Finite Termination of Asynchronous Iterative Algorithms

Serap A. Savari[1,2] *Dimitri P. Bertsekas*[3]

Abstract: We consider n-processor distributed systems where the i^{th} processor executes asynchronously the iteration $x_i = f_i(x)$. It is natural to terminate the iteration of the i^{th} processor when some local condition, such as $x_i - f_i(x)$: "small", holds. However, local termination conditions of this type may not lead to global termination because of the asynchronous character of the algorithm. In this paper, we propose several approaches to modify the original algorithm and/or supplement it with an interprocessor communication protocol so that this difficulty does not arise. Some of the resulting procedures can be recast as termination detection schemes for arbitrary finite, distributed computations.

1. Introduction

Convergent asynchronous iterative algorithms are interesting because they offer a greater flexibility of implementation and a faster convergence than their synchronous counterparts. Asynchronous iterations have received a lot of attention (see [1] and [2]), but the issue of termination has been virtually overlooked. There is an extensive literature on the termination detection of asynchronous algorithms that are executed for a finite number of steps (see, for example, [1] and [3-10]). However, in general, there is no bound on the number of iterations performed by asynchronous iterative algorithms. This termination problem was first recognized and formulated in [11] and [2, §9], and the solution proposed to this implementation difficulty is to modify the asynchronous iterative algorithm so that it terminates in finite time and then to use one of the many available protocols for termination detection. The modification to the underlying algorithm suggested in [2] is guaranteed to terminate for many, but not all, classes of convergent asynchronous iterations. In this paper, we will specify an alternate modification that will lead to termination for a larger set of asynchronous iterative algorithms at the expense of increased communication overhead. We are also going to consider an alternate approach to the issue of

[1]The authors are with the Laboratory for Information and Decision Systems and the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, USA.

[2]Supported by an AT&T Bell Laboratories GRPW Fellowship.

[3]Supported by NSF under Grant 9300494-DMI, and the ARO under Grant DAALO3-92-G-0309.

termination in which the finite termination of the algorithm and termination detection in finite time are addressed jointly.

To be more precise, we will investigate the following problem. Let $X_1, \dots, X_n$ be given sets, and let $X = \prod_{i=1}^{n} X_i$ be their Cartesian product. Any $x \in X$ can be represented as $x = (x_1, \dots, x_n)$, where $x_i \in X_i$ for each i. For $i \in \{1, \dots, n\}$, let $f_i : X \mapsto X_i$ be a given function and define $f : X \mapsto X$ by $f(x) = (f_1(x), f_2(x), \dots, f_n(x))$ for every $x \in X$. Ideally, we would like to find some $x^* \in X$ for which $x^* = f(x^*)$; such an x^* is called a *fixed point* of the function f. Toward this end, we will consider distributed asynchronous versions of the iteration

$$x := f(x).$$

We assume that we are given a message-passing system with n processors, each having its own local memory and communicating with the other processors over a reliable communication network. By reliable communication, we mean that information sent over the network will eventually be received without errors at its destination. Each processor can communicate "directly" with a subset of the other processors called its *neighbors.* For each pair (i, j) of neighboring processors, we assume that there is a communication capability by means of which processors i and j can send information to each other; this capability may be established through a direct communication link between the two processors or via a multi-hop path in the communication network. For convenience, we assume that neighboring processors are connected by a communication link. If $f_i(x)$ depends on x_j and $j \neq i$, we require processors i and j to be neighbors; in this situation, we say that processor i is a *dependent neighbor* of processor j and that processor j is an *essential neighbor* of processor i. Furthermore, any processor j can be reached from any other processor i through a sequence of neighbors; if this were not the case, the iteration $x := f(x)$ could be decomposed into two or more smaller and independent iterations.

Let t be an integer "time" variable used to index the sequence of physical times at which the events of interest of this system occur. Note that t may have little relation with "real time." At time t, each processor i stores a vector $x^i(t) = (x^i_1(t), \dots, x^i_n(t))$, where for all $j \neq i$, $x^i_j(t) = x^j_j(\tau^i_j(t))$ for some $0 \leq \tau^i_j(t) \leq t$; this model captures the situation in which the processors do not necessarily have access to the most recent value of the components of x. To simplify notation, we also write

$$x_i(t) = x_i^i(t), \qquad i \in \{1, \ldots, n\}$$

and we view the vector $x(t)$ defined by

$$x(t) = (x_1(t), \ldots, x_n(t))$$

as the "nominal" iterate of the algorithm at time t. Assume that the iteration is initialized with a vector $x(0)$. The i^{th} processor updates $x_i(t)$ at a set of times T^i, so that

$$x_i(t+1) := \begin{cases} f_i(x^i(t)), & t \in T^i \\ x_i(t), & t \notin T^i \end{cases} \tag{1}$$

We assume that the sets T^i are infinite and that whenever processor i revises x_i, it eventually sends the new value to all of its dependent neighbors.

In general, a convergent asynchronous iterative algorithm does not necessarily produce a fixed point in a finite number of steps. A practical implementation of an asynchronous iterative algorithm must terminate after a finite number of iterations. For this reason, we usually have to be satisfied with finding some $x \in X$ which is in the neighborhood of a fixed point of f. To be more precise, there is a *local termination set* $S_i \subset X$ associated with each processor i, and we would like to obtain a vector x belonging to the *global termination set* $S = \bigcap_{i=1}^n S_i$ after a finite number of iterations at each processor.

For the procedures of section 2 (distributed termination), we will assume the following regarding the asynchronous algorithm:

Assumption 1: In order to determine if a given $x = (x_1, \ldots, x_n) \in S_i$, processor i needs only x_i and the set of x_j for which j is an essential neighbor of i.

Assumption 2: For each execution of the algorithm such that the sets of times T^i are infinite and $\lim_{t\to\infty} \tau_j^i(t) = \infty$ for all i and j, there exists $\bar{t}$ such that $x^i(t) \in S_i$ for all $t \geq \bar{t}$.

Assumption 2 relates to the total asynchronism assumptions of [1, §6.1]. Chapter 6 of this reference describes several algorithms for which Assumption 2 is satisfied.

For the procedures of section 3 (supervised termination), we make Assumption 1 and the following assumption in place of Assumption 2:

Assumption 3: There exists $\bar{t}$ for which $(x_1(t_1), x_2(t_2), \ldots, x_n(t_n)) \in S$ whenever $t_1, \ldots, t_n \geq \bar{t}$.

Assumption 3 is implied from Assumption 2 if there is some $U \subset S$ such that $U = \prod_{i=1}^{n} U_i$, $U_i \subset X_i$ for all i, and $x(t) \in U$ for all sufficiently large t. However, Assumption 3 holds for certain iterative algorithms that do not satisfy Assumption 2 when the convergence of the iteration is sensitive to the choice of the sets T^i and the amount of communication delay between the processors. Assumptions 1-3 are valid for many iterative asynchronous algorithms and corresponding local termination sets S_i (see, for example, [1], [2], [12], and [13]).

To demonstrate that the issue of finite termination is not trivial, we consider a "natural" approach to the problem. We implement the following modification to the asynchronous algorithm. Processor i performs the iteration $x_i = f_i(x^i(t))$ at time $t \in T^i$ if $x^i(t)$ does not belong to the local termination set S_i; if the component x_i obtained is different from the stored component $x_i^i(t)$, then processor i sends x_i to its dependent neighbors and saves x_i as $x_i^i(t+1)$. If a message x_j from processor j arrives at processor i, the message is stored as x_j^i. For convenience, suppose that the messages sent from any processor to any other processor are received in the order in which they are transmitted; this assumption, called FIFO (for first in, first out), can be enforced via an appropriate data link control scheme (see [1, §1.3.2]). Unfortunately, even if we know that the original asynchronous iteration satisfies $x(t) \in S$ for all sufficiently large t, we cannot always conclude that the modified procedure will terminate. This is illustrated by the following example.

Example: Suppose that each x_i is a real number, $n = 2$, and for $i \in \{1, 2\}$, $S_i = \{x \in \Re^2 : \mid f_i(x) - x_i \mid < \epsilon\}$, for some $0 < \epsilon < 1$. Let the iteration f be defined by

$$f_1(x) = \frac{x_1 x_2}{\epsilon}, \qquad f_2(x) = \frac{x_2}{2}$$

The original asynchronous iteration converges to $x^* = (0, 0)$ from any initial vector $x(0)$. Therefore, for all sufficiently large t, $x(t)$ belongs to the global termination set. Let us consider the behavior of the modified algorithm when $x(0) = (2, 1.5\epsilon)$. Because the local termination condition at processor 2 holds at time 0, processor 2 never iterates. Since $x(0) \notin S_1$, processor 1 executes the non-convergent iteration $x_1 := 1.5x_1$ infinitely often; consequently, termination will not occur.

Hence, the local termination rules do not necessarily ensure the global termination of the computation in the absence of further communication between the processors.

We will consider a variety of algorithms that comprehensively address the issue of termination. There are essentially three aspects to the problem of finite termination of a convergent distributed asynchronous iterative algorithm with a vector that is a member of S. These are

1. finite termination of the algorithm,
2. termination detection in finite time, and
3. construction of $x \in S$.

The procedures we will specify can be categorized into two broad approaches called *distributed termination* and *supervised termination.* Protocols with distributed termination decompose the three issues and algorithms using supervised termination address them jointly.

Another important difference between the two approaches relates to whether or not they interfere with the execution of the original asynchronous algorithm. In the distributed approach, the original iteration is modified so that the update $x_i := f_i(x)$ is executed at processor i only when $x^i \notin S_i$. Once there are no more updates to be done, a solution $x \in S$ is constructed by a separate protocol.

By contrast, in the supervised approach, the algorithm is left intact. Instead, a supervisory process is used that takes *snapshots* of the system, collects a *potential solution vector* $\overline{x}$ consisting of a component $\overline{x}_i$ from each processor i, checks whether $\overline{x} \in S$, and if so, terminates the computation. The snapshots and the construction of the potential solution vector $\overline{x}$ are transparent to the original algorithm.

The supervised approach can be implemented in a number of different ways. In Section 3, we describe three possibilities (and several subvariations), which differ in the degree of centralization of the supervisory process. In the most centralized version, the test $\overline{x} \in S$ is conducted at a single special processor who collects the components $\overline{x}_i$ from the other processors using essentially a polling scheme. This special processor is distinct from the n processors executing the iteration. In the other versions, the test $\overline{x} \in S$ is distributed among all the processors, who individually test the corresponding condition $\overline{x} \in S_i$ and "vote" on whether termination is warranted. The results of the vote are collected by a special processor who acts to terminate the computation if all votes are positive and to restart the testing process otherwise. In the second processing scenario, the special processor does not participate in the iteration, while in the last processing situation, the special processor is involved in the iteration. We conclude Section 3 by using the supervised approach to derive protocols for the termination detection of finite, distributed computations. These termination detection schemes rely upon snapshots of the number of messages sent and received at each of the processors.

2. Distributed Termination

Distributed termination was introduced in [11] and also in [2] as a viable approach to the termination problem for a special class of asynchronous iterations. The procedure considered in [2] to modify the asynchronous iterative algorithm so that it terminates in finite time is identical to the "natural" approach specified in the introduction, except that processor i broadcasts x_i to all of the other processors whenever $x_i \neq x_i^i(t)$. For the case considered in [2], x is an element of a Euclidean space and for each i, the local termination sets S_i are

$$S_i = \{x \; : \| \, f_i(x) - x_i \, \| < \; \epsilon\}.$$

For any subset I of the set of processors and any vector $\vec{\theta} \in \Theta^I = \{\{\theta_i\}_{i \in I} \; : \; \theta_i \in X_i \text{ for all } i \in I\}$, let $f^{I,\vec{\theta}}(x)$ be defined by

$$f^{I,\vec{\theta}}(x) = \begin{cases} \theta_i, & i \in I \\ f_i(x), & i \notin I \end{cases} .$$

It is established in [2] that for the special case of the problem considered there, if the original iteration $x := f(x)$ satisfies an "independent convergence property" in which any asynchronous version of the iteration $x = f^{I,\vec{\theta}}(x)$ converges for any choice of I and $\vec{\theta} \in \Theta^I$, then the modified procedure specified above is guaranteed to terminate in finite time. Some families of functions that satisfy the independent convergence property, such as the class of weighted supremum norm contractions, are identified in [2]. In the approach of [2], termination is detected by using a standard protocol. However, to guarantee that the ultimately obtained vector x belongs to the global termination set S, the FIFO assumption on the link transmissions is needed.

Under Assumptions 1 and 2, we will propose a procedure to modify the underlying algorithm so that it terminates in finite time and is compatible with the set of known termination detection protocols (see, for example, [1] and [3-10].) To follow the terminology frequently used in the literature on termination detection protocols, a processor is said to be *idle* if it satisfies its local termination condition; otherwise it is called *active*. We presuppose that an idle processor cannot transmit messages and that it remains idle until a message from another processor is received. The algorithm has terminated by time t if all processors are idle at time t and there is no message in transit along any communication link at time t. The modifications to the algorithm specified below are similar to the ones considered in the introduction; to avoid the situations in which the earlier modifications result in no termination, we assume that in addition to sending messages containing the current value of x_i, processor i is able to send a different kind of packet called a *request message* and store an *on-off request variable*, R^i, which is initially off. The asynchronous algorithm is modified in the following way. When processor i updates x_i via the iteration $x_i = f_i(x^i)$, i sends the new value of x_i to its dependent neighbors if $x_i \neq x_i^i$, sets x_i^i to the updated value of x_i, and sets R^i to off; if the current x^i does not belong to S_i, processor i sends a request message to all processors $j \neq i$. If processor i receives a request message, it sets R^i to on, and if it receives x_j from processor j, it stores the updated value as x_j^i. Processor i iterates at time $t \in T^i$ if $x^i(t)$ is not a member of S_i or if R^i is on at time t. Processor i is idle when its current x^i belongs to S_i and its current R^i is off; otherwise it is active. Hence, as desired, idle processors do not iterate. If processor i is idle and receives x_j from processor j, it updates x_j^i and remains idle if and only if the new x^i belongs to S_i. Termination has occurred when all processors are idle and there are no messages or request messages in transit.

We have the following result.

Proposition 1: Under the preceding assumptions, the modified algorithm terminates. Furthermore, if the communication links that carry x_i are FIFO and t is the time at which termination occurs, then $x(t) \in S$.

Proof: To arrive at a contradiction, suppose that the modified algorithm does not terminate.

Then there is a non-empty subset I of processors that iterate infinitely often. Hence, there is some processor $j \in I$ for which x^j is not a member of S_j infinitely often; if this were not the case, there would be no request messages sent after some time and so all processors eventually become idle and the modified algorithm terminates. Since x^j does not belong to S_j infinitely often, processor j sends a request message to every other processor infinitely often. Consequently, all of the processors iterate infinitely often. By our assumptions on the original asynchronous algorithm, this implies that $x^i(t) \in S_i$ for all i and all t sufficiently large. This contradicts our earlier conclusion that x^j does not belong to S_j infinitely often. Hence, the modified algorithm terminates at some time t^*. If the communication channels that carry x_i are FIFO, then at time t^*, the value of x_i stored by all of the dependent neighbors of processor i will be equal to $x_i^i(t^*)$. Since the definition of termination indicates that $x^i(t^*) \in S_i$ for all i, the modified algorithm terminates with $x(t^*) \in S$. $\square$

It seems plausible that the modified algorithm will terminate under the weaker condition that iterating processors send request messages only to their essential neighbors. The following example demonstrates that this hypothesis can be false.

Example: Suppose that each x_i is a scalar and $n = 3$. Let the iteration f be defined by

$$f_1(x) = x_1 x_2, \qquad f_2(x) = x_2 x_3, \qquad f_3(x) = \left(1 - \frac{\epsilon}{2}\right) x_3,$$

where $0 < \epsilon < 1$. This asynchronous algorithm will converge to $x^* = (0,0,0)$ from any initial vector $x(0)$. For $i \in \{1,2,3\}$, let $S_i = \{x \in \Re^3 \ : \ | f_i(x) - x_i | \ < \ \epsilon\}$. Let us consider the behavior of the modified algorithm when $x(0) = (1,2,1)$ and iterating processors send request messages only to their essential neighbors. Processor 3 is initially idle and it will not iterate unless it receives a request message from processor 2. Processor 2 also initially satisfies it local termination condition; it will iterate after it receives a request message from processor 1. Note that if processor 2 receives a request message while $x_3^2 = 1$, x_2 will not change upon the resulting iteration and hence will not send a request message to processor 3. As a consequence, processor 3 always remains idle, and processor 2 iterates from time to time, but it never changes x_2. What happens to processor 1? Since $x_1(0) = 1$, processor 1 is initially active, and it executes the nonconvergent iteration

$x_1 := 2x_1$ infinitely often. Hence, in order for an implementation of this algorithm to terminate, it is necessary for processor 1 to send processor 3 a request message.

The approach of this section requires a broadcast mechanism for the request messages. In [14], various broadcasting protocols such as flooding and the shortest path topology algorithm are studied in detail. Unfortunately, each processor may initiate a large number of request messages and therefore the broadcasts cause a potentially excessive communication overhead for the computing system. We propose the following solution to this practical difficulty. The implementation is similar to the one discussed so far. Now all requests are numbered. Each processor i stores the highest *request number* n^i it has received or generated, where the initial n^i are arbitrary integers. If processor i iterates and finds that the updated vector x^i is not a member of S_i, it increments n^i by one, numbers the request with the new n^i, and sends the request to its neighbors. If processor j receives a request numbered n, it discards the request if $n \leq n^j$; otherwise, it sets $n^j := n$, sets R^j to on and relays the request (with the number n unchanged) to its neighbors. The results of the proposition will still hold for the new implementation. This is because the iteration of processor i infinitely many times implies that n^i and the request numbers of the neighbors of processor i will increase to infinity. In fact, each processor must iterate infinitely often since any processor can be reached from processor i through a sequence of neighbors; this is enough to establish that the proposition remains valid.

Finally, we address the issue of constructing $x \in S$. We note that if the communication links do not necessarily satisfy the FIFO assumption, then for each i, there is no guarantee that the values of x_i stored by processor i and its dependent neighbors will be consistent upon termination; as a consequence, it is possible that the algorithm will terminate with a vector x which is not a member of the global termination set. Hence, we assume that the communication links are FIFO. When termination is detected using, for example, one of the standard termination detection protocols, we require a special processor called the coordinator to send a *termination message* to each processor via a pre-determined spanning tree. Upon receiving a termination message, each processor sends its parent a packet containing its value and its identity number. These packets are propagated through the tree to the coordinator, which can use the spanning tree to find out how many packets to expect.

Since all the communication links constituting the spanning tree are FIFO, we can use a different procedure in which the coordinator does not need to know the number of processors. In this procedure, each processor sends its parent all the packets it has received and then sends its own packet as soon as it has sent the packets from its children in the tree. The remaining practical issues are the selection of a coordinator and the construction of a spanning tree of the processors, if they are not already provided. Choosing a coordinator is equivalent to the problem of identifying the leader in an asynchronous network, which is discussed in [15]. The algorithm in [1, §8.1, Example 1.3] can be easily generalized to find a procedure to produce a spanning tree when the coordinator is known.

3. Supervised Termination

The approach of the last section has some important shortcomings. In general, we need the FIFO assumption to guarantee that we will terminate with some vector x belonging to the global termination set S; in many applications, this assumption is inconvenient. Recall that in the modified version of the algorithm, processor i iterates at only those times $t \in T^i$ for which either $x^i(t)$ does not satisfy processor i's local termination condition or processor i's on-off request variable is on. Hence, because there are fewer iterations, the modified version of the asynchronous iterative algorithm seems likely to take longer than the original version to enter S; moreover, we can not produce a vector x belonging to S until *after* the modified algorithm terminates and termination is detected. Perhaps the most serious flaw in the approach is the need for Assumption 2, which presupposes that *any* asynchronous execution of the algorithm is guaranteed to enter the global termination set in finite time. There are convergent asynchronous iterative algorithms called *partially asynchronous iterative algorithms* (see [2]) which are known to converge under certain bounds on the amount of asynchronism in the computing system. It is conceivable that for one of these algorithms, the original version of the algorithm enters the global termination set while the modified version does not because processors iterate less often in the modified version, and consequently send their values to their dependent neighbors less frequently. By replacing Assumption 2 with Assumption 3, we avoid this problem. Furthermore, the approach we will consider in this section does not suffer from any of the drawbacks listed above.

The idea in the new approach is that processors execute the (original) algorithm until they receive notification to terminate. From time to time, a *snapshot* will be taken of the system in the form of a *potential solution vector*, and the resulting information will be processed by a *supervisor* or a *coordinator*, which can subsequently determine if the system is ready to terminate the computation. Unlike the well-known snapshot algorithm of [8] (see, for example, [1]), we do not impose a FIFO assumption on communication between any two processors. Instead, we require Assumption 3.

We will consider algorithms under the following three processing situations:

1. There is a supervisor which communicates directly with all of the processors and is able to determine if a given vector $x \in X$ is in the global termination set.

2. There is a coordinator which communicates directly with all of the processors, but this coordinator is unable to determine on its own whether some vector x is a member of S.

3. One of the processors involved in the iteration assumes the role of coordinator, but the processor is not aware of the entire network topology.

Obviously, for a system with a large number of processors, these assumptions vary from least viable to most practicable. We discuss protocols for all three scenarios because the guidelines we develop for the second and third processing situations build on the ideas in the procedures for the previous one. In every case, we assume that each processor involved in the computation carries out its computations and its communications with its neighboring processors independently of the termination protocol, until it receives an order to terminate. In particular, none of the protocols we will subsequently discuss affect the execution of the underlying algorithm until the processors obtain a termination directive.

We will conclude this section by converting some of the supervised termination procedures into termination detection protocols for arbitrary finite, distributed computations.

3.1. First Processing Situation

We consider four protocols for the first processing situation. We first give a broad overview of the way all of the procedures work, and then we describe the remaining operations specific to each algorithm. In every procedure, each processor i participating in the iteration uses local information to ascertain if the system is involved in a snapshot and also to determine the single instance during a snapshot at which processor i sends its current value of x_i, denoted $\overline{x}_i$, to the supervisor. The supervisor collects these *potential solution values* and stores them in a *potential solution vector* $\overline{x} = (\overline{x}_1, \ldots, \overline{x}_n)$. When the vector is complete, the supervisor verifies if $\overline{x}$ is a member of the global termination set. If this is the case, the supervisor sends a *termination order* to each processor and finishes with $\overline{x}$ as the solution to the problem. Otherwise, the supervisor discards the potential solution vector and performs some other tasks to prepare the system for the next snapshot. Note that the assumption of the existence of a $\overline{t}$ for which $(x_1(t_1), x_2(t_2), \ldots, x_n(t_n)) \in S$ whenever $t_1, \ldots, t_n \geq \overline{t}$ implies that each protocol will eventually terminate.

The procedures are presented in the order of increasing complexity. The idea is that as the protocols become more intricate, the supervisor is expected to test fewer potential solution vectors before terminating with a solution. This suggests that the more complicated protocols have a larger termination delay; however, in the case of a large system, the validation of a potential solution vector may require considerable computational resources and the penalty of an increase in termination delay may be more than offset by the need for fewer potential solution vector verifications.

For the first protocol, each processor i maintains a binary flag b_i, initially set to zero, which indicates if the processor may initiate a snapshot. Whenever $b_i = 0$ and processor i satisfies its local termination condition, i changes b_i to one. Furthermore, processor i immediately sends the supervisor its current value $\overline{x}_i = x_i^i$. We assume that if the supervisor processes a potential solution vector and decides that termination is inappropriate, it subsequently sends each processor a special message called a *repeat signal*. Upon receiving a repeat signal, processor i resets b_i to zero.

The second protocol is very similar to the first one. The only difference is that the second protocol introduces a delay between the time b_i is changed from zero to one and the time that processor i sends the supervisor its value $\overline{x}_i = x_i^i$. We now assume that each processor i sends the supervisor an *initiation message* upon changing b_i from zero to one. Furthermore, the supervisor maintains an *initiation buffer*, initially empty, that contains at most n initiation messages. When the supervisor has received an initiation message from each processor, i.e., its initiation buffer is full, it empties the buffer and sends a special message called a *query message* to each processor. Upon receiving a query message, processor i sends the supervisor a message containing $\overline{x}_i = x_i^i$ as soon as $x^i \in S_i$ and discards the query message. We suspect that the second protocol often requires fewer potential solution vector validations than the first protocol because the transmission of initiation and query messages introduces a delay that acts like a synchronizer, so that the vector $\overline{x}$ seen by the supervisor is closer to being consistent with the vectors x^i stored by the processors during some interval of time.

The last two procedures are very similar and employ slightly different features from the second protocol. The third and fourth protocols are differentiated from the second by the rules specifying when processor i should change b_i from one to zero and when the supervisor should send each processor a query message. The repeat signals employed by the first two procedures are not needed for the third and fourth protocols. Instead, in addition to sending messages containing its value and initiation messages, each processor can also send the supervisor special *cancellation messages* notifying that its termination condition is no longer satisfied. More specifically, whenever processor i has $b_i = 1$ and $x^i \notin S_i$, it sends a cancellation message to the supervisor and sets $b_i = 0$. Rather than keeping an initiation buffer, the supervisor maintains a binary flag W, which indicates if it is waiting for the reply to a query, and for each i, it keeps a counter c_i, which records the difference between the number of initiation and cancellation messages it has received from processor i. Initially, W is set to zero, and $b_i = c_i = 0$ for all i. When the supervisor receives an initiation message from processor i, it increments c_i by one. When a cancellation message from processor i arrives at the supervisor, the supervisor decrements c_i by one. Note that c_i is not necessarily a binary variable unless the links that carry initiation and cancellation messages are FIFO; however, the assumption that there is some $\overline{t}$ for which $x^i(t) \in S_i$ for all $t \geq \overline{t}$ and all i implies that there is some t^* such that for all i, $c_i = 1$ at all times greater than t^*. For the third (fourth) protocol, whenever $W = 0$ and $c_i \geq 1$ ($c_i = 1$) for all i, the supervisor sends a query to each processor and sets $W = 1$. The supervisor changes W back to zero whenever it processes a potential solution vector and decides not to terminate. Intuitively, the third and fourth procedures require fewer potential solution vector verifications than the second because query messages are sent when it's more likely that all of the processors simultaneously satisfy their local termination condition. The fourth procedure results in a larger termination delay than the third procedure, but will probably involve few potential solution tests before time t^*.

3.2. Second Processing Situation

The second processing situation differs from the first one in a few ways. The supervisor of the first scenario is replaced by a coordinator which can communicate directly with each processor, but is unable to determine on its own if a given vector $x \in X$ satisfies the global termination condition. Therefore, to compensate for this impediment, we presuppose that the processors in the second processing situation are more powerful than their counterparts in the first scenario. In addition to the computation, storage, and interprocessor communication capability needed to execute the underlying algorithm, the processors also have the means to take and test a *snapshot* of the system. Each processor i has the responsibility to determine if the *potential solution vector* associated with a snapshot satisfies i's local termination condition and provide the coordinator with this information. The role of the coordinator is to collect the results of a snapshot from the processors, determine and inform the processors if termination is appropriate or inappropriate, and store the solution vector. Except for the first protocol we will consider, the coordinator also has the function of gathering additional information from the processors for the purpose of requesting snapshots. In the first protocol, a processor initiates a snapshot solely on the basis of local information.

Each of the protocols discussed for the first processing situation can be modified to work for the second computing scenario. The adaptations of the second, third and fourth protocols of the last subsection maintain the flags b_i introduced earlier; the modification of the first protocol does not. For the procedures which store b_i, processor i continues to use the sames rules and mechanisms as before to update b_i and to notify the coordinator about the changes in the flag. The coordinator uses the same expedients introduced in the last subsection to decide when to send each processor a *query message*. In the first processing situation, a query message to a processor was a request for the processor to transmit its value as soon as it satisfies its local termination condition; now, a query message to a processor is a request for the processor to initiate a snapshot if it is not currently engaged in one.

Furthermore, for each protocol, we assume that each processor i can store a binary flag q_i, which is zero when processor i can initiate a snapshot of the system, and a potential solution vector $\overline{x}^i$ consisting of its own potential solution value $\overline{x}_i$ and the potential solution values of its essential neighbors; there is no need for the coordinator to store a potential solution vector before the algorithm has terminated. Initially, for each processor i, $q_i = 0$, $\overline{x}_i = x_i(0)$, and the rest of the potential solution vector is empty. For the first protocol, processor i initiates a snapshot when $q_i = 0$ and $x^i \in S_i$. For the other three protocols, processor i initiates a snapshot whenever it receives a query message from the coordinator while $q_i = 0$. For all of the procedures, the execution of a snapshot is identical, except for some minor variations. If processor i initiates a snapshot, it sets $q_i = 1$, saves $\overline{x}_i = x_i^i$ as its own potential solution value as soon as $x^i \in S_i$ and sends a special message called a *test message* containing the new value of $\overline{x}_i$ to each of its dependent neighbors. If processor i receives a test message containing $\overline{x}_j$ while $q_i = 0$, it sets $q_i = 1$, discards the current value of $\overline{x}_i$, stores $\overline{x}_j$ in its potential solution vector, saves $\overline{x}_i = x_i^i$ as soon as $x^i \in S_i$ and sends a test message containing the new value of $\overline{x}_i$ to each of its dependent neighbors. If processor i receives a test message containing $\overline{x}_j$ while $q_i = 1$, it stores $\overline{x}_j$ in its potential solution vector. For the last three protocols, if processor i is not an initiator of the current snapshot, a query message subsequently reaches the processor and is ignored if $q_i = 1$ upon arrival. When $\overline{x}^i$ is complete, processor i checks if $\overline{x}^i$ satisfies the local termination condition, empties the potential solution vector except for $\overline{x}_i$, and sends the coordinator a special message called a *token*; if the local termination condition is satisfied, processor i transmits a *white token* to the coordinator and otherwise sends a *black token.* The coordinator maintains a *token buffer* containing at most n tokens; initially, the token buffer is empty. When the coordinator's token buffer is full, the coordinator checks if all the tokens are white. If they are, the coordinator sends a termination order to each processor and awaits the solution value from each processor; otherwise, it empties the buffer, transmits a repeat signal to each processor, and for the third and fourth protocols, sets W back to zero. When a processor obtains a termination order, it terminates with its potential solution value as its final value and sends a copy of $\overline{x}_i$ to the coordinator. When processor i receives a repeat signal, it sets q_i to zero and, as we indicated earlier, for the second protocol it also sets b_i to zero. We note that the value of $\overline{x}_i$ stored in the potential solution vectors of processor i and its essential neighbors are consistent. Furthermore, we recall that by Assumption 3, there is some $\overline{t}$ for which $(x_1(t_1), \ldots, x_n(t_n)) \in S$ for all $t_1, \ldots, t_n \geq \overline{t}$. Hence, the protocol terminates in finite time with a vector $\overline{x}$ belonging to the global termination set.

We suspect that again there is an increase of synchronism going from the first protocol to the fourth one because the delays introduced by the query generation in the more complex procedures are likely to result in a decrease in the difference between the time processor i sets q_i to one and the time i sends test messages to its dependent neighbors. Consequently, as the procedures become increasingly complicated, they probably have fewer snapshots and a larger termination delay.

3.3. Third Processing Situation

For the third processing situation, one of the processors participating in the iteration, say processor j, undertakes the role of coordinator. We presuppose that information passes between the coordinator and every other processor by means of a spanning tree rooted at processor j; a synopsis of our approach to selecting a coordinator and constructing a spanning tree is provided at the end of Section 2. Furthermore, the processors are more powerful than their analogues in the second processing situation. In particular, they are now required to store and send their parents or children in the spanning tree the type of packets that their counterparts in the second processing scenario transmit to or receive from the coordinator, respectively. For the protocols we consider, we assume that each non-leaf processor maintains a token buffer, which is initially empty, and whose size is given by the number of children it has in the spanning tree.

We adapt the first two protocols specified for the second processing situation to the third computing scenario. For both protocols, we assume that each processor i maintains the flag q_i and the potential solution vector $\overline{x}^i$ and uses the rules itemized in the last subsection to initialize these expedients, change q_i from zero to one, and update $\overline{x}^i$. The snapshots differ in two ways from their parallels for the second processing situation. First, only the coordinator is able to initiate a snapshot. The other change is an adjustment in the way the coordinator obtains the results of a snapshot from and sends messages to the processors. Upon storing a complete potential solution vector, each processor checks if that vector satisfies the local termination condition; if it does, the processor becomes *white* and otherwise it turns *black*. Each *leaf* processor that has tested its potential solution vector sends its parent a token of its color. Except for the coordinator, each *non-leaf* processor that has tested its potential solution vector and has a full token buffer sends its parent a white token if it is white and the token buffer contains no black tokens, and it sends a black token otherwise. Upon sending a token, processor i empties its potential solution vector except for $\overline{x}_i$, discards the tokens in its token buffer, and loses its color. Since the coordinator is the only processor to ever initiate a snapshot, processor $i \neq j$ can view the snapshot to be over, and hence, reset q_i to zero, upon sending its parent a token. When the *coordinator* has tested its potential solution vector and has a complete token buffer, it broadcasts a termination order if it is white and possesses no black tokens; otherwise, it empties its potential solution vector. As before, we assume that there will eventually be a snapshot of the system belonging to the global termination set, and hence, both of the protocols terminate in finite time. Once the processors receive the termination directive, the procedure specified at the end of the section on distributed termination is used to construct the solution vector. We give the remaining details specific to each protocol below.

For the first procedure, the coordinator employs only local information in deciding whether or not to initiate a snapshot; a snapshot is initiated when $q_j = 0$ and $x^j \in S_j$. The coordinator changes q_j back to zero every time it processes a snapshot and determines that termination is inappropriate.

For the second protocol, aside from keeping the flag q_i and a token buffer, processor i also maintains a binary flag b_i and an initiation buffer, initially empty, which is the same size as its token buffer. If $i \neq j$, b_i is zero when processor i is able to send its parent an initiation message; b_j is set to zero when there is no snapshot in progress. When $x^i \in S_i$, a *leaf* processor i sends its parent an initiation message and sets $b_i = 1$. When $x^i \in S_i$ and its initiation buffer is full, a *non-leaf* processor $i \neq j$ sends its parent an initiation message, sets $b_i = 1$, and empties the initiation buffer. When $x^j \in S_j$ and its initiation buffer is full, the *coordinator* sets $q_j = b_j = 1$, initiates a snapshot, and empties its initiation buffer. When the *coordinator* has tested its potential solution vector and has a complete token buffer, it broadcasts a termination order if it is white and holds no black tokens; otherwise, it sets $b_j = 0$, empties its potential solution vector, and sends its children a repeat signal. Upon receiving a repeat signal, processor i sets $b_i = 0$ and propagates the signal to its children, if it has any. Using the same reasoning we applied to the first two computing scenarios, we again surmise that the additional complexity of the second protocol relative to the first results in a decrease in the number of snapshots and an increase in the termination delay.

3.4. Application to a Termination Detection Scheme

For a finite, distributed computation, each processor is in either an *active* state or an *idle* state at any time. An active processor can send *primary messages*, i.e., messages pertaining to the underlying computation, to its dependent neighbors, and it may become idle at any time. An idle processor cannot send primary messages and it may remain idle or turn active upon receiving a primary message. The computation has terminated if all processors are idle and there are no primary messages in transit. We assume that the computation eventually terminates and we consider the problem of detecting termination. This is a well-known problem, which has been studied in many sources, e.g., [1] and [3-10].

A simple way to gather information about the number of undelivered primary messages in the system is to have each of the processors from time to time send the supervisor a *secondary message,* i.e., a report on the number of primary messages it has recently sent and received. The supervisor can then detect termination when the counts of the received and sent messages are equal, and in addition, all processors are idle. Termination detection protocols based on message counting were first considered in [9] and [10]. We present and establish the validity of a class of termination detection schemes that use *snapshots* to account for the primary messages that have passed through the system. A snapshot is a time interval during which every processor participating in the computation sends the supervisor exactly one secondary message and which ends when the supervisor has received and processed each of these secondary messages. Our schemes and presentation are similar to those in [9, pp. 95-97]. In particular, the supervisor in our protocols plays the same role as the central process in [9]. The procedures in §3.1 - §3.3 provide specific (and new) implementations of these termination detection schemes.

In every scheme, each processor i participating in the computation updates $\mathcal{S}_i$, the number of primary messages it has sent, and $\mathcal{R}_i$, the number of primary messages it has received, between consecutive snapshots. Initially, $\mathcal{S}_i = \mathcal{R}_i = 0$, and the appropriate variable is incremented when a primary message is sent or received. Each processor uses local information to find out if the system is involved in a snapshot and to select the single moment during a snapshot at which it sends the supervisor a secondary message consisting of its current counts and resets

$\mathcal{S}_i$ and $\mathcal{R}_i$ to zero. For example, since all processors must be idle in order for termination to occur, we require that a processor be idle at the instant it sends the supervisor a secondary message. The supervisor maintains $\mathcal{S}$ and $\mathcal{R}$, which estimate the cumulative number of primary messages that have been sent and received in the system, respectively. Initially, $\mathcal{S} = \mathcal{R} = 0$, and when a set of message counts from processor i arrives at the supervisor, $\mathcal{S}$ and $\mathcal{R}$ are increased by $\mathcal{S}_i$ and $\mathcal{R}_i$, respectively. At the end of a snapshot, the supervisor decides that the computation has terminated if $\mathcal{S}$ is equal to $\mathcal{R}$, and if $\mathcal{R}_i$ is equal to zero for all processors i. Otherwise, the supervisor performs some other tasks to prepare the system for the next snapshot, which will take place within a finite time. We will establish that for this class of schemes, the supervisor will detect termination within a finite time after it occurs and that it will never incorrectly conclude that the computation has terminated. Then we will provide the final details characterizing the individual protocols.

We have the following result.

Proposition 2: If the underlying computation terminates at time t^*, then within a finite time after t^*, the supervisor's record of the cumulative number of primary messages sent will be equal to its record of the cumulative number of primary messages received, and there will subsequently be a snapshot during which all processors will report that they have not received any new primary messages.

Proof: For all i, as processor i sends or receives a primary message, it simultaneously tallies the message in $\mathcal{S}_i$ or $\mathcal{R}_i$, respectively, and transmits these message counts at the next instant it sends the supervisor a secondary message. The time between the transmission and reception of a secondary message, and the time between the transmission of successive secondary messages from the same processor are assumed to be finite. By the end of the second snapshot to be completed after t^*, the supervisor knows of every primary message that was sent and subsequently received during the computation, and hence, $\mathcal{S}$ and $\mathcal{R}$ must be equal. Furthermore, since the computation has terminated by the end of the first snapshot that finishes after t^*, in the following snapshot, each processor will be idle and report that it has received no new messages. $\square$

We next establish that the supervisor never incorrectly decides that the computation has terminated.

Proposition 3: If at the end of a snapshot, the supervisor finds that $\mathcal{R}_i = 0$ for all processors i and that $\mathcal{S} = \mathcal{R}$, then it can conclude that the underlying computation has terminated.

Proof: For this snapshot, let t_i denote the instant at which processor i sends the supervisor a secondary message. We will show that for all j, processor j will never receive a primary message after t_j. Since processor j is idle at time t_j, this will prove the proposition. A primary message is called a *bad message* if it arrives at its destination, say processor j, after time t_j. We will demonstrate that there cannot be any bad messages in the system. To arrive at a contradiction, suppose there are bad messages in the system and let m be the bad message with the earliest time of reception, say t_r. Assume that m was sent at time t_s by processor i to processor j. There are two cases to consider.

1. Suppose $t_s > t_i$. Then processor i must have received a bad message prior to $t_s < t_r$. This contradicts the assumption that m is the bad message with the earliest time of reception.

2. Suppose $t_s < t_i$. Let $\mathcal{T}_e$ represent the earliest of the times t_k. Then either m is in transit at time $\mathcal{T}_e$, or $t_s \in [\mathcal{T}_e, t_i]$. We will show that either scenario is impossible.

   First, we need to introduce some additional notation. For any time t, let $\mathcal{S}^k(t)$ and $\mathcal{R}^k(t)$ represent the cumulative number of primary messages that have been sent and received, respectively, by processor k by time t, let $\Delta^{(k,l)}(t)$ symbolize the number of primary messages from processor k to processor l that are in transit at time t. Clearly,

   $$\sum_k \mathcal{S}^k(t) \;=\; \sum_l \mathcal{R}^l(t) \;+\; \sum_{(k,l)} \Delta^{(k,l)}(t). \tag{2}$$

   Since $\mathcal{R}_k = 0$, processor k did not receive any primary messages in the interval $[\mathcal{T}_e, t_k]$. Hence,

   $$\mathcal{R} \;=\; \sum_l \mathcal{R}^l(t_l) \;=\; \sum_l \mathcal{R}^l(\mathcal{T}_e). \tag{3}$$

   We also have that

   $$\begin{aligned} \mathcal{S} &= \sum_k \mathcal{S}^k(t_k) \\ &\geq \sum_k \mathcal{S}^k(\mathcal{T}_e), \quad \text{by monotonicity of } \mathcal{S}^k(\cdot) \end{aligned}$$

$$= \sum_{l} \mathcal{R}^l(\mathcal{T}_e) + \sum_{(k,l)} \Delta^{(k,l)}(\mathcal{T}_e), \quad \text{by (2)}. \tag{4}$$

Since $\mathcal{S} = \mathcal{R}$, it follows from (3) and (4) that there are no primary messages in transit at time $\mathcal{T}_e$ and that for all k, processor k did not send any messages in the interval $[\mathcal{T}_e, t_k]$.$\Box$

To conclude this section, we will explain how to convert the protocols in §3.1 to termination detection protocols. Aside from the alterations in the snapshot described above, there are two additional modifications needed. The first is that we need to replace the condition "processor i satisfies (doesn't satisfy) its local termination condition" with "processor i is idle (active)." The other change is that instead of sending the supervisor a message containing $\overline{x}_i$, processor i transmits a secondary message consisting of its current message counts. As in §3.1, the more complicated protocols are likely to generate larger termination delays in exchange for fewer snapshots. The protocols in §3.3 can also be recast into this framework. We omit the details.

4. Conclusions

The problem of obtaining iteratively a vector satisfying a global termination condition using local criteria at the processors of an asynchronous distributed system is surprisingly delicate. We gave two general approaches and several algorithms that use additional interprocessor communications to ensure that a vector with the desired property is constructed at one of the processors. An analytical comparison of these algorithms in terms of communication efficiency and termination delay appears to be difficult. For many common architectures such as star networks and broadcast rings, the implementation of these procedures is very simple. In the distributed approach, the termination protocol affects the communication delays of the iterative algorithm, possibly also affecting its convergence properties. Generally, for the supervised approach, it would seem that the communication requirements increase as the termination protocol becomes more distributed. However, the computation requirements, including memory, to check the various termination conditions increase as the termination protocol becomes more centralized. The supervised approach can also be used to create protocols for the termination detection of arbitrary finite, distributed computations.

References

[1] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*, Prentice-Hall, New Jersey 1989.

[2] D. P. Bertsekas and J. N. Tsitsiklis, "Some aspects of parallel and distributed algorithms — a survey," *Automatica* Vol. 27, No. 1, 3-21, 1991.

[3] E. W. Dijkstra and C. S. Scholten, "Termination detection for diffusing computations," *Inform. Process. Lett.* Vol. 11, 1-4, 1980.

[4] N. Francez, "Distributed termination," *ACM Trans. Programming Languages Syst.* Vol. 2, No. 1, 42-55, Jan. 1980.

[5] N. Francez and M. Rodeh, "Achieving distributed termination without freezing," *IEEE Trans. Software Eng.* Vol. SE-8, 287-292, May 1982.

[6] E. W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren, "Derivation of a termination algorithm for distributed computations," *Inform. Process. Lett.* Vol. 16, 217-219, 1983.

[7] R. W. Topor, "Termination detection for distributed computations," *Inform. Process. Lett.* Vol. 18, 33-36, 1984.

[8] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Trans. Comput. Syst.* Vol. 3, 63-75, 1985.

[9] D. Kumar, "A class of termination detection algorithms for distributed computations," *Proc. Fifth Conf. Foundations Software Technol. & Theoret. Comput. Sci.,* New Delhi, India, Dec. 16-18, 1985; *Lecture Notes in Computer Science, 206* Berlin, Germany: Springer-Verlag, 73-100.

[10] D. Kumar, "Development of a class of distributed termination detection algorithms," *IEEE Trans. Knowledge and Data Eng.* Vol. 4, No. 2, Apr. 1992.

[11] D. P. Bertsekas and J. N. Tsitsiklis, "Convergence rate and termination of asynchronous iterative algorithms," *Proc. 1989 Intern. Conference on Supercomputing,* 461-470, Irakleion, Crete, 1989.

[12] D. Chazan and W. L. Miranker, "Chaotic relaxation," *Lin. Algebra and Appl.* Vol. 2, 199-222, 1969.

[13] G. M. Baudet, "Asynchronous iterative methods for multiprocessors," *J. ACM* Vol. 15, 226-244, 1978.

[14] D. P. Bertsekas and R. G. Gallager, *Data Networks*, Prentice-Hall, New Jersey 1987.

[15] R. G. Gallager, P. A. Humblet, and P. M. Spira, "A distributed algorithm for minimum-weight spanning trees," *ACM Trans. Programming Languages Syst.* Vol. 5, No. 1, 66-77, Jan. 1983.