Technical Report No. 98-420

A Case Study in Real–Time Parallel Computation: Correcting Algorithms *

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

January 8, 1999

Abstract

A *correcting* algorithm is one that receives an endless stream of corrections to its initial input data and terminates when all the corrections received have been taken into account. We give a characterization of correcting algorithms based on the theory of *data–accumulating* algorithms. In particular, it is shown that any correcting algorithm exhibits superunitary behavior in a parallel computation setting if and only if the static counterpart of that correcting algorithm manifests a strictly superunitary speedup. Since both classes of correcting and data–accumulating algorithms are included in the more general class of *real–time* algorithms, we show in fact that many problems from this class manifest superunitary behavior. Moreover, we give an example of a real–time parallel computation that pertains to neither of the two classes studied (namely, correcting and data–accumulating algorithms), but still manifests superunitary behavior.

Because of the aforementioned results, the usual measures of performance for parallel algorithms (that is, speedup and efficiency) lose much of their ability to convey effectively the nature of the phenomenon taking place in the real–time case. We propose therefore a more expressive measure that captures all the relevant parameters of the computation. Our proposal is made in terms of a graphical representation. We state as an open problem the investigation of such a measure, including finding an analytical form for it.

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

1 Introduction

A well-known result concerning the limits of parallel computation, and called the *speedup theorem*, states that when two or more processors are applied to solve a given computational problem, the decrease in running time is at most proportional to the increase in the number of processors [4, 12]. Since, according to this result, the ratio of the speedup achieved to the number of processors used is at most 1, the behavior of any parallel algorithm is at most *unitary*. By contrast, if a parallel algorithm were to be found which provides a speedup larger than the number of processors it uses, then this algorithm would contradict the speedup theorem and would be said to exhibit *superunitary* behavior.

One of the principal shortcomings of the speedup theorem is that it is stated for, and applies solely to, a conventional computing environment, where all the input is available at the beginning of the computation, and no restrictions are imposed on the completion time of the algorithm. In particular, such a conventional environment does not assume any *real–time* constraints. Typically, such constraints are divided into two main classes: (i) the jobs performed by an algorithm have real–time deadlines, and (ii) the entire input set is not available initially, so that more data arrive in real-time during the computation.

The first of these classes was considered from the point of view of the parallel computation in [1, 2], where it is shown that such constraints can lead to superunitary behavior. The study of the second class of constraints began in [8], where the class of *data–accumulating algorithms* (or *d-algorithms* for short) is introduced. In this paradigm, data that arrive during the computation are considered as an endless stream, the algorithm terminating when all the currently arrived data have been processed. The study of d-algorithms was continued in [5, 6], where it is shown that such an algorithm manifests superunitary behavior if and only if the parallel speedup for the static case is strictly larger than 1. Moreover, an interesting link between real–time input and real–time deadlines was discovered in [5]: For each d-algorithm there is a finite deadline (that is, if the running time of the algorithm exceeds some finite amount that does not depend on the shape of the data arrival law, then that algorithm never terminates).

Two extensions of the d-algorithm class are proposed in [8]. The first extension concerns *maintenance algorithms* (or *m-algorithms*). An m-algorithm is one that works on an endless, real–time, input stream, such that its work is divided into stages of fixed time length. At the beginning of each stage, an m-algorithm processes the data that arrived during the previous stage (updating some data structure), after which it releases the control to another application, that performs some work on the updated data structure. The analysis of m-algorithms can be easily performed once the theory of d-algorithms is established, as shown in [8].

The second extension is given in [8] as the class of *correcting algorithms* (*c-algorithms* henceforth), where the input data are available at the beginning of the computation, but there is an endless stream of corrections to the initial input set. Again, a c-algorithm terminates when all the arrived corrections have been taken into account. It is mentioned in [8] that the same approach can be used to study such algorithms as the one applied to d-algorithms. Indeed, we start our paper by reducing the class of c-algorithms to that of d-algorithms. Therefore, c-algorithms have the same properties as d-algorithms. In particular, they manifest superunitary behavior if and only if the static case offers even the slightest speedup (that is, a speedup larger than 1).

We believe that there are many practical uses of c-algorithms, examples of which are presented in this paper, namely, c-algorithms for algebraic computations and for graph processing. An interesting subclass is also identified in which the process of updating the solution when a correction arrives is no faster than rebuilding the solution from scratch.

Since the speedup can grow unbounded for parallel c-algorithms, and this growth depends on the arrival law, the usual measures of performance (that is, speedup and efficiency) lose most of their ability to convey effectively the nature of the phenomenon which occurs in this case. This is mainly because these measures of performance fail to take into consideration the parameters of the real–time input arrival law. Hence, we propose a new and more expressive measure of performance, in the form of a graphical representation. However, an open problem suggested by this paper is to find an analytical form corresponding to the proposed measure, and to assess its effectiveness. We believe that such a measure will shed a new light on the analysis of real-time parallel algorithms.

The paper is organized as follows. In section 2 the relevant concepts concerning the traditional limits on the power of parallel algorithms are presented, along with a definition of the class of d-algorithms. In section 3 we formally define the notion of a c-algorithm, and analyze this class in section 4. Two classes of problems are described in section 5 that are important—both for practical and theoretical reasons—in the context of our study of c-algorithms. Parallel computation in a general real-time environment is the subject of section 6. A paradigm involving a set of objects in perpetual motion is discussed and a new measure of performance for parallel real-time computations is proposed. Conclusions and open problems are offered in section 7.

In the following, a proposition is a result proved elsewhere. Whenever possible, the name of the original theorem is provided. The word "iff" stands for the phrase "if and only if". For definiteness, we use the Random Access Machine (RAM) and the COMBINING Concurrent-Read Concurrent-Write Parallel Random Access Machine (CRCW PRAM), throughout the paper as our models of sequential and parallel computation, respectively [1].

2 Preliminaries

The most common measure of the performance of a parallel algorithm is the *speedup*. Given some problem Π, the speedup of an algorithm that uses p_1 processors over an algorithm that uses p_2 processors with respect to problem Π is the ratio $S(p_2, p_1) = T_\Pi(p_2)/T_\Pi(p_1)$, $p_1 > p_2 > 0$, where $T_\Pi(x)$ is the running time of the best x-processor algorithm that solves Π. In many cases, this definition is used to compare a parallel algorithm with a sequential one, that is, $p_2 = 1$ [1].

Efficiency is usually defined as the ratio of the speedup over the number of processors, that is, $E(1,p) = S(1,p)/p$ [1]. Henceforth we omit the first argument and denote the efficiency simply by $E(p)$. However, in some cases [2], efficiency is defined in a more general way: $E(p_1,p_2) = \frac{T_\Pi(p_1)}{T_\Pi(p_2)} \times \frac{p_1}{p_2}$, where T_Π is defined as above.

In a conventional environment, the speedup (and hence the efficiency) of a parallel algorithm is bounded by the following results [1, 4, 12].

Proposition 2.1 (Speedup Theorem [12]) *$S(1,p) \leq p$ or, equivalently, $E(1,p) \leq 1$.* □

Proposition 2.2 (Brent's Theorem [4]) *If a computation C can be performed in time t with q operations and sufficiently many processors that perform arithmetic operations in unit time, then C can be performed in time $t + (q-1)/p$ with p such processors.* □

These two propositions state that, when more processors solve a given problem, the increase in performance is at most proportional to the increase in the number of processors. However, both of these 'theorems' fail in a number of unconventional computational environments, as shown in [1, 2, 8]. Specifically, the bounds that they set do not hold in such environments. Thus, for example, a speedup asymptotically larger than p (that is, a *superlinear speedup*) is possible with p processors in certain real-time computations, indicating that the speedup theorem is too pessimistic. Similarly, a 'slowdown' asymptotically larger than q/p (that is, a *superlinear slowdown*) can be incurred, pointing to the fact that Brent's theorem is too optimistic.

Since our study of c-algorithms relies heavily on the theory of d-algorithms, we summarize here the relevant properties of the latter. Conforming to [8], a standard algorithm, working on a non-varying set of data, is referred to as a *static* algorithm. On the other hand, an algorithm for which the input data arrive while the computation is in progress, and the computation terminates when all the currently arrived data have been treated, is called a *d-algorithm*. The size of the set of processed data is denoted by N. The arrival law is denoted by $f(n,t)$, where n denotes the number of input data available at the beginning of the computation, and t denotes the time.

It is important to emphasize the difference between the time complexity and the running time in the data-accumulating paradigm. For a d-algorithm, we denote by $C(N)$ the *time complexity* (or just *complexity* for short), which is a function of the amount of processed data N. However, since N itself is a function of time, the actual *running time*, denoted by t, is obtained by solving an implicit equation of the form $t = C(N)$. Similarly, the parallel time complexity is different from the parallel running time, the former being denoted by $C_p(N)$, and the latter by t_p. (In general, we emphasize the parallel case by adding the subscript p.) Note that, in the static case, the time complexity and the running time, as defined here, are identical.

The form proposed in [8] for the data arrival law is

$$f(n,t) = n + kn^{\gamma}t^{\beta}, \tag{1}$$

where k, γ, and β are positive constants. In the case of a sequential d-algorithm, the running time is given by the solution of the following implicit equation [8]

$$t = c_d(n + kn^{\gamma}t^{\beta})^{v}, \tag{2}$$

where the static counterpart of the d-algorithm in discussion has a complexity of $c_d n^v$, for a positive constant c_d.

In the parallel case, we have a similar equation for the running time:

$$t_p = \frac{c_{dp}(n + kn^{\gamma}t_p^{\beta})^{v}}{S'(1,P)}, \tag{3}$$

conforming to a result given in [8] and improved in [6], where $S'(1,P)$ is the speedup offered in the static case by an algorithm that uses P processors and c_{dp} is a positive constant.

The size of the whole input data set will be denoted by N_ω. Since the input data set is virtually endless in the data-accumulating paradigm, we will consider N_ω to be either large enough or tending to infinity.

The following two examples are mentioned in [7], but in another context. However, they fit the d-algorithms paradigm.

Example 2.1 Given two power series

$$\begin{aligned} g(x) &= g_0 + g_1 x + g_2 x^2 + \cdots, \\ h(x) &= h_0 + h_1 x + h_2 x^2 + \cdots, \end{aligned}$$

where g_i and h_i are real numbers for any nonnegative i, compute the sum $g+h$ of the two series. The coefficients g_i and h_i come in pairs (g_i, h_i) in real time, $N(\tau)$ such pairs arriving before time τ, where $N(\tau) = n + kn^{\gamma}\tau^{\beta}$. The result should be reported at that time t when a polynomial of degree $N(t)$ is available as a (partial) result. $\square$

The problem described in example 2.1 can be solved by a d-algorithm, since the result reaches the degree $N(t)$ only if $N(t)$ pairs of coefficients have been processed.

The coefficients of the sum of the two power series, denoted by s_i, are given by $s_i = g_i + h_i$. Therefore, the computation of the first n such coefficients takes n time units (one addition only is necessary for each such coefficient). The complexity in the static case in thus n. Then, the running time of the d-algorithm that solves the problem is given by $t = n^{\alpha} + kn^{\gamma}t^{\beta}$. We have hence a d-algorithm of linear complexity.

Example 2.2 Given two power series as in example 2.1, compute the product of the two series. The coefficients of the two series come in pairs, in real time, respecting the same arrival law as in example 2.1. The result should be reported at that time t when a polynomial of degree $N(t)^2$ is available as a (partial) result. □

The computation in example 2.2 is also one that can be performed by a d-algorithm since, in order to compute a result of degree $N(t)^2$, the first $N(t)$ coefficients of both series should be available. In order to compute the i-th coefficient r_i of the result one can use the Cauchy product rule: $r_i = \sum_{k=0}^{i} g_k h_{i-k}$. We need hence i multiplications and i additions in order to compute the i-th coefficient of the result. Therefore, the computation of the first n coefficients of the result is $\sum_{i=1}^{n} 2i = O(n^2)$. A d-algorithm that solves the problem has a running time $t = (n^\alpha + kn^\gamma t^\beta)^2$ and a quadratic complexity. Note that we cannot find a better algorithm, since the i-th coefficient of the product r_i depends on all the coefficients g_k and h_k, $0 \leq k \leq i$.

3 Correcting algorithms

A c-algorithm is defined in [8] as being an algorithm which works on an input data set of n elements, all available at the beginning of computation, but $V(n,t)$ variations of the n input data occur with time. For the sake of consistency with the analysis of d-algorithms, we denote by $f(n,t)$ the sum $n + V(n,t)$. We call the the function V the *corrections arrival law*.

We propose a definition of a c-algorithm that is similar to the one of a d-algorithm [6]:

Definition 3.1 (C-algorithm) An algorithm A is a c-algorithm if

1. A works on a set of n data which is available at the beginning of computation. However, $V(n,t)$ corrections to the initial input data occur with time, and A terminates at some time t when all the corrections received up to time t have been processed.
2. For any input data set, there is at least one corrections arrival law V such that, for any value of n, A terminates in finite time, where V has the following properties: (i) V is strictly increasing with respect to t, and (ii) $V(n, C(n)) > 0$, where $C(n)$ is the complexity of A. Moreover, A immediately terminates if the initial data set is null ($n = 0$).

□

The rationale behind this definition is the same as in the d-algorithm case: The first item is the definition introduced in [8], while the second one implies that there is at least one corrections arrival law such that a correction occurs before the algorithm in question has finished processing the initial input data.

Note that algorithms that correct themselves when their input changes have been studied [9, 10], but they do not fall within the theory of c-algorithms, since this theory assumes a real–time component, namely the corrections arrival law. We will call such algorithms, where the corrections arrival law is not considered, *dynamic algorithms*[1]. Also, if some c-algorithm A uses a dynamic algorithm for correction processing, that dynamic algorithm will be called the *static case* of A.

It remains to define the form of a correction in order to complete the definition of a c-algorithm. Generally, we consider that a correction consists of a tuple (i, v), where i denotes the index of the datum that is corrected, $1 \leq i \leq n$, and v is the new value for that datum.

An important consideration regards the quantity in terms of which the complexity analysis is expressed. It is not reasonable to give complexity results with respect to n only, since processing the corrections generally takes an important part of the computation time (it is consistent to assume that most of the data that are corrected are already considered; therefore, a c-algorithm will have at least to insert the new (corrected) datum into the solution; however, such an algorithm may also have to remove the effects of the old value). It seems more consistent, though, to consider the quantity $f(n,t)$ as the basis for complexity analysis, where t denotes the termination time. We will denote $f(n,t)$ by N, as in the case of d-algorithms.

We can modify examples 2.1 and 2.2 in order to fit the new paradigm of c-algorithms as follows.

Example 3.1 Given two polynomials of fixed degree n

$$\begin{aligned} g(x) &= g_0 + g_1 x + g_2 x^2 + \cdots + g_n x^n, \\ h(x) &= h_0 + h_1 x + h_2 x^2 + \cdots + h_n x^n, \end{aligned}$$

where g_i and h_i are real numbers for any i, $0 \leq i \leq n$, compute the sum $g+h$ of the two polynomials. The coefficients g_i and h_i are given at the beginning of the computation, but corrections to the coefficients arrive in real time, $V(n,t)$ such corrections arriving before time t, where $V(n,t) = kn^\gamma t^\beta$. The result should be reported at that time t when all the corrections received before t have been considered. □

Example 3.2 Given two polynomials as in example 3.1, compute the product of the two polynomials. The coefficients of g and h are given at the beginning of the computation, but corrections to them arrive in real time, $V(n,t)$ such corrections arriving before time t, where $V(n,t) = kn^\gamma t^\beta$. The result should be reported at that time t when all the corrections received before t have been considered. □

[1]Sometimes called "incremental algorithms".

4 Complexity

We analyze here the complexity of a general form of a c-algorithm. Throughout the rest of the paper, it is assumed that an algorithm which applies one update only has a complexity of $C_u(n)$, where $C_u(n) = c_u n^{\epsilon}$ and ϵ is a nonnegative constant[2]. Note that such a c-algorithm has to perform some initial processing. Moreover, such a processing may perform more computations than required for merely building the solution in the static case, in order to facilitate a fast update. Generally, we consider the computations that are performed before considering any correction as having a complexity of $C'(n) = cn^{\alpha}$, for some positive constants α and c. It is worth noting that the complexity of an update C_u is always upper bounded by C' since, in the worst case, the algorithm has to build a new solution for the whole data set, and the complexity of such a processing is given by C'.

Concerning the parallel case, we consider that the parallel speedup for both the initial processing and update algorithms is the same, and we denote it by $S'(1,P)$. This implies that the complexity of the parallel initial processing algorithm is $c_p n^{\alpha_p}$. Also, the complexity of the parallel update algorithm becomes $c_{up} n^{\epsilon_p}$. Moreover, we assume for easier calculations that the ratios $\frac{c_u}{c}$ and $\frac{c_{up}}{c_p}$ are equal.

Our analysis uses the following corrections arrival law (introduced in examples 3.1 and 3.2)

$$V(n,t) = kn^{\gamma} t^{\beta}, \tag{4}$$

since this was the arrival law mainly used when studying d-algorithms [5, 6].

4.1 Processing the corrections one at a time

We first consider the case in which the c-algorithm processes the incoming corrections one at a time.

Theorem 4.1 *Given a problem Π that requires an initial computation of complexity $C'(n) = cn^{\alpha}$, and an algorithm A_u for Π that processes an update in time $C_u(n) = c_u n^{\epsilon}$, $\epsilon \geq 0$, there is a c-algorithm A that solves Π. Moreover, the properties of A (namely, the time complexity, the running time, and the parallel speedup) working on n initial data and with the corrections arrival law given by (4) are the same as the properties of some d-algorithm A_d of complexity cN, which works on $n' = n^{\alpha}$ initial data and with the data arrival law $f(n,t) = n + k'n^{\gamma'} t^{\beta}$, $\gamma' = \frac{\gamma+\epsilon}{\alpha}$ and $k' = kc_u/c$.*

[2] It should be emphasized that we refer here to an updating algorithm which applies one update at a time. Therefore, the complexity of such an update depends only on the amount of data that is available at the beginning of the computation. (Note, however, that in section 4.2 we consider the case in which a bundle of updates are taken into account at once.)

Proof. We build the algorithm A, starting from the algorithm that performs the initial processing (denoted by A') and A_u. A will perform the following computations:

1. Compute the solution for the initial amount of data n, using A'. This takes $C'(n)$ time.
2. For each newly arrived correction, compute a new solution, using A_u. Terminate after this only if no new correction arrived while the solution is recomputed; otherwise, repeat this step. The complexity of this step is $C_u(n)$.

The complexity of A is then $C(N) = C'(n) + C_u(n)(N - n)$, and, considering the form of $C'(n)$ and $f(n,t)$, this gives the following implicit equation for the termination time:

$$\begin{aligned} t &= cn^\alpha + c_u n^\epsilon (n + kn^\gamma t^\beta - n) \\ &= c(n^\alpha + \frac{c_u}{c} kn^{\gamma+\epsilon} t^\beta). \end{aligned}$$

Then, considering the form of n', γ', and k', we reach the desired form:

$$t = c(n' + k'n'^{\gamma'} t^\beta). \tag{5}$$

But this is the same as the implicit equation for the termination time of a linear d-algorithm [8]. Therefore, the time complexity of A is $C(N) = cN$, $N = n' + k'n'^{\gamma'} t^\beta$, and the termination time for A is the same as for a d-algorithm of complexity $C(N)$, since both are solutions to the same equation.

With regard to the parallel implementation, considering that the static case of A provides a speedup $S'(1, P)$ when using P processors, we have, analogously, the following implicit equation for the running time:

$$t_p = \frac{c_p(n' + k'n'^{\gamma'} t^\beta)}{S'(1, P)}. \tag{6}$$

But, again, equation (6) is identical to the equation obtained in [6] for the parallel case of d-algorithms.

In order to complete the proof, we have to prove that A terminates for at least one corrections arrival law and for any value of n'. But it has been proved in [8] that a d-algorithm of complexity cN terminates for any initial amount n of input data if $\beta < 1$. □

Corollary 4.2 *With n', γ', and k' as in theorem 4.1,*

1. *For a problem admitting a sequential c-algorithm and a parallel P-processor c-algorithm, $P = \xi(n' + k'n'^{\gamma'} t_p^\beta)^\delta$, such that the speedup for the static case is $S'(1, P) = \xi_1(n' + k'n^{\gamma'} t_p^\beta)^s$, where $S'(1, P) > 1$ for any strictly positive values of n' and t_p, we have for $c_p/S'(1, P) < c$:*

$$\frac{t}{Pt_p} \to N_\omega \, for \, n \to \frac{1}{k'c},$$

where $\beta = \gamma' = 1$, and $0 \leq \delta \leq 1$, $0 \leq s \leq 1$. For all values of β and γ',

$$\frac{t}{t_p} > \frac{c}{c_p} S'(1, P).$$

2. *For any problem admitting a sequential c-algorithm and a parallel P-processor c-algorithm such that the speedup for the static case is $S'(1, P) = 1$, and for any data arrival law such that either $\beta \leq 1$, or $\gamma' \geq 1$ and $1/2 \leq k'c^\beta(\beta - 1)$, the speedup of the parallel c-algorithm is $S(1, P) = 1$.*

3. *For any problem admitting a sequential c-algorithm and a parallel P-processor c-algorithm such that the speedup for the static case is $S'(1, P) = 1$, and for any data arrival law such that either $\beta \leq 1$, or $\gamma' > 1$ and n' is large enough, the speedup of the parallel c-algorithm is $S(1, P) = 1$.*

Proof. These results were proved in [6] with respect to d-algorithms. But, conforming to theorem 4.1, the properties of c-algorithms are the same, hence the results follow immediately. □

Corollary 4.2 contradicts the speedup theorem. More precisely, the first item shows that the speedup of any c-algorithm grows unbounded as long as the static counterpart of that c-algorithm manifests a superunitary speedup. On the other hand, according to items 2 and 3, if the speedup for the static case is unitary, then the c-algorithm manifests unitary speedup as well.

Lets us refer to examples 3.1 and 3.2. The complexity of the initial processing in example 3.1 is n, since the coefficients s_i need to be computed. Then, each correction can be handled in unit time: Suppose that some coefficient g_i is corrected. Then, only the coefficient s_i of the product needs to be recomputed, and this is done by adding the new value of g_i to (the old value of) h_i. Therefore, the complexity of A_u for this problem is unitary. Then, the running time of the c-algorithm is given by $t = n^\alpha + kn^\gamma t^\beta$. One can note that the expression for the running time of the c-algorithm that solves the problem in example 3.1 is identical to the d-algorithm that solves the corresponding problem. The reason for this identity is that the update is fast, and no initial computation other than the computation of the initial solution is required.

On the other hand, in the case of the problem from example 3.2, assume that the value of some g_i is corrected. Since we deal with worst–case analysis, we can assume without loss of generality that $i \geq n/k$ for some constant k. But then all the coefficients r_j, $j \leq i$ should be recomputed, since all these coefficients depend on g_i. This computation requires $O(n^2)$ operations. Therefore, the running time is given by $t = n^2 + kn^{\gamma+2}t^\beta$. Note that the complexity of the update algorithm is the same as the complexity of the algorithm that rebuilds the solution from scratch. As we note in section 5.1, this is not a singular case.

4.2 Processing the corrections in bundles

One may wonder why a c-algorithm cannot consider the incoming corrections in bundles of size b instead of one by one. We now analyze this possibility.

We denote by A_u^b the best known algorithm that applies b corrections. Let the complexity of A_u^b be $C_u^b(n, b)$, and the complexity of A_u, the (best known) algorithm that applies one correction only, as in theorem 4.1, be $C_u(n)$.

Suppose that algorithm A of theorem 4.1 terminates at some time t for some initial data and some corrections arrival law. Further, let A_{bw} be an algorithm that performs the same processing as A, except that, instead of applying each correction immediately, it waits until some number b of corrections have been accumulated and applies them at once using algorithm A_u^b described above. Note that we do not impose any restriction on b; it may be either a constant, or a function of either n or t or both. For simplicity, we assume that the complexity of the initial processing $C'(n)$ is the same for both A and A_{bw}. We believe that this is a reasonable assumption, since the processes are essentially the same in the two c-algorithms. We have the following immediate result:

Lemma 4.3 *Given some solution σ to a problem Π and a set of input data of size n, let A_u^i be the best known algorithm that receives σ and i corrections, and computes a solution σ_u for the corrected input. Analogously, let A_u^j be an algorithm that behaves in the same way as A_u^i, except that it processes some set of j corrections, $j > i$. If we denote the complexity of A_u^i by C'_u, and the complexity of A_u^j by C''_u, then $C'_u \leq C''_u$.*

Proof. Assume that $C'_u > C''_u$. But then one can construct an algorithm for performing i corrections only in time C''_u as follows: When only i corrections are sought, apply A_u^j to perform the requested corrections, together with $j - i$ corrections that change nothing (that is, "correct" $j - i$ values to their original value). Technically speaking we have j corrections, and thus we can use A_u^j in order to perform them. But we obtained in this way an algorithm for updating i elements which is better than A_u^i, and this contradicts the assumption that A_u^i is the best known algorithm that performs such an update. □

Corollary 4.4 *Given some solution σ to a problem Π and a set of input data of size n, let A_u^b be the best known algorithm that receives σ and b corrections, and computes a solution σ_u for the corrected input. Also, let A_u (of complexity C_u) be the best known algorithm that receives σ and one correction only and returns the corresponding corrected solution. Then, $C_u^b \geq C_u$ for any strictly positive b.*

Proof. By induction over b, using lemma 4.3. □

The following upper bound on the update time for b corrections is also immediate:

Lemma 4.5 *Given some solution σ to a problem Π and a set of input data of size n, let A_u^b be the best known algorithm that receives σ and b corrections, and computes a solution σ_u for the corrected input. Also, let A_u (of complexity C_u) be the best known algorithm that receives σ and one correction only and returns the corresponding corrected solution. Then, $C_u^b \leq bC_u$ for any strictly positive b.*

Proof. Again, assuming that $C_u^b > bC_u$, one can build an algorithm that corrects σ when b input data change and which is better than A_u^b, by simply applying A_u b times, contradicting in this way the hypothesis that A_u^b is the best algorithm for this process. □

Note that the problem in example 3.1 does not admit an updating algorithm that processes a bundle of size b at a time, and whose running time is smaller than b times the time required for performing a single update. On the other hand, the updating algorithm for the problem in example 3.2 has a time complexity of $O(n^2)$ regardless of the number of corrections that are considered.

4.3 Optimality considerations

Recall that in theorem 4.1 an algorithm A is given which solves the corresponding problem Π correctly. However, nothing is said in the theorem about the optimality of A. It is important to note that, if A is not optimal, there is no guarantee that corollary 4.2 holds. This issue is now studied. We show in what follows that there are instances of the input for which there is indeed an algorithm better than A, but that corollary 4.2 holds for this new algorithm as well.

As stated at the beginning of section 4, we consider the complexity of A_u as being $C_u(n) = c_u n^\epsilon$. Also, let the complexity of A_u^b be $C_u^b(n) = c_u b^{\epsilon_b} n^\epsilon$. Then, corollary 4.4 and lemma 4.5 imply that ϵ_b is a positive number, no larger than 1.

Let A_b be a variant of algorithm A_{bw} for which the buffer size is correctly chosen such that A_b does not wait until b corrections have arrived, but instead processes whatever corrections are available whenever it has time to do so (that is, as soon as it finishes processing the previous bundle).

Lemma 4.6 *A_b is no better than A iff either $\epsilon_b = 1$, or $\beta > 1$, or $\beta = 1$ and $\beta k c_u n^{\epsilon+\gamma} \geq 1$.*

Proof. The running time t_b of A_b is given by

$$\begin{aligned} t_b &= cn^\alpha + \sum_{i=1}^{m} C_u^b(b_i, n), \\ &= cn^\alpha + c_u n^\epsilon \sum_{i=1}^{m} b_i^{\epsilon_b}, \end{aligned} \tag{7}$$

where m is the number of times A_u^b has been invoked, and, when A_u^b is invoked the i-th time, the size of the buffer b is b_i.

But, since $0 \leq \epsilon_b \leq 1$, and $b_i \geq 1$, $1 \leq i \leq m$, we have

$$\sum_{i=1}^{m} b_i^{\epsilon_b} \leq \sum_{i=1}^{m} b_i. \tag{8}$$

Also, note that exactly all the buffered corrections are processed, that is,

$$\sum_{i=1}^{m} b_i = V(n, t_b). \tag{9}$$

It follows from (7), (8) and (9) that

$$t_b \leq cn^{\alpha} + kc_u n^{\epsilon+\gamma} t_b^{\beta}. \tag{10}$$

On the other hand, we found in the proof of theorem 4.1 that the corresponding equation for the running time of A is

$$t = cn^{\alpha} + kc_u n^{\epsilon+\gamma} t^{\beta}. \tag{11}$$

Let $R(n,t)$ be the function $R(n,t) = cn^{\alpha} + kc_u n^{\epsilon+\gamma} t^{\beta} - t$. Then, from relation (10) we have

$$R(n, t_b) \geq 0,$$

and from relation (11) it follows that

$$R(n, t) = 0.$$

The above two relations yield

$$R(n,t) \leq R(n, t_b), \tag{12}$$

for any size of the initial input n.

Moreover, since there is at least one i such that $b_i > 1$, then $\sum_{i=1}^{m} b_i^{\epsilon_b} = \sum_{i=1}^{m} b_i$ iff $\epsilon_b = 1$. Therefore, we can further restrict relation (12) as follows:

$$R(n,t) < R(n,t_b) \quad \text{if} \quad \epsilon_b < 1, \tag{13}$$

$$R(n,t) = R(n,t_b) \quad \text{if} \quad \epsilon_b = 1. \tag{14}$$

The derivative of R with respect to t is

$$\frac{\partial R}{\partial t} = \beta k c_u n^{\epsilon+\gamma} t^{\beta-1} - 1.$$

For $\epsilon_b < 1$, considering relation (13), we have:

1. If $\beta > 1$, then $\frac{\partial R}{\partial t} > 0$ for t large enough. Therefore, for such t, R is strictly increasing with respect to its second argument. In this case, given relation (12), it follows that $t < t_b$. In other words, A_b is no better than A in this case.

2. If $\beta < 1$, then, $\frac{\partial R}{\partial t} < 0$ for t large enough. Hence, R is strictly decreasing with respect to t. Therefore, analogously, $t > t_b$. In this case, A is no better than A_b.

3. If $\beta = 1$, then $\frac{\partial R}{\partial t}$ is positive iff $\beta k c_u n^{\epsilon+\gamma} > 1$. Therefore, if $\beta k c_u n^{\epsilon+\gamma} \geq 1$, then we are in the same case as in item 1 above. Hence A_b is no better than A. Otherwise, analogously to item 2, A is no better than A_b.

On the other hand, when $\epsilon_b = 1$, relation (14) holds. But R is strictly monotonic with respect to its second argument. Therefore, in this case, $t = t_b$, and hence A_b is no better than A, their running times being, in fact, identical. □

We found therefore that A is optimal iff either $\epsilon_b = 1$, or $\beta > 1$, or $\beta = 1$ and $\beta k c_u n^{\epsilon+\gamma} \geq 1$. In the other cases, A_b may be better than A. Recall here that A_{bw} is the algorithm that waits for b corrections to have accumulated before processing them (unlike A_b which processes any available corrections as soon as it has time to do so). It is somewhat intuitive that A_{bw} is no better than A_b, but we give a proof for this below.

Lemma 4.7 *A_{bw} is no better than A_b when $\beta \leq 1$.*

Proof. We use the relations

$$\left(\sum_{i=1}^{m} b_i\right)^a \leq \sum_{i=1}^{m} b_i^a \tag{15}$$

for any $m > 0$, $b_i \geq 1$, $1 \leq i \leq m$, and $0 \leq a \leq 1$, and

$$\left(\sum_{i=1}^{m} b_i\right)^a \geq \sum_{i=1}^{m} b_i^a \tag{16}$$

for m and b_i as above, and $a \geq 1$. These relations can be easily proved by induction over m.

When speaking of A_{bw} we denote by m_w and b_{wi} the values that correspond to m and b_i in the case of A_b, respectively. We denote by t_{bw} the running time of A_{bw}, which has the form

$$t_{bw} = cn^{\alpha} + c_u n^{\epsilon} \sum_{j=1}^{m_w} b_{wj}^{\epsilon_b} + t_w, \tag{17}$$

where t_w is the extra time generated by the waiting process. We denote by t_{wi} the waiting time before each invocation of the algorithm that applies the corrections, and by a_i the number of corrections A_{bw} waits for before each such invocation. Note that $t_w = \sum_{i=1}^{m_w} t_{wi}$. Moreover, the number of corrections that arrive during t_{wi} is given by $a_i = kn^\gamma((t+tw_i)^\beta - t^\beta)$ for some positive t. But $(t+tw_i)^\beta - t^\beta \leq t_{wi}^\beta$ by relation (15) for $m = 2$. Therefore, $a_i \leq kn^\gamma t_{wi}^\beta$.

But, for any i, $1 \leq i \leq m_w$, $b_{wi+1} = kn^\gamma((t+t_{wpi})^\beta - t^\beta) + a_{i+1}$, where t_{wpi} represents the time in which the previous buffer of corrections is processed. But then $t_{wpi} = c_u n^\epsilon b_{wi}^{\epsilon_b}$, and, as above, $(t+t_{wpi})^\beta - t^\beta \leq t_{wpi}$. It follows that

$$b_{wi+1} \leq kn^\gamma (c_u n^\epsilon)^{\epsilon_b} b_{wi}^{\beta\epsilon_b} + kn^\gamma t_{wi}^\beta.$$

The above relation implies that $ab_{wi}^{\beta\epsilon_b} \geq b_{wi+1} - kn^\gamma t_{wi}^\beta$, where we denote $kn^\gamma(c_u n^\epsilon)^{\epsilon_b}$ by a. But, since $\beta \leq 1$ and $b_{wi} \geq 1$, $ab_{wi}^{\epsilon_b} \geq ab_{wi}^{\beta\epsilon_b}$. Therefore,

$$b_{wi}^{\epsilon_b} \geq \frac{b_{wi+1} - kn^\gamma t_{wi}^\beta}{a}.$$

Note that $\sum_{i=1}^{m_w} b_{wi} = kn^\gamma t_{bw}^\beta$, since all the corrections that arrive before the termination time must be processed. Then, by summation over i of the above relation we have

$$\sum_{i=1}^{m_w} b_{wi}^{\epsilon_b} \geq \frac{kn^\gamma}{a} t_{bw}^\beta - \frac{kn^\gamma}{a} \sum_{i=1}^{m_w} t_{wi}^\beta.$$

But then, from relation (17),

$$t_{bw} \geq cn^\alpha + \frac{b}{a} t_{bw}^\beta - \frac{b}{a} \sum_{i=1}^{m_w} t_{wi}^\beta + t_w, \tag{18}$$

where $b = c_u n^\epsilon kn^\gamma$. Considering relations (18) and (10) we have

$$g(t_{bw}) - g(t_b) \geq t_w - \frac{b}{a} \sum_{i=1}^{m_w} t_{wi}^\beta + \left(b - \frac{b}{a}\right) t_{bw}^\beta, \tag{19}$$

where $g(t) = t - (b/a)t^\beta$. Let us denote the right hand side of relation (19) by G. If $G \geq 0$, then relation (19) leads to $g(t_{bw}) \geq g(t_b)$ and then it is immediate that $t_{bw} \geq t_b$, since g is an increasing function for any $t \geq 1$. Therefore, in order to complete the proof, we show that G is positive.

The time between the arrivals of two consecutive corrections at some time t is

$$\Delta(n,t) = \left(\frac{1}{kn^\gamma} + t^\beta\right)^{1/\beta} - t. \tag{20}$$

We can assume without loss of generality that A_{bw} waits for d corrections, where $d \geq 2$, and at least two corrections are waited for at some time $t_i \geq t_{bw}/2$, $1 \leq i \leq d$ (otherwise, A_{bw} is the same as A_b in the asymptotic case). Then,

$$
\begin{aligned}
t_w &\geq \sum_{i=1}^{d} \Delta(n, t_i) \\
&\geq \sum_{i=1}^{d} (1/kn^\gamma + t_i^\beta)^{1/\beta} - \sum_{i=1}^{d} t_i^\beta \\
&\geq \left(\sum_{i=1}^{d} t_i\right)^{1/\beta} - \sum_{i=1}^{d} t_i^\beta + d(1/kn^\gamma)^{1/\beta} \\
&\geq t_{bw} - t_{bw}^\beta. \qquad (21)
\end{aligned}
$$

Assume now that $t_{bw}^\beta < \sum_{i=1}^{m_w} t_{wi}^\beta$. Then, $kn^\gamma t_{bw}^\beta < \sum_{i=1}^{m_w} kn^\gamma t_{wi}^\beta$, therefore the number of corrections processed by A_{bw} is smaller than the number of corrections for which A_{bw} waits, which is absurd. Therefore

$$
bt_{bw}^\beta \geq \frac{b}{a} \sum_{i=1}^{m_w} t_{wi}^\beta. \qquad (22)
$$

Considering relations (21) and (22), we have

$$
G \geq t_{bw} - \left(1 + \frac{b}{a}\right) t_{bw}^\beta.
$$

But then G is obviously positive for t_{bw} large enough, since the left hand side of the above relation tends to infinity when t_{bw} tends to infinity. This completes the proof, since the fact that G is positive implies that $t_{bw} \geq t_b$ as shown in relation (19). □

Note that, for the problem in example 3.1, A is always optimal, since, for this case, $\epsilon_b = 1$. On the other hand, in the problem of example 3.2, $\epsilon_b = 0$, and hence A is not optimal in some cases (namely, when $\beta < 1$, or $\beta = 1$ and $\beta k c_u n^{\epsilon+\gamma} < 1$), for which the optimal algorithm is A_b, as shown by lemma 4.7.

The proof of lemma 4.7 contains yet another interesting result. We show there that $b_{wi+1} \leq ab_{wi}^{\beta\epsilon_b} + a_i$. Since A_b does not wait for any correction, we have analogously $b_i \leq ab_i^{\beta\epsilon_b}$. But, applying this relation recursively i times we observe that

$$
b_{i+1} \leq a^{(1-(\beta\epsilon_b)^i)/(1-\beta\epsilon_b)} (b_1)^{(\beta\epsilon_b)^i}.
$$

But $0 \leq \beta\epsilon_b \leq 1$, hence $b^{i+1} \leq b_1$. On the other hand, $b_1 = kn^\gamma (cn^\alpha)^\beta$, since the first invocation of the algorithm that processes the corrections happens exactly after the initial processing terminates, that is, after cn^α time units. Therefore, in the case of A_b, the size of the buffer is bounded by a quantity that does not depend on time, but only on n. Then, the running time of A_b is given by an equation of the form

$$
t_b = c(n'' + k'' n''^{\gamma''} t_b^\beta),
$$

which is similar to the form obtained in the case of A in theorem 4.1. Hence corollary 4.2 holds for A_b. Lemmas 4.6 and 4.7 therefore imply:

Theorem 4.8 *The c-algorithm of theorem 4.1 is optimal iff the corrections arrival law has the following property: Either $\beta > 1$, or $\beta = 1$ and $\beta k c_u n^{\epsilon+\gamma} \geq 1$. If this property does not hold, then the optimal algorithm is A_b, where A_b processes at once all the corrections that arrived and have not been processed yet.*

However, corollary 4.2 holds for A_b as well. □

5 Related Work

In the following, we consider two interesting classes of problems for which dynamic algorithms were studied, and analyze them in the environment offered by the c-algorithm paradigm.

5.1 Algebraic Problems

The theory of dynamic algorithms for algebraic problems has been recently investigated in [10], where the following lower bounds are established for a number of problems, each of size n:

Proposition 5.1 [10] *Any dynamic algorithm for the following problems requires $\Omega(n)$ time per update:*

- *polynomial evaluation at a single point,*
- *multipoint evaluation, where evaluation points are allowed to change,*
- *the chirp z-transform, where z is allowed to change,*
- *polynomial reciprocal,*
- *solving a triangular Toeplitz system of equations, and*
- *extended GCD problem.*

□

Each problem in proposition 5.1 is known to have an algorithm that solves the problem in the absence of any correction in $O(n)$ time. Therefore, none of these problems admits an update algorithm whose performance is better than the naive algorithm that rebuilds the whole solution using the corrected input. Moreover, based on proposition 5.1, it is mentioned in [10] that many other algebraic problems have this property.

Problems such as those in proposition 5.1, for which no fast update algorithm is possible, form an interesting class. For problems in this class, the algorithms implied by theorem 4.1 are not optimal. Indeed, the time complexity of the algorithm that performs however many corrections is the same as the complexity of an algorithm that performs only one correction. This complexity is the same as the complexity of the static algorithm for the given problem. Therefore, the algorithm of theorem 4.1 is not optimal, as shown in theorem 4.8.

We say that we have an "updating asynergy" in the case when applying many corrections at once is no better than applying those corrections independently. The algebraic problems described above form a rather large class where such an asynergy does not hold. However, to our knowledge, no other approach to applying many corrections at once has been previously considered in the literature (except for the trivial case in which the updating algorithm is no better than the algorithm that builds the solution from scratch).

5.2 Graph Problems

Dynamic algorithms for graph processing were investigated in [9], where the cost of such algorithms is measured with respect to the size of changes in the input (denoted by $||\delta||$) instead of the size of the input. A dynamic algorithm is called *bounded* if its complexity depends only on $||\delta||$, and *unbounded* otherwise [9].

The size of a change δ, that is, $||\delta||$, denotes the sum of the cardinalities of both the set of edges and vertices affected by that change. In other words, $||\delta||$ is a measure of the size of the change in the input. A computational hierarchy is found in [9]:

1. problems of polynomial complexity in $||\delta||$ are: attribute updating, priority ordering, all-pairs shortest path (with strictly positive edge weights), and single source/sink shortest path (with strictly positive edge weights),

2. problems whose complexities are exponential in $||\delta||$ are: circuit annotation and weighted circuit annotation, and

3. unbounded problems, such as single source/sink reachability, and single source/sink shortest path (with positive edge weights).

On the other hand, the key point in the theory of c-algorithms is the ability of a c-algorithm to terminate while receiving corrections in real time. Since $||\delta||$ is a parameter which depends on the specific correction that is applied, the corrections arrival law cannot be expressed in terms of $||\delta||$. Therefore, for the purpose of analyzing c-algorithms, this measure is inadequate. However, an interesting research direction that can put together the results from [9] and our results would be to see if δ can be bounded, even if for particular shapes of the input or for particular problems.

6 Parallel Real-Time Computations

Corollary 4.2 characterizes the parallel behavior of c-algorithms. More precisely, it shows that even the slightest superunitary speedup manifested in the static case grows unbounded in the data-accumulating environment. On the other hand, if the static case provides only unitary speedup, then no improvement can be obtained in the real-time environment.

These properties of c-algorithms were found previously to hold for various related computational settings [5, 6, 8]. In fact, we believe that there are many more paradigms in the context of real-time computations for which superunitary behavior manifests itself, as suggested by the following example.

6.1 A Moving Objects Paradigm

Within an $n \times n$ grid, where n is a positive integer, n objects are in perpetual movement. Each object moves from one grid point (x_1, y_1), $0 \le x_1, y_1 \le n-1$, to another grid point (x_2, y_2), $0 \le x_2, y_2 \le n-1$, according to a given equation of motion that is a function of elapsed time. Specifically, if the n objects occupy certain positions at time t, they all occupy new positions at time $t+1$.

If the location of an object at time t is known, it is possible to compute its location at time t', for $t' > t$, using the object's equation of motion. This computation requires $k_1 n^2$ unit-time operations, for some positive constant k_1.

Once the positions of the n objects are known at a specific moment in time, a function ϕ of these positions is to be computed. An example of such function is the centroid of the n positions. Computing ϕ requires $k_2 n$ unit-time operations, for some positive constant k_2. Note that, assuming that ϕ is reported at some time τ, then it should be computed with respect to the positions of the n objects at time τ.

Sequential solution. The sequential processor first scans the grid in $k_3 n^2$ time, where k_3 is a positive constant, to determine an initial location for each object. It then computes a new location for each object, that is, the new position reached by the object when all such computations of new positions have been performed and the function ϕ has been computed. Specifically, suppose that the initial position of the ith object was identified at time t_i, $1 \le t_i \le n^2$. A new position at time t_i' is computed, where $t_i' = t_i + (t_n - t_i) + (k_1 n^2 \times n) + k_2 n$. This requires $k_1 n^3$ unit-time operations. Finally, the function ϕ is computed in $k_2 n$ unit-time operations. The entire algorithm runs in $O(n^3)$ time.

Parallel solution. If n^2 processors are available, each examines one grid point. The n processors that locate the n objects collectively compute ϕ in k_2 time units. The entire algorithm takes k_4 unit-time operations for some positive constant k_4.

Speedup. The parallel algorithm offers a speedup of $O(n^3)/k_4 = O(n^3)$, and an efficiency of $O(n^3)/n^2 = O(n)$.

The moving objects paradigm thus achieves a superunitary speedup, clearly suggesting that the speedup theorem does not hold in real-time computations. It should be clear that this paradigm fits in a class which is in some sense the

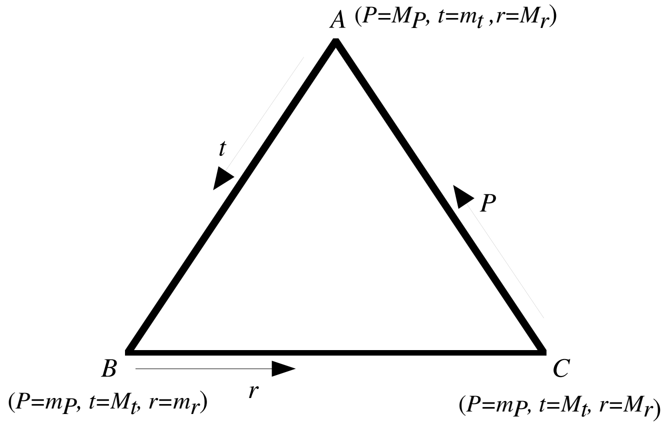

Figure 1: A representation of the relation between the number of processors, the data arrival rate, and the running time for a parallel algorithm.

complement of that of c-algorithms: Here, the corrections are known (through the motion functions), but the initial locations are not. However, it is interesting to note that if the initial locations of the moving objects are known, the speedup is still $O(n^3)$, while an improved efficiency of $O(n^2)$ results (the running times of both the sequential and parallel algorithms remain unchanged, but the number of processors required in the parallel case is only n, since the objects do not have to be found initially). Thus, this version of the problem also manifests superunitary behavior.

6.2 A Graphical Representation

In the classical study of parallel algorithms, the main measure of performance is the speedup (note that the efficiency measure derives from the speedup). Therefore, a graphical representation involves a two-dimensional plot in a Cartesian space: One coordinate represents the running time, and the second represents the number of processors. However, if the input data arrive in real-time, a third parameter must be considered, namely the arrival rate, which can substantially affect the performance. Therefore, we propose a new graphical representation. Such a representation was used in [3], but in another context (that is, to represent time, information, and energy, as originally proposed in [13]).

The diagram we propose is a triangle ABC, as in figure 1. On the edge AB the time increases from m_t (in A) to M_t, on the edge BC the arrival rate increases from m_r to M_r, and the number of processors increases from m_P to M_P on the edge CA, where m_i are some minimal and M_i some maximal values for each measure ($M_i > m_i$, $i \in \{t, r, P\}$). Each discrete point in this triangle represents a combination of these three quantities for a computer working on the problem.

It remains to establish how, given some triangular coordinates P, r, and t,

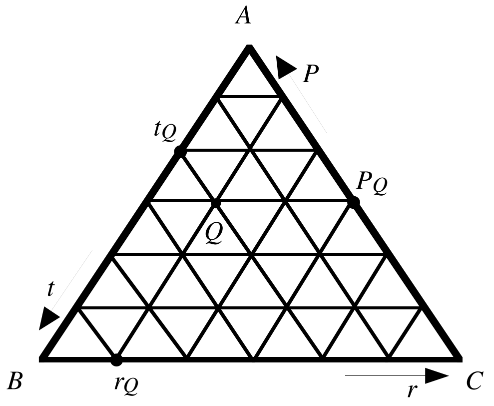

Figure 2: The main triangle divided into smaller triangles.

the point corresponding to this situation is determined. We propose two such methods:

1. The point be the center of the circle inscribed in the triangle whose vertices are the points corresponding to P, r, and t on the appropriate edges[3], or

2. The main triangle is divided is smaller triangles (as shown in figure 2). Then, based on this triangulation, the location of the point is established. Note that, given a point Q, each of the lines $\overline{Qt_Q}$, $\overline{QP_Q}$, and $\overline{Qr_Q}$ intersect the edges of the triangle twice. One should establish therefore a consistent method for determining which of the two intersection points establishes the coordinate of Q. In the example from figure 2, the intersection points closer to A (for the edge AB), to B (for BC), and to C (for CA) were chosen, respectively.

The first method seems more likely to lead to an analytical expression, but the second one is simpler.

Note that, if the time or the number of processors are straightforward measures, one needs to find a meaningful measure for the arrival rate. In the c-algorithms or d-algorithms case, this measure should be a function of n, γ, and β, since these are the parameters of the arrival law.

For example, A (minimum time, maximum rate, maximum number of processors) may represent a parallel computer with a large number of processors receiving the input at a high rate (e.g., β is large), and solving the problem at hand in minimum time and optimal cost. The vertex B (maximum time, minimum data, minimum number of processors) represents a sequential computer receiving the input data at a low rate and solving the problem at hand in

[3]We consider that the center of the circle inscribed in a degenerate triangle, whose edges are collinear, is the inner edge of that triangle.

maximum time but optimal cost. Finally, the vertex C (maximum time, maximum data, minimum number of processors) represents a sequential computer receiving a high rate of input data and failing to solve the problem at hand (the single processor, unable to cope with such a large arrival rate, gives up after M_t time units).

Note that the triangle doesn't have to be equilateral (though it may be made equilateral in any case by appropriately scaling the measures corresponding to its edges). The lengths of its edges may be determined in the case of d-algorithms using the following result stated in [5] and improved in [6]:

Proposition 6.1 *For the polynomial data arrival law given by relation (1), let A be any P-processor d-algorithm with time complexity $\Omega(N^v)$, $v > 1$. If A terminates, then its running time is upper bounded by a constant T that does not depend on n but depends on $S'(1, P)$.* □

This result establishes a maximum running time for a d-algorithm, given a maximum available number of processors. Moreover, the triangle in figure 1 may be easily scaled, when the conditions (that is, the number of processors or the characteristics of the arrival law) change. However, one should find a consistent way of representing the shape of the data arrival law (on the edge BC).

7 Conclusions

We started this paper by offering a characterization of c-algorithms, while presenting at the same time a general method to build an optimal c-algorithm starting from a dynamic algorithm. The analysis was completed by reducing it to the theory of d-algorithms. As a consequence, it was found that the speedup theorem does not hold for the class of c-algorithms (corollary 4.2). This result provides yet another class of problems that contradict this theorem. Moreover, the moving objects paradigm of section 6.1 suggests that there are many more real-time computations in which the speedup theorem no longer holds.

On the other hand, the usual measure employed in evaluating parallel algorithms (namely, the speedup and related measures), proved to be of less use in a real-time environment, where the characteristics of the input arrival law constitute an important parameter, that the speedup measure fails to consider. Therefore, we proposed in section 6 a new way of representing the performances of real-time parallel algorithms, which is simple, intuitive, and capable of capturing all the relevant parameters of the computation, while showing how they vary with respect to (and in connection with) one another. We believe that the main open problem introduced by this paper is to provide a thorough analysis of the properties and uses of such a representation, which may put in a new light the analysis of parallel real-time algorithms.

Another open problem is the updating asynergy phenomenon defined in section 5. Even though we may tend to believe that, except for the trivial case, the update asynergy is inherent, it is important to recall that the parallel asynergy theorem[4] stated in [4] was later invalidated, at least for real-time processing, in [1, 2, 5, 8] and in this paper. Until this updating asynergy is proved, algorithm designers who build algorithms to be used in the real-time framework analyzed in this paper should consider the possibility of handling many corrections at once, since such an approach has both a theoretical and practical use: From the theoretical point of view, the updating asynergy may be proven false in this way, while in practice such an approach may lead to faster c-algorithms, as shown in theorem 4.8.

Note that the graph processing c-algorithms may prove very useful in practice, due to the development of mobile computing. Right now, if a host is moving, the rerouting of the packets addressed to that host is accomplished through procedures like tunneling [14]. Such procedures—while having the advantage of being computationally inexpensive— lengthen the communication chain and may lead to an increase in network traffic. On the other hand, it may be interesting to investigate algorithms that recompute the routing tables on the fly, since we believe that a large number of mobile hosts may lead otherwise to traffic congestion. It is immediate that one of the possible paradigms that accommodates such a process is the theory of c-algorithms on graphs. Note that two solutions to such a problem have been proposed in [11], where simulation results were presented. One can obtain analytical results by applying the results in this paper to the problem presented in [11].

We analyzed only one of the multiple real-time paradigms that can be imagined. As mentioned in section 1, another useful paradigm of this type is presented in [8], namely, the maintenance algorithms (or m-algorithms), whose study is based on the theory of d-algorithms. Since our paper reduced the analysis of c-algorithms to that of the d-algorithms, the extension of c-algorithms to something that may be called maintenance correcting algorithms (mc-algorithms?) is immediate and hence it is not presented here.

References

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

[2] S. G. Akl, L. Fava Lindon, *Paradigms Admitting Superunitary Behaviour in Parallel Computation*, Parallel Algorithms and Applications, 11, 1997, 129–153.

[3] J.D. Barrow, *Impossibility: The Limits of Science and the Science of Limits*, Oxford University Press, Oxford, 1998, p. 146.

[4] R. P. Brent, *The Parallel Evaluation of General Arithmetic Expressions*, Journal of the ACM, 21(2), 1974, 201–206.

[4]This principle was not given a name in [4]. It later became known as Brent's theorem. The term "parallel asynergy" was introduced in [2].

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

[6] S. D. Bruda, S. G. Akl, *The Characterization of Data-Accumulating Algorithms*, Technical Report No. 98-418, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, August 1998.

[7] D. Knuth, *The Art of Computer Programming*, Vol. 2, Seminumerical Algorithms, Addison-Wesley, 1969.

[8] F. Luccio, L. Pagli, *Computing with Time–Varying Data: Sequential Complexity and Parallel Speed–up*, Theory of Computing Systems, 31(1), 1998, 5–26.

[9] G. Ramalingam, T. Reps, *On the Computational Complexity of Dynamic Graph Problems*, Theoretical Computer Science, 158, 1996, 233–277.

[10] J. H. Reif, *On Dynamic algorithms for Algebraic Problems*, Journal of Algorithms, 22, 1997, 347–371.

[11] M. Singhal, D. P. Agrawal, *A Distributed Connectivity Algorithm for Ad Hoc Networks*, Proceedings of the Fourth International Conference on Computer Science and Informatics, Research Triangle Park, North Carolina, October 1998, 146–149.

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

[13] D.T. Spreng, *On time, information, and energy conservation*, ORAU/IEA-78-22(R), Institute for Energy Analysis, Oak Ridge Assoc. Universities, Oak Ridge, Tennessee, 1978.

[14] A. S. Tanenbaum, *Computer Networks*, Prentice Hall, Upper Saddle River, New Jersey, 3rd edition, 1996.