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}@cs.queensu.ca

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.

1 Introduction

A well known result concerning the limits of parallel computation, 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, 13].

One of the principal shortcomings of the speedup theorem is that it is stated for, and applies solely to, a conventional computing environment. In particular, such a conventional environment does not assume any *real-time* constraints, typically 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 [9], 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 [6, 7, 10], 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.

An extension for the class of d-algorithms is proposed in [10] as the class of *correcting algorithms* (*c-algorithms* henceforth), where the input data are available at the beginning of the computation, but there

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

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 [10] that the same approach can be used to study such algorithms as the one applied to d-algorithms. Indeed, we start our paper by showing how a characterization of c-algorithms using the already developed theory of d-algorithms is possible. More precisely, we show that, given some c-algorithm, its characteristics (that is, running time, complexity, parallel speedup) are identical to the characteristics of a (hypothetical) d-algorithm. Therefore, c-algorithms have the same properties as d-algorithms. In particular, they manifest superunitary behavior iff the static case offers even the slightest speedup (that is, a speedup larger than 1).

We believe that there are many 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. In fact, as we shall see, speedup and efficiency become unfair measures, since the definition of running time varies from the static to the data-accumulating model. Moreover, 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.

2 Preliminaries

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].

Given some problem Π, the *speedup* of an algorithm A_p that uses p_1 processors is the ratio $S(1,p_1) = T_\Pi(1)/T_\Pi(p_1)$, where $T_\Pi(p_1)$ is the running time of A_p when solving Π, and $T_\Pi(1)$ is the running time of the best known sequential algorithm working on Π. In a conventional environment, the speedup of a parallel algorithm is bounded by the following results: Conforming to the *speedup theorem* [13], $S(1,p) \leq p$. Furthermore, *Brent's theorem* [4] states that "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." The speedup theorem and Brent's theorem are valid in a conventional computational environment. However, both of them fail in a number of unconventional environments, as shown in [1, 2, 10]. Specifically, speedup theorem is too pessimistic, while Brent's theorem is too optimistic.

2.1 D-Algorithms

The notion of d-algorithm was introduced in [9], and is further studied in [7, 10].

Definition 2.1 [7] An algorithm A is a d-algorithm if: (I) A works on a set of data which is not entirely available at the beginning of computation. Data come while the computation is in progress (conforming to some specified data arrival law), and A terminates when all the currently arrived data have been processed before another datum arrives. (II) For any input data set, there is at least one data arrival law f such that, for any value of n, A terminates in finite time, where f has the following properties: (i) f is increasing with respect to t, and (ii) $f(n, C(n)) > n$, where $C(n)$ is the complexity of A. Moreover, A immediately terminates if the initial data set is null ($n = 0$). $\square$

The size of the set of processed data is denoted by N. The data arrival law is denoted by $f(n, \tau)$, where n denotes the number of input data available at the beginning of the computation, and τ denotes the time. That is, $f(n, \tau)$ denotes the amount of input data that are accessible to the d-algorithm at any time τ. Moreover, we have the constraint $f(n, 0) = n$, since n denotes the size of the initial input. Given the termination condition, if we denote by t the running time of some d-algorithm, then $N = f(n, t)$.

The form proposed in [10] for the data arrival law is $f(n, t) = n + kn^{\gamma}t^{\beta}$, where k, γ, and β are positive constants. Such a form of the arrival law is rather flexible, and its polynomial form eases the reasoning about algorithms that use it. In particular, note that, when $\gamma = 0$, the amount of data that arrive in one time unit is independent of the size of initial data set. If $\beta = 1$, then the data flow is constant during the time, while in the case $\beta > 1$ the flow of data actually increases with time. Similarly, when $\beta < 1$, fewer and fewer data arrive as time increases.

Definition 2.2 Consider a given problem Π, and let A be a d-algorithm for Π, working on a varying set of data of size N. Consider now an algorithm A_s that perform the same computations as A, except that A_s works on the N data as if they are available at time 0. Then, if A and A_s use the same number of processors, A_s is called the *static version* of A. $\square$

Note that this definition of the static version of a d-algorithm is somehow different from the definition from [10]. Indeed, the definition from [10] is similar with definition 2.2, except that "perform the same computation" is replaced by "solves the same problem," and, in addition, A_s is the best known algorithm that solves the given problem. However, as noted in [7], such a definition is of little relevance, since there are d-algorithms whose performance is inherently worse than in the static case as defined in [10]. Therefore we chose the above definition, as suggested in [7]. In addition, this definition allows us to define the time complexity of a d-algorithm in a more elegant manner, as we shall see below.

Example 2.1 In order to make a clear distinction between the two definitions of the static version, let us consider the problem of sorting a sequence of numbers. Consider then the d-algorithm A that solves this problem and performs the following computations: Sort by some efficient method the initial data set, putting the result in an array; then, insert each datum that arrive into the sorted array. Conforming to definition 2.2, the static version A_s of A is an algorithm that receives the amount of data processed by A, sorts what was the initial sequence for A, and then inserts each additional datum into the already sorted array. Considering that $N > n$, the time complexity of A_s is $\Theta(N^2)$. On the other hand, the static version of A as defined in [10] takes the whole sequence of length N and sorts it using some efficient method. The complexity is now $\Theta(N \log N)$.

It is important to emphasize the difference between the time complexity and the running time in the data-accumulating paradigm. In the static case, the time complexity $C(n)$ of some algorithm A_s on input w, $|w| = n$, is defined as the running time of A_s. Such a complexity is a function of n, and there is no difference between the notions of time complexity and running time. We define the time complexity in the data-accumulating case in a similar manner, that is, as a function of the size N of the processed input.

Definition 2.3 Given some d-algorithm A, some data arrival law f, and some initial data set of size n, suppose that A terminates at some time t, where t depends on both n and f. We call t the *running time* of A.

Given the amount of data N that is processed by A, the *time complexity* $C(N)$ of A (or just *complexity* for short) is defined as the termination time of A_s, expressed as a function of N, where A_s is the static version of A. $\square$

Note that definition 2.3 is valid in both the sequential and parallel case. However, for clarity, we use the notations t and $C(N)$ for the sequential case, while we add the subscript p when explicitly referring to the parallel case. That is, when speaking of a parallel d-algorithm, we denote its running time by t_p and its (time) complexity by $C_p(N)$.

By contrast to the static case, in the case of d-algorithms the time complexity and the running time are different from each other. Indeed, consider some d-algorithm A, working on some input where the size of the initial data set is n and new data arrive while the computation is in progress, according to some arrival law f. Starting from the analysis of the static version A_s of A, one can easily compute the complexity $C(N)$ of A. However, the complexity is not a very useful measure, since N itself is a function of time. The running time 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. To further clarify the concepts of complexity and running time, let us consider the following examples, mentioned in another context in [8].

Example 2.2 *Given two power series $g(x) = g_0 + g_1x + g_2x^2 + \cdots$ and $h(x) = h_0 + h_1x + h_2x^2 + \cdots$, 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.*

The problem described in this example 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. The complexity in the static case in thus n. Based on the static complexity, it is easy to see that the complexity of a d-algorithm that solves the problem is $C(N) = N$. We have therefore a d-algorithm of linear complexity. The running time of such a d-algorithm is given then by an equation of the form $t = C(N)$. That is, the running time can be obtained by solving the implicit equation $t = n + kn^\gamma t^\beta$.

Example 2.3 *Given two power series as in example 2.2, with $h_0 \neq 0$, compute the series $r(x) = g(x)/h(x)$. The coefficients of the two series come in pairs, in real time, respecting the same arrival law as in example 2.2. The result should be reported at that time t when a polynomial of degree $N(t)$ is available as a (partial) result.*

The coefficients r_i has the form $r_i = (g_i - \sum_{k=0}^{i-1} r_k h_{i-k})/h_0$. The coefficient r_n can be thus computed only when g_n and h_n are available. Moreover, r_n does not depend on any coefficient g_j or h_j, with $j > n$. Therefore, this computation is also one that can be performed by a d-algorithm. In order to compute the i-th coefficient r_i of the result one needs to compute all the coefficients r_k, $0 \le k < i$, and then insert them in the above formula. Therefore, the computation of the first n coefficients require $O(n^2)$ time. A d-algorithm that solves the problem has a quadratic complexity ($C(N) = O(N^2)$), and a running time given by $t = (n + kn^\gamma t^\beta)^2$. Note that we cannot find a better algorithm, since the i-th coefficient of the product r_i depends on all the coefficients r_k, $0 \le k < i$.

In general, as shown in [10], in the case of a sequential d-algorithm, the running time is given by the solution of the following implicit equation: $t = c_d(n+kn^\gamma t^\beta)^v$, where the static counterpart of the d-algorithm in discussion has a complexity of c_dN^v, for a positive constant c_d. The complexity of such a d-algorithm is also c_dN^v. In the parallel case, we have a similar equation for the running time: $t_p = \frac{c_{dp}(n+kn^\gamma t_p^\beta)^v}{S'(1,P)}$, conforming to a result given in [10] and improved in [7], 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 parallel complexity is in this case $c_{dp}N^v/S'(1, P)$.

3 Correcting algorithms

A c-algorithm is defined in [10] as being an algorithm which works on an input data set of n elements, all available at the beginning of computation, but $V(n, \tau)$ variations of the n input data occur with time. We call the the function V the *corrections arrival law*. More precisely, given some time τ and some initial input of size n, the quantity $V(n, \tau)$ represents the number of corrections that arrived in the time interval $[0, \tau]$. We consider that $V(n, 0) = 0$ (no corrections at time 0). We denote by $f(n, \tau)$ the sum $n + V(n, \tau)$. We propose a definition of a c-algorithm that is similar to the one of a d-algorithm [7]:

Definition 3.1 (C-algorithm) An algorithm A is a c-algorithm if: (I) 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. (II) 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 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 [10], 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. 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.

Note that algorithms that correct themselves when their input changes have been studied [11, 12], 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].

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). It seems more consistent 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. The size of the whole, virtually endless, input data set (including the corrections) will be denoted by N_ω.

We can modify examples 2.2 and 2.3 in order to fit the new paradigm of c-algorithms as follows (these examples will be used henceforth in order to illustrate the properties of c-algorithms).

Example 3.1 Given two polynomials $g(x) = g_0 + g_1 x + g_2 x^2 + \cdots + g_n x^n$ and $h(x) = h_0 + h_1 x + h_2 x^2 + \cdots + h_n x^n$ of fixed degree n, 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, with $h_0 \neq 0$, compute the quotient of the two polynomials $r(x) = g(x)/h(x)$. 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 also called "incremental algorithms." The reader should not confuse the notion of c-algorithm with the notion of dynamic algorithm. The former assumes a real-time arrival law for the input and a specific termination condition. The terminology is indeed confusing, but these terms were already introduced elsewhere [10, 11, 12], hence we will use them as they are.

Throughout the rest of the paper, it is assumed that an algorithm A_u 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. However, it may be useful sometimes to consider simultaneously a bundle of corrections of size b. Let an algorithm that performs this processing be A_u^b, of complexity $C_u^b(n, b)$. Now, considering the parallel implementation of A_u and A_u^b, we make the following assumption:

Claim 1 For some problem Π solvable by a c-algorithm, let A_u be the best P-processor algorithm that considers a correction, and let A_u^b be the best known P-processor algorithm that considers a bundle of corrections of size b. Also, let the speedup manifested by A_u and A_u^b be $S_u(1, P)$ and $S_u^b(1, P)$, respectively. Then, $S_u(1, P) = S_u^b(1, P)$. □

In other words, the speedup manifested by A_u is the same as the speedup manifested by A_u^b. We believe that this is a reasonable assumption, since the computation performed by A_u^b is essentially the same as the computation performed by A_u (except that there may be some initial manipulation like sorting the bundle of corrections; however, such a manipulation manifests linear speedup, and hence does not affect the overall speedup of A_u^b).

Note that a c-algorithm has to perform some initial processing, that 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.

Definition 3.2 Consider a given problem Π, and let A be a c-algorithm for Π, working on a varying set of data of size N. Consider now an algorithm A_s that perform the same computations as A, except that A_s works on the N data as if they are available at time 0. Then, if A and A_s use the same number of processors, A_s is called the *static version* of A.

Given some corrections arrival law V, and some initial data set of size n, suppose that A terminates at some time t, where t depends on both n and V. We call t the *running time* of A.

Given the amount of data N that is processed by A, the *time complexity* $C(N)$ of A (or just *complexity* for short) is defined as the termination time of A_s, expressed as a function of N. □

The static version of a c-algorithm, as well as the running time and the time complexity, are hence defined similarly to the case of d-algorithms. Again, definition 3.2 is valid in both the sequential and parallel case. We denote by $S'(1, P)$ the parallel speedup for the static case.

4 Complexity

We analyze here the complexity of a general form of a c-algorithm. Since this was the arrival law considered when studying d-algorithms [6, 7, 10], our analysis uses the following corrections arrival law (introduced in examples 3.1 and 3.2):

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

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

Theorem 4.1 *Consider some problem Π solvable by a dynamic algorithm. Let A_s be the best known dynamic algorithm that solves Π, and let A_{ps} be a parallel version of A_s, which uses P processors and exhibit a speedup $S'(1, P)$. Then, there exist a sequential c-algorithm A and a P-processor c-algorithm A_p that solve Π. Moreover, when A and A_p work on an initial set of data of size n and with the corrections arrival law* *given by (1), there exist n', k', and γ', where γ' and k' are constants and n' is a function of n, such that the properties of A and A_p (namely, time complexity, running time, and parallel speedup) are the same as the properties of some sequential d-algorithm A_d and some P-processor d-algorithm A_{pd}, which work on n' initial data and with the data arrival law $f(x,t) = x + k'x^{\gamma'}t^{\beta}$, where the complexity of A_d is cN, and the parallel speedup manifested in the static case by A_{pd} is $S'(1,P)$.*

Proof. Assuming that such d-algorithms A_d and A_{pd} exist, recall that the running time t of A_d is given by $t = cN$, where $N = f(n', t)$. That is, the relations from which all the properties of A_d and A_{pd} can be derived are

$$t = c(n' + k'n'^{\gamma'}t^{\beta}), \tag{2}$$

$$t_p = \frac{c_p(n' + k'n'^{\gamma'}t^{\beta})}{S'(1,P)}. \tag{3}$$

Therefore, if we show that there exist n', γ', and k' such that we can consider two c-algorithms A and A_p whose running times respect relations (2) and (3), respectively, we complete the proof. We do this in what follows.

The computation performed by A_s can be split in two parts: the initial processing, and the processing of one correction. Let the algorithm A' that performs the initial processing be of complexity $C'(n) = cn^{\alpha}$, and the algorithm A_u that processes a correction be of complexity $C_u(n) = c_u n^{\epsilon}$, $\epsilon \geq 0$.

The algorithm A performs then the following computations: (a) compute the solution for the initial amount of data n, using A' (this takes $C'(n)$ time), and (b) 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 this gives the following implicit equation for the termination time: $t = cn^{\alpha} + c_u n^{\epsilon}(n + kn^{\gamma}t^{\beta} - n) = c(n^{\alpha} + \frac{c_u}{c}kn^{\gamma+\epsilon}t^{\beta})$. Then, considering $n' = n^{\alpha}$, $\gamma' = \frac{\gamma+\epsilon}{\alpha}$, and $k' = kc_u/c$, we obtain for the running time of A the implicit equation (2).

With regard to the parallel implementation, considering that the static version of A provides a speedup $S'(1,P)$ when using P processors, and that the complexity of A is $C(N)$, it is immediate that the complexity in the parallel case is $C(N)/S'(1,P)$. But then relation (3) as the implicit equation for the running time of A_p follows immediately. This is enough to prove that the properties of A and A_p are identical to the properties of A_d and A_{pd}.

However, in order to finalize the proof, we have to take into consideration the second item from definition 3.1. That is, 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 [10] that a d-algorithm of complexity cN terminates for any initial amount n of input data if $\beta < 1$. □

4.1 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. 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 a form of theorem 4.1 holds for this new algorithm as well. Specifically, one may wonder why a c-algorithm cannot consider the incoming corrections in bundles of size b instead of one by one.

Recall that 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. We have the following immediate observation:

Lemma 4.2 *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, $bC_u \geq C_u^b \geq C_u$ for any strictly positive b.*

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). For simplicity, we assume that the complexity of the initial processing $C'(n)$ is the same for A, A_{bw}, and A_b. We believe that this is a reasonable assumption, since the processes are essentially the same in the two c-algorithms. It follows immediately from claim 1 that

Lemma 4.3 *Given A, A_{bw}, and A_b as above, let S_1, S_2 and S_3 be the speedup of the static version of A, A_{bw}, and A_b, respectively, when implemented using P processors. Then, $S_1 = S_2 = S_3$.*

Note that the problem in example 3.1 does not admit an updating algorithm that processes a bundle of size b at a time, $b < n$, and whose running time is smaller than b times the time required for performing a single update[2]. On the other hand, the updating algorithm for the problem in example 3.2 has a time complexity independent on the number of corrections that are considered. Indeed, assume that some coefficient h_i changes. Since we consider a worst case analysis, we can assume without loss of generality that $i \geq n/k$ for some constant k. Then, all the coefficients r_j, $0 \leq j < i$ should be recomputed. However, note that r_j cannot be recomputed before the values of all r_k, $0 \leq k < j$ are known. Therefore, updating the solution requires $\Omega(i^2) = \Omega(n^2)$ time. But this is also the complexity of the algorithm that recomputes the solution from scratch, therefore, it takes the same time to consider any number of corrections.

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, lemma 4.2 implies that ϵ_b is a positive number, no larger than 1.

Lemma 4.4 *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 (sketch). The running time t_b of A_b is given by $t_b = cn^\alpha + c_u n^\epsilon \sum_{i=1}^m b_i^{\epsilon_b}$, 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. Note that exactly all the buffered corrections are processed, that is, $\sum_{i=1}^m b_i = V(n, t_b)$. Then, it follows that

$$t_b \leq cn^\alpha + kc_u n^{\epsilon+\gamma} t_b^\beta. \tag{4}$$

Let $R(n,t)$ be the function $R(n,t) = cn^\alpha + kc_u n^{\epsilon+\gamma} t^\beta - t$. 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 have from relation (4)

[2] Since we are dealing with worst case analysis, we do not consider exceptional cases like the one when some corrections that need to be considered are on the same index.

$$R(n,t) < R(n,t_b) \quad \text{if} \quad \epsilon_b < 1, \tag{5}$$
$$R(n,t) = R(n,t_b) \quad \text{if} \quad \epsilon_b = 1. \tag{6}$$

When $\epsilon_b = 1$, relation (6) holds. Therefore, in this case, A_b is no better than A. For $\epsilon_b < 1$, considering relation (5), we have:

1. If $\beta > 1$, then $\frac{\partial R}{\partial t} > 0$ for t large enough. In this case, given relation (5), 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. Therefore, analogously, 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. Otherwise, A is no better than A_b. □

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

Proof (sketch). 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

$$t_{bw} = cn^\alpha + c_u n^\epsilon \sum_{j=1}^{m_w} b_{wj}^{\epsilon_b} + t_w, \tag{7}$$

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, it can be showed that $a_i \leq k n^\gamma t_{wi}^\beta$.

But, for any i, $1 \leq i \leq m_w$, $b_{wi}^{\epsilon_b} \geq \frac{b_{wi+1} - kn^\gamma t_{wi}^\beta}{a}$. 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 and from relation (7), $t_{bw} \geq cn^\alpha + \frac{b}{a} t_{bw}^\beta - \frac{b}{a} \sum_{i=1}^{m_w} t_{wi}^\beta + t_w$, where $b = c_u n^\epsilon k n^\gamma$. Considering relation (4) we have then

$$\rho(t_{bw}) - \rho(t_b) \geq t_w - \frac{b}{a} \sum_{i=1}^{m_w} t_{wi}^\beta + \left(b - \frac{b}{a}\right) t_{bw}^\beta, \tag{8}$$

where $\rho(t) = t - (b/a)t^\beta$. Let us denote the right hand side of relation (8) by $\mathcal{R}$. It can be proved that $\mathcal{R}$ is positive. But then relation (8) leads to $\rho(t_{bw}) \geq \rho(t_b)$ and it is immediate that $t_{bw} \geq t_b$, since ρ is an increasing function for any $t \geq 1$. □

> 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 4.5.

The proof of lemma 4.5 contains yet another interesting result. We show there that $b_{wi+1} \leq a b_{wi}^{\beta\epsilon_b} + a_i$. Since A_b does not wait for any correction, we have analogously $b_i \leq a b_i^{\beta\epsilon_b}$. But, applying this relation recursively i times we observe that $b_{i+1} \leq a^{(1-(\beta\epsilon_b)^i)/(1-\beta\epsilon_b)} (b_1)^{(\beta\epsilon_b)^i}$. However, $0 \leq \beta\epsilon_b \leq 1$, 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. 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. Starting from this expression, one can follow the same reasoning as in the proof of theorem 4.1 to derive an implicit equation for the parallel case of A_b similar to equation (3). Lemmas 4.4 and 4.5 therefore imply:

Theorem 4.6 *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.*

Moreover, let A_b work on n initial data and with the corrections arrival law given by (1), and let A_{bp} be the P-processor parallel implementation of A_b, where the static version of A_{bp} exhibit a speedup $S'(1, P)$. Then, there exist n'', k'', and γ'', where γ'' and k'' are constants and n'' is a function of n, such that the properties of A_b and A_{bp} (namely, time complexity, running time, and parallel speedup) are the same as the properties of some sequential d-algorithm A_d and some P-processor d-algorithm A_{pd}, which work on n'' initial data and with the data arrival law $f(x, t) = x + k''x^{\gamma''}t^{\beta}$, where the complexity of A_d is cN, and the parallel speedup manifested in the static case by A_{pd} is $S'(1, P)$. □

Corollary 4.7 *Consider some problem Π solvable by a c-algorithm. With A, A_b, n', n'', γ', γ'', k', and k'' as in theorem 4.1 and theorem 4.6, given some corrections arrival law V of the form (1) and some initial data set of size n, define*

$$\mathcal{P}(x, y) = \begin{cases} x & \text{if the optimal algorithm for } \Pi \text{ on } n \text{ and } V \text{ is } A \\ y & \text{otherwise.} \end{cases}$$

Let $n_1 = \mathcal{P}(n', n'')$, $\gamma_1 = \mathcal{P}(\gamma', \gamma'')$, and $k_1 = \mathcal{P}(k', k'')$. Then,

1. For a problem admitting a sequential c-algorithm and a parallel P-processor c-algorithm ($P = \xi(n_1 + k_1 n_1^{\gamma_1} t_p^{\beta})^{\delta}$) of complexity cN^{ι} and $c_p N^{\iota_p}$, respectively, such that the speedup for the static case is $S'(1, P) = \xi_1(n_1 + k_1 n^{\gamma_1} t_p^{\beta})^{s}$, 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 $n_1 \to \frac{1}{k_1 c}$, where $\beta = \gamma_1 = 1$, and $0 \leq \delta \leq 1$, $0 \leq s \leq 1$. For all values of β and γ_1, $\frac{t}{t_p} > \frac{c}{c_p} S'(1, P)$.

2. Consider any problem admitting a sequential c-algorithm and a parallel P-processor c-algorithm such that the speedup for the static case is $S'(1, P) = 1$. Then, for any data arrival law such that either $\beta \leq 1$, or $\gamma_1 \geq 1$ and $1/2 \leq k_1 c^{\beta}(\beta - 1)$, or $\gamma_1 > 1$ and n is large enough, the speedup of the parallel c-algorithm is $S(1, P) = 1$.

Proof. These results were proved in [7] with respect to d-algorithms, whose properties are the same with the properties of c-algorithms conforming to theorems 4.1 and 4.6. In addition, conforming to lemma 4.3, A_b is better than A iff the P-processor implementation of A_b is better than the P-processor implementation of A (that is, there exists no situations where A is better in the sequential case, but the optimal algorithm for the parallel case is A_b). The result follows immediately. □

Corollary 4.7 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 item 2, 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 $t = n + kn^{\gamma}t^{\beta}$. 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, considering a correction takes the same time as rebuilding the solution from scratch, namely $\Theta(n^2)$, as shown in section 4.1. Therefore, the running time of the algorithm that considers one correction at a 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.

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 [12], where the following lower bounds are established for a number of problems, each of size n: 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; extended GCD problem.

Each problem mentioned above is known to have an algorithm that solves it 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, it is mentioned in [12] that many other algebraic problems have this property.

Such kind of problems, 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 4.6.

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 [11]. Instead of analyzing the complexity of a dynamic algorithm for graph processing in terms of the size of entire input, such algorithms are analyzed in terms of an adaptive parameter $\|\delta\|$ that captures the size of the changes in the input and output. In graph problems, the input and output values can be associated with vertices of the input graph. Given a correction, the set of vertices whose values change is denoted by δ. Then, the sum of the number of vertices in δ and the number of edges incident on some vertex in δ is denoted by $\|\delta\|$.

The approach used in [11] is to analyze the computational complexity of dynamic algorithms on graphs in terms of $||\delta||$ rather than the size of the whole graph provided as input. Some general classes of dynamic algorithms are identified: Such algorithms are said to be *bounded* if, for all input data sets and for all corrections that can be applied this set, the time it takes to generate the updated solution depends only on $||\delta||$, and not on the size of the whole input. Otherwise, a dynamic algorithm is called *unbounded.* A refining of the class of bounded algorithms leads to a computational hierarchy [11]:

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).

Dynamic algorithms can sometimes be fruitfully analyzed in terms of the parameter $||\delta||$. However, there are algorithms for which such an analysis is not possible (namely, the unbounded ones). In addition, 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 (moreover, $||\delta||$ may take any value between 1 and the sum of the cardinalities of the vertex set and the edge set in the input graph; for this reasons $||\delta||$ is called an *adaptive* measure), 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 [11] and our results would be to see if δ can be bounded, even if for particular shapes of the input or for particular problems.

On the other hand, while the fact that $||\delta||$ is adaptive is a feature that makes this measure inappropriate for the analysis of c-algorithms, this feature can offer some benefits in an indirect way: One can draw conclusions on the nature of the correction that is to be processed with respect to the set δ associated with that correction. Based on these conclusions, one may find updating algorithms that processes many corrections at once faster than the time required to process the corrections individually. That is, such a measure can lead to a positive answer to the *updating asynergy* problem, which we introduced in section 5.1.

6 Parallel Real-Time Computations

Corollary 4.7 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. These properties of c-algorithms were found previously to hold for various related computational settings [6, 7, 10]. 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), $1 \leq x_1, y_1 \leq n$, to another grid point (x_2, y_2), $1 \leq x_2, y_2 \leq n$, 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. Moreover, these motion functions depend on some "external parameters," that do not change with time. Computing the motion function for any object requires $k_1 n^2$ unit-time operations, for some positive constant k_1.

Example 6.1 As an example of motion function, assume that each grid point (x, y) has an associated "acceleration vector" $\vec{i}(x, y)$ (defined by its components $i_x(x, y)$ and $i_y(x, y)$ on the two axis of the grid). At time 0, the velocity $\vec{v}(j)$ imposed on some object j depends on each $\vec{i}(x, y)$, and this dependence weakens with the distance: If the initial coordinates of object j are (x_j, y_j), then the velocity of this object is $\vec{v}(j) = \vec{\sum}_{1 \leq x,y \leq n} (1/d_j(x, y))\vec{i}(x, y)$, where $d_j(x, y) = 1 + \sqrt{(x - x_j)^2 + (y - y_j)^2}$, and we denote by $\vec{\sum}$ the usual vector composition operator. After this initial "push," the objects move freely (that is, the velocity of some object does not change), bouncing off the edges of the grid. Given two moments in time t and t', $t < t'$, and the location (x_j, y_j) of some object j at time t, the location (x'_j, y'_j) of this object at time t' is easily computed in constant time. For example, if the x-axis component $v_x(j)$ of $\vec{v}(j)$ is positive, then $x'_j = \lfloor x_j + 2(n - x_j) - (t' - t)v_x(j) \rfloor$ if object j bounces, and $x'_j = \lfloor x_j + (t' - t)v_x(j) \rfloor$ otherwise. The test whether the object bounces is also computed in constant time, and the other cases are similar[3]. Finally, since computing $\vec{v}(j)$ takes $\Theta(n^2)$ time, it follows that the computation of the motion function of some object takes $\Theta(n^2)$ time.

Once the positions of the n objects are known at a specific moment in time, a function ϕ (for example, the centroid of the n objects) of these positions is to be computed. Computing ϕ requires $k_2 n$ unit-time operations on a sequential machine for some positive constant k_2. When n processors are available, the computation of ϕ takes constant time. 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', where $\tau - \tau'$ is a positive constant.

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 and the "acceleration vector" for each grid point. Then the algorithm computes the new position reached by each object when all such computations of new positions have been performed and the function ϕ can be computed. Specifically, suppose that the initial position of the i-th object was identified at time t_i, $1 \leq t_i \leq n^2$. A new position at time t'_i is computed, where $t'_i = t_i + (n^2 - t_i) + (k_1 n^2 \times n) + k_2 n$. 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.

Moreover, the algorithm that is described above is optimal. Indeed, before the function ϕ can be computed, the algorithm needs to know the position of all the points. Assume now that, at some moment τ_1 the algorithm knows the current positions of the n objects and it is ready to compute ϕ. However, this computation takes $k_2 n$ time, hence the result will be available no earlier than at the time $\tau_1 + k_2 n$. The difference between this time and the time at which the positions of the objects are known (τ_1) is clearly not constant, therefore the algorithm needs to compute new positions for each of the n objects before computing ϕ. This takes $\Omega(n^3)$ time, and this establishes the lower bound matched by the above algorithm.

Parallel solution. If n^2 processors are available, then each can examine 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. Specifically, let ϕ be the centroid of the n objects, and the grid be stored in the $n \times n$ array *grid*. An algorithm that solves the problem is the following ($\mathcal{O}(x, y)$ is true iff, at the moment of its invocation, $grid(x, y)$ contains an object):

algorithm MovingCentroid($grid$; (x, y))
1. **for** $i \leftarrow 1 \ldots n^2$ **do in parallel** $\{ x_i \leftarrow 0;\ y_i \leftarrow 0 \}$;
2. **for** $i \leftarrow 1 \ldots n$, $j \leftarrow 1 \ldots n$ **do in parallel**
   if $\mathcal{O}(i, j)$ **then** $\{ x_{n(i-1)+j} \leftarrow i;\ y_{n(i-1)+j} \leftarrow j \}$;
3. **for** $i \leftarrow 1 \ldots n^2$ **do in parallel** $\{ x \overset{\Sigma}{\leftarrow} x_i;\ y \overset{\Sigma}{\leftarrow} y_i \}$;
4. $x \leftarrow x/n$; $y \leftarrow y/n$;

[3] We assumed here an idealized world, in which the objects reach their initial velocity instantaneously, and there is no friction. Introducing more constraints is certainly feasible, but is not in the scope of this paper.

The above algorithm uses two variables x_i and y_i for each processor i, $1 \leq i \leq n^2$, which are initialized with 0 (step 1). Then, those processors that found some point set their associated variables accordingly (step 2). After this, each processor i writes the values stored in the variables x_i and y_i to x and y, respectively (step 3). Recall that our parallel model is the COMBINING CRCW PRAM, and hence all the values that are written to x and y are added together in step 3. Finally, x and y are divided by the number n of objects (step 4). Note that a processor that did not find any object writes 0 to both x and y, and such an operation does not have any effect, since 0 is the identity element for addition. It is immediate that each step of the algorithm can be performed in constant time, and hence the whole algorithm takes constant time as well. Therefore, the difference between the time at which the coordinates of the objects are found and the time at which ϕ is reported is constant, as required.

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

Note that the moving objects paradigm is a real-time problem because of its real-time restrictions. Moreover, such a problem 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 complement of that of c-algorithms: Here, the corrections are known (through the motion functions), but the initial locations are not.

6.2 A Graphical Representation

A graphical representation for speedup 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 (to represent time, information, and energy).

The diagram we propose is a triangle ABC from figure (1.a). On the edge AB the time increases from m_t to M_t, on the edge CB 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, $i \in \{t, r, P\}$. Each discrete point in this triangle represents a combination of these three quantities for a computer working on the problem.

One should establish how to determine a point in the diagram from its coordinates (or vice-versa). We propose the following method: Given some point T, the coordinates of T are established based by the three lines that contain T and are parallel to the edges of the triangle, as shown in figure (1.b). Note that, each of the lines $\overline{TS'}$, $\overline{TQ''}$, and $\overline{TT'}$ intersect the edges of the triangle twice. In the example from figure (1.b), the intersection points closer to A (for the edge AB), to B (for BC), and to C (for CA) were chosen, respectively.

Example 6.2 Consider the problem of a c-algorithm that computes the quotient of two polynomials. This problem was introduced in example 3.2. The arrival law is $V(n,t) = t^\beta$ (that is, $k = 1$ and $\gamma = 0$), and n is fixed. Recall that the complexity in the static case is cn^2. Clearly, a parallel static algorithm for this problem offers a linear speedup. The equivalent diagram for this problem is shown in figure (1.b).

When $\beta = 0$ and $P = 1$, we are in the sequential static case. The running time is then $t = cn^2$, and this situation corresponds to the point C_1 in the figure, whose coordinates are (A', C', C''). When the number of processors is increased to $P = cn^2$ and β is still 0, then we have a parallel static algorithm that solves the problem in one time unit, represented by point C_2. Note that all the points corresponding to the static case are located on the segment C_1C_2. As expected, if β increases and the number of processors remains 1, then the running time increases (the point Q in figure (1.b) with coordinates (Q'', Q', C'')). Moreover, from point Q, if the number of processors is increased (and β remains the same), then the running time decreases (point S).

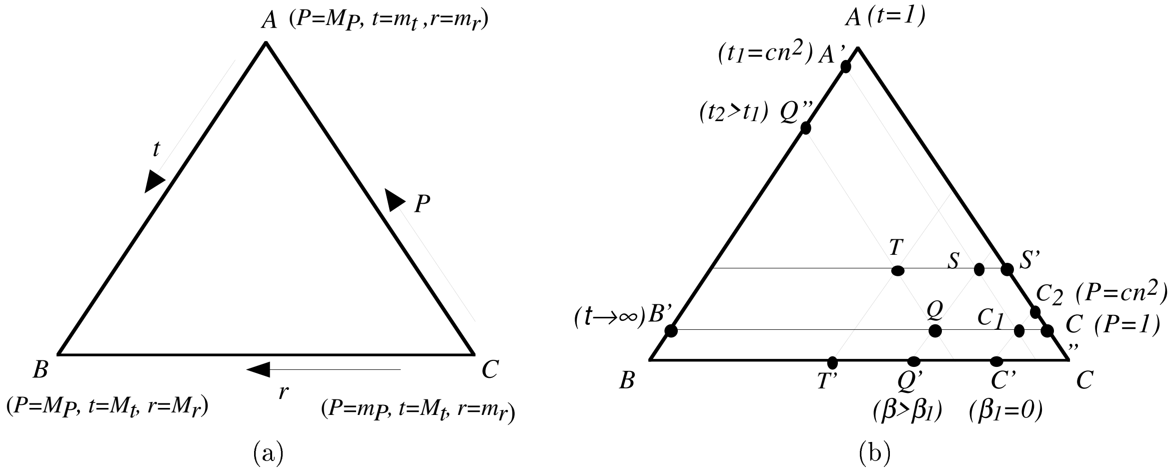

Figure 1: A representation of the relation between the number of processors, the data arrival rate, and the running time for a parallel algorithm.

The triangle doesn't have to be equilateral (although it may be made equilateral by appropriately scaling the measures corresponding to its edges). Moreover, the triangle in figure (1.a) may be easily scaled, when the conditions (that is, the number of processors or the characteristics of the arrival law) change. However, if the time or the number of processors are straightforward measures, one should find a consistent way of representing the shape of the data arrival law (on the edge BC). In example 6.2 we used β only as such a measure, but this is a very restricted case.

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. More precisely, given a some problem solvable by a c-algorithm, we showed that there is a (hypothetical) d-algorithm whose properties are the same as the properties of the optimal c-algorithm for that problem. As a consequence, it was found that the speedup theorem does not hold for the class of c-algorithms (corollary 4.7). 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 consider that this representation is more clear than the traditional 3-axe representation for at least two reasons: First, as the traditional representation, our measure shows the variation of speedup with respect to the number of processors and the characteristics of the arrival law. However, it also identifies in a precise manner the limit points (namely, the three vertices of the triangle), including those combinations that cannot lead to a successful computation. Therefore, by looking at the graphical representation, one can easily identify the critical characteristics of the system. Second, a two-dimensional representation is printed and read easier than a three-dimensional one.

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, 6, 10] 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 4.6.

Graph processing c-algorithms may prove useful in practice, due to the development of mobile computing. Indeed, one of the approaches for the routing problem in ad hoc networks [5] consists in recomputing the routing tables on the fly, as the network topology changes. The results in this paper can be used for the analysis of such computations.

References

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

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

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

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

[5] J. Broch, D. A. Maltz, D. B. Johnson, Y.-C. Hu, *A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols*, in The Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking, 1998, Dallas, TX, pp. 85-97.

[6] S. D. Bruda, S. G. Akl, *On the Data-Accumulating Paradigm*, in Proc. of the Fourth International Conference on Computer Science and Informatics, Research Triangle Park, NC, 1998, pp. 150–153.

[7] S. D. Bruda, S. G. Akl, *The Characterization of Data-Accumulating Algorithms*, Theory of Computing Systems, 33 (2000), pp. 85–96.

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

[9] F. Luccio, L.Pagli, *The p-Shovelers Problem (Computing with time-varying data)*, in Proc. of the Fourth IEEE Symposium on Parallel and Distributed Processing, 1992, pp. 188–193.

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

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

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

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

[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].