular stream, unknown in advance, brings data that allow the problem to be solved in n time units. By contrast, the n data in each one of the remaining $n-1$ streams lead to a solution in 2^n time units. All streams are equally likely to be the one stream containing the data leading to a solution in n time units.

In addition, the data in the n streams satisfy the following conditions:

1. The n data streams are totally disjoint: No two streams contain the same datum, and no subset of one stream can be combined with a subset of another stream to solve the computational problem at hand.

2. There is a deadline constraint: Processing of the data in a stream must begin *at the latest* when the last datum in the stream has been received. Otherwise, the n data of that stream become obsolete as they are replaced by n new values in the fixed-size buffer.

Sequential Solution. At random, $\mathcal{S}$ chooses *one* of the n streams and uses the n data supplied by that stream to compute a solution to the problem at hand. In the worst case, $\mathcal{S}$ selects one of the $n-1$ streams that cause the computation to run for 2^n time units, before arriving to a solution. Note that this worst case occurs with probability $(n-1)/n$. This probability approaches 1 as n increases. In other words, with near certainty, the running time of $\mathcal{S}$ is $T_1 = n + 2^n$ time units, which is optimal when solving this problem sequentially.

Parallel Solution. On $\mathcal{P}$ each of the n processors monitors one of the n streams and attempts to solve the computational problem using the n data provided by that stream. One (and only one) of the processors attains a solution n time units after the arrival of the nth datum in its chosen stream. That processor now sends a message across the linear array to inform the remaining $n-1$ processors to stop their calculations. The computation by $\mathcal{P}$ therefore requires $T_n = 3n-1$ time units at worst.

Speedup. In the worst case, the speedup provided by $\mathcal{P}$ over $\mathcal{S}$ is $\mathit{speedup}(1,n) = (n+2^n)/(3n-1)$, which is superlinear in n, and evidently the speedup theorem fails in this case as well.

What happens if $\mathcal{P}$ has only q processors, where $2 \le q < n$? From among the n possible streams, the q processors choose at random q streams to monitor. In the worst case, none of the processors chooses a stream whose data lead to a solution in n time units. This worst case occurs with probability $1-(q/n)$. Therefore, if n is large and q is small, the running time required by $\mathcal{P}$ is $T_q = n + 2^n$ time units, almost certainly. By comparison with the n-processor solution, $\mathit{slowdown}(q,n) = (n+2^n)/(3n-1)$. This slowdown is superlinear in $\lceil n/q \rceil$, regardless of the value of q, and the slowdown theorem does not hold.

3.2.3 A Metaphor Revisited

Let us reconsider the pursuit evasion on a ring metaphor, in the case where there are two entities X and Y in pursuit of entity B. Suppose here that each of X and Y moves at $1/k$ the speed of both the single pursuer A and the entity B, where k is a positive integer larger than 1. Again here, X and Y (despite their sluggishness) *always* seize B.

For the equivalent computational paradigm, let us suppose that each of the n processors of $\mathcal{P}$ is n times slower computationally than the single processor of $\mathcal{S}$. This, as mentioned at the end of Section 2.1, is contrary to the convention adopted in the literature and throughout this paper, whereby the processors of $\mathcal{P}$ and $\mathcal{S}$ are identical. Specifically, we assume here that a processor of the parallel computer still takes one time unit to receive a constant number of fixed-size data as input, and produce a constant number of fixed-size data as output; however, it now requires n time units to execute the same (arithmetic and logical) operations performed by $\mathcal{S}$ in *one time unit*. In the worst case, therefore, the number of time units now needed by $\mathcal{P}$ to solve the computational problem of Section 3.2.2 is on the order of n^2, whereas $\mathcal{S}$, as before, spends on the order of 2^n time units on the same problem. The parallel computer continues to achieve a speedup superlinear in n, despite the unreasonable assumption made about its processors in favor of the sequential computer.

3.3 Furniture Moving And One-Way Functions

Our third and last example of superlinear speedup deals with the situation where the sequential solution is more than n times slower than the parallel one because it performs unnecessary computations, or must wait for data.

3.3.1 The Furniture-Moving Metaphor

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. This is 240 times faster than the single mover [2].

3.3.2 The One-Way Functions Paradigm

Moving the piece of furniture was easy if one did not need to take it apart. We now describe an equivalent computational paradigm.

A function f is said to be *one-way* if the function itself takes little time to compute, but (to the best of our knowledge) its inverse f^{-1} is computationally prohibitive. For example, let $x_1, x_2, \ldots, x_n$ be a sequence of integers. It is easy to compute the sum of a given subset of these integers; however, starting from the sum, no efficient algorithm is known to determine the subset of integers that make it up.

Suppose that in order to solve a certain problem, it is required to compute $g(x_1, x_2, \ldots, x_n)$, where g is some function of n variables. For example, $g(x_1, x_2, \ldots, x_n) = x_1^2 + x_2^2 + \cdots + x_n^2$, might be such a function. The computation of g requires n time units. The inputs $x_1, x_2, \ldots, x_n$ needed to compute g are received as n pairs of the form $\langle x_i, f(x_1, x_2, \ldots, x_n)\rangle$, for $i = 1, 2, \ldots, n$.

The function f possesses the following property: Computing f from $x_1, x_2, \ldots, x_n$ is done in n time units; on the other hand, extracting x_i from $f(x_1, x_2, \ldots, x_n)$ takes 2^n time units.

Because the function g is to be computed in real time, there is a deadline constraint: If a pair is not processed within one time unit of its arrival, it becomes obsolete (it is overwritten by other data in the fixed-size buffer in which it was stored).

Sequential Solution. The n pairs arrive simultaneously and are stored in a buffer waiting in queue to be processed. The pair $\langle x_1, f(x_1, x_2, \ldots, x_n)\rangle$ is the first to be read by $\mathcal{S}$. At this point, the other $n-1$ pairs are no longer available. In order to retrieve $x_2, x_3, \ldots, x_n$, the sequential processor needs to invert f. This requires $(n-1) \times 2^n$ time units. It then computes $g(x_1, x_2, \ldots, x_n) = x_1^2 + x_2^2 + \cdots + x_n^2$. Consequently, $T_1 = 1 + (n-1) \times 2^n + (n-1)$ time units. Clearly, this is optimal for $\mathcal{S}$ considering the time required to obtain the data.

Parallel Solution. Once the n pairs are received, they are processed by the parallel computer immediately. Processor P_i reads the pair $\langle x_i, f(x_1, x_2, \ldots, x_n)\rangle$ and computes x_i^2, for $i = 1, 2, \ldots, n$. In particular, P_1 sets a variable g' equal to x_1^2 and sends it to P_2. The processors of $\mathcal{P}$ now compute $g(x_1, x_2, \ldots, x_n)$ in $n-1$ time units: P_i receives g' from P_{i-1} and sends $g' = g' + x_i^2$ to P_{i+1}, for $2 \leq i \leq n-1$, and to the output if $i = n$ (at which point $g' = x_1^2 + x_2^2 + \cdots + x_n^2$). Therefore, $T_n = n$.

Speedup. The speedup provided by $\mathcal{P}$ over $\mathcal{S}$, namely, $\mathit{speedup}(1, n) = ((n-1) \times 2^n + n)/n$, is superlinear in n and thus contradicts the speedup theorem. What if only q processors are available on $\mathcal{P}$, where $2 \leq q < n$? In this case, only q of the n variables (for example, $x_1, x_2, \ldots, x_q$) are read directly from the input buffer (one by each processor). Meanwhile, the remaining $n-q$ variables vanish and must be extracted from $f(x_1, x_2, \ldots, x_n)$. It follows that $T_q = 1 + \lceil (n-q)/q \rceil \times 2^n + \lceil (n-q)/q \rceil + (q-1)$. Therefore, $\mathit{slowdown}(q, n) = T_q/T_n$ is superlinear in $\lceil n/q \rceil$ and, once again, the slowdown theorem is violated.

3.3.3 Variation On A Paradigm

As in Section 3.3.2, it is required to compute $g(x_1, x_2, \ldots, x_n) = x_1^2 + x_2^2 + \cdots + x_n^2$. Here, however, the input consists of n independent and parallel streams. Each stream is a cyclic permutation of $x_1, x_2, \ldots, x_n$.

Within a stream each value is separated from the next by 2^n time units. Each time n values are received simultaneously (one in each stream) they are all distinct and hence form a complete set $x_1, x_2, \ldots, x_n$. A value not processed within one time unit of its arrival becomes obsolete (perhaps replaced by other data in the fixed-size buffer).

Because each value received is replaced after one time unit of its arrival, $\mathcal{S}$ cannot read more than one value from among the n supplied by the n streams simultaneously at any given time. In order to gather $x_1, x_2, \ldots, x_n$, the sequential computer therefore 'locks' onto one stream and thus waits $(n-1) \times 2^n$ time units before receiving the nth value in the set. Initially, a variable g' is 0; when an input value is received, it is squared and added to g', until $x_1^2 + x_2^2 + \cdots + x_n^2$ is obtained. The sequential computation requires $T_1 = 1 + (n-1) \times 2^n$ time units, which is optimal for $\mathcal{S}$. By contrast, on $\mathcal{P}$, the n processors receive $x_1, x_2, \ldots, x_n$ simultaneously (one value per stream per processor). Computation of $g(x_1, x_2, \ldots, x_n)$ then proceeds immediately (there is no need to wait). The running time is $T_n = n$ time units. Consequently, for this variant, the speedup is also superlinear in n, the number of processors used by the parallel computer.

4 QUALITY-UP

The primary purpose of parallel computation is the fast execution of computational tasks that are too slow to perform sequentially. As a consequence, interest in parallel computation to date has naturally focused on the speedup provided by parallel algorithms over their sequential counterparts. 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. In this section we show that within the real-time mode of computation, some classes of problems have the property that a solution to a problem in the class, when computed in parallel, is far superior in quality than the best one obtained on a sequential computer. What constitutes a better solution depends on the problem under consideration. Thus, for example, 'better' means 'closer to optimal' for optimization problems, 'more accurate' for numerical problems, and 'more secure' for cryptographic problems.

The improvement in the quality of a solution to some computational problem, achieved through parallelism, is measured by a ratio known as the *quality-up* [1]. Let V_1 be the value of the solution to the problem obtained (sequentially) on $\mathcal{S}$. Similarly, let V_n be the value of the solution to the same problem obtained (in parallel) on $\mathcal{P}$ using n processors. Then

$$\textit{quality-up}(1, n) = \frac{V_n}{V_1}.$$

By the simulation principle, a sequential computer should be able to compute *the same* solution as the one obtained in parallel. Consequently, one should not expect $\textit{quality-up}(1, n)$ to exceed 1 (never mind be superlinear in n). Therefore, to be able to demonstrate that a parallel computer can obtain a better solution to a computational problem than one derived sequentially would alone be an interesting (and even surprising) observation in its own right. Yet it is shown in what follows that the improvement in quality can be *arbitrarily high* (and certainly superlinear in the number of processors used by the parallel computer). Moreover, as with the speedup, the superlinear quality-up is provable and consistent. Because the notion of a 'better' solution is so closely related to the particular computational problem of interest, we present three specific examples, one from each of the areas mentioned above, namely, optimization, numerical computation, and cryptography. The last example illustrates the case where the improvement in quality grows without bound.

It is important to note that, in each case, V_1 and V_n are defined appropriately so that, if indeed there is an improvement in quality due to parallel computation, the ratio V_n/V_1 is greater than 1. When the purpose is to *maximize* a quantity (such as, for example, the profit in an optimization problem, or the level of security in a cryptographic application), the choice of V_1 and V_n is straightforward. By contrast, particular care must be paid to the definition of V_1 and V_n in problems where it is required to *minimize* a quantity, such as, for example, the cost of the solution in an optimization problem, or the amount of error in a numerical computation. The latter situation is illustrated in Section 4.2 where the value of a numerical solution is measured by the inverse of the error it contains.

4.1 Fast Decisions And Discrete Maximization

The family of optimization problems, as used in this paper, is defined as follows. Each problem in the family takes as input a finite set of data. It is required to select a subset of this set, consisting of m elements (where m is a positive integer, which may or may not be given). The selected subset must satisfy certain conditions, known as the *constraints*, that are germane to the problem being solved. Among all subsets satisfying the constraints, we are to find the one maximizing (or minimizing) a given function of m arguments, called the *objective function*. Often, when the exact maximum (or minimum) is difficult to obtain, an *approximation* of the optimal value is computed. This form of optimization is commonly referred to as *discrete* or *combinatorial* optimization.

Evidently, we are interested here in solving optimization problems in a real-time setting. Our purpose is to demonstrate the ability of a parallel algorithm to do better than the best sequential algorithm when maximizing an objective function in real time. In this context, a *better* solution is one that is *closer to maximum*.

4.1.1 The Prize Behind A Door Metaphor

In a certain game there are 10 doors, each of which hiding 10 boxed prizes. It takes about one minute to open a box. A contestant has 10 minutes to choose 10 prizes, one from behind each door. Evidently, there is no time to compare the prizes behind any given door before making a selection. Of course it would be nice (if the rules allowed it) to enlist the help of 9 friends (one per door), thereby guaranteeing that the best (that is, the most valuable, or most appealing, or most appropriate, and so on) 10 prizes are chosen.

4.1.2 The Combinatorial Optimization Paradigm

In the present application, there are n input streams, where $n > 1$. Each time unit, one datum is received from each input stream. The data are numbers in the interval 1 to n^w, for some $w > 1$. The data within each stream are viewed as blocks of length n. For each stream, it is required to choose the largest datum in each received block (or an approximation thereof).

This being a real-time computation, there is inevitably a deadline condition: A selection must be made for each block when the last datum of that block has been received, *at the latest*.

Sequential Solution. A sequential computer cannot monitor all streams simultaneously in real time (it has only one processor!). Similarly, while buffering may be possible, $\mathcal{S}$ cannot wait for a complete block to arrive from each stream, then choose the largest value in each of these n blocks, due to the deadline constraint. Therefore, the only thing a sequential computer can do is to select one value arbitrarily from the block currently being received in each stream. For example, it may scan the streams consecutively, devoting one time unit to each stream, whereby it picks the first value from the first stream, the second value from the second stream, and so on. Once the nth value of the nth stream has been selected, a phase of the computation would have been completed and the next phase, identical to the previous one, begins. Note here that no other strategy for obtaining the sequential solution is provably superior to the arbitrary selection of n values.

Parallel Solution. An n-processor computer monitors all streams simultaneously in real time. Each processor is in charge of one stream: it reads successive values received, keeping track of the largest one in the current block.

Quality-up. In the worst case, each value selected by the sequential computer is 1, while the largest value in that block is n^w. On the other hand, the parallel computer always finds the largest value. This means that $\textit{quality-up}(1, n) = n^w$, that is, the improvement in quality is a polynomial in n. Because $w > 1$, this improvement is a superlinear function of the number of processors used by $\mathcal{P}$.

4.2 Zeroing In On The Answer And Minimizing The Error

A further class of computational problems is now identified in which parallelism provides solutions that are better than ones obtained sequentially. Specifically, we study the class of *numerical computations*. In this context, a solution is 'better' if it is 'more accurate', and our analysis focuses on the reduction in the size of the error, achieved through parallelism. It is appropriate to begin, therefore, by defining the notion of *error*. Typically, 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_{approx}. Then, the *numerical error* in A_{approx} is defined as $A_{\text{exact}} - A_{\text{approx}}$.

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

$$|A_{\text{exact}} - A_{\text{approx}}| \leq \frac{K}{h(r)},$$

where K is a constant that depends on the problem at hand, r is a parameter of the algorithm (such as, for example, the number of iterations performed), and $h(r)$ is an increasing function of r.

4.2.1 Searching For The Needle In A Haystack Metaphor

You live in a big city and need to make an important but urgent purchase. There are dozens of dealers and several times as many models of the item of interest. By calling on the members of your family to assist you, your search is narrowed to a handful of options.

4.2.2 The Numerical Computation Paradigm

Suppose that $f(x)$ is a continuous function, such as $e^x - \cos x$, for example. Further, let a and b be two values of the variable x such that $f(a) \times f(b) < 0$. A *zero* of f, that is, a value x_{exact} for which $f(x_{\text{exact}}) = 0$, is guaranteed to exist in the interval $[a, b]$. An iterative numerical algorithm computes an approximation x_{approx} to x_{exact} as follows. 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 $f(a_1) \times f(m_1) < 0$, 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_{approx} of x_{exact} is obtained, that is, until for some $r \geq 1$, $|b_r - a_r| < \alpha$, where α is a small positive number chosen such that the desired accuracy is obtained. When the latter condition is satisfied, $x_{\text{approx}} = (a_r + b_r)/2$. Because

$$|x_{\text{exact}} - x_{\text{approx}}| \leq \frac{|b_r - a_r|}{2} \leq \frac{|b_{r-1} - a_{r-1}|}{2^2} \leq \cdots \leq \frac{|b_1 - a_1|}{2^r},$$

the error bound is

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

The algorithm would have performed r iterations to obtain x_{approx}.

Now consider the following computational environment:

1. A computer system receives a stream of inputs in real time.

2. The numerical problem to be solved here is to find a zero x_{approx} for a continuous function $f(x)$ that falls between $x = a$ and $x = b$. At the beginning of each time unit, a new 3-tuple $\langle f, a, b \rangle$ is received by the computer system.

3. It is required that $\langle f, a, b \rangle$ be processed as soon as it is received and that x_{approx} be produced as output as soon as it is computed. Furthermore, one output must be produced at the end of each time unit (with possibly an initial delay before the first output is produced).

4. The operations of reading $\langle f, a, b \rangle$, performing *one* iteration of the algorithm, and producing x_{approx} as output once it has been computed, can be performed within one time unit, as defined in Section 2.1.

Sequential Solution. Here, there is a single processor whose task is to read each incoming 3-tuple, to compute x_{approx}, and to produce the latter as output. Recall that the computational environment we assumed dictates that a new input 3-tuple be received at the beginning of each time unit, and that such an input be processed immediately upon arrival. Therefore, $\mathcal{S}$ must have finished processing a 3-tuple before the next one arrives. It follows that, within the one time unit available, the algorithm can perform no more than one iteration on each input $\langle f, a, b\rangle$. The approximate solution computed by $\mathcal{S}$ is $x_{\text{approx}} = m_1$. This being the *only* option available, it is by default the *best* solution possible sequentially.

Parallel Solution. When solving the problem on the n-processor computer, it is evident that processor P_1 must be designated to receive the successive input 3-tuples, while it is the responsibility of P_n to produce x_{approx} as output. The fact that each 3-tuple needs to be processed as soon as it is received implies that the processor must be finished processing a 3-tuple before the next one arrives. Since a new 3-tuple is received every time unit, processor P_1 can perform only one iteration on each 3-tuple it receives. Unlike the sequential solution, however, the present algorithm can perform additional iterations. This is done as follows. Once P_1 has executed its single iteration on $\langle f, a_1, b_1\rangle$, it sends $\langle f, a_2, b_2\rangle$ to P_2, and turns its attention to the next 3-tuple arriving as input. Now P_2 can execute an additional iteration before sending $\langle f, a_3, b_3\rangle$ to P_3. This continues until $x_{\text{approx}} = (a_n + b_n)/2$ is produced as output by P_n. Meanwhile, $n-1$ other 3-tuple inputs co-exist in the array (one in each of P_1, P_2, $\ldots$, P_{n-1}), at various stages of processing. One time unit after P_n has produced its first x_{approx}, it produces a second, and so on, so that an output emerges from the array every time unit. Note that each output x_{approx} is the result of applying n iterations to the input 3-tuple, since there are n processors and each executes one iteration.

Quality-up. In what follows we derive a bound on the size of the error in x_{approx} for the sequential and parallel solutions. Let the *accuracy* of the solution be defined as the inverse of the *error*.

Sequentially, one iteration of the bisection algorithm is performed to obtain x_{approx}, that is, $r = 1$. In the worst case, $|x_{\text{exact}} - x_{\text{approx}}| = |b-a|/2$.

In parallel, each 3-tuple input is subjected to n iterations of the bisection algorithm, where each processor performs one iteration. Therefore, $r = n$. In the worst case, $|x_{\text{exact}} - x_{\text{approx}}| = |b-a|/2^n$.

By defining quality-up as the ratio of the parallel accuracy to the sequential accuracy, in the worst case, *quality-up*$(1, n) = 2^{(n-1)}$. This suggests that increasing the number of processors by a factor of n leads to an increase in the level of accuracy by a factor on the order of 2^n. In other words, the improvement in quality is exponential in n, the number of processors on $\mathcal{P}$.

4.3 Gift-Wrapping And Secrecy

The purpose of contemporary cryptography is the protection of digital data. The latter may be, for example, personal, commercial, financial, or military information. 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, text, numbers, sounds, or images. 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:

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

for $1 \leq i \leq n$, 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 a 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.

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 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 section we assume that a cryptosystem implemented using a long key is more secure than the same cryptosystem implemented using a shorter key.

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

4.3.1 The Gift-Wrapping Metaphor

A person has a number of last-minute holiday gifts that need to be wrapped and mailed. One layer of paper wrapping offers no guarantee of protection against damage and exposure while a gift is in transit. On the other hand, adding more layers per package is time-consuming, and the post office is about to close. As it turns out, several people working together are able to wrap each gift securely and mail all the packages on time.

4.3.2 The Cryptography Paradigm

The problem to be solved is defined as follows:

1. A computer system receives a stream of plaintext data in real time, one block M at the beginning of each time unit.

2. No block received can be stored in plaintext form. Therefore, an input block must be processed as soon as it arrives to produce an encrypted output block. The latter is then immediately stored in some memory or transmitted over an insecure channel.

3. An encrypted block is to be produced as output at the end of each time unit (with possibly an initial delay before the first output is produced).

4. The operations of reading a block, performing one iteration of the encryption function E, and finally storing (or transmitting) the resulting block, together require one time unit.

5. For a given positive integer n greater than 1, performing n iterations (or more) of the encryption function E renders the system unconditionally secure. In other words, without knowledge of the encryption/decryption keys, n iterations of the encryption function E are unbreakable with current mathematical knowledge and present (and foreseeable) computers. Specifically, given C_n, an opponent cannot feasibly recover M.

   On the other hand, for any $x < n$, a system that performs x iterations of the encryption function E is effectively breakable without knowledge of any cryptographic key used. Specifically, an opponent can with reasonable computational effort recover M from C_x.

   By way of illustration, suppose that the computations performed by an opponent who tries to break the system require $10^{x(1+\lfloor x/n \rfloor)}$ time units, where x is the number of iterations of the encryption function E in the current implementation. Let $n = 5$ and assume that one time unit lasts one second. For $x < 5$, it takes less than three hours to obtain M from C_x. When the number of iterations is 5, however, three-hundred years are not enough to break the system.

Sequential Solution. Suppose that the computer system receiving the real-time input is $\mathcal{S}$, 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 one time unit separates consecutive blocks, only one iteration of the encryption function E can be performed on a block before the latter is stored or transmitted, and this is the best that can be done sequentially under the conditions imposed. Regarding running time, if the plaintext consists of w blocks, the sequential computer requires w time units to encrypt all blocks.

Parallel Solution. We now consider the case in which the computer system receiving the real-time input is $\mathcal{P}$. Naturally, when a linear array of 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 time unit, 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 array (one in each of the other processors) at various stages of encryption. One time unit 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 time unit. If there are w blocks in all, P_n transmits or stores the final encrypted block $n+(w-1)$ time units after the first plaintext block arrives at P_1. Each input plaintext block therefore undergoes n encryption iterations. Note that, in the absence of real-time deadlines, the same computation would require wn time units sequentially.

Quality-up. 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 parameter. Let the *security value* U be a quantity that expresses the level of security offered by a cryptosystem. For an unconditionally secure cryptosystem, $U = 1$. At the other extreme, $U = 0$ when a cryptosystem is guaranteed to be breakable.

Suppose that two implementations of a cryptosystem have security values U_1 and U_2, respectively, where $U_2 > U_1$. The *improvement in security* provided by the second implementation is given as U_2/U_1.

In the context of our discussion, we define U 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, $U = \lfloor x/n \rfloor$. The sequential implementation executes one iteration of E, and consequently its security value U_s is 0. For the parallel implementation, the number of iterations is n, resulting in a security value U_p of 1. Hence, the improvement in security provided by $\mathcal{P}$ over $\mathcal{S}$ is *quality-up*$(1, n) = U_p/U_s$. In other words, when the number of processors is n, this improvement is without bound.

5 CONCLUSION

We have demonstrated that within the real-time mode of computation a parallel computer can provide a significant improvement over the performance of a sequential computer. This improvement, in either the quality of a solution or the speed with which it is obtained, can be superlinear in the number of processors used by the parallel computer. At the heart of this result is the fact that, when computing with deadlines, the principle of simulation does not make sense. Indeed, the idea that the computations performed in parallel can somehow be replicated sequentially to achieve the same results (with respect to speed and quality) is without meaning in this context. Similar results appear in [7, 8, 9, 17].

The various computational paradigms described in this paper can be modified in several different ways. For example, to cite only three alternatives, the data-arrival rate may not be constant, certain *outputs* may be used as *inputs* in subsequent computations, or the same computation may span several time intervals. The latter variant is particularly interesting as it suggests new measures for the quality of a solution. Suppose that data arrive in real time as additions or corrections to an instance of a problem [3]. For example, these data may be new vertices for a given graph along with their associated (weighted) edges, new values for some entries of the coefficient matrix of a system of linear equations, and so on. For each newly arrived set of data, an updated solution must be computed (for example, a new minimum-weight spanning tree of the weighted graph, or a new solution to the system of linear equations). Assume now that this takes place over a long period of time, that is, over several time intervals, with each interval bringing a new data set for the present problem. If we were to use the approach presented in this paper when comparing sequential and parallel solutions, then we would perform the comparison at a certain point in time, typically after the first interval. However, more appropriate criteria may be used here. As time passes, the sequential, and perhaps the parallel, solutions may drift further and further away from the optimal. In order to compare them, one can use:

1. The *accumulated error* in the sequential solution versus the *accumulated error* in the parallel solution after a certain number t of time intervals, where $t > 0$. Specifically, if the sequential and parallel errors during the ith interval are $\rho_s(i)$ and $\rho_p(i)$, respectively, then after t intervals, the accumulated sequential and parallel errors are $\phi(\rho_s, t)$ and $\phi(\rho_p, t)$, respectively, where ϕ is a function of the error and the number of intervals. For example, it may be the case that $\phi(\rho_s, t) = \sum_i^t \rho_s(i)$ and $\phi(\rho_p, t) = \sum_i^t \rho_p(i)$, or $\phi(\rho_s, t) = \prod_i^t \rho_s(i)$ and $\phi(\rho_p, t) = \prod_i^t \rho_p(i)$, and so on. The ratio $\phi(\rho_s, t)/\phi(\rho_p, t)$ may then be used to compare the two solutions. (Note that, for computational convenience, $\phi(\rho_p, t)$ is defined to be 1 if the parallel solution is error-free after t intervals.)

2. The *cumulative error* $E_s(t)$ in the sequential solution versus the *cumulative error* $E_p(t)$ in the parallel solution after t time intervals. Using the notation just introduced, $E_s(t)$ and $E_p(t)$ could be defined as $\sum_{i=1}^t \phi(\rho_s, i)$ and $\sum_{i=1}^t \phi(\rho_p, i)$, respectively. For example, let $\rho_s(i) = e_s$ and $\rho_p(i) = e_p$, for all i, where e_s and e_p are nonnegative constants, and suppose that $\phi(\rho_s, t) = \sum_i^t \rho_s(i)$ and $\phi(\rho_p, t) = \sum_i^t \rho_p(i)$. In this case, $\phi(\rho_s, t) = te_s$, $\phi(\rho_p, t) = te_p$, $E_s(t) = t(t+1)e_s/2$, and $E_p(t) = t(t+1)e_p/2$. The sequential and parallel solutions are compared using the ratio $E_s(t)/E_p(t)$. (Again, if the parallel solution contains no error, $E_p(t) = 1$.)

The second criterion, namely, the cumulative error, is especially relevant if a penalty is incurred at the end of each interval, this penalty being a function of the accumulated error in the solution. Assume, for example, that the sequential and parallel penalties at the end of interval t are $\phi(\rho_s, t)$ and $\phi(\rho_p, t)$, respectively. This means that the cumulative penalties will be equal to the cumulative errors. Also, let the number of processors on the parallel computer be n, with $t = n$, $e_s = 2$, and $e_p = 0$. Then, when $\phi(\rho_s, t) = te_s$ and $\phi(\rho_p, t) = te_p$, the ratio of the cumulative sequential penalty to the cumulative parallel penalty, after n intervals, that is $E_s(n)/E_p(n)$, is *quadratic* in the number of processors. Similarly, when $\phi(\rho_s, t) = e_s^t$ and $\phi(\rho_p, t) = e_p^t$, this ratio is *exponential* in n.

We conclude with two open problems:

1. In every one of the examples of Section 4 (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 is at best linear in n. It is therefore tempting to ask: Can a superlinear speedup and a superlinear improvement in quality be achieved simultaneously?

2. This paper focused on real-time computation. An obvious question is: Do other modes of computation exist in which it is possible for a parallel computer with n processors to obtain a faster and/or better solution than the one derived sequentially, such that the improvement is superlinear in n?

6 ACKNOWLEDGMENTS

I am grateful to Stefan Bruda, Marius Nagy, and Naya Nagy for reading this manuscript and offering valuable comments.

References

[1] S.G. Akl, Parallel real-time computation: Sometimes quantity means quality, *Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks*, Dallas, Texas, December 2000, 2–11.

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

[3] S.G. Akl, S.D. Bruda, Parallel real-time optimization: Beyond speedup, *Parallel Processing Letters*, 9, 1999, 499–509.

[4] S.G. Akl and L. Fava Lindon, Paradigms for superunitary behavior in parallel computations, *Journal of Parallel Algorithms and Applications*, 11, 1997, 129–153.

[5] R.S. Barr and B.L. Hickman, Parallel simplex for large pure network problems: Computational testing and sources of speedup, *Operations Research*, 42, 1994, 65–80.

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

[7] S.D. Bruda, and S.G. Akl, A case study in real-time parallel computation: Correcting algorithms, Journal of Parallel and Distributed Computing, to appear.

[8] S.D. Bruda, and S.G. Akl, Real-Time Computation: A Formal Definition and its Applications, *Proceedings of the Workshop on Advances in Parallel and Distributed Computational Models*, San Francisco, California, April 2001.

[9] S.D. Bruda, and S.G. Akl, The characterization of data-accumulating algorithms, *Theory of Computing Systems*, 33, 2000, 85–96.

[10] D.P. Helmbold and C.E. McDowell, Modeling speedup(n) greater than n, *IEEE Transactions on Parallel and Distributed Systems*, 1, 1990, 250–256.

[11] R. Janssen, A note on superlinear speedup, *Parallel Computing*, 4, 1987, 211–213.

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

[13] 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.

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

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

[16] H.R. Lewis and C.H. Papadimitriou, *Elements of the Theory of Computation*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[17] M. Nagy and S.G. Akl, Real-time minimum vertex cover for two-terminal series-parallel graphs, Technical Report No. 2000-441, Department of Computing and Information Science, Queen's University, Kingston, Ontario, October 2000.

[18] M. Thorin, *Real-Time Transaction Processing*, Macmillan, London, 1992.