Technical Report No. 98-418

The Characterization of Data-Accumulating 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

August 25, 1998

Abstract

A data-accumulating algorithm (d-algorithm for short) works on an input considered as a virtually endless stream. The computation terminates when all the currently arrived data have been processed before another datum arrives. In this paper, the class of d-algorithms is characterized. It is shown that this class is identical to the class of on-line algorithms. The parallel implementation of d-algorithms is then investigated. It is found that, in general, the speedup achieved through parallelism can be made arbitrarily large for almost any such algorithm. On the other hand, we prove that for d-algorithms whose static counterparts manifest only unitary speedup, no improvement is possible through parallel implementation.

1 Introduction

Researchers in the area of parallel computation are always seeking to find limits to the performance of parallel algorithms. The most cited result in this connection states that the decrease in the running time of a parallel algorithm that solves some problem is at most proportional to the increase in the number of processors [4, 13]. Such algorithms are said to manifest a behaviour that is *at most unitary*, since, according to this result, the ratio of the speedup achieved to the number of processors used is at most 1. By contrast, an algorithm that would manifest a speedup larger than the number of processors used would be said to exhibit *superunitary* behaviour.

The first observation of superunitary behaviour was based on parallel search algorithms, which have been found to exhibit such a behaviour on particular shapes of the search space [8]. Later, additional examples of such algorithms were found [3], this time manifesting

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

superunitary behaviour in all instances of the solved problem. These algorithms use unconventional, yet realistic paradigms. This direction is continued in [2]. Finally, another approach led to a new paradigm where superunitary behaviour is manifested, namely the *data-accumulating paradigm*.

In the data-accumulating paradigm, introduced in [10], the input is considered as a virtually endless stream. An algorithm pertaining to this paradigm, called a *data-accumulating algorithm* or *d-algorithm* for short, terminates when all the currently arrived data have been processed before another datum arrives. This paradigm is studied further in [9] and [5], where complexity-related properties are derived for both the parallel and sequential cases. Even though the study of d-algorithms started from the desire to find paradigms in which a parallel approach can lead to superunitary behaviour, few things have been said about the performance of parallel implementations of d-algorithms. More precisely, this performance was investigated only for those d-algorithms whose static version[1] manifests a unitary behaviour.

In this paper we characterize the class of d-algorithms. First, we show that it is precisely the same as the well-known class of on-line algorithms. This result basically shows that a d-algorithm is an on-line algorithm for which the termination time is imposed by some real-time restriction (namely the shape of the data arrival law). Therefore, given some problem, the best on-line algorithm for that problem is also the best d-algorithm for it, and one can rely on this when designing d-algorithms for various problems. The identity between d-algorithms and on-line algorithms also leads to an interesting discussion on the notion of optimality of d-algorithms. This discussion is outlined in the last section.

In the second part of the paper we study how a parallel implementation affects the performance of d-algorithms. We address the most general case. That is, we do not restrict ourselves to those d-algorithms for which the static counterpart is work-optimal (that is, manifests unitary behaviour). We find that, as long as the speedup in the static case is larger than one, the speedup of the d-algorithm can be made arbitrarily large (that is, if the parallel static implementation generates even the slightest improvement, then, in the data-accumulating case, this improvement through parallelism becomes unbounded). On the other hand, when the static case manifests a unitary speedup, then the parallel d-algorithm will keep this property (thus, if no improvement is possible for the static case through parallelism, then neither does parallelism help in the data-accumulating case).

The paper is organized as follows. The next section briefly summarizes those results concerning d-algorithms that are used in our study. Section 3 starts by presenting a formal definition for d-algorithms, and then proves the equivalence between the classes of d-algorithms and on-line algorithms. Section 4 presents the afordmentioned results concerning the parallel implementation of d-algorithms. We conclude in section 5. In the following, a proposition is a result proved elsewhere. Throughout the paper we use the Random Access Machine (RAM) and the Parallel Random Access Machine (PRAM) as our sequential and parallel computational models respectively [2], unless otherwise stated. The word “iff” stands for the phrase “if and only if”.

[1] The static version of a d-algorithm A solves the same problem as A, but the whole input is available at the beginning of computation, as explained in the next section.

2 The Data-Accumulating Paradigm

We present here the necessary preliminaries concerning the data-accumulating paradigm, conforming to [9], also summarizing the notations used through the paper. 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 is called a *d-algorithm*. For such an algorithm, the computation terminates when all the currently arrived data have been treated. The size of the set of processed data is denoted by N.

Consider a given problem Π. Let the best known static algorithm for Π be A'. Then, a d-algorithm A for Π working on a varying set of data of size N is *optimal* iff its running time $T(N)$ is asymptotically equal to the time $T'(N)$, where $T'(N)$ is the time required by A' working on the N data as if they were available at time 0. Generally, when speaking about some property X of a d-algorithm A , we denote by X' the corresponding property of A', the static counterpart of A. When referring to the parallel case, we add the subscript p.

We will denote the arrival law by $f(n,t)$, where n denotes the amount of input data available at the beginning of the computation, and t denotes the time. That is, the amount of data processed by a d-algorithm will be given by the implicit equation $N = f(n, T(N))$. Note that this leads to an implicit equation for either N or $T(N)$, since N is a function of the elapsed time.

Note the difference between the running time and the (time) complexity in the data-accumulating paradigm. We denoted the running time of such an algorithm by $T(N)$. However, since N itself is a function of time, the actual running time is not a function of N anymore, it being obtained by solving an implicit equation of the form $t = T(N)$. The first form of the running time (that is, as a function of N) is referred to as the *time complexity* (or just complexity for short) of the d-algorithm in discussion, while the second form (the solution of the implicit equation) is referred to as the *running time* and is denoted by t. For the same reasons, the parallel running time is different from the parallel complexity, the former being denoted by $T_p(N)$, and the latter by t_p. Note that, in the static case, the running time and the time complexity as defined here are identical.

The form proposed in [9] 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 what follows, when we refer to a particular form of the data arrival law we use the above expression. It is shown in [9] that the termination time of a (parallel or sequential) d-algorithm of complexity $O(N^{\alpha})$ is finite for any $\alpha\beta < 1$.

We consider in section 4 problems that are solvable in polynomial time, that is, $T'(N) = O(N^{\alpha'})$, where α' is a positive constant. This implies that a d-algorithm has a time complexity of $T(N) = cN^{\alpha}$, for some positive constants c and α. Therefore, the complexity of a parallel d-algorithm has the form $T_p(N) = c_p N^{\alpha''}$, with c_p and α'' positive constants. The number of processors used by the parallel algorithm is denoted by P.

From the properties summarized in the above two paragraphs, it results that, in the case of a sequential d-algorithm, the running time is given by the solution of the following implicit equation

$$t = c(n + kn^{\gamma}t^{\beta})^{\alpha}. \tag{2}$$

The size of the whole input data set will be denoted by N_{ω}. Since the input data set in virtually endless in the data-accumulating paradigm, we will consider N_{ω} to be either large enough or tending to infinity. When considering N_{ω} to be infinite, it is obvious that some d-algorithm terminates in finite time iff it terminates before considering the whole input data set. By abuse of notation we also say this when N_{ω} is considered finite (that is, we say that the d-algorithm *terminates in finite time* iff it terminates before considering all its N_{ω} input data, no matter whether N_{ω} is finite or not).

3 Characterizing D-Algorithms

We characterize here the class of d-algorithms. But, first of all, we need a formal definition for such algorithms.

Definition 3.1 An algorithm A is a d-algorithm if

1. 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, and A terminates when all the currently arrived data have been processed before another datum arrives.

2. 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 strictly 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 ($n = 0$). □

In other words, the first condition is the obvious definition, implicitly given in [9]. The second condition means that A stops for some increasing data arrival law, such that *at least one* new datum arrives before A finishes the processing of the initial set of n data. If this condition is not stated, then any algorithm A_1 may be considered a d-algorithm for the following reason: Let the complexity of A_1 be $C_1(N)$ and let the data arrival law be $f(n,t) = n + kn^{\gamma}([t/C_1(n)])^{\beta}$, where $[x] = x$ if $x > 1$ and $[x] = 0$ otherwise. Note that, in this case, $f(n, C_1(n)) = n$. That is, no new data arrive before time $C_1(n)$. But, at this time, A_1 would have processed all its input data and would have simply stopped, without considering any other datum. Obviously, any algorithm will have this property.

We use the following notations: We denote by D_i the i-th datum in the input stream. The ordering is naturally defined as follows: D_j is examined before D_i is examined for the first time iff $i > j$.

We say that an algorithm A *is able to terminate at point* k if, before visiting any $D_{k'}$, $k' > k$, it has built a solution identical to the solution returned by A when working on the input set $D_1, \ldots, D_k$. If the algorithm A is able to terminate at some point, that point will be denoted by N_j, $j \in \{1, 2, \ldots\}$. Note that N (the amount of data processed by a d-algorithm) is also a termination point, but we will use N only for this purpose, in order to avoid confusion.

3.1 A Turing Machine Model

Our characterization of d-algorithms is made more convenient through the use of the following Turing machine model.

Definition 3.2 A Turing machine M which models an algorithm that is able to terminate at some point other than N_ω is the tuple (K, Σ, δ, h'), K being the (finite) set of states, Σ the (finite) tape alphabet, δ the transition function, and h' the initial state. The machine M has two tapes, as in [6]: The first tape is the (read-only) input tape, and the second one is the working tape. In addition, M is deterministic, except that it has to model the ability to terminate at some point. For this purpose, we allow a designated state h' to have two output transitions as follows: $\delta(h', x) = (h, x)$, and $\delta(h', x) = (q, z)$, where h denotes the halting state. With the above exception, δ is deterministic. Moreover, no other state is allowed to go directly to h. That is, the halting state h is replaced by an "optional halting" one (namely, h'). Note that the optional halting state h' is also the initial state. □

The definition above models a d-algorithm. More precisely, the algorithm A corresponding to such a machine M can terminate before the whole input is considered, namely, when M enters the state h'. Once in h', M's choice of halting or continuing to work models the ability of A to terminate eventually when it is able to output a solution for the currently arrived data and there is no arrived but yet unprocessed datum. Note that it is required that the state h' be entered at least once before the end of input data in order for A to be considered a d-algorithm (since, conforming to definition 3.1, there is at least one data arrival law for which A terminates, and this termination is modeled by the nondeterminism of h'). Since a d-algorithm should immediately terminate on an empty initial input, we impose h' as the initial state.

Generally, we assume that any algorithm (whether or not modeled by such a machine M) eventually terminates after considering all its input data. That is, when N_ω is finite, M's initial state h' is reached again some time after M visits all the data on the input tape.

Lemma 3.1 *A Turing machine M as in definition 3.2, working on any sufficiently large input data set N_ω, is able to terminate at some point $N_1 < N_\omega$, N_1 being constant with respect to N_ω, iff it is able to terminate at two finite points N_1 and N_2 strictly smaller than N_ω and constant with respect to N_ω.*

Proof. The "only if" part is immediate. We provide a proof for the "if" part.

When M halts at the point N_1 it must have reached the special state h'. Obviously, this happened after some constant number of steps (since both K and Σ are of constant size, and the number of tape cells visited is N_1 which is constant as well). Therefore, we have a cycle, from h' (the initial state) back to h', after a number of steps bounded by some constant ζ. Assume now that M chooses not to halt at the point N_1 and instead goes to another state q. But the state h' is accessible from q (otherwise, M won't halt even after processing all the N_ω input data) and, since M already reached h' for an arbitrary input, it will reach it again, after a number of steps bounded by ζ and after visiting a constant number of new tape cells, because M is deterministic. But this point is the point N_2 whose existence we want to prove. Note that N_2 does not depend on N_ω but only on the graph of δ, and hence we have completed the proof. □

Theorem 3.2 *A Turing machine M as in definition 3.2, working on any input data set of size N_ω, where N_ω tends to infinity, is able to terminate at some finite point N_1 iff it is able to terminate at all of the points in a countably infinite set $S \subseteq \{1, 2, \ldots, N_\omega\}$, where S has the following properties: (i) the least element of S is upper bounded by a finite constant ζ, and (ii) the distance between any two consecutive elements in S is upper bounded by ζ.*

Proof. Again, the "only if" part is immediate. But the "if" part is easily proved by induction over the size of S, using the theorem's premise (the existence of N_1) as inductive hypothesis and lemma 3.1 for the inductive step. □

The theorem above says that, if an algorithm working on a set of data of unbounded size can terminate at some finite point, then such a point is not unique. In fact, the set of such points is an infinite countable set (denoted by S in the theorem).

For any alphabet X and positive integer y, let X^y be the set of all the words of length y over the alphabet X. Given a constant ζ, one can compact a Turing machine's tape by simply considering $\Sigma^\zeta \cup \{\#\}$, where # is the blank symbol, as the tape alphabet instead of Σ, then "folding" each sequence of ζ non-blank tape cells into one cell, and finally modifying the function δ accordingly (see for example the proof given in [11] of the fact that a k-tape Turing machine can be simulated by a one-tape Turing machine). We have thus the following corollary:

Corollary 3.3 *A Turing machine M as in definition 3.2, working on any input data set of size N_ω, where N_ω tends to infinity, is able to terminate at some finite point N_1 iff it is able to terminate at all of the points in the set $\{1, 2, \ldots, N_\omega\}$.* □

3.2 On Line Algorithms

The notion of an *on-line* algorithm was introduced in order to define a class of algorithms for which the size of the input may be unknown at the beginning of computation. Basically, such an algorithm processes each input datum D_k without looking ahead at any datum $D_{k'}$, $k' > k$. This can be useful when either the input is not entirely available at the beginning of computation, or the input is virtually infinite, but a (partial) solution, based on some finite subset of the input is acceptable. By contrast, an algorithm that needs to know all the input in advance is called an *off-line* algorithm. From the above informal characterization for the on-line class, one can already identify a strong similarity between on-line algorithms and d-algorithms. In this section we formally show that these two classes are in fact identical.

There are many implicit definitions of on-line algorithms [1, 7, 12]. In [1], an *on-line execution* of some sequence of instructions σ is defined as requiring that the instructions in σ be executed from left to right, executing the i-th instruction without looking at any following instruction. An on-line algorithm is defined in [12] as an algorithm that cannot look ahead at its input. A similar definition in terms of Turing machines can be found in [6]. Finally, an on-line algorithm A is defined in [7] as having the property that A can determine the result of N input data without knowing N in advance, such that it is possible to run the algorithm until the end of the input data, or to run it until a certain condition is met. We assume here the latter definition, since the definition given in [12] leaves the way of reporting the result unclarified. However, if the definition in [12] is completed in a natural way (that is, an on-line algorithm A is able to report a (partial) solution after processing each datum), we reach the definition given in [7].

Also, we should stress again that a Turing machine defined as in definition 3.2 models any d-algorithm. The nondeterministic choice of halting or continuing to work (modeled by the state h') should be viewed as the decision made conforming to the first item in definition 3.1 (that is, whether no new data arrived during the current computation).

With the above two paragraphs in mind, corollary 3.3 leads to the following result, where $\mathcal{D}$ and $\mathcal{O}$ denote the class of d-algorithms and on-line algorithms, respectively.

Theorem 3.4 $\mathcal{D} = \mathcal{O}$.

Proof. Clearly, corollary 3.3 proves the inclusion $\mathcal{D} \subseteq \mathcal{O}$. It also proves $\mathcal{O} \subseteq \mathcal{D}$, except that the second point of definition 3.1 is not accounted for. Therefore, in order to complete the proof, we have to show that, for any on-line algorithm A and any size n of the initial data set, there is a data arrival law f such that, when working on a data-accumulating input set, A terminates in finite time, and considers at least $n+1$ data.

Let the complexity of A be $C(n)$. In general, $C(n)$ depends on the actual values of the input data. For any positive integer n_1, denote by t_1 a lower bound on $C(n_1)$, and let t_2 be an upper bound on $C(n_1+1)$, for any possible input data sets of size n_1 and n_1+1, respectively. It is easy to build a function $f(n,t)$, strictly increasing with respect to its second argument, such that $f(n_1, 0) = n_1$, $f(n_1, t_1) = n_1 + 1.1$, and $f(n_1, t_2) = n_1 + 1.5$ (for example, this could be done by interpolation). But such a function is the one we are searching for, considering n_1 as the initial amount of data: The function f is strictly increasing with respect to the second argument and $f(n_1, t_1) > n_1 + 1$, meaning that at least one new datum arrived before t_1. But $C(n_1) > t_1$; therefore, A needs to consider that new input datum. On the other hand, analogously, no other datum arrives between t_1 and t_2, and $C(n_1 + 1) < t_2$. That is, A terminates at some time less than t_2. The behaviour of A working on an initial data set of size n_1 and under the data arrival law f clearly satisfies the requirements stated in the second item of definition 3.1. □

4 On the Parallel Speedup

In this part we analyze how a parallel implementation influences the performance of a d-algorithm. The main measure used for evaluating a parallel algorithm is the *speedup*, defined as follows.

Given some problem Π, the speedup provided by an algorithm that uses p_1 processors over an algorithm that uses p_2 processors with respect to problem Π is the ratio $S(p_2, p_1) = \tau_\Pi(p_2)/\tau_\Pi(p_1)$, $p_1 > p_2 > 0$, where $\tau_\Pi(x)$ is the running time of the best x-processor algorithm that solves Π. In many cases [2], this definition is used to compare a parallel algorithm with a sequential one, that is, $p_2 = 1$. In the following, the amount of input data N_ω is considered tending to infinity.

We start by quoting the main result from [9] concerning parallel d-algorithms.

Proposition 4.1 *For a problem admitting an optimal sequential d-algorithm obeying relation* $t = c(n + kn^\gamma t^\beta)^\alpha$ *and an optimal parallel d-algorithm obeying relation* $t_p = \frac{c_p(n+kn^\gamma t_p^\beta)^\alpha}{P}$ *we have:*

1. *For* $\alpha = \beta = \gamma = 1$,
$$\frac{t}{Pt_p} = \frac{c}{c_p}\frac{1 - (c_p/P)kn}{1 - ckn}.$$

2. *For* $c_p/P < c$,
$$\frac{t}{Pt_p} \to N_\omega \, for \, n \to \frac{1}{kc^{1/\alpha}},$$
where $\alpha\beta = \gamma = 1$, *and* $P = \xi(n + kn^\gamma t_p^\beta)^\delta$, *with some constants* ξ, $\xi > 0$, *and* δ, $0 \leq \delta \leq \alpha$.

3. *For all values of* α, β, γ,
$$\frac{t}{Pt_p} > \frac{c}{c_p}.$$

□

Here, a discussion on the number of processors is in order. In proposition 4.1, P is considered a polynomial in n and t_p. That is, it depends at first sight on the elapsed time. This may be considered unrealistic, since no machine is expected in the near future to be able to increase its number of processors during the execution of an algorithm. However, the termination time t_p depends only on the initial data arrival law, the initial amount of input data, and the speedup. Hence, one can compute a value for P in advance, provided that the initial amount of data and the data arrival law are known. Consequently, we will retain this form for P. In addition, the case in which P is constant is covered by the expression for P in proposition 4.1, since $\delta = 0$ is a legal exponent.

Let us first take a look at how the implicit equation for the parallel running time has been derived. Generally,

$$t_p = c_p T_p'(n + kn^\gamma t_p^\beta). \tag{3}$$

Only *work-optimal* parallel algorithms[2] are considered in [9]. In this case, a static parallel algorithm requires time $T_p'(N) = O(N^\alpha/P)$, and the implicit equation for the running time

[2] A parallel algorithm is said to be *work-optimal* if the product of its worst case running time and the number of processors it uses is of the same order as the worst case running time of the best known sequential algorithm solving the same problem. Usually, such parallel algorithms are called simply *optimal* [2]. However, we will keep the terminology from [9], because we already used the qualifier "optimal" for d-algorithms.

of a parallel d-algorithm follows immediately. However, in the case of a non-work-optimal parallel static algorithm, we have the relation $S'(1,P) = T'(N)/T'_p(N)$ and thus $T'_p(N) = T'(N)/S'(1,P)$ which leads to $T'_p(N) = O(N^\alpha/S'(1,P))$. In this general case, the implicit equation for the parallel running time becomes

$$t_p = \frac{c_p(n + kn^\gamma t_p^\beta)^\alpha}{S'(1,P)}. \tag{4}$$

Note that the only change is the replacement of the number of processors P by the speedup of the static algorithm $S'(1,P)$ corresponding to the d-algorithm in discussion. Keeping this in mind, the following extension of proposition 4.1 is immediate.

Theorem 4.2 *For a problem admitting a sequential d-algorithm and a parallel d-algorithm such that the speedup for the static case is $S'(1,P) > 1$ we have:*

1. *For $\alpha = \beta = \gamma = 1$,*
$$\frac{t}{t_p} = \frac{c}{c_p}\frac{1 - (c_p/S'(1,P))kn}{1 - ckn}S'(1,P).$$
2. *For $c_p/S'(1,P) < c$,*
$$\frac{t}{S'(1,P)t_p} \to N_\omega \, for \, n \to \frac{1}{kc^{1/\alpha}},$$
where $\alpha\beta = \gamma = 1$.
3. *For all values of α, β, γ,*
$$\frac{t}{t_p} > \frac{c}{c_p}S'(1,P).$$

□

Corollary 4.3 *For a problem admitting a sequential d-algorithm and a parallel P-processor d-algorithm, $P = \xi(n + kn^\gamma t_p^\beta)^\delta$, such that the speedup for the static case is $S'(1,P) = \xi_1(n + kn^\gamma t_p^\beta)^\epsilon$, $S'(1,P) > 1$ for any strictly positive values of n and t_p, we have for $c_p/S'(1,P) < c$:*

$$\frac{t}{Pt_p} \to N_\omega \, for \, n \to \frac{1}{kc^{1/\alpha}},$$

where $\alpha\beta = \gamma = 1$, and $0 \le \delta \le \alpha$, $0 \le \epsilon \le \alpha$.

Proof. Conforming to formula (4), we have

$$t_p = (c_p/\xi_1)(n + knt_p^\beta)^{\alpha-\epsilon}. \tag{5}$$

But, since $\alpha\beta = 1$, it follows that $(\alpha - \epsilon)\beta < 1$, and hence the solution t_p of equation (5) is finite for any finite value of n [9]. But note that, in our case, $n \to \frac{1}{kc^{1/\alpha}}$, and thus it is finite.

Then, both P and $S'(1,P)$ are finite at the point t_p since they are polynomials in n and t_p. But we have by theorem 4.2 that $\frac{t}{S'(1,P)t_p} \to N_\omega$ and, obviously, $\frac{t}{Pt_p} = \frac{t}{S'(1,P)t_p}\frac{S'(1,P)}{P}$ (here we use P and $S'(1,P)$ to denote the number of processors and the static speedup evaluated at the point t_p). But then $\frac{t}{Pt_p}$ equals an infinite quantity multiplied by a finite quantity, and therefore it is infinite, as desired. □

Note that the result of corollary 4.3 is general. It does not apply only to work-optimal algorithms as the result in proposition 4.1. Indeed, the case $\epsilon < \delta$ is covered as well, for any small ϵ. By corollary 4.3 we found out that, at least for some data arrival laws, the speedup of any parallel d-algorithm can be made arbitrarily large, even if, in the static case, the parallel algorithm is not work-optimal (work-optimality is assumed in [9] when proving proposition 4.1). On the other hand, it is not an accident that we specified $S'(1,P) > 1$ in theorem 4.2 and corollary 4.3:

Theorem 4.4 *For any problem admitting a sequential d-algorithm and a parallel d-algorithm such that the speedup for the static case is $S'(1,P) = 1$, and for any data arrival law such that either $\alpha\beta \leq 1$, or $\gamma \geq 1$ and $1/2 \leq kc^\beta(\alpha\beta - 1)$, the speedup of a parallel d-algorithm is $S(1,P) = 1$.*

Proof. When $S'(1,P) = 1$, equation (4) become $t_p = c_p(n + kn^\gamma t_p^\beta)^\alpha$. Also, recall that the implicit equation for the running time in the sequential case is $t = c(n + kn^\gamma t^\beta)^\alpha$. Thus, the complexity of the static parallel algorithm is precisely the same as the complexity of the sequential algorithm. But then we have $c = c_p$, because the d-algorithm relies on static processing. We have then

$$\frac{t}{t_p} = \left(\frac{1 + kn^{\gamma-1}t^\beta}{1 + kn^{\gamma-1}t_p^\beta}\right)^\alpha,$$

which leads to

$$X(n,t) = X(n,t_p), \tag{6}$$

where the function X is $X(n,t) = t^{-1/\alpha}(1 + kn^{\gamma-1}t^\beta)$. Therefore, in order to prove that the speedup is unitary (that is, $t = t_p$) it is enough to prove that $X(n,\cdot)$ is a one to one function for any n. For this purpose, we will prove that $X(n,\cdot)$ is a strictly monotonic function and hence we will complete the proof. We have

$$\frac{\partial X}{\partial t} = \frac{1}{\alpha}t^{-(\alpha+1)/\alpha}(kn^{\gamma-1}(\alpha\beta - 1)t^\beta - 1).$$

1. If $\alpha\beta \leq 1$, then it is immediate that $\frac{\partial X}{\partial t} < 0$ for any n, because $kn^{\gamma-1}(\alpha\beta - 1)t^\beta \leq 0$.

2. If $\alpha\beta > 1$, then we have $\frac{\partial X}{\partial t}(n,t_0) = 0$, and $\frac{\partial X}{\partial t}(n,t) > 0$ for any $t > t_0$, where $t_0^\beta = 1/(kn^{\gamma-1}(\alpha\beta - 1))$. But the algorithm must process at least the initial set of data n and one more datum (conforming to definition 3.1). That is, $t \geq c(n+1)^\alpha$. Suppose now that t_0 is a possible value for the termination time. Then, $t_0 \geq c(n+1)^\alpha$ as well. This leads to

$$(n+1)^{\alpha\beta} \leq \frac{1}{kc^{\beta}n^{\gamma-1}(\alpha\beta-1)}.$$

Since $\alpha\beta > 1$ and $n \geq 1$ (for if $n = 0$ both the parallel and the sequential d-algorithms will immediately terminate and the speedup is obviously 1), we have $(n+1)^{\alpha\beta} > 2$ and then the above formula implies that

$$n^{\gamma-1} < \frac{1}{2kc^{\beta}(\alpha\beta-1)}. \tag{7}$$

Again, $n > 1$ and $\gamma \geq 1$, implying that $1 < 1/(2kc^{\beta}(\alpha\beta-1))$, that is, $1 > 2kc^{\beta}(\alpha\beta-1)$. This clearly contradicts the theorem's hypothesis. Therefore, our assumption that t_0 is a legal termination time is false. But then, for all possible values of the termination time, $X(n,\cdot)$ is monotonic, and this result holds for any n.

□

We impose in the above theorem a rather limited form for the data arrival law, but no restriction on n. It is easy though to put the problem in a different way.

Corollary 4.5 *For any problem admitting a sequential d-algorithm and a parallel d-algorithm such that the speedup for the static case is $S'(1,P) = 1$, and for any data arrival law such that either $\alpha\beta \leq 1$, or $\gamma > 1$ and n is large enough, the speedup of a parallel d-algorithm is $S(1,P) = 1$.*

Proof. The situation is analogous to the one in theorem 4.4, hence the proof is almost the same. More precisely, the only difference is the way in which the falsity of relation (7) is proved: In this case the relation is immediately false, since $\gamma - 1 > 0$ and hence the inequality does not hold for $n \geq (1/(2kc^{\beta}(\alpha\beta-1)))^{1/(\gamma-1)}$. □

Finally, some properties concerning the parallel speedup of sorting d-algorithms are derived in [5]. In particular, a limit $t''_B(P)$ is found on the running time of any P-processor algorithm. That is, when the running time of such an algorithm exceeds $t''_B(P)$, that algorithm never terminates. Moreover, such a limit holds for any d-algorithm of complexity $\Omega(N^{\alpha})$, $\alpha > 1$. The most important result in [5] concerning the parallel case is as follows:

Proposition 4.6 *For the polynomial data arrival law given by relation (1), let A be any P-processor d-algorithm with time complexity $\Omega(N^{\alpha})$, $\alpha > 1$. If A terminates, then its running time is upper bounded by a constant T that does not depend on n but depends on P.* □

We can now extend this result simply by observing that, in the case of a non-work-optimal parallel static algorithm, the number of processors P should be replaced by the speedup function S', as justified by formula (4). Thus we have:

Theorem 4.7 *For the polynomial data arrival law given by relation (1), let A be any P-processor d-algorithm with time complexity $\Omega(N^{\alpha})$, $\alpha > 1$. If A terminates, then its running time is upper bounded by a constant T that does not depend on n but depends on $S'(1,P)$.* □

5 Conclusions

Theorem 3.4 is an important result, because it characterizes the class of d-algorithms as being exactly the class of on-line algorithms. When working with d-algorithms, one can take advantage of this result, since on-line algorithms have already been designed for various problems (e.g., the on-line algorithms for manipulation of power series [7]).

As an immediate consequence of theorem 3.4, it is easier to know whether some problem does not admit an optimal d-algorithm (where the notion of optimality is the one defined in [9] and summarized in section 2 of this paper): If a given problem admits an off-line algorithm with a complexity asymptotically smaller than the lower bound for the complexity in the on-line case, then one cannot build an optimal d-algorithm.

As an example, sorting does not admit an optimal d-algorithm, because the best known (off-line) algorithm has a complexity of $O(n \log n)$ [1], while it is immediate that an on-line sorting algorithm has a complexity of $\Omega(n^2)$ (such an algorithm has to insert each of its input data into the already sorted sequence, one by one, because, at any time, we should have a sorted sequence of the already processed data). The same result is obtained in [5], though with a lot more effort.

However, considering theorem 3.4, the above notion of optimality no longer makes sense since, given some problem, once the lower bound in the on-line case has been established for that problem, a d-algorithm has no chance to beat it. Therefore, we suggest the following definition of optimality: Given some problem Π, a d-algorithm solving Π is optimal iff its complexity matches the lower bound for the complexity of on-line algorithms solving Π. Using this definition, it follows that sorting does admit an optimal d-algorithm, namely the one found in [5] which has a complexity of $\Theta(N^2)$.

Concerning the parallel case, we found that, when the parallel implementation of a static algorithm offers some (however small) speedup, then the d-algorithm based on that static algorithm will efficiently exploit this feature, such that the speedup may grow without bound for that d-algorithm. On the other hand, for those problems that take no advantage at all of a parallel implementation in the static case, a d-algorithm will manifest no speedup.

For example, consider the following *list scanning* problem defined in [9]: Given only a starting and an ending point in a linked list, it is required that the list be scanned between those points, some processing being required for each visited node; in the data-accumulating case, new nodes may be inserted in the list while the scanning is in progress [9]. In light of the results in this paper, it is unlikely that a parallel d-algorithm for the list scanning problem would admit any speedup, since a parallel static algorithm for this problem is likely to manifest unitary speedup only, as shown in [2], where the same problem (in the static case) is independently found and analyzed (exercise 6.13).

References

[1] A. V. Aho, J. E. Hopcroft, J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[2] S. G. Akl, *Parallel Computation: Models and Methods*, Prentice-Hall, 1997.

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

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

[5] S. D. Bruda, S. G. Akl, *On the Data-Accumulating Paradigm*, Proceedings of the Fourth International Conference on Computer Science and Informatics, 1998.

[6] J. Hartmanis, P. M. Lewis II, R. E. Stearns, *Classifications of Computations by Time and Memory Requirements*, Proceedings of the IFIP Congress 65, 1965, 31-35.

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

[8] T.-H. Lai, S. Sahni, *Anomalies in Parallel Branch-and-Bound Algorithms*, Communications of the ACM, 27, 1984, 594–602.

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

[10] F. Luccio, L. Pagli, *The p-Shovelers Problem (computing with time-varying data)*, Proceedings of the IEEE Symposium on Parallel and Distributed Processing, 1992, 188–193.

[11] H. R. Lewis, C. H. Papadimitriou, *Elements of the Theory of Computation*, Prentice-Hall, 1981.

[12] F. P. Preparata, M. I. Shamos, *Computational Geometry. An Introduction*, Springer-Verlag, 1985.

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