

# 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  
{bruda,akl}@cs.queensu.ca

## 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

The most cited result with respect to the limits of parallel computation 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 [3, 12]. The first observation that contradicts this result was based on parallel search algorithms, which have been found to exhibit superunitary behaviour on particular shapes of the search space [7]. Later, additional examples of such algorithms were found [1, 2], this time manifesting superunitary behaviour in all instances of the solved problem. 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 [9], 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, termi-

nates when all the currently arrived data have been processed before another datum arrives. This paradigm is studied further in [10] and [4], where complexity-related properties are derived for both the parallel and sequential cases.

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. The identity between *d-algorithms* and *on-line algorithms* 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 find that, as long as the speedup in the static case<sup>1</sup> is larger than 1, the speedup of the *d-algorithm* can be made arbitrarily large. On the other hand, when the static case manifests a unitary speedup, then the parallel *d-algorithm* will keep this property.

In the following, a proposition is a result proved elsewhere. The word “iff” stands for the phrase “if and only if”.

## 2. The data-accumulating paradigm

We present here the necessary preliminaries concerning the data-accumulating paradigm, conforming to [10] (see also the definition 3.1 from section 3). A standard algorithm, working on a non-varying set of data, is referred to as a *static algorithm*. For a *d-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  $\Pi$ . Let the best known static algorithm for  $\Pi$  be  $A'$ . Then, a *d-algorithm*  $A$  for  $\Pi$  work-

---

<sup>1</sup>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.

---

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

ing 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. When referring to the parallel case, we add the subscript  $p$ . We will denote the arrival law by  $f(n, t)$ . The amount of data processed by a d-algorithm will be given then by the implicit equation  $N = f(n, T(N))$ .

Note the difference between the running time and the (time) complexity in the data-accumulating paradigm. Since  $N$  itself is a function of time, the actual running time is 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  $t$ . Note that, in the static case, the running time and the time complexity as defined here are identical.

The form proposed in [10] for the data arrival law is

$$f(n, t) = n + kn^\gamma t^\beta, \quad (1)$$

where  $k$ ,  $\gamma$ , and  $\beta$  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 [10] that the termination time of a 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. This implies that a d-algorithm has a time complexity of  $T(N) = cN^\alpha$ , for some positive constants  $c$  and  $\alpha$ . The number of processors used by a parallel algorithm is denoted by  $P$ .

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

### 3. Characterizing d-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, and (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 ( $n = 0$ ).  $\square$

The first condition is the definition implicitly given in [10]. 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.

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  $D_1, \dots, D_k$ . If the algorithm  $A$  is able to terminate at some point, that point will be denoted by  $N_j$ ,  $j \in \{1, 2, \dots\}$ . Note that  $N$  (the amount of data processed by a d-algorithm) is also a termination point.

#### 3.1. A Turing machine model

**Definition 3.2** A Turing machine  $M$  which models an algorithm that is able to terminate at some point other than  $N_\omega$  is the tuple  $(K, \Sigma, \delta, h')$ ,  $K$  being the (finite) set of states,  $\Sigma$  the (finite) tape alphabet,  $\delta$  the transition function, and  $h'$  the initial state. The machine  $M$  has two tapes, as in [5]: 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,  $\delta$  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.  $\square$

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. Since a d-algorithm should immediately terminate on an empty initial input, we impose  $h'$  as the initial state.

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

*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  $\Sigma$  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  $\zeta$ . 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 at all) and, since  $M$  already reached  $h'$  for an arbitrary input, it will reach it again, after a number of steps bounded by  $\zeta$  and after visiting a constant number of new tape cells, because  $M$  is deterministic.  $\square$

**Theorem 3.2** *A Turing machine  $M$  as in definition 3.2, working on any input data set of size  $N_\omega$ , where  $N_\omega$  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  $S \subseteq \{1, 2, \dots, N_\omega\}$ , where  $S$  has the following properties: (i) the least element of  $S$  is upper bounded by a finite constant  $\zeta$ , and (ii) the distance between any two consecutive elements in  $S$  is upper bounded by  $\zeta$ .*

*Proof.* By induction on the size of  $S$ , using lemma 3.1.  $\square$

Given a constant  $\zeta$ , 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  $\Sigma$ , then “folding” each sequence of  $\zeta$  non-blank tape cells into one cell, and finally modifying the function  $\delta$  accordingly (see for example the proof given in [8] 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  $N_\omega$ , where  $N_\omega$  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  $\{1, 2, \dots, N_\omega\}$ .*  $\square$

### 3.2. On-line algorithms

Basically, an *on-line* algorithm processes each input datum  $D_k$  without looking ahead at any datum  $D_{k'}$ ,  $k' > k$ . By contrast, an algorithm that needs to know all the input in advance is called an *off-line* algorithm. 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.

An on-line algorithm is defined in [11] as an algorithm that cannot look ahead at its input. A similar definition in terms of Turing machines can be found in [5]. Finally, an

on-line algorithm  $A$  is defined in [6] 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 [11] leaves the way of reporting the result unclarified. However, if the definition in [11] 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 [6].

With the above 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  $C(n)$ . For any positive integer  $n$ , denote by  $t_1$  a lower bound on  $C(n)$ , and let  $t_2$  be an upper bound on  $C(n + 1)$ , for any possible input data sets of size  $n$  and  $n + 1$ , respectively. But then one can build a function  $f(n_1, t)$ , strictly increasing with respect to  $t$ , such that  $f(n, 0) = n$ ,  $f(n, t_1) = n + 1.1$ , and  $f(n, t_2) = n + 1.5$ .  $\square$

## 4. On the parallel speedup

The main measure used for evaluating a parallel algorithm is the *speedup*, defined as follows. Given some problem  $\Pi$ , the speedup provided by an algorithm that uses  $p_1$  processors over an algorithm that uses 1 processor with respect to problem  $\Pi$  is the ratio  $S(1, p_1) = \tau_\Pi(1)/\tau_\Pi(p_1)$ ,  $p_1 > 0$ , where  $\tau_\Pi(x)$  is the running time of the best  $x$ -processor algorithm that solves  $\Pi$ .

We start by quoting the main result from [10] 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  $c_p/P < c$ ,  $\frac{t}{Pt_p} \rightarrow N_\omega$  for  $n \rightarrow \frac{1}{kc^{1/\alpha}}$ , where  $\alpha\beta = \gamma = 1$ , and  $P = \xi(n + kn^\gamma t_p^\beta)^\delta$ , with some constants  $\xi$ ,  $\xi > 0$ , and  $\delta$ ,  $0 \leq \delta \leq \alpha$ .

3. For all values of  $\alpha, \beta, \gamma, \frac{t}{Pt_p} > \frac{c}{c_p}$ .

□

Let us first take a look at how the implicit equation for the parallel running time has been derived. Generally,  $t_p = c_p T'_p(n + kn^\gamma t_p^\beta)$ . Only *work-optimal* parallel algorithms<sup>2</sup> are considered in [10]. In this case, a static parallel algorithm requires time  $T'_p(N) = O(N^\alpha/P)$ , and the implicit equation for the running time of a parallel d-algorithm follows immediately. However, in the case of a non-work-optimal parallel static algorithm, we have analogously the relation  $T'_p(N) = O(N^\alpha/S'(1, P))$ . Then,

$$t_p = \frac{c_p(n + kn^\gamma t_p^\beta)^\alpha}{S'(1, P)}. \quad (2)$$

The following extension of proposition 4.1 is hence 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  $c_p/S'(1, P) < c, \frac{t}{S'(1, P)t_p} \rightarrow N_\omega$  for  $n \rightarrow \frac{1}{kc^{1/\alpha}}$ , where  $\alpha\beta = \gamma = 1$ .
3. For all values of  $\alpha, \beta, \gamma, \frac{t}{t_p} > \frac{c}{c_p} S'(1, P)$ .

□

**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  $c_p/S'(1, P) < c: \frac{t}{Pt_p} \rightarrow N_\omega$  for  $n \rightarrow \frac{1}{kc^{1/\alpha}}$ , where  $\alpha\beta = \gamma = 1$ , and  $0 \leq \delta \leq \alpha, 0 \leq \epsilon \leq \alpha$ .

*Proof.* Conforming to formula (2), we have  $t_p = (c_p/\xi_1)(n + kn^\gamma t_p^\beta)^{\alpha-\epsilon}$ . But, since  $\alpha\beta = 1$ , the solution  $t_p$  of the above equation is finite for any finite value of  $n$  [10]. Note that, in our case,  $n \rightarrow \frac{1}{kc^{1/\alpha}}$ . 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  $\frac{t}{S'(1, P)t_p} \rightarrow N_\omega$  and, obviously,  $\frac{t}{Pt_p} = \frac{t}{S'(1, P)t_p} \frac{S'(1, P)}{P}$ .

<sup>2</sup>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* [1]. However, we will keep the terminology from [10], because we already used the qualifier “optimal” for d-algorithms.

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 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  $\epsilon$ . 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 (2) 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. We have then  $X(n, t) = X(n, t_p)$ , 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.

If  $\alpha\beta \leq 1$ , then it is immediate that  $\frac{\partial X}{\partial t} < 0$  for any  $n$ . On the other hand, 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). 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

$$n^{\gamma-1} < \frac{1}{2kc^\beta(\alpha\beta - 1)}. \quad (3)$$

But this clearly contradicts the theorem’s hypothesis. But then, for all possible values of the termination time,  $X(n, \cdot)$  is monotonic. □

**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. The only difference in the proof is the way in which the falsity of relation (3) is proved: In this case the relation is immediately false. □

Finally, the most important result in [4] 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  $P$ .*  $\square$

We can now extend this result:

**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  $S'(1, P)$ .*  $\square$

## 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. 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 [10] 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 algorithm has a complexity of  $O(n \log n)$  [13], while it is immediate that an on-line sorting algorithm has a complexity of  $\Omega(n^2)$ . The same result is obtained in [4], 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  $\Pi$ , a  $d$ -algorithm solving  $\Pi$  is optimal iff its complexity matches the lower bound for the complexity of on-line algorithms solving  $\Pi$ . Using this definition, it follows that sorting does admit an optimal  $d$ -algorithm, namely the one found in [4] 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 [10]: 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 [10]. 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 [1], where the same problem (in the static case) is independently found and analyzed (exercise 6.13).

## References

- [1] S. G. Akl. *Parallel Computation: Models and Methods*. Prentice-Hall, Upper Saddle River, NJ, 1997.
- [2] S. G. Akl and L. Fava Lindon. Paradigms admitting superunitary behaviour in parallel computation. *Parallel Algorithms and Applications*, 11:129–153, 1997.
- [3] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, Apr. 1974.
- [4] S. D. Bruda and S. G. Akl. On the data-accumulating paradigm. In *Proceedings of the Fourth International Conference on Computer Science and Informatics*, pages 150–153, Research Triangle Park, NC, October 1998.
- [5] J. Hartmanis, P. M. Lewis II, and R. E. Stearns. Classification of computations by time and memory requirements. In *Proceedings of the IFIP Congress 65*, pages 31–35, Washington, DC, 1965.
- [6] D. Knuth. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*. Addison-Wesley, Reading, MA, 1969.
- [7] T. H. Lai and S. Sahni. Anomalies in parallel branch-and-bound algorithms. *Communications of the ACM*, 27(6):594–602, June 1984.
- [8] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [9] F. Luccio and L. Pagli. The p-shovelers problem (computing with time-varying data). In *Proceedings of the 4th IEEE Symposium on Parallel and Distributed Processing*, pages 188–193, 1992.
- [10] F. Luccio and L. Pagli. Computing with time-varying data: Sequential complexity and parallel speed-up. *Theory of Computing Systems*, 31(1):5–26, Jan./Feb. 1998.
- [11] F. P. Preparata and M. I. Shamos. *Computational Geometry. An Introduction*. Springer-Verlag, New York, NY, 1985.
- [12] J. R. Smith. *The Design and Analysis of Parallel Algorithms*. Oxford University Press, 1993.
- [13] J. D. Ullman, A. V. Aho, and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.