Technical Report No. 98-417

On the Data–Accumulating Paradigm

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

May 4, 1998

Abstract

In the data–accumulating paradigm, the input is an endless stream. A computation is considered to be finished when all the already received data are processed before another datum arrives. We study sorting algorithms in this paradigm. First, we consider the data arrival law as being polynomial in time. We prove the existence of an upper bound on the running time, and hence on the amount of processed data, for any polynomial arrival law. This contradicts previous research, which states that such a limit exists only for particular cases. We extend this result to a large class of data-accumulating algorithms besides sorting. On the other hand, we show that, if the number of processors is large enough, then the sorting algorithm terminates for any polynomial arrival law. Then, we analyze an example of a non-continuous data arrival law. We find similar properties for the sorting algorithm, namely, (i) the existence of an upper bound on the running time, and (ii) termination of the algorithm for any such arrival law when there are sufficiently many processors. Therefore, we conjecture that these properties are not dependent on the expression of the arrival law.

1 Introduction

Researchers in the area of parallel computation have always striven to establish limits on the power of parallel algorithms. One such result 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 [5, 10]. Since, according to this result, the ratio of the speedup achieved to the number of processors used is at most 1, the behavior of any parallel algorithm is at most *unitary*. By contrast, suppose that a parallel algorithm is found to provide a speedup larger than the number of processors it uses. Then this algorithm would contradict the aforementioned result and would be said to exhibit *superunitary* behavior. No such algorithm seemed to exist until some parallel search algorithms appeared to demonstrate superunitary behavior when operating on particular instances of the problem to be solved (see, for example, [6]). However, since this property manifested itself only for some special inputs, no general conclusion could be drawn.

Problems that admit superunitary behavior in all instances were later discovered. They are based on unconventional (yet realistic) paradigms. Some computations that can be described as a disjunction of tasks are considered in [3]. Environments in which such problems exhibit superunitary behavior have been investigated; they include, for example, the existence of time critical tasks that must be solved in a specified time, the high cost of inverting a computation, and so on. The common property of such situations is the existence of a threshold value for the number of processors. Once the number of processors is smaller than this threshold, more work is required to solve the problem. This direction was continued in [2], where new paradigms are investigated, such as (i) memory filling computations, in which each iteration of the algorithm produces a value and an address where that value is to be stored, and the value produced by an iteration depends on the previously computed values and addresses, or (ii) dealing with several input streams, when the completion of the computation requires that all the streams be monitored in real time.

In the classical study of algorithms, input data are considered to be available at the beginning of computation. No additional data come while the computation proceeds, and the algorithm will eventually terminate when the input data have been processed. But what happens when the data arrive *while* the computation is in progress? If the arrival rate is sufficiently high, then a sequential algorithm will be simply overwhelmed by the stream of data and will not be able to process it fast enough. A parallel algorithm, on the other hand, may succeed to handle all the incoming data. This paradigm is called the *data–accumulating paradigm* [7, 8, 9]. The data are considered here as being an endless stream. An algorithm terminates when all the data that were already received have been processed. It is shown that the speedup offered by a parallel algorithm can be made arbitrarily large by modifying the parameters of the data arrival law and/or the number of processors. The data arrival law considered in [7, 8, 9] is polynomial with respect to the elapsed time.

In this paper we continue the investigation of data–accumulating algorithms. As a case study we consider sorting algorithms, which are basic elements for various applications. It has been shown [7] that, if the data arrive too fast, then a successful algorithm (that is, one that terminates) must have a running time upper bounded by a constant. In other words, it must process all the received data (before a new datum arrives) in a time which is no larger than that constant. This, of course, is not always possible. In fact, when the running time exceeds that constant, then the algorithm never terminates. Indeed, our results indicate that such a constant upper bound for the running time exists for *any* polynomial data arrival law. This is a negative, yet important, result because it establishes a limit that was not previously known. However, we also show that any arrival law can be handled in finite time with a sufficiently large number of processors. Subsequently, we extend these results in two ways. First, we consider another data arrival law which is significantly different from the polynomial one, being a non-continuous function. Second, we show that the constant upper bound mentioned above is not a particular feature of sorting, but it occurs for a large class of problems.

The paper is organized as follows. In section 2 we summarize the results concerning the old upper bound for speedup. In section 3 we formalize the class of data–accumulating algorithms, and in section 4 we study sorting algorithms with a data arrival law conforming to this paradigm, also generalizing the results obtained. We conclude in the last section.

In the following, a proposition is a result proved elsewhere. Where possible, the name of the original theorem is provided. Throughout the paper we use the RAM and the PRAM as our sequential and parallel computational models, respectively [2].

2 Classical Speedup Bounds

The most common measure of the performance of a sequential algorithm is the *speedup*. Given some problem Π, the speedup of an algorithm that uses p_1 processors over an algorithm that uses p_2 processors with respect to problem Π is the ratio $S(p_2,p_1) = T_\Pi(p_2)/T_\Pi(p_1)$, $p_1 > p_2 > 0$, where $T_\Pi(x)$ is the running time of the best x-processor algorithm that solves Π. In many cases [2] this definition is used to compare a parallel algorithm with a sequential one, that is, $p_2 = 1$.

The *efficiency* is defined [2] as the ratio of the speedup over the number of processors, that is, $E(1,p) = S(1,p)/p$. When the efficiency is defined in this way, we will omit the first argument and denote the efficiency simply by $E(p)$. However, in some cases [3], the efficiency is defined in a more general way: $E(p_1,p_2) = \frac{T_\Pi(p_1)}{T_\Pi(p_2)} \times \frac{p_1}{p_2}$, where T_Π is defined as above.

In a conventional environment, the speedup (and hence the efficiency) of a parallel algorithm is bounded by the following results [2, 5, 10].

Proposition 2.1 (Speedup Theorem [10]) *$S(1,p) \leq p$ or, equivalently, $E(1,p) \leq 1$.*

Proposition 2.2 (Brent's Theorem [5]) *If a computation C can be performed in time t with q operations and sufficiently many processors that perform arithmetic operations in unit time, then C can be performed in time $t + (q-1)/p$ with p such processors.*

These two propositions state that, when more processors solve a given problem, then the increase in performance is at most proportional to the increase in the number of processors.

3 Data–Accumulating Paradigm

Some paradigms for which the bounds from propositions 2.1 and 2.2 do not hold are presented by Luccio and Pagli:

- The input data are not all available at the beginning of the computation; more data come while the algorithm is operating [7, 8, 9];
- PRAM processors may fail; then, the algorithm must be organized such that, for any processor failure, all pending operations can still be executed on the remaining processors [8];
- Computing in a secure environment; some actions are performed by several distinct agents in order to certify the reliability of the result [8].

We focus in this paper on the first of these paradigms. Two variants of it have been investigated [7]: (i) the computation terminates when all the data currently arrived have been treated, and (ii) data are considered an endless stream, but the computation is divided into stages, where the work in each stage is based on the data set updated with the data received in the previous stage. However, results obtained for the first variant may easily be extended to the second one. Therefore, we are concerned henceforth only with the case in which the computation terminates when all the currently received data have been processed. Algorithms pertaining to this class are called *data-accumulating algorithms* or, for short, *d-algorithms*.

A standard algorithm, working on a non-varying set of data, is referred to as a *static* algorithm. 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* if and only if 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 $t = 0$.

The data arrival law can have any form. However, only non-decreasing functions make sense. The form proposed in [7] is

$$N = n + kn^{\gamma}t^{\beta}, \tag{1}$$

where n is the size of the initial data set and k, γ, and β are positive constants.

The problems considered in [7] are solvable in polynomial time, that is, $T'(N) = O(N^{\alpha})$, where α is a positive constant, with T' defined as above. This implies that an optimal d-algorithm must have a time complexity of $T(N) = cN^{\alpha}$, for some positive constant c[1]. If the d-algorithm terminates at time t, then we have $t = cN^{\alpha}$. Considering the data arrival law given by relation (1), the termination time t can be expressed by the implicit function

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

[1] Note, however, that our study in the present paper covers d-algorithms that have a time complexity of cN^{α}, even if they are not optimal.

It is shown in [7] that the d-algorithm will terminate for n in some interval that depends on the constants involved only for certain values of α, β, and γ. For example, it is claimed that, if $\gamma = 1$, for $\alpha\beta < 1$, then n grows as $(1/(kc^{1/\alpha}))t^{(1-\alpha\beta)/\alpha}$. That is, for any n, the d-algorithm will eventually terminate, even if its running time can be very large (the expression of t as a function of n may be exponential). For $\alpha\beta \geq 1$, on the other hand, the algorithm terminates only for bounded values of n; moreover, for $\alpha\beta > 1$, the running time itself is upper bounded by a constant (and this bound is not present in the other cases).

The parallel d-algorithm is defined in the same manner as the sequential one, except that its execution time $T_p(N)$ is compared to the execution time $T'_p(N)$ of a parallel static algorithm that uses the same number of processors $P(N)$ as the parallel d-algorithm. The equation for the parallel optimal d-algorithm is similar to the one obtained for the sequential case:

$$t_p = \frac{c_p(n + kn^\gamma t_p^\beta)^\alpha}{P}, \tag{3}$$

where the subscript p denotes the parallel case and P is the number of processors. It is assumed in [7] that P is a polynomial function of N.

We quote the following main results obtained in [7]:

Proposition 3.1 *For a problem admitting an optimal sequential d-algorithm obeying relation (2) and an optimal parallel d-algorithm obeying relation (3) we have*

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

For $c_p/P < c$ we have:

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

where $\alpha\beta = 1$, $\gamma = 1$, and $P = \xi(n + kn^\gamma t_p^\beta)^\delta$, with some constants ξ, $\xi > 0$, and δ, $0 \leq \delta \leq \alpha$, and

$$\frac{t}{Pt_p} > \frac{c}{c_p},$$

for all values of α, β, γ.

Proposition 3.2 (Corollary) *For optimal d-algorithms the bound on parallel speedup of proposition 2.1 is violated for proper values of the parameters. In particular, this always occurs for $c_p = c$.*

This result gives a precise formula for parallel speedup when $\alpha = \beta = \gamma = 1$ and shows that the quantity t/Pt_p may become arbitrarily large. This implies that, for optimal d-algorithms, the bound for parallel speedup from proposition 2.1 does not hold for proper values of the parameters (and this always occurs for $c_p = c$). Also, Brent's theorem (proposition 2.2) does not hold.

In the following sections we study d-algorithms for the problem of sorting a sequence of numbers in non-decreasing order. A new restriction on the running time is derived, namely, that a constant upper bound exists on the running time which does not depend on β or n (as previously thought), but only on α. On the other hand, propositions 3.1 and 3.2 still hold, having even a stronger form (namely, data arriving according to *any* law can be processed in finite time with sufficiently many processors). We also generalize these results for a large class of problems.

4 Sorting

In the following, we study sorting with respect to the data–accumulating paradigm. Note that a similar problem is investigated in [7], but there the result is a search tree. This implies an $O(\log n)$ access time, which may be unacceptable for a large data structure. Therefore, an $O(1)$ access time data structure is preferred in many applications. That is, the sorting algorithms discussed here will output an *array* of sorted elements. Henceforth, we refer to such processing as *sorting on a linear structure* (or simply *sorting*, for short, when there is no ambiguity).

One may say that an array is a static data structure and thus it is not suitable for varying data. However, arrays that extend themselves in order to accommodate an increasing amount of data are known (for example the `Vector` data type in the Java language [4]).

For comparison purposes, we use a static sorting algorithm whose running time is $\Theta(n \log n)$ [1].

4.1 The Sequential Case

By definition, data accumulates as the computation proceeds. Generally, we consider that the incoming data are buffered until some amount q is reached, and then the buffered data are inserted into the already sorted sequence (for the non-buffered case simply set q to 1). The time complexity of such an operation is given by the following lemma.

Lemma 4.1 *Let the length of the already sorted sequence be l. Then, the time complexity of inserting q new elements into the sorted sequence is $\Theta(q \log q + l)$, for any q and l such that either $q = 1$ or $l \geq (q \log q)/(q-1)$.*

Proof. In order to prove the upper bound on the time complexity, note that it is enough to sort the newly arrived sequence (this will take $O(q \log q)$ time) and then merge it with the old sequence (in time $O(q + l)$), for an overall running time of $O(q \log q + l)$.

For the lower bound, we have the following:

Supposing that the distance between the insertion point of some element and the end of the buffer is $\Omega(l)$ (which is the general case), the time required for inserting a datum into some sorted sequence is also $\Omega(l)$, since all data that are behind the insertion point should be shifted to make room for the new one. But, if the newly arrived data are not sorted, then one should insert them into the buffer one by one. Therefore, the insertion time for the whole sequence is $\Omega(ql)$.

On the other hand, the number of operations needed to sort q data is $\Omega(q \log q)$ [1], while merging the sorted sequences requires $\Omega(q + l)$ operations [1]. Hence, the lower bound for the entire process in this case is $\Omega(q \log q + l)$.

Therefore, the overall lower bound is given by $\Omega(q \log q + l)$ provided that $q = 1$ or $ql \geq q \log q + l$, that is, $l \geq (q \log q)/(q-1)$. □

Note that most implementation details were left out of the proof. These details, however, do not affect the complexity. For example, *two* buffers are actually needed: one of them contains data that are processed (that is, sorted and merged), and the other accumulates data that continue to arrive. After the first buffer is processed, the functions of the two buffers are switched and the computation continues.

We did not consider the case in which more than q elements come before the processing of the already arrived data is finished. But this case is equivalent to the one in which the data arrive too fast and the d-algorithm never stops. Indeed, as seen below, the necessary condition (5) for the algorithm to terminate in finite time covers this case.

Considering the data arrival law of relation (1), the time t_q in which the buffer is filled is given by the equation

$$q = kn^{\gamma} t_q^{\beta}. \tag{4}$$

At some time t_x, when the buffer is filled and is ready to be inserted, the length of the already sorted sequence will be $l = n + kn^\gamma(t_x^\beta - t_q^\beta)$, because all the arrived data have been inserted, except the data which are in the buffer (otherwise, some elements are lost). We will consider for simplicity that $\beta = 1$, but similar results can be obtained for other values as well. In this context, the time interval between two arrivals is $dt = 1/kn^\gamma$. Suppose that the algorithm stops at time t. This means that, at time t, all the buffered data have been inserted before another datum arrives. That is, the time required to insert the buffer (given by lemma 4.1) should be no greater than dt:

$$dt \geq q \log q + n + kn^\gamma(t - t_q), \tag{5}$$

or, considering relation (4) and the expression for dt,

$$\frac{1}{kn^\gamma} \geq q \log q + n + kn^\gamma(t - \frac{q}{kn^\gamma}). \tag{6}$$

Simple calculations let us derive the following bound on the computation time:

$$t \leq \frac{1}{(kn^\gamma)^2} + \frac{1}{kn^\gamma} q(1 - \log q) - \frac{n}{kn^\gamma}. \tag{7}$$

This imposes a limit on the running time of any sequential sorting d-algorithm that terminates for the polynomial data arrival law given by relation (1). Henceforth, the right-hand side of relation (7) will be denoted by t'_B[2].

We are now ready to determine the complexity of sorting on a linear structure.

Theorem 4.2 *Under the polynomial data arrival law given by relation (1), the running time of any sequential d-algorithm for sorting on a linear structure is* $\Theta(N^2)$.

Proof. Let us assume first that q is a constant. Specifically, q does not depend on elapsed time (however, if it is a function of n, then the proof is not affected). By lemma 4.1, the time required to insert the q elements from the buffer is $\Theta(q \log q + l)$, where l is the number of already sorted elements. This time is $\Theta(l)$, since q is constant. But l increases by q each time a new buffer is inserted. The initial size of l is the amount of initial data, namely, n. Therefore, the total time required for sorting N elements is

$$\Theta\left(\sum_{l=n}^{N/q} ql\right) = \Theta(N^2), \tag{8}$$

as claimed.

However, one may consider that the size q of the buffer varies with time. We have

$$\begin{aligned}
\frac{\partial t'_B}{\partial t} &= \frac{\partial}{\partial t}\left(\frac{1}{(kn^\gamma)^2} + \frac{1}{kn^\gamma} q(1 - \log q) - \frac{n}{kn^\gamma}\right) \\
&= -\frac{1}{kn^\gamma} \log q \frac{\partial q}{\partial t}.
\end{aligned}$$

Obviously, $q \geq 1$ and $t'_B > 0$ (otherwise, the algorithm never terminates). If there is some constant (with respect to the time) Q such that $q \leq Q$ everywhere, then relation (8) holds (because q may then be approximated by Q and the rate of growth does not change). Therefore, it is enough to consider the case of q being an increasing function. Then, $\frac{\partial q}{\partial t} > 0$, and this implies $\frac{\partial t'_B}{\partial t} < 0$, because $\log q > 0$ for $q > 1$. That is, t'_B is a decreasing function. Since $t'_B(0)$ is finite, there exists

[2]This notation was chosen in order to be consistent with the notation from [7]. There, t_B denotes an upper bound on the running time. Obviously, relation (7) also defines an upper bound.

some Q such that $t'_B(q) \leq 0$, for all $q \geq Q$. But, from relation (7), the termination time of the algorithm is less than t'_B. Therefore, when $q \geq Q$, the upper limit for the termination time is negative, which means that the algorithm never stops. Thus the values q for which the algorithm terminates are bounded again and we are in the case covered by relation (8). This completes the proof. □

Corollary 4.3 *Sorting on a linear structure does not admit an optimal d-algorithm.*

4.2 Extending the Upper Bound on the Running Time

In the previous section we established an upper bound on the running time only for proving the complexity of sorting d-algorithms. In this section, we analyze this bound. We begin by considering sorting d-algorithms, and then generalize our results for other d-algorithms.

First, note that a side consequence of the proof of theorem 4.2 is that the best value for the buffer size q is the minimal possible, that is, 1, because t'_B is a decreasing function with respect to q. In other words, there is no reason to buffer data; it is better to insert each arrived datum in linear time. Hence, we will consider $q = 1$. Second, when obtaining relation (7), we considered only the rate of growth of the functions involved. If we want a quantitative result, then the time required in order to insert one element into the already sorted sequence ($q = 1$) is cl, for some constant c. Relation (6) becomes in this case

$$\frac{1}{kn^\gamma} \geq c(n + kn^\gamma t), \tag{9}$$

and hence

$$t'_B = \frac{1}{c(kn^\gamma)^2} - \frac{n}{kn^\gamma}. \tag{10}$$

Relation (10) defines the limit in terms of n. Therefore, it can be as small as desired (in particular, smaller than t_B) for appropriate values of n.

The limits obtained in [7] for the running time and the initial amount n of data for $\gamma = 1$ are

$$t_B = \frac{1}{(k(\alpha\beta - 1))^{1/\beta}},$$

and

$$n_B = \frac{(\alpha\beta - 1)^{(\alpha-1)/\alpha\beta}}{c^{1/\alpha}k^{1/\alpha\beta}\alpha\beta}.$$

But, in our case, $\alpha = 2$ by theorem 4.2 and $\beta = 1$. Then, the two relations above become

$$t_B = \frac{1}{k},$$

$$n_B = \frac{1}{2c^{1/2}k^{1/2}}.$$

For n_B as above, t'_B becomes $3/k$. Therefore, our limit turns out to be too optimistic (even if of the same order). However, it provides a way of characterizing sorting d-algorithms. Also, note that t'_B depends on the initial amount of data n. In fact, we can provide a stronger result for the general case.

We have considered $\beta = 1$. But this implies that the product $\alpha\beta$ is larger than 1, and the existence of a limit on the running time in this case was established in [7]. More interesting is the situation where $\beta \leq 1/2$, for $\alpha\beta \leq 1$. Under these conditions, no limit on the running time is known. We now study this case.

Theorem 4.4 *For the polynomial data arrival law given by relation (1), if a sorting d-algorithm terminates, then its running time is upper bounded by a constant T that does not depend on n.*

Proof. We will have the restriction $\gamma = 1$ for easier calculations. Note that this does not make the problem less general, since both k and n are constants with respect to time. Therefore, for any term kn^γ one can choose another constant k', such that $kn^\gamma = k'n$ ($k' = kn^{\gamma-1}$).

The time dt after which a new datum arrives is given by

$$1 = kn^\gamma((t + dt)^\beta - t^\beta),$$

for some moment t. That is,

$$(t + dt)^\beta - t^\beta = \frac{1}{kn^\gamma}. \tag{11}$$

On the other hand, relation (5) in the general case becomes

$$dt \geq q \log q + n + kn^\gamma(t^\beta - \frac{q}{kn^\gamma}),$$

which leads to

$$\begin{aligned} dt + t &\geq q \log q + n + kn^\gamma(t^\beta - \tfrac{q}{kn^\gamma}) + t && \Leftrightarrow \\ dt + t &\geq q(\log q - 1) + n + kn^\gamma t^\beta + t && \Leftrightarrow \\ (dt + t)^\beta &\geq (q(\log q - 1) + n + kn^\gamma t^\beta + t)^\beta. \end{aligned}$$

By relation (11), the last expression becomes

$$\frac{1}{kn^\gamma} \geq (q(\log q - 1) + n + kn^\gamma t^\beta + t)^\beta - t^\beta. \tag{12}$$

In particular, for $\gamma = 1$,

$$\frac{1}{kn} \geq (q(\log q - 1) + n + knt^\beta + t)^\beta - t^\beta. \tag{13}$$

But the complexity of the sorting algorithm is $O(N^2)$ by theorem 4.2. That is, the running time is given by

$$t = cN^2 = c(n + kn^\gamma t^\beta)^2,$$

and, for $\gamma = 1$, this implies

$$n = \frac{t^{1/2}}{c(1 + kt^\beta)}. \tag{14}$$

From relations (13) and (14) we have

$$\begin{aligned} \tfrac{qc(1+kt^\beta)}{kt^{1/2}} &\geq (q(\log q - 1) + \tfrac{t^{1/2}}{c} + t)^\beta - t^\beta && \Leftrightarrow \\ \tfrac{qc}{k} &\geq \tfrac{t^{1/2}}{(1+kt^\beta)} \left((q(\log q - 1) + \tfrac{t^{1/2}}{c} + t)^\beta - t^\beta\right). \end{aligned}$$

Let us denote the term $\frac{t^{1/2}}{(1+kt^\beta)}$ by $b(t)$ and $(q(\log q - 1) + \frac{t^{1/2}}{c} + t)^\beta - t^\beta$ by $a(t)$. Then, the above expression becomes

$$\frac{qc}{k} \geq b(t) \times a(t). \tag{15}$$

But

$$\frac{\partial b(t)}{\partial t} = \frac{t^{-1/2}}{(1+kt^{\beta})^2}(1/2 + k(1/2-\beta)t^{\beta}),$$

and hence, for $\beta \leq 1/2$, $\frac{\partial b(t)}{\partial t} > 0$. That is, $b(t)$ is an increasing function. Analogously, $a(t)$ is an increasing function as well:

$$\begin{aligned}
\frac{\partial a(t)}{\partial t} &= \beta(\frac{t^{-1/2}}{c}+1)(q(\log q-1)+\frac{t^{1/2}}{c}+t)^{\beta-1} - \beta t^{\beta-1} \\
&> \beta\left((q(\log q-1)+\frac{t^{1/2}}{c}+t)^{\beta-1} - t^{\beta-1}\right) \ \ [\text{because}\, t^{-1/2}/c > 0] \\
&> 0 \ \ [\text{because}\, q(\log q-1)+t^{1/2}/c > 0].
\end{aligned}$$

Therefore, $b(t) \times a(t)$ is increasing. Moreover, for $\beta < 1/2$,

$$\lim_{t\to\infty} b(t) = \infty, \tag{16}$$

since $b(t)$ is the ratio of two polynomials in t and the degree of the numerator (namely, $t^{1/2}$) is larger than the degree of the denominator (namely, $1+kt^{\beta}$).

But $a(t)$ is a strictly increasing function as shown above and

$$\begin{aligned}
a(1) &= (1/c+1)^{\beta} - 1 \\
&> 0
\end{aligned}$$

(for suppose that $(1/c+1)^{\beta} - 1 \leq 0$; then, $1/c+1 \leq 1$ and this implies that $c \leq 0$, which is absurd). But then $a(t) > 0$ for any $t \geq 1$. Therefore,

$$\lim_{t\to\infty} a(t) > 0. \tag{17}$$

By relations (16) and (17),

$$\lim_{t\to\infty} b(t) \times a(t) = \infty,$$

for $\beta < 1/2$.

For $\beta = 1/2$, $lim_{t\to\infty} b(t) = 1/k$. In this case, $a(t) \geq (t^{1/2}/c+t)^{1/2} - t^{1/2}$. Suppose now that $(t^{1/2}/c+t)^{1/2} - t^{1/2} < t^{1/8}$. This implies $t^{1/2}/c + t < t^{1/4} + t + 2t^{5/8}$. That is, $t^{4/5} < ct^{2/5} + 2c$, which is obviously false for t large enough. Hence, for such a t, $a(t) \geq t^{1/8}$ and thus $lim_{t\to\infty} a(t) = \infty$. But then, again, $\lim_{t\to\infty} b(t) \times a(t) = \infty$.

The facts that $b(t) \times a(t)$ is an increasing function and that its limit is infinite imply that there exists some finite T such that $b(t) \times a(t) > \frac{qc}{k}$ for any $t > T$. But then, such a t larger than T will contradict the condition necessary for algorithm termination and given by relation (15). Hence, T is an upper bound for the running time and this completes the proof. $\square$

Note that the theorem implicitly gives an upper bound for the maximum amount of data which can be processed, because this amount is given by $N = n + kn^{\gamma}t^{\beta}$ and its upper bound is obviously $n + kn^{\gamma}T^{\beta}$. We contradict by theorem 4.4 the results derived in [7], where it is claimed that such a bound does not exist for $\alpha\beta < 1$.

In the case of sorting on a linear structure we found an upper bound on the running time for any data arrival law. But sorting is not the only case in which such a bound exists.

Theorem 4.5 *For the polynomial data arrival law given by relation (1), let A be any 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.*

Proof. It is enough to consider the case $\beta \leq 1/\alpha$, because in the other case (namely, $\alpha\beta > 1$) the limit has been already found [7].

Let $\epsilon = \alpha - 1$. This implies $\epsilon > 0$. If the algorithm terminates at some finite time t, then N data have been processed, where $N = n + kn^\gamma t^\beta$. That is, the time for processing one datum is $cN^\alpha/N = cN^\epsilon$ for some positive constant c. Following the same idea as the one used for deriving relation (5), we have

$$dt \geq c(n + kn^\gamma t^\beta)^\epsilon,$$

which is similar to relation (5). Therefore, analogously to the proof of theorem 4.4, we obtain for $\gamma = 1$

$$\frac{qc}{k} \geq \frac{t^{1/2}}{(1 + kt^\beta)} \left((c(n + knt^\beta)^\epsilon + t)^\beta - t^\beta\right). \tag{18}$$

But the left-hand side of this relation is increasing, because $c(n+knt^\beta)^\epsilon > 0$, and it is immediate that the limit of the right-hand side is infinite. Hence, the limit T is derived in the same way as in the proof of theorem 4.4. □

4.3 The Parallel Case

Recall that P is the number of processors in the parallel model.

Lemma 4.6 *The process described in lemma 4.1 admits linear speedup.*

Proof. Sorting q elements admits linear speedup [2] (page 179). Inserting the buffer into the previously sorted sequence may be achieved by using an optimal merging algorithm [2] (page 209). □

Conforming to lemma 4.6, relation (6) becomes in the parallel case

$$\frac{P}{kn^\gamma} \geq q \log q + n + kn^\gamma (t - \frac{q}{kn^\gamma}).$$

This implies

$$t \leq \frac{P}{(kn^\gamma)^2} + \frac{1}{kn^\gamma} q(1 - \log q) - \frac{n}{kn^\gamma}. \tag{19}$$

As expected, relations (7) for the sequential case and (19) for the parallel one are similar. Therefore, all the above sequential results hold for the parallel case as well. That is, the best value for q is 1 (buffering no longer helps), and a similar limit for the running time can be found. This limit is

$$t''_B(P) = \frac{P}{c_p(kn^\gamma)^2} - \frac{n}{kn^\gamma}. \tag{20}$$

Note the close similarity with the limit obtained for the sequential case, given by relation (10). It is easy to see that theorem 4.4 holds for the parallel case as well.

Theorem 4.7 *If the running time of a sequential sorting d-algorithm is larger than t'_B and the running time for a P-processor parallel sorting d-algorithm is less than $t''_B(P)$, then the speedup $S(1, P)$ is infinite.*

Proof. The theorem is established by recalling that t'_B and $t''_B(P)$ are the respective upper bounds on the running times of sequential and parallel sorting d-algorithms that terminate under the polynomial data arrival law given by relation (1). $\square$

The term "infinite speedup" means that the sequential d-algorithm never stops (since its running time is larger than t'_B), while the parallel one produces a result in finite time. The following characterization is similar to proposition 3.1.

Theorem 4.8 *For the problem of sorting, if $c_p < cP$, then*

$$S(1,P) > P \times \frac{c}{c_p},$$

or, alternatively,

$$E(P) > \frac{c}{c_p}.$$

Proof. Following the same idea as in [7], from relations (2) and (3) with $\alpha = 2$ we have

$$S(1,P) = \frac{t}{t_p} = P\frac{c}{c_p}\left(\frac{n + kn^{\gamma}t^{\beta}}{n + kn^{\gamma}t_p^{\beta}}\right)^2$$

But it is immediate that $t > t_p$ if $c_p < cP$ and therefore $(n + kn^{\gamma}t^{\beta})/(n + kn^{\gamma}t_p^{\beta}) > 1$. $\square$

It is consistent to assume $c_p < cP$, because usually c_p and c are close to each other. However, $c_p < cP$ will hold for P large enough.

Theorem 4.9 *If the running time of a P-processor sorting d-algorithm is larger than $t''_B(P)$, then the algorithm can be rescheduled for P' processors, $P' > P$, such that the speedup $S(P,P')$ is infinite.*

In other words, if we have enough processors, then the d-algorithm will stop each time.

Proof. If the running time t is larger than $t''_B(P)$, then the P-processor algorithm will never stop. But t''_B is a strictly increasing function with respect to P, therefore there exists some P' such that $t \leq t''_B(P')$. $\square$

We can also give a generalization of theorem 4.5 for the parallel case.

Theorem 4.10 *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.*

Proof. It is enough to replace the first term from the right-hand side of relation (18) in the proof of theorem 4.5 by

$$\frac{P \times t^{1/2}}{(1 + kt^{\beta})}.$$

Then, the proof is analogous, because this replacement introduces a multiplicative constant which does not change the sign of the derivative. Also, the constant P does not change the limit. $\square$

Finally, it is worth pointing out that a closer look at the proof of proposition 3.1 given in [7] reveals that the optimality of the d-algorithm in question is not used to establish the result. Therefore:

Theorem 4.11 *Proposition 3.1 holds for any optimal or non-optimal d-algorithm with polynomial running time.*

Note that we extended proposition 3.1 for a non-optimal d-algorithm (sorting) by theorem 4.8.

4.4 Using Another Data Arrival Law

Until now, we have considered a polynomial data arrival law for the sorting problem, but we expect similar results for other expressions. As an example, we consider here a totally different law:

$$N(t) = q(n) + q(n)\lfloor t/r(n) \rfloor, \tag{21}$$

for some fixed n, where q and r are some functions from $\mathbb{N}$ to $\mathbb{N}$. In plain English, data arrive in bundles of $q(n)$ elements each $r(n)$ time units. This law is interesting because it extends the analysis of d-algorithms to non-continuous functions.

Lemma 4.12 *For the data arrival law given by relation (21) the minimum value for the size of the buffer q is $q(n)$.*

Proof. By relation (21), $q(n)$ data arrive together (at precisely the same time). All of them must be temporarily stored into some buffer until they are inserted into the sorted sequence. □

Note that, in this case, the time interval dt between two data arrivals is $r(n)$. By the same method used to obtain t'_B we have $dt \geq q \log q + N(t - t_q)$. But t_q is null (since the whole bundle of data arrives at once) and $q = q(n)$ by lemma 4.12. Hence,

$$\lfloor t/r(n) \rfloor \leq \frac{r(n)}{q(n)} - (1 + \log q(n)). \tag{22}$$

Relation (22) is the limit for the running time of the sequential algorithm. The limit for a P-processor algorithm may be derived analogously, by considering lemma 4.6 (sorting and merging admit linear speedup):

$$\lfloor t/r(n) \rfloor \leq P\frac{r(n)}{q(n)} - (1 + \log q(n)). \tag{23}$$

The two relations above are very similar to the ones obtained for the polynomial arrival law. Therefore, theorems 4.7, 4.9, and 4.8 hold.

As an example, consider $q(n) = n$ and $P = n$. Then, relation (23) becomes:

$$\lfloor t/r(n) \rfloor \leq r(n) - 1 - \log n.$$

Obviously, $\lfloor t/r(n) \rfloor \geq 0$. Therefore, $r(n) - 1 - \log n \geq 0$. That is,

$$r(n) \geq 1 + \log n. \tag{24}$$

But, intuitively, n processors will succeed to sort the first bundle of n data in $t = O(\log n)$ time. If $r(n)$ is less than t, then new data arrive before the old data have been processed. The system has no chance to handle the current data in $r(n)$ time (and after this time another bundle arrives). Thus the algorithm never stops. On the other hand, if $r(n)$ is large enough (that is, larger than t), then the processing is finished before the second bundle arrives, therefore the algorithm stops at time t.

Note that the lower bound on $r(n)$ obtained in relation (24) is $\Omega(\log n)$. Hence, our result matches in terms of rate of growth the observation in the previous paragraph. The constant "1" in relation (24) stands for merging the sorted buffer with the already sorted sequence (which is simply a copy operation, since the old sequence is void). This means that the equation is not too general but not too restrictive either. We ignored all the constants and thus relation (23) is valid only in terms of rate of growth, but it is simple to introduce constants as in the previous section.

5 Conclusions

In our study of data-accumulating algorithms for sorting, we have taken a different approach than the one in [7]: we first obtained a bound for the running time and, based on this result, we were able to characterize sorting d-algorithms in terms of complexity. This bound also helped us find the optimal size of the input buffer. Note that a sorting (d-)algorithm updates the data structure in a time interval that depends on the number of already processed input data. This is the reason for which the upper bound on the running time t'_B exists. Therefore, we expected similar upper bounds for other problems with this property, such as the construction of search trees (problems 3 and 4 in [7]). Our expectations were justified as shown in theorems 4.5 and 4.10.

Theorems 4.5 and 4.10 are the main results of this paper. They prove the existence of an upper bound on the running time for a large class of algorithms. It has been claimed [7] that the existence of such a limit depends on both α and β. That is, it depends on both the complexity of the d-algorithm and the data arrival law. We have shown that, in fact, the bound depends only on the complexity of the d-algorithm, and this dependence is weak: if the complexity of the d-algorithm is more than linear, then such a bound exists. This is a serious limitation of d-algorithms. However, if the parallel case is considered, this limitation is not that important, since theorems 4.7 and 4.9 show that any arrival law can be handled in finite time if the number of processors is sufficiently large. This shows the importance of parallelism for the theory of d-algorithms.

Considering a data arrival law other than the polynomial one, we found that properties of sorting d-algorithms do not change significantly. Based on this, we conjecture that the general properties of such algorithms hold for any type of data arrival law.

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] K. Arnold, J. Gosling, *The Java Programming Language*, Java Series, Sun Microsystems, 1996.

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

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

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

[8] F. Luccio, L. Pagli, G. Pucci, *Three non Conventional Paradigms of Parallel Computation*, F. Meyer auf der Heide, B. Monien, A. L. Rosenberg (Eds.) Parallel Architectures and Their Efficient Use, Springer Lecture Notes in Computer Science 678, 1992, 166–175.

[9] 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.

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