Technical Report No. 2001-453

DISCRETE STEEPEST DESCENT IN REAL TIME[*]

Selim G. Akl
Department of Computing and Information Science
Queen's University
Kingston, Ontario K7L 3N6
Canada

Email: akl@cs.queensu.ca
Phone: (613) 533 6062
Fax: (613) 533 6513

November 28, 2001

Abstract

A general framework is proposed for the study of real-time algorithms. The framework unifies previous algorithmic definitions of real-time computation. In it, state space traversal is used as a model for computational problems in a real-time environment. The proposed framework also employs a paradigm, known as discrete steepest descent, for algorithms designed to solve these problems. Sequential and parallel algorithms for traversing a state space by discrete steepest descent are then analyzed and compared. The analysis measures the value (or worth) of a computed solution. The quantity used in the evaluation may be the time required by an algorithm to reach the solution, the quality of the solution obtained, or any similar measure. The value of a real-time solution obtained in parallel is shown to be consistently superior to that of a solution computed sequentially by an amount superlinear in the size of the problem.

Key words and phrases: Real-time computation, state space traversal, discrete steepest descent, parallelism, superlinear speedup, superlinear quality-up.

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

1 INTRODUCTION

Several recent studies have compared the behavior of sequential algorithms to that of their parallel counterparts in a real-time computational environment. Two criteria were used to assess the performance of these algorithms:

1. **Running time:** The primary purpose of parallel computation is to perform time-consuming computations *faster* than is possible sequentially. Parallel algorithms are evaluated using the *speedup* that they achieve over their sequential counterparts. This measure of performance is defined as follows. Let T_1 be the time required (in the worst case) by the best sequential algorithm for solving the problem at hand. Similarly, let T_n be the time required (also in the worst case) by the n-processor parallel algorithm being evaluated, where $n > 1$. Denoting the speedup by $\mathit{speedup}(1, n)$, we have:

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

   A number of real-time computations have been described in recent publications for which $\mathit{speedup}(1, n)$ is superlinear in n. Examples include data-accumulating computations [1, 10, 13, 17, 18, 19], computations with multiple data streams [1, 12], and computations involving one-way functions [1, 4]. In these computations, $\mathit{speedup}(1, n)$ is typically on the order of n^x, or even x^n, for some $x > 1$.

2. **Solution quality:** A second equally important motivation for using parallel computation is that a solution to a problem computed in parallel can in some circumstances be *better* than that obtained sequentially. The improvement in the quality of a solution is measured by a ratio known as the *quality-up* [3]. Let V_1 be the value of the solution to the problem obtained sequentially. Similarly, let V_n be the value of the solution to the same problem obtained in parallel using n processors. Then

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

   As with speedup, it was shown recently 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 when compared to the best solution obtained on a sequential computer. Specifically, $\mathit{quality\text{-}up}(1, n)$ in these cases is superlinear in n. Evidently, what constitutes a better solution depends on the problem under consideration. Thus, for example, 'better' means 'closer to optimal' for optimization problems [2, 8, 15, 20, 21], 'more accurate' for numerical problems [7], and 'more secure' for cryptographic problems [6].

Different models of computation were used to develop the parallel solutions described in the cited references. They include the linear array, the reconfigurable multiple bus model, and the parallel random-access machine. Furthermore, and partly as a consequence of the particular choice of a computational model, the parallel algorithms used by previous studies varied widely in their design.

This paper proposes a generalization of these earlier efforts. Specifically, four aspects of real-time computation are generalized:

1. *State space traversal* is used as a general framework for the computation to be performed (that is, the *problem* to be solved).

2. *Discrete steepest descent* is used as a general paradigm for the *algorithm* to be applied when solving the problem at hand.

3. The *model* of parallel computation is as general as can be for our purpose: Any model that is able to compute the minimum of n values using n processors in time that grows as the logarithm of n could be adopted.

4. The *number of stages traversed* in going from the initial to the final state of the computation measures the *value* or *worth* of a solution.

Sequential and parallel algorithms for traversing a state space by discrete steepest descent are analyzed and compared. The analysis measures the value (or worth) of a solution, which may be the time required by an algorithm to reach such solution, or the quality of the solution obtained, and so on. The value of a real-time solution obtained in parallel is shown to be consistently superior to that of one computed sequentially.

The remainder of this paper is organized as follows. We begin in Section 2 by providing an algorithmic definition of real-time-computation. Section 3 introduces state space traversal as a computational framework. The algorithmic paradigm to be applied to the solution of problems within state space traversal, namely, discrete steepest descent, is defined in Section 4. An interpretation of state space traversal within a real-time environment is developed in Section 5. Section 6 outlines the characteristics required for sequential and parallel models of computation to be used for the design of algorithms. Sequential and parallel solutions to the problem of state space traversal by discrete steepest descent in real time are designed and analyzed in Sections 7 and 8, respectively. Some conclusions are offered in Section 9.

2 WHAT IS REAL-TIME COMPUTATION?

The notion of *real time* is an intuitive one. The term evokes simultaneity, liveness, and immediacy; it even conveys a sense of urgency and the need for an instantaneous action. Yet, *real-time computation* is difficult to capture. The numerous definitions in the literature attest to this fact [11]. One reason for this situation is the fact that, in today's fast-paced society where computers can perform billions of operations per second, *every* computation seems to be performed in real time. Whether we are dealing with a bank machine or running a flight simulator, we have become accustomed to expect the result of our computations *now.*

In a recent textbook on the subject [16], the authors acknowledge the difficulty in delineating what computations to characterize as real-time ones. They proceed to define real-time computations as those computations in which failure to produce the answers after a certain time would be catastrophic, involving loss of human life.

This paper avoids the extremes of the previous two paragraphs. Our working definition of real-time computation is an algorithmic one. It can be adapted to apply at any point on the above spectrum, depending on the circumstances and requirements of a particular application.

2.1 An Algorithmic Definition

In *real-time* computation all the data required by an algorithm are *not* available when it starts working on the problem to be solved. Instead, the algorithm receives its data *from an external source*, one or several at a time, *during* the computation, and must incorporate the newly arrived inputs in the solution obtained so far.

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, especially (as pointed out earlier) when human lives are at stake.

In this paper, we make the following assumptions:

1. A *time unit* is the length of time required by an algorithm to read a constant number of fixed-size data from the input, perform a fixed-number of constant-time operations (such as adding two numbers, comparing two numbers, and so on), and produce a constant number of fixed-size data as output.

2. Time is divided into *intervals.* Each time interval is $\mathcal{T}$ time units long, where $\mathcal{T}$ depends on a number of factors, including the problem being solved, the environment in which the computation is performed, and the timeliness requirements imposed by the application.

3. The external source provides at most $\mathcal{N}$ new data at the beginning of each time interval, where $\mathcal{N}$ also depends on the problem being solved.

4. The algorithm for solving the problem at hand can, based on its calculations, directs the external source as to which data to provide when the next batch is to be sent. Should the algorithm not issue such directions, the external source selects the inputs by itself and sends them to the algorithm.

5. All deadlines are *tight*, that is, they are measured in terms of one time interval, and they are *firm*, meaning that missing a deadline causes the computation to fail.

3 STATE SPACE TRAVERSAL

A certain computation has the following characteristics:

1. There is an initial configuration of data called the START state and a goal configuration called the FINISH state. These two states are distinct and well defined for a given instance of the problem to be solved.

2. It is required to go from the START state to the FINISH state by a series of transformations. This takes the data through a sequence of intermediate states $S(1)$, $S(2)$, ..., $S(m)$, where $1 \leq m \leq M$, for some positive integer M. For convenience, we denote the START state in what follows by $S(0)$. We also use a sequence $r(0)$, $r(1)$, ..., $r(m)$, in which $r(i)$ is the index of the state selected (perhaps among several) to be $S(i)$. Evidently, $r(0) = 0$.

3. Each of the states $S(i)$, $1 \leq i \leq m$, is obtained as follows. Beginning at $S(0)$, n states can be reached, denoted by $a_{r(0),1}(1)$, $a_{r(0),2}(1)$, ..., $a_{r(0),n}(1)$, where n is a positive integer larger than 1. From among these states, one is selected as $S(1)$, say $a_{r(0),k}(1)$, where $1 \leq k \leq n$. Thus, $r(1) = k$. Now, starting at $S(1)$, n new states can be reached,

namely, $a_{r(1),1}(2)$, $a_{r(1),2}(2)$, …, $a_{r(1),n}(2)$, and one of these is chosen as $S(2)$. This continues until $S(m)$ is selected among $a_{r(m-1),1}(m)$, $a_{r(m-1),2}(m)$, …, $a_{r(m-1),n}(m)$. Now $S(m)$ leads directly to FINISH. This is illustrated in Fig. 1.

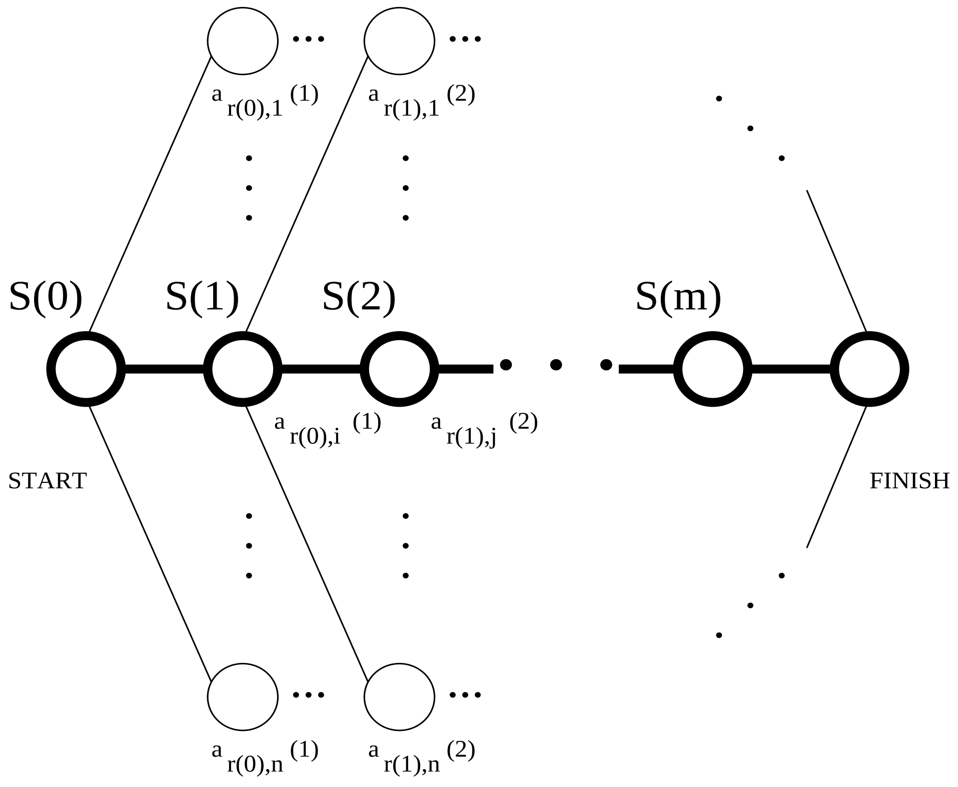

Figure 1: State space traversal.

4. The sequence $S(1)$, $S(2)$, …, $S(m)$ satisfies the following properties:

   (a) Regardless of how $S(i)$ is selected, $1 \leq i \leq m$, a sequence of transformations always exists that leads from START to FINISH.

   (b) The value of m is neither a constant, nor is it known in advance. For some choices of $S(i)$, the sequence $S(1)$, $S(2)$, …, $S(m)$ may be larger or shorter than for other choices. However, m is at least 1 and at most M, for some given positive integer M (as stated earlier).

   (c) A function $d(x, y)$ measures the distance from one state x to another state y as the number of intermediate states on a shortest path from x to y. Suppose that for some i, where $2 \leq i \leq m$, the same rule is used to obtain $S(i-1)$ and $S(i)$. Then, regardless of how $S(i-1)$ and $S(i)$ are computed, $S(i)$ is guaranteed to be closer to FINISH than $S(i-1)$, for $i \geq 2$, that is, $d(S(i), \text{FINISH}) < d(S(i-1), \text{FINISH})$. In other words, the structure of the problem is such that, when one moves from state $S(i-1)$ to state $S(i)$ the distance to the goal state is always reduced (it never stays the same or grows), provided that the algorithm used to reach $S(i-1)$ is also used to reach $S(i)$. In Fig. 1, $S(i)$ is closer to FINISH than $S(i-1)$ for $i = 1, 2, \ldots, m$.

Henceforth, we refer to this computation as State Space Traversal (SST) [9]. Note that SST describes a computational framework (that is, a setting within which a family of computations sharing a set of properties can be defined), rather than a specific computational problem. It may be helpful in what follows to think of the index i, where $1 \leq i \leq m$, as denoting the ith *stage* of the state space. Thus $S(i)$ is the state reached in the ith stage of the computation.

4 DISCRETE STEEPEST DESCENT

We begin we a simple metaphor. A skier at the top of a mountain wants to reach the valley as quickly as possible. The fastest way to accomplish this is to take the path of *steepest descent*. All other routes lead to the same destination but take more time.

Now back to our problem, suppose that we wish to go from START to FINISH while visiting the fewest number of intermediate states. In other words, m is to be minimized. To this end we can choose $S(i)$, for every i, $1 \leq i \leq m$, as follows:

1. There are at most n states, namely,

$$a_{r(i-1),1}(i), a_{r(i-1),2}(i), \ldots, a_{r(i-1),n}(i),$$

   to choose from, where $1 \leq r(i-1) \leq n$.

2. Among the states that can be reached, we select $S(i)$ as $a_{r(i-1),k}(i)$, where $1 \leq k \leq n$, such that

$$d(a_{r(i-1),k}(i), \text{FINISH})$$

   is minimum.

The above approach is referred to as the *Steepest Descent Algorithm* (SDA). It is a discrete version of the well-known *steepest descent method* [23] and a variant of *best-first search* [22] and the *greedy algorithm* [14]. We assume that when SDA is used, as described above, $m = m_{\text{SDA}} = n$. Otherwise, $m = M = \Omega(2^n)$. In Fig. 1, the sequence $S(1)$, $S(2)$, …, $S(m)$ selected by SDA is shown with thick lines. It is important to note that if for any i, $i \geq 1$, $S(i)$ is not chosen as the state for which the distance to FINISH is the smallest, then, as a result, the number of stages of the computation is at least exponential in n.

5 STATE SPACE TRAVERSAL IN REAL TIME

In a real-time environment, SST has the same characteristics described in Section 3 with one important difference: The reachable states change as the computation proceeds. Specifically,

1. We assume that $\mathcal{N} = n$ and $\mathcal{T} = c_{\mathcal{T}} \log n$, for a positive integer $c_{\mathcal{T}}$.
2. Each stage of the SST computation is identified with a time interval. Thus, $S(i)$ is reached during the ith time interval, for $1 \leq i \leq m$.
3. All but one of the states in the set $\{a_{r(i-1),1}(i), a_{r(i-1),2}(i), \ldots, a_{r(i-1),n}(i)\}$ that are reachable during time interval i, become unreachable during time interval $i+1$ (and thereafter).
4. It is not possible to predict which states become unreachable.
5. State $S(i)$, selected during time interval i, is the only state still reachable during time interval $i+1$. This state is determined in one of two ways as follows:
   (a) Either $S(i)$ has indeed been selected and reached by the algorithm during time interval i,
   (b) Or the algorithm fails to notify the source of its choice during time interval i and the external source selects $S(i)$ arbitrarily.
6. It is the role of the external source to provide all the states reachable from $S(i)$ during time interval $i+1$.

As a result, none of the potential intermediate states $a_{j,k}(i)$, where $1 \leq j, k \leq n$ and $1 \leq i \leq m$, are known in advance.

With these observations, we now define the real-time version of SST:

1. At the end of time interval i, where $i \geq 1$,
   (a) Either the algorithm informs the external source of its choice for $S(i)$; for example, $S(i)$ is $a_{r(i-1),j}(i)$, for some $1 \leq j \leq n$, which means that $r(i) = j$.
   (b) Or (the algorithm having failed to inform the source of its choice) the external source selects $a_{r(i-1),j}(i)$ arbitrarily.
2. At the beginning of time interval $i+1$, where $i \geq 1$, the external source
   (a) Makes all states $a_{r(i-1),\ell}(i)$, where $1 \leq \ell \leq n$ and $\ell \neq j$, unreachable, and

   (b) Transmits to the algorithm all the states $a_{r(i),k}(i+1)$, for $1 \leq k \leq n$, that are reachable from $S(i)$.

3. During time interval $i+1$, where $i \geq 1$, $S(i+1)$ is selected.

Note that there are no explicit deadlines to be met. The only difference between one solution and another is the total length of time required to go from START to FINISH. An implicit deadline is imposed by the source which needs to be told, by the end of time interval i, what values to send at the beginning of time interval $i+1$.

6 MODELS OF COMPUTATION

In order to study the effect of parallelism on real-time state space traversal, we define two distinct models of computation to solve the problem at hand.

The first model of computation is *sequential*: It consists of one processor along with some memory and internal registers. This is the standard model used in conventional algorithm analysis. Despite its familiarity, however, there are many instances of this model. To be specific, in what follows we assume that the Random Access Machine (RAM) version of the model [5] is the one used to solve the SST problem sequentially. For a given i, where $1 \leq i \leq m$, this model requires $c_{\mathcal{S}}n$ time units, where $c_{\mathcal{S}}$ is a positive constant, in order to perform the following two steps:

1. Compute $d(a_{r(i-1),j}(i), \text{FINISH})$, for all $1 \leq j \leq n$, and

2. Find the minimum of the n values computed in step 1.

Note that $c_{\mathcal{S}}n > \mathcal{T}$. In what follows, we assume for simplicity that $c_{\mathcal{S}} = c_{\mathcal{T}}$.

The second model of computation is *parallel*: It consists of several processors, each executing its algorithm. These processors work simultaneously on the solution to a computational problem. Here, again, there are many options from which to choose. Our only requirement is that the model be able to execute steps 1 and 2 above in $c_{\mathcal{P}} \log n$ time units, where $c_{\mathcal{P}}$ is a positive constant no larger than $c_{\mathcal{T}}$. Possibly the simplest model satisfying this condition is the (complete) *binary tree of processors*, with n leaf processors [5]. Note here that the number of processors used by the parallel model (that is, $2n-1$) is a linear function of the size of the problem (namely, the number n of states received from the external source at the beginning of each time interval). Again for simplicity, we assume in what follows that $c_{\mathcal{P}} = c_{\mathcal{T}}$.

Both models described use the same type of processors. In particular, the sequential processor has the same computational capabilities as each parallel processor. Furthermore, both processors run at the same speed (which we assume to be the maximum speed possible theoretically). Finally, note that we adopt for each processor the common definition of *time unit*, as given in Section 2.1. This is the unit traditionally used to measure the running time of an algorithm [5]. 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 RAM and each processor on the binary tree).

7 SEQUENTIAL SOLUTION

In order to compute $S(i)$, for any $i \geq 1$, the sequential algorithm needs to determine among

$$a_{r(i-1),1}(i), a_{r(i-1),2}(i), \ldots, a_{r(i-1),n}(i)$$

that state $a_{r(i-1),k}(i)$, for which $d(a_{r(i-1),k}(i), \text{FINISH})$ is the smallest. This would require $c_{\mathcal{S}}n$ time units. However, because $\mathcal{T} = c_{\mathcal{T}} \log n$, only $\log n$ of the n candidate states can be examined and one of them chosen as $S(i)$. The probability that the latter is indeed $a_{r(i-1),k}(i)$ equals $\log n/n$. Therefore, with probability approaching 1, the number of stages required in this case to reach FINISH is $\Omega(2^n)$.

8 PARALLEL SOLUTION

For each set of states

$$a_{r(i-1),1}(i), a_{r(i-1),2}(i), \ldots, a_{r(i-1),n}(i)$$

received from the external source at the beginning of time interval i, the parallel algorithm computes $S(i)$ as that state $a_{r(i-1),k}(i)$, for which $d(a_{r(i-1),k}(i), \text{FINISH})$ is the smallest. This requires $c_{\mathcal{P}} \log n$. Since $\mathcal{T} = c_{\mathcal{T}} \log n$, and $c_{\mathcal{P}} = c_{\mathcal{T}}$, the entire computation requires n stages.

9 CONCLUSION

The analyses of Sections 7 and 8 show that with overwhelming probability, the ratio of the number of stages traversed sequentially to the number of stages traversed in parallel is exponential in the number of processors used by the parallel algorithm. This suggests that the number of stages m may serve to compare the performances of the sequential and parallel algorithms, respectively. Indeed, the number of stages traversed by an algorithm can be interpreted in general as a measure of the *value* or *worth* of a computed solution. It is not difficult to see that the meaning of m can be specialized to fit the needs of a particular analysis. Thus, for example:

1. In some computational problems, we may want m to represent the *running time* of the solution. In these cases, a smaller number of stages means a faster and less expensive solution to compute.

2. In other circumstances, m may be related to the *timeliness* of a solution. A result requiring a large number of computational stages would be out-of-date in this context.

3. Finally, m may stand for the *quality* of a solution. Here, the value of a solution may decrease with an increasing number of stages traversed.

The following comments are now in order regarding some of the assumptions made in this paper.

1. The state space defined in Section 3 has two charateristics: Every state has n successor states and there is always a path from any state to the FINISH state. Both of these assumptions were made to simplify the analysis. The first assumption guarantees that all paths will offer the same computational challenge to every algorithm. The second assumption guarantees that all algorithms (particularly those not using steepest descent) will be able to obtain a solution. Clearly, both of these conditions can be easily lifted, leading to a more general definition of the state space.

2. In Section 5 we assumed that the length of a time interval, namely, $\mathcal{T}$ is $c_{\mathcal{T}} \log n$, for a positive integer $c_{\mathcal{T}}$. The value of $\mathcal{T}$ is the implicit deadline in the computation: At the end of the ith time interval the algorithm must inform the external source of its choice for $S(i)$. It may be argued that in some applications such a time interval is too long for large values of n and hence the computation may not qualify as being performed in 'real time'. This point is a valid one: While in most real-life computations the size of the problem does not grow without bound (and therefore $\log n$ is usually very small for all practical purposes), it is indeed true that in many circumstances a tighter deadline (in fact a constant-time one) may be required. We note here that our choice of $\mathcal{T} = c_{\mathcal{T}} \log n$ was motivated by a desire to use as weak a model of parallel computation as possible while still obtaining a reasonable performance (the idea being that the weaker the model, the more powerful the result). As it turns out, the weakest eligible model is one that computes a simple function of n variables (specifically, the minimum) in time that grows as the logarithm of n. Evidently, a stronger model, that is, one that can compute a function of this sort in constant time, could also be used. Such a model would easily meet a constant-time deadline.

3. As described in Section 4, the steepest descent algorithm chooses $S(i)$ among $a_{r(i-1),1}(i)$, $a_{r(i-1),2}(i)$, …, $a_{r(i-1),n}(i)$, as the state $a_{r(i-1),k}(i)$, for which $d(a_{r(i-1),k}(i), \text{FINISH})$ is minimum. One further generalization can be obtained by extending the way in which the next state $S(i)$ is computed in the specification of SST. For example, we may have:

$$S(i) = \mathcal{F}(a_{r(i-1),1}(i), a_{r(i-1),2}(i), \ldots, a_{r(i-1),n}(i)),$$

for some function $\mathcal{F}$. Thus, $\mathcal{F}$ may be defined such that $S(i)$ is equal to one of the arguments of $\mathcal{F}$; for instance $\mathcal{F}$ may return that state which minimizes the distance to FINISH (as in Section 4). In other cases, $S(i)$ is obtained by combining the values of the arguments of $\mathcal{F}$; for example, $S(i)$ may be the average of these values. Regardless of how $\mathcal{F}$ is defined, however, the following holds:

   (a) If $S(i)$ is obtained by applying $\mathcal{F}$ to *all* of the $a_{r(i-1),k}(i)$, for $1 \leq k \leq n$, then $m = n$.

   (b) If $S(i)$ is obtained by any other means, then $m = 2^n$. Specifically, if $S(i)$ is computed by applying $\mathcal{F}$ to *some* of the $a_{r(i-1),k}(i)$'s, or by choosing $S(i)$ arbitrarily as one of the $a_{r(i-1),k}(i)$'s then $m = 2^n$.

It remains as an interesting avenue for future research to uncover additional practical problems that fit the general framework for real-time computation proposed in this paper.

10 Acknowledgments

I would like to thank Stefan Bruda, Marius Nagy, and Amber Simpson for several useful comments.

References

[1] S.G. Akl, Superlinear performance in real-time parallel computation, *Proceedings of the Thirteenth Conference on Parallel and Distributed Computing and Systems*, Anaheim, California, August 2001, pp. 505–514.

[2] S.G. Akl, Nonlinearity, maximization, and parallel real-time computation, *Proceedings of the Twelfth Conference on Parallel and Distributed Computing and Systems*, Las Vegas, Nevada, November 2000, pp. 31–36.

[3] 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, pp. 2–11.

[4] S.G. Akl, Secure file transfer: A computational analog to the furniture moving paradigm, *Proceedings of the Eleventh Conference on Parallel and Distributed Computing and Systems*, Cambridge, Massachusetts, November 1999, pp. 227–233.

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

[6] S.G. Akl and S.D. Bruda, Improving a solution's quality through parallel processing, *The Journal of Supercomputing*, Vol. 19, No. 2, 2001, pp. 219–231.

[7] S.G. Akl and S.D. Bruda, Parallel real-time numerical computation: Beyond speedup III, *International Journal of Computers and their Applications*, Special Issue on High Performance Computing Systems, Vol. 7, No. 1, March 2000, pp. 31–38.

[8] S.G. Akl and S.D. Bruda, Parallel real-time optimization: Beyond speedup, *Parallel Processing Letters*, Vol. 9, No. 4, December 1999, pp. 499–509.

[9] A. Barr and E.A. Feigenbaum, Eds., *The Handbook of Artificial Intelligence*, Vol. I, William Kaufmann, Los Altos, California, 1981.

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

[11] S.D. Bruda and S.G. Akl, On the necessity of formal models for real-time parallel computations, *Parallel Processing Letters*, Vol. 11, Nos. 2 & 3, June & September 2001, pp. 353–361.

[12] S.D. Bruda and S.G. Akl, Pursuit and evasion on a ring: An infinite hierarchy for parallel real-time systems, to appear in *Theory of Computing Systems.*

[13] S.D. Bruda and S.G. Akl, A case study in real-time parallel computation: Correcting algorithms, *Journal of Parallel and Distributed Computing*, Vol. 61, No. 5, May 2001, pp. 688–708.

[14] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms*, The MIT Press, Cambridge, Massachusetts, 2001.

[15] M. Gendreau, G. Laporte, and F. Semet, A dynamic model and parallel tabu search heuristic for real-time ambulance relocation, *Parallel Computing*, Vol. 27, 2001, pp. 1641–1653.

[16] C.M. Krishna and Kang G. Shin, *Real-Time Systems*, Mc-Graw Hill, New York, 1997.

[17] 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, pp. 188–193.

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

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

[20] N. Nagy and S.G., Akl, The maximum flow problem: A real-time approach, *Proceedings of the Thirteenth Conference on Parallel and Distributed Computing and Systems*, Anaheim, California, August 2001, pp. 515–525.

[21] M. Nagy and S.G. Akl, Real-time minimum vertex cover for two-terminal series-parallel graphs, *Proceedings of the Thirteenth Conference on Parallel and Distributed Computing and Systems*, Anaheim, California, August 2001, pp. 526–534.

[22] J. Pearl, *Heuristics*, Addison-Wesley, Reading, Massachusetts, 1984.

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