Automatica, Vol. 27, No. 1, pp. 3–21, 1991
Printed in Great Britain.

0005–1098/91 $3.00 + 0.00
Pergamon Press plc

Survey Paper

Some Aspects of Parallel and Distributed Iterative Algorithms—A Survey*†

DIMITRI P. BERTSEKAS‡§ and JOHN N. TSITSIKLIS‡

Iterative methods suitable for use in parallel and distributed computing systems are surveyed. Both synchronous and asynchronous implementations are discussed. A number of theoretical issues regarding the validity of asynchronous algorithms are addressed.

Key Words—Computational methods; distributed data processing; iterative methods; parallel processing; asynchronous algorithms; parallel algorithms; distributed algorithms.

Abstract—We consider iterative algorithms of the form $x := f(x)$, executed by a parallel or distributed computing system. We first consider synchronous executions of such iterations and study their communication requirements, as well as issues related to processor synchronization. We also discuss the parallelization of iterations of the Gauss–Seidel type. We then consider asynchronous implementations whereby each processor iterates on a different component of x, at its own pace, using the most recently received (but possibly outdated) information on the remaining components of x. While certain algorithms may fail to converge when implemented asynchronously, a large number of positive convergence results is available. We classify asynchronous algorithms into three main categories, depending on the amount of asynchronism they can tolerate, and survey the corresponding convergence results. We also discuss issues related to their termination.

1. INTRODUCTION

PARALLEL AND DISTRIBUTED computing systems have received broad attention motivated by several different types of applications. Roughly speaking, *parallel* computing systems consist of several tightly coupled processors that are located within a small distance of each other. Their main purpose is to execute jointly a computational task and they have been designed with such a purpose in mind: communication between processors is fast and reliable. *Distributed* computing systems are somewhat different in a number of respects. Processors are loosely coupled with little, if any, central coordination and control, and interprocessor communication is more problematic. Communication delays can be unpredictable, and the communication links themselves can be unreliable. Finally, while the architecture of a parallel system is usually chosen with a particular set of computational tasks in mind, the structure of distributed systems is often dictated by exogenous considerations. Nevertheless, there are several algorithmic issues that arise in both parallel and distributed systems and that can be addressed jointly. To avoid repetition, we will mostly employ in the sequel the term "distributed", but it should be kept in mind that most of the discussion applies to parallel systems was well.

There are at least two contexts where distributed computation has played a signficant role. The first is the context of information acquisition, information extraction, and control, within spatially distributed systems. An example is a sensor network in which a set of geographically distributed sensors obtain information on the state of the environment and process it cooperatively. Another example is provided by data communication networks in which certain functions of the network (such as correct and timely routing of messages) have to be controlled in a distributed manner, through the cooperation of the computers residing at the nodes of the network. Other applications are possible in the quasistatic decentralized control of large scale systems whereby certain parameters (e.g. operating points for each subsystem) are to be optimized locally, while taking into account interactions with neighboring subsystems. The second important context for parallel or distributed computation is the solution of very large computational problems in which no single processor has sufficient computational power to tackle the problem on its own.

The ideas of this paper are relevant to both contexts, but our presentation will emphasize large scale numerical computation issues and iterative methods in particular. Accordingly, we shall consider

* Received 10 November 1988; revised 12 February 1990; received in final form 11 March 1990. The original version of this paper was presented at the IFAC/IMACS Symposium on Distributed Intelligence Systems which was held in Varna, Bulgaria during June 1988. The published Proceedings of this IFAC Meeting may be ordered from: Pergamon Press plc, Headington Hill Hall, Oxford OX3 0BW, U.K. This paper was recommended for publication in revised form by Editor K. J. Åström.

† Research supported by the NSF under Grants ECS-8519058 and ECS-8552419, with matching funds from Bellcore, Du Pont and IBM, and by the ARO under Grant DAAL03-86-K-0171.

‡ Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, MA 02139, U.S.A.

§ Author to whom all correspondence should be addressed.

algorithms of the form $x := f(x)$ where $x = (x_1, \ldots, x_n)$ is a vector in $\mathscr{R}^n$ and $f : \mathscr{R}^n \mapsto \mathscr{R}^n$ is an iteration mapping defining the algorithm. In many interesting applications, it is natural to consider distributed executions of this iteration whereby the ith processor updates x_i according to the formula

$$x_i := f_i(x_1, \ldots, x_n), \tag{1.1}$$

while receiving information from other processors on the current values of the remaining components.

Our discussion of distributed implementations of iteration (1.1) focuses on mechanisms for interprocessor communication and synchronization. We also consider asynchronous implementations and present a survey of the convergence issues that arise in the face of asynchronism. These issues are discussed in more detail in Bertsekas and Tsitsiklis (1989b) where proofs of most of the results quoted here can be found.

Iteration (1.1) can be executed *synchronously* whereby processors perform an iteration, communicate their results to the other processors, and then proceed to the next iteration. In Section 2, we introduce two alternative synchronous iterations, namely Jacobi type and Gauss–Seidel type iterations, and discuss briefly their parallelization. In Section 3, we indicate that synchronous parallel execution is feasible even if the underlying computing system is inherently asynchronous (i.e. no processor has access to a global clock) provided that certain synchronization mechanisms are in place. We review and compare three representative synchronization methods. We also discuss some basic communication problems that arise naturally in parallel iterations, assuming that processors communicate using a point-to-point communication network. Then, in Section 4, we provide a more detailed analysis of the required time per parallel iteration. In Section 5, we indicate that the synchronous execution of iteration (1.1) can have certain drawbacks, thus motivating *asynchronous* implementations whereby each processor computes at its own pace while receiving (possibly outdated) information on the values of the components updated by the other processors. An asynchronous implementation of iteration (1.1) is not mathematically equivalent to its synchronous counterpart and an otherwise convergent algorithm may become divergent. It will be seen that asynchronous iterative algorithms can display several and different convergence behaviors, ranging from divergence to guaranteed convergence in the face of the worst possible amount of asynchronism and communication delays. We classify the possible behaviors in three broad classes; the corresponding convergence results are surveyed in Sections 6, 7 and 8, respectively. In Section 9, we address some difficulties that arise if we wish to terminate an asynchronous distributed algorithm in finite time. Finally, Section 10 contains our conclusions and a brief discussion of future research directions.

2. JACOBI AND GAUSS–SEIDEL ITERATIONS

Let $X_1, \ldots, X_p$, be subsets of the Euclidean spaces $\mathscr{R}^{n_1}, \ldots, \mathscr{R}^{n_p}$, respectively. Let $n = n_1 + \cdots + n_p$, and let $X \subset \mathscr{R}^n$ be the Cartesian product $X = \prod_{i=1}^{p} X_i$. Accordingly, any $x \in \mathscr{R}^n$ is decomposed in the form $x = (x_1, \ldots, x_p)$, with each x_i belonging to $\mathscr{R}^{n_i}$. For $i = 1, \ldots, p$, let $f_i : X \mapsto X_i$ be a given function and let $f : X \mapsto X$ be the function defined by $f(x) = (f_1(x), \ldots, f_p(x))$ for every $x \in X$. We want to solve the fixed point problem $x = f(x)$. To this end we will consider the iteration

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

We will also consider the more general iteration

$$x_i := \begin{cases} f_i(x) & \text{if } i \in I \\ x_i & \text{otherwise,} \end{cases} \tag{2.1}$$

where I is a subset of the component index set $\{1, \ldots, p\}$, which may change from one iteration to the next.

We are interested in the distributed implementation of such iterations. While some of the discussion applies to shared memory systems, we will focus in this and the next two sections on a message-passing system with p processors, each having its own local memory and communicating with the other processors over a communication network. We assume that the ith processor has the responsibility of updating the ith component x_i according to the rule $x_i := f_i(x)$. It is implicitly assumed here that the ith processor knows the form of the function f_i. In the special case where $f(x) = Ax + b$, where A is an $n \times n$ matrix and $b \in \mathscr{R}^n$, this amounts to assuming that the ith processor knows the *rows* of the matrix A corresponding to the components assigned to it. Other implementations of the linear iteration $x := Ax + b$ are also possible. For example, each processor could be given certain *columns* of A. We do not pursue this issue further and refer the reader to McBryan and Van der Velde (1987) and Fox *et al.* (1988) for discussions of alternative matrix storage schemes.

For implementation of the iteration, it is seen that if the function f_j depends on x_i (with $i \neq j$), then processor j must be informed by processor i on the current value of x_i. To capture such data dependencies, we form a directed graph $G = (N, A)$, called the *dependency graph* of the algorithm, with nodes $N = \{1, \ldots, p\}$ and with arcs $A = \{(i, j) \mid i \neq j \text{ and } f_j \text{ depends on } x_i\}$. We assume that for every arc (i, j) in the dependency graph there is a communication capability by means of which processor i can relay information to processor j. We also assume that messages are received correctly within a finite but otherwise arbitrary amount of time. Such communication may be possible through a direct communication link joining processors i and j or it could consist of a multi-hop path in a communication network. The discussion that follows applies to both cases.

An iteration in which all of the components of x are simultaneously updated [$I = \{1, \ldots, p\}$ in (2.1)], is sometimes called a *Jacobi* type iteration. In an alternative form, the components of x are updated one at a time, and the most recently computed values of the other components are used. The resulting iteration is often called an iteration of the

Gauss–Seidel type and is described mathematically by

$$x_i(t+1) = f_i(x_1(t+1), \dots, x_{i-1}(t+1), x_i(t), \dots, x_p(t)), \quad i = 1, \dots, p. \tag{2.2}$$

In a serial computing environment, Gauss–Seidel iterations are often preferable. As an example, consider the linear case where $f(x) = Ax + b$, and A has non-negative elements and spectral radius less than one. Then, the classical Stein–Rosenberg theorem [see e.g. Bertsekas and Tsitsiklis (1989b, p. 152)] states that both the Gauss–Seidel and the Jacobi iterations converge at a geometric rate to the unique fixed point of f; however, in a serial setting where one Jacobi iteration takes as much as one Gauss–Seidel iteration, the rate of convergence of the Gauss–Seidel iteration is always faster. Surprisingly, in a parallel setting this conclusion is reversed, as we now describe in a somewhat more general context.

Consider the sequence $\{x^J(t)\}$ generated by the Jacobi iteration

$$x^J(t+1) = f(x^J(t)), \quad t = 0, 1, \dots \tag{2.3}$$

and the sequence $\{x^G(t)\}$ generated by the Gauss–Seidel iteration (2.2), started from the same initial condition $x(0) = x^J(0) = x^G(0)$. The following result is proved in Tsitsiklis (1989) generalizing an earlier result of Smart and White (1988):

Proposition 1. Suppose that $f: \mathcal{R}^n \to \mathcal{R}^n$ has a unique fixed point x^*, and is monotone, that is, it satisfies $f(x) \le f(y)$ if $x \le y$. Then, if $f(x(0)) \le x(0)$, we have

$$x^* \le x^J(pt) \le x^G(t), \quad t = 0, 1, \dots$$

and if $x(0) \le f(x(0))$, we have

$$x^G(t) \le x^J(pt) \le x^*, \quad t = 0, 1, \dots.$$

Proposition 1 establishes the faster parallel convergence of the Jacobi iteration, for certain initial conditions, assuming that a Gauss–Seidel iteration takes as much parallel time as p Jacobi iterations. It has also been shown in Smart and White (1988) that if in addition to the assumptions of Proposition 2.1, f is linear (and thus satisfies the assumptions of the Stein–Rosenberg theorem), the rate of convergence of the Jacobi iteration is faster than the rate of convergence of the Gauss–Seidel iteration. An extension of this result that applies to asynchronous Jacobi and Gauss–Seidel iterations is also given in Bertsekas and Tsitsiklis (1989a).

The preceding comparison of Jacobi and Gauss–Seidel iterations assumes that a Jacobi iteration is executed in one time step, and that the Gauss–Seidel iteration cannot be parallelized (so that a full update of all the components $x_1, \dots, x_p$ requires p time steps). This is the case when the number of available processors is p and the dependency graph describing the structure of the iteration is complete (every component depends on every other component), so that no two components can be updated in parallel. A Gauss–Seidel iteration can still converge faster, however, if it can be parallelized to the point where it requires the same number of time steps as the corresponding Jacobi iteration; this can happen if the number of available processors is less than p and the dependency graph is sufficiently sparse, as we now illustrate.

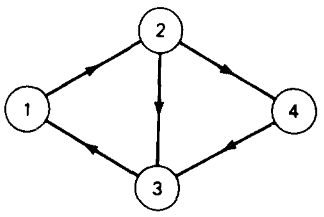


FIG. 1. A dependency graph.

Consider the dependency graph of Fig. 1. A corresponding Gauss–Seidel iteration is described by

$$\begin{aligned}
x_1(t+1) &= f_1(x_1(t), x_3(t)) \\
x_2(t+1) &= f_2(x_1(t+1), x_2(t)) \\
x_3(t+1) &= f_3(x_2(t+1), x_3(t), x_4(t)) \\
x_4(t+1) &= f_4(x_2(t+1), x_4(t))
\end{aligned}$$

and its structure is shown in Fig. 2. We notice here that $x_3(t+1)$ and $x_4(t+1)$ can be computed in parallel. In particular, a *sweep*, that is, an update of all four components, can be performed in only three stages. On the other hand, a different ordering of the components leads to an iteration of the form

$$\begin{aligned}
x_1(t+1) &= f_1(x_1(t), x_3(t)) \\
x_3(t+1) &= f_3(x_2(t), x_3(t), x_4(t)) \\
x_4(t+1) &= f_4(x_2(t), x_4(t)) \\
x_2(t+1) &= f_2(x_1(t+1), x_2(t))
\end{aligned}$$

which is illustrated in Fig. 3. We notice here that $x_1(t+1)$, $x_3(t+1)$, and $x_4(t+1)$ can be computed in parallel, and a sweep requires only two stages.

The above example motivates the problem of choosing an ordering of the components for which a sweep requires the least number of stages. The solution of this problem, given in Bertsekas and Tsitsiklis (1989b, p. 23) is as follows:

Proposition 2. The following are equivalent:

(i) There exists an ordering of the variables such that a sweep of the corresponding Gauss–Seidel algorithm can be performed in K parallel steps.

(ii) We can assign colors to the nodes of the dependency graph so that at most K different colors

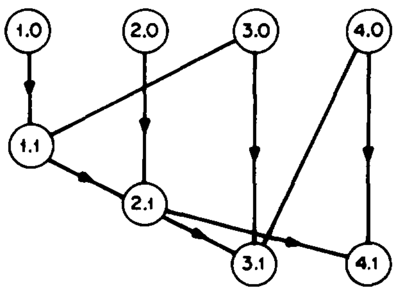


FIG. 2. The data dependencies in a Gauss–Seidel iteration.

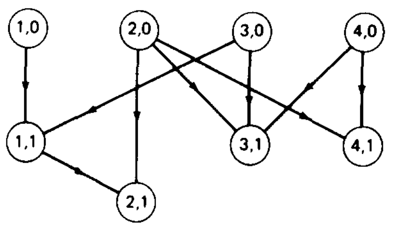


FIG. 3. The data dependencies in a Gauss–Seidel iteration for a different updating order.

are used and so that each subgraph obtained by restricting to the set of nodes with the same color has no directed cycles.

A well known special case of the above proposition arises when the dependency graph G is symmetric; that is, the presence of an arc $(i, j) \in A$ also implies the presence of the arc (j, i). In this case there is no need to distinguish between directed and undirected cycles, and the coloring problem of Proposition 2 reduces to coloring the nodes of the dependency graph so that no two neighboring nodes have the same color.

Unfortunately, the coloring problem of Proposition 2 is intractable (NP-hard). On the other hand, in several practical situations the dependency graph G has a very simple structure and the coloring problem can be solved by inspection. Furthermore, it can be shown that if the dependency graph is a tree or a two-dimensional grid, only two colors suffice, so a Gauss–Seidel sweep can be done in two steps, with roughly half the components of x being updated in parallel at each step. In this case, while with n processors the Jacobi method is as fast or faster than Gauss–Seidel, the reverse is true when using $n/2$ processors (or more generally, any number of processors with which a Gauss–Seidel step can be completed in the same time as the Jacobi iteration).

Even with unstructured dependency graphs, reasonably good colorings can be found using simple heuristics; see Zenios and Lasken (1988) and Zenios and Mulvey (1988), for examples. Let us also point out that the parallelization of Gauss–Seidel methods by means of coloring is very common in the context of the numerical solution of partial differential equations; see, for example, Ortega and Voigt (1985) and the references therein.

A related approach for parallelizing Gauss–Seidel iterations, which is fairly easy to implement, is discussed in Barbosa (1986) and Barbosa and Gafni (1987). In this approach, a new sweep is allowed to start before the previous one has been completed and for this reason, one obtains, in general, somewhat greater parallelism than that obtained by the coloring approach.

We finally note that the order in which the variables are updated in a Gauss–Seidel sweep may have a significant effect on the convergence rate of the iteration. Thus, completing a Gauss–Seidel sweep in a minimum number of steps is not the only consideration in selecting the grouping of variables to be updated in parallel; the corresponding rate of convergence must also be taken into account.

3. SYNCHRONIZATION AND COMMUNICATION ISSUES

We say that an execution of iteration (2.1) is *synchronous* if it can be described mathematically by the formula

$$x_i(t+1) = \begin{cases} f_i(x_1(t), \ldots, x_p(t)) & \text{if } t \in T^i \\ x_i(t) & \text{otherwise.} \end{cases} \quad (3.1)$$

Here, t is an integer-valued variable used to index different iterations, not necessarily representing real time, and T^i is an infinite subset of the index set $\{0, 1, \ldots\}$. Thus, T^i is the set of time indices at which x_i is updated. With different choices of T^i one obtains different algorithms, including Jacobi and Gauss–Seidel type of methods. We will later contrast synchronous iterations with asynchronous iterations, where instead of the current component values $x_j(t)$, earlier values $x_j(t-d)$ are used in (3.1), with d being a possibly positive and unpredictable "communication delay" that depends on i, j and t.

3.1. *Synchronization methods*

Synchronous execution is certainly possible if the processors have access to a global clock, and if messages can be reliably transmitted from one processor to another between two consecutive "ticks" of the clock. Barring the existence of a global clock, synchronous execution can be still accomplished by using synchronization protocols called *synchronizers*. We refer the reader to Awerbuch (1985) for a comparative complexity analysis of a class of synchronizers and we continue with a brief discussion of three representative synchronization methods. These methods will be described for the case of Jacobi type iterations, but they can be easily adapted for the case of Gauss–Seidel iterations as well.

(*a*) *Global synchronization.* Here the processors proceed to the $(t+1)$st iteration, also referred to as phase, only after every processor i has completed the tth iteration and has received the value of $x_j(t)$ from every j such that $(j, i) \in A$. Global synchronization can be implemented by a variety of techniques, a simple one being the following: the processors are arranged as a spanning tree, with a particular processor chosen to be the root of the tree. Once processor i has computed $x_i(t)$, has received the value of $x_j(t)$ for every j such that $(j, i) \in A$, and has received a phase termination message from all its "children" in the tree, it sends a phase termination message to its "father" in the tree. Phase termination messages thus propagate towards the root. Once the root has received a phase termination message from all of its children, it knows that the current phase has been completed and sends a phase initiation message to its children, which is propagated along the spanning tree. Once a processor receives such a message it can proceed to the next phase. (See Fig. 4 for an illustration.)

(*b*) *Local synchronization.* Global synchronization can be seen to be rather wasteful in terms of the time

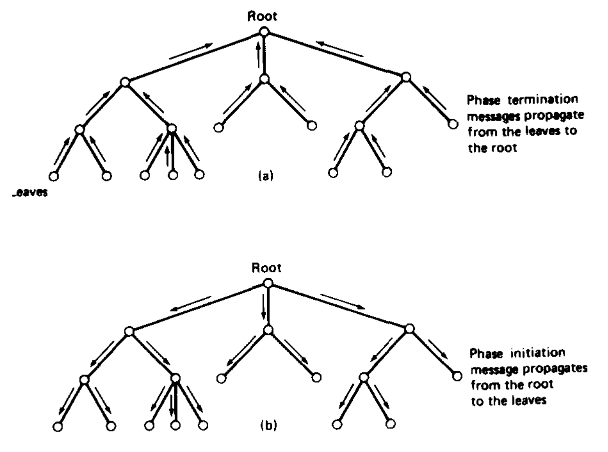

Root
Phase termination messages propagate from the leaves to the root
Leaves
(a)
Root
Phase initiation message propagates from the root to the leaves
(b)

FIG. 4. Illustration of the global synchronization method.

required per iteration. An alternative is to allow the ith processor to proceed with the $(t+1)$st iteration as soon as it has received all the messages $x_j(t)$ it needs. Thus, processor i moves ahead on the basis of local information alone, obviating the need for propagating messages along a spanning tree.

It is easily seen that the iterative computation can only proceed faster when local synchronization is employed. Furthermore, this conclusion can also be reached even if a more efficient global synchronization method were possible whereby all processors start the $(t+1)$st iteration immediately after all messages generated by the tth iteration have been delivered. (We refer to this hypothetical and practically unachievable situation as the ideal global synchronization.) Let us assume that the time required for one computation and the communication delays are bounded above by a finite constant and are bounded below by a positive constant. Then it is easily shown that the time spent for a number K of iterations under ideal global synchronization is at most a constant multiple of the corresponding time when local synchronization is employed.

The advantage of local synchronization is better seen if communication delays do not obey any *a priori* bound. For example, let us assume that the communication delay of every message is an independent exponentially distributed random variable with mean one. Furthermore, suppose for simplicity, that each processor sends messages to exactly d other processors, where d is some constant (i.e. the outdegree of each node of the dependency graph is equal to d). With global synchronization, the real time spent for one iteration is roughly equal to the maximum of dp independent exponential random variables and its expectation is, therefore, of the order of $\log(dp)$. Thus, the expected time needed for K iterations is of the order of $K\log(pd)$. On the other hand, with local synchronization, it turns out that the expected time for K iterations is of the order of $\log p + K\log d$ [joint work with C. H. Papadimitriou; see Bertsekas and Tsitsiklis (1989b, p. 104)]. If K is large, then local synchronization is faster by a factor roughly equal to $\log(pd)/\log d$. Its advantage is more pronounced if d is much smaller than p, as is the case in most practical applications. Some related analysis and experiments can be found in Dubois and Briggs (1982).

(*c*) *Synchronization via rollback.* This method, introduced by Jefferson (1985), has been primarily applied to the simulation of discrete-event systems. It can also be viewed as a general purpose synchronization method but it is likely to be inferior to the preceding two methods in applications involving solution of systems of equations. Consider a situation where the message $x_j(t)$ transmitted from some processor j to some other processor i is most likely to take a fixed default value known to i. In such a case, processor i may go ahead with the computation of $x_i(t+1)$ without waiting for the value of $x_j(t)$, by making the assumption that $x_j(t)$ will take the default value. In case that a message comes later which falsifies the assumption that $x_j(t)$ has the default value, then a *rollback* occurs; that is, the computation of $x_i(t+1)$ is invalidated and is performed once more, taking into account the correct value of $x_j(t)$. Furthermore, if a processor has sent messages based on computations which are later invalidated, it sends *antimessages* which cancel the earlier messages. A reception of such an antimessage by some other processor k could invalidate some of k's computations and could trigger the transmission of further antimessages by k. This process has the potential of explosive generation of antimessages that could drain the available communication resources. On the other hand, it is hoped that the number of messages and antimessages would remain small in problems of practical interest, although insufficient analytical evidence is available at present. Some probabilistic analyses of the performance of this method can be found in Lavenberg *et al.* (1983) and Mitra and Mitrani (1984).

3.2. *Single and multinode broadcasting*

Regardless of whether the implementation is synchronous or not, it is necessary to exchange some information between the processors after each iteration. The interprocessor communication time can be substantial when compared to the time devoted to computations, and it is important to carry out the message exchanges as efficiently as possible. There are a number of generic communication problems that arise frequently in iterative and other algorithms. We describe a few such tasks related to message broadcasting.

In the first communication task, we want to send the same message from a given processor to every other processor (we call this a *single node broadcast*). In a generalized version of this problem, we want to do a single node broadcast simultaneously from all nodes (we call this a *multinode broadcast*). A typical example where a multinode broadcast is needed arises in the iteration $x := f(x)$. If we assume that there is a separate processor assigned to component x_i, $i = 1, \ldots, p$, and that the function f_i depends on all components x_j, $j = 1, \ldots, p$, then, at the end of an iteration, there is a need for every processor to send

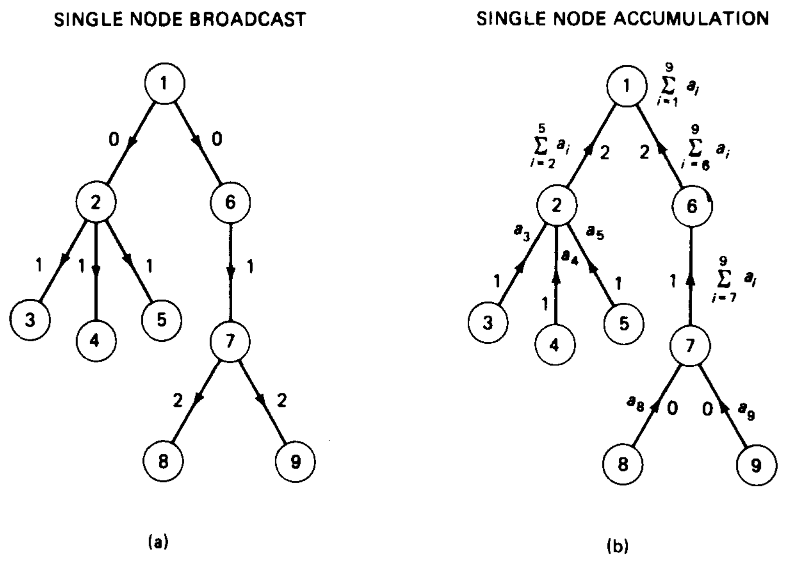


FIG. 5. (a) A single node broadcast uses a tree that is rooted at a given node (which is node 1 in the figure). The time next to each link is the time that transmission of the packet on the link begins. (b) A single node accumulation problem involving summation of n scalars $a_1, \ldots, a_n$ (one per processor) at the given node (which is node 1 in the figure). The time next to each link is the time at which transmission of the "combined" packet on the link begins, assuming that the time for scalar addition is negligible relative to the time required for packet transmission.

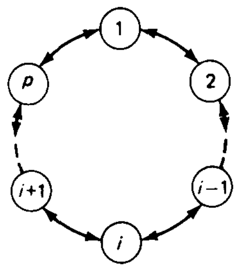
the value of its component to every other processor, which is a multinode broadcast.

Clearly, to solve the single node broadcast problem, it is sufficient to transmit the given node's message along a spanning tree rooted at the given node, that is, a spanning tree of the network together with a direction on each link of the tree such that there is a unique path from the given node (called the *root*) to every other node. With an optimal choice of such a spanning tree, a single node broadcast takes $\Theta(r)$-time,† where r is the diameter of the network, as shown in Fig. 5(a). To solve the multinode broadcast problem, we need to specify one spanning tree per root node. The difficulty here is that some links may belong to several spanning trees; this complicates the timing analysis, because several messages can arrive simultaneously at a node, and require transmission on the same link with a queueing delay resulting.

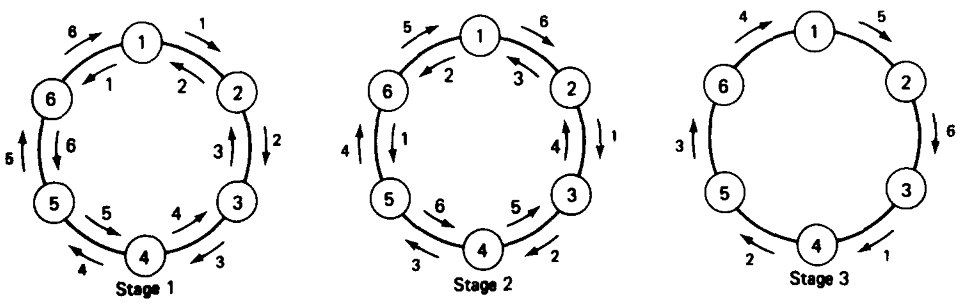
There are two important communication problems that are dual to the single and multinode broadcasts, in the sense that the spanning tree(s) used to solve one problem can also be used to solve the dual in the same amount of communication time. In the first problem, called *single node accumulation*, we want to send to a given node a message from every other node; we assume, however, that messages can be "combined" for transmission on any communication link, with a "combined" transmission time equal to the transmission time of a single message. This problem arises, for example, when we want to form at a given node a sum consisting of one term for each node, as in an inner product calculation [see Fig. 5(b)]; we can view addition of scalars at a node as "combining" the corresponding messages into a single message. The second problem, which is dual to a multinode broadcast, is called *multinode accumulation*, and involves a separate single node accumulation at each node. It can be shown that a single node (or multinode) accumulation problem can be solved in the same time as a single node (respectively multinode) broadcast problem, by realizing that an accumulation algorithm can be viewed as a broadcast algorithm running in reverse time, as illustrated in Fig. 5. As shown in Fig. 4, global synchronization can be accomplished by a single node broadcast followed by a single node accumulation.

Algorithms for solving the broadcast problems just described, together with other related communication problems, have been developed for several popular architectures (Nassimi and Sahni, 1980; Saad and Shultz, 1987; McBryan and Van der Velde, 1987; Ozveren, 1987; Bertsekas *et al.*, 1989; Bertsekas and Tsitsiklis, 1989b; Johnsson and Ho, 1989). Table 1 gives the order of magnitude of the time needed to solve each of these problems using an optimal algorithm. The underlying assumption for the results of this table is that each message requires unit time for transmission on any link of the interconnection network, and that each processor can transmit and receive a message simultaneously on all of its incident links. Specific algorithms that attain these times are given in Bertsekas *et al.* (1989) and Bertsekas and Tsitsiklis (1989b, Section 1.3.4). In most cases these algorithms are optimal in that they solve the problem in the minimum possible number of time steps. Figure 6 illustrates a multinode broadcast algorithm for a ring

† The notation $h(y) = \Theta(g(y))$, where y is a positive integer, means that for some $c_1 > 0$, $c_2 > 0$, and $y_0 > 0$, we have $c_1|g(y)| \le h(y) \le c_2|g(y)|$ for all $y \ge y_0$.



(a)



(b)

FIG. 6. (a) A ring of p nodes having as links the pairs $(i, i+1)$ for $i = 1, 2, \ldots, p-1$, and $(p, 1)$. (b) A multinode broadcast on a ring with p nodes can be performed in $\lceil (p-1)/2 \rceil$ stages as follows: at stage 1, each node sends its own packet to its clockwise and counterclockwise neighbors. At stage $2, \ldots, \lceil (p-1)/2 \rceil$, each node sends to its clockwise neighbor the packet received from its counterclockwise neighbor at the previous stage; also, at stages $2, \ldots, \lceil (p-2)/2 \rceil$, each node sends to its counterclockwise neighbor the packet received from its clockwise neighbor at the previous stage. The figure illustrates this process for $p = 6$.

with p processors, which attains the minimum number of steps.

Using the results of Table 1, it is also shown in Bertsekas and Tsitsiklis (1989b) that if a hypercube is used, then most of the basic operations of numerical linear algebra, i.e. inner product, matrix–vector multiplication, matrix–matrix multiplication, power of a matrix, etc., can be executed in parallel in the same order of time as when communication is instantaneous. In some cases this is also possible when the processors are connected with a less powerful interconnection network such as a square mesh. Thus, communication affects only the "multiplying constant" as opposed to the order of time needed to carry out these operations. Nonetheless, with a large number of processors, the effect of communication delays on linear algebra operations can be very substantial.

4. ITERATION COMPLEXITY

We now try to assess the potential benefit from parallelization of the iteration $x := f(x)$. In particular, we will estimate the order of growth of the required time per iteration, as the dimension n increases. Our analysis is geared towards large problems and the issue of speedup of iterative methods using a large number of processors. We will make the following assumptions:

(a) All components of x are updated at each iteration. (This corresponds to a Jacobi iteration. If a Gauss–Seidel iteration is used instead, the time per iteration cannot increase, since by updating only a subset of the components, the computation per iteration will be reduced and the communication problem will be simplified. Based on this, it can be seen that the order of required time will be unaffected if in place of a Jacobi iteration, we perform a Gauss–Seidel sweep with

TABLE 1. SOLUTION TIMES OF OPTIMAL ALGORITHMS FOR THE BROADCAST AND ACCUMULATION PROBLEMS USING A RING, A BINARY BALANCED TREE, A d-DIMENSIONAL MESH (WITH THE SAME NUMBER OF PROCESSORS ALONG EACH DIMENSION), AND A HYPERCUBE WITH p PROCESSORS. THE TIMES GIVEN FOR THE RING ALSO HOLD FOR A LINEAR ARRAY

Problem	Ring	Tree	Mesh	Hypercube
Single node broadcast (or single node accumulation)	$\Theta(p)$	$\Theta(\log p)$	$\Theta(p^{1/d})$	$\Theta(\log p)$
Multinode broadcast (or multinode accumulation)	$\Theta(p)$	$\Theta(p)$	$\Theta(p)$	$\Theta(p/\log p)$

a number of steps which is fixed and independent of the dimension n.)

(b) There are n processors, each updating a single scalar component of x at each iteration. (One may wish to use fewer than n processors, say p, each updating an n/p-dimensional component of x, in order to economize on communication. We argue later, however, that under our assumptions, choosing $p < n$ cannot improve the order of time required per iteration, although it may reduce this time by a constant factor. In practice, of course, the number of available processors is often much less than n, and it is interesting to consider optimal utilization of a limited number of processors in the context of iterative methods. In this paper, however, we will not address this issue, prefering to concentrate on the potential and limitations of iterative computation using massively parallel machines with an abundant number of processors.)

(c) Following the execution of their assigned portion of the iteration, the processors exchange the updated values of their components by means of a communication algorithm such as a multinode broadcast. The subsequent synchronization takes negligible time. (This can be justified by noting that local synchronization can be accomplished as part of the communication algorithm and thus requires no additional time. Furthermore, global synchronization can be done by means of a single node broadcast followed by a single node accumulation. Thus the time required for global synchronization grows with n no faster than a multinode broadcast time. Therefore, if the communication portion of the iteration is done by a multinode broadcast, the global synchronization time can be ignored when estimating the order of required time per iteration.)

We estimate the time per iteration as

$$T_{\mathrm{COMP}} + T_{\mathrm{MNB}},$$

where T_{COMP} is the time to compute the updated components $f_i(x)$, and T_{MNB} is the time to exchange the updated component values between the processors as necessary. If there is overlap of the computation and communication phases due to some form of pipelining, the time per iteration will be smaller than $T_{\mathrm{COMP}} + T_{\mathrm{MNB}}$ but its order of growth with n will not change. We consider several hypotheses for T_{COMP} and T_{MNB}, corresponding to different types of computation and communication hardware, and structures of the functions f_i. In particular, we consider the following cases, motivated primarily by the case where the system of equations $x = f(x)$ is linear:

Small T_{COMP}: $(= \Theta(1))$. One example for this case is when the iteration functions f_i are linear and correspond to a very sparse system (the maximum node degree of the dependency graph is $\Theta(1)$). Another example is when the system solved is linear and dense, but each processor has vector processing capability allowing it to compute inner products in $\Theta(1)$ time.

Medium T_{COMP}: $(= \Theta(\log n))$. An example for this case is when the system solved is linear and dense, and each processor can compute an inner product in $\Theta(\log n)$ time. It can be shown that this is possible if each processor is itself a message-passing parallel processor with $\log n$ diameter.

Large T_{COMP}: $(= \Theta(n))$. An example for this case is when the system solved is linear and dense, and each processor computes inner products serially in $\Theta(n)$ time.

Also the following are considered for the communication time T_{MNB}:

Small T_{MNB}: $(= \Theta(1))$. An example for this case is when special very fast communication hardware is used, making the time for the multinode broadcast negligible relative to T_{COMP} or relative to the communication software overhead at the message sources. Another example is when the processors are connected by a network that matches the form of the dependency graph, so that all necessary communication involves directly connected nodes. For example when solving partial differential equations, the dependency graph is often a grid resulting from discretization of physical space. Then, with processors arranged in an appropriate grid, communication can be done very fast.

Medium T_{MNB}: $(= \Theta(n/\log n))$. An example for this case is when the multinode broadcast is performed using a hypercube network (cf. Table 1).

Large T_{MNB}: $(= \Theta(n))$. An example for this case is when the multinode broadcast is performed using a ring network or a linear array (cf. Table 1).

Table 2 gives the time per iteration $T_{\mathrm{COMP}} + T_{\mathrm{MNB}}$ for the different combinations of cases. In the worst case, the time per iteration is $\Theta(n)$, and this time is faster by a factor n than the time needed to execute serially the linear iteration $x := Ax + b$ when the matrix A is fully dense. In this case, the speedup is proportional to the number of processors n and the benefit from parallelization is very substantial. This thought, however, must be tempered by the realization that the parallel solution time still increases at least linearly with n, unless the number of iterations needed to solve the problem within practical accuracy decreases with n—an unlikely possibility.

Table 2. Time per iteration $x := f(x)$ under a variety of assumptions for the computation time per iteration T_{COMP} and the communication time per iteration T_{MNB}. In the cells above the diagonal, the computation time is the bottleneck, and in the cells below the diagonal, the communication time is the bottleneck

	T_{COMP}: $\Theta(1)$	T_{COMP}: $\Theta(\log n)$	T_{COMP}: $\Theta(n)$
T_{MNB}: $\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$
T_{MNB}: $\Theta(n/\log b)$	$\Theta(n/\log n)$	$\Theta(n/\log n)$	$\Theta(n)$
T_{MNB}: $\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

In the best case of Table 2, the time per iteration is bounded irrespectively of the dimension n, offering hope that with special computing and communication hardware, some extremely large practical problems can be solved in reasonable time.

Another interesting case, which is not covered by Table 2, arises in connection with the linear iteration $x := Ax + b$, where A is an $n \times n$ fully dense matrix. It can be shown that this iteration can be executed in a hypercube network of n^2 processors in $\Theta(\log n)$ time [see, for example, Bertsekas and Tsitsiklis (1989b)]. While it is hard to imagine at present hypercubes of n^2 processors solving large $n \times n$ systems, this case provides a theoretical limit for the time per iteration for unstructured linear systems in message-passing machines.

Consider now the possibility of using $p < n$ processors, each updating an n/p-dimensional component of x. The computation time per iteration will then increase by a factor n/p, so the question arises whether it is possible to improve the order of growth of the communication time in the cases where T_{MNB} is the iteration time bottleneck. The cases of medium and large T_{MNB} are of principal interest here. In these cases the corresponding times measured in message transmission time units are $\Theta(p/\log p)$ and $\Theta(p)$, respectively. Because, however, each message involves n/p values, its transmission time grows linearly with n/p, so the corresponding time T_{MNB} becomes $\Theta(n/\log p)$ and $\Theta(n)$, respectively. Thus the order of time per iteration is not improved by choosing $p < n$, at least under the hypotheses of this section.

5. ASYNCHRONOUS ITERATIONS

Asynchronous iterations have been introduced by Chazan and Miranker (1969) (under the name *chaotic relaxation*) for the solution of linear equations. In an asynchronous implementation of the iteration $x := f(x)$, processors are not required to wait to receive all messages generated during the previous iteration. Rather, each processor is allowed to keep iterating on its own component at its own pace. If the current value of the component updated by some other processor is not available, then some outdated value received at some time in the past is used instead. Furthermore, processors are not required to communicate their results after each iteration but only once in a while. We allow some processors to compute faster and execute more iterations than others, we allow some processors to communicate more frequently than others, and we allow the communication delays to be substantial and unpredictable. We also allow the communication channels to deliver messages out of order, i.e. in a different order than the one they were transmitted.

There are several potential advantages that may be gained from asynchronous execution [see Kung (1976) for a related discussion].

(a) *Reduction of the synchronization penalty.* There is no overhead such as the one associated with the global synchronization method. In particular, a processor can proceed with the next iteration without waiting for all other processors to complete the current iteration, and without waiting for a synchronization algorithm to execute. Furthermore, in certain cases, there are even advantages over the local synchronization method as we now discuss. Suppose that an algorithm happens to be such that each iteration leaves the value of x_i unchanged. With local synchronization, processor i must still send messages to every processor j with $(i, j) \in A$ because processor j will not otherwise proceed to the next iteration. Consider now a somewhat more realistic case where the algorithm is such that a typical iteration is very likely to leave x_i unchanged. Then each processor j with $(i, j) \in A$ will be often found in a situation where it waits for rather uninformative messages stating that the value of x_i has not changed. In an asynchronous execution, processor j does not wait for messages from processor i and the progress of the algorithm is likely to be faster. A similar argument can be made for the case where x_i changes only slightly between iterations. Notice that the situation is similar to the case of synchronization via rollback, except that in an asynchronous algorithm processors do not roll back even if they iterate on the basis of outdated and later invalidated information.

(b) *Ease of restarting.* Suppose that the processors are engaged in the solution of an optimization problem and that suddenly one of the parameters of the problem changes. (Such a situation is common and natural in the context of data networks or in the quasistatic control of large scale systems.) In a synchronous execution, all processors should be informed, abort the computation, and then reinitiate (in a synchronized manner) the algorithm. In an asynchronous implementation no such reinitialization is required. Rather, each processor incorporates the new parameter value in its iterations as soon as it learns the new value, without waiting for all processors to become aware of the parameter change. When all processors learn the new parameter value, the algorithm becomes the correct (asynchronous) iteration.

(c) *Reduction of the effect of bottlenecks.* Suppose that the computational power of processor i suddenly deteriorates drastically. In a synchronous execution the entire algorithm would be slowed down. In an asynchronous execution, however, only the progress of x_i and of the components strongly influenced by x_i would be affected; the remaining components would still retain the capacity of making unhampered progress. Thus the effects of temporary malfunctions tend to be localized. The same argument applies to the case where a particular communication channel is suddenly slowed down.

(d) *Convergence acceleration due to a Gauss–Seidel effect.* With a Gauss–Seidel execution, convergence often takes place with fewer updates of each component, the reason being that new information is incorporated faster in the update formulas. On the other hand Gauss–Seidel iterations are generally less parallelizable. Asynchronous algorithms have the potential of displaying a Gauss–Seidel effect because newest information is incorporated into the computations as soon as it becomes available, while retaining maximal parallelism as in Jacobi-type algorithms.

A major potential drawback of asynchronous algorithms is that they cannot be described mathematically by the iteration $x(t+1)=f(x(t))$. Thus, even if this iteration is convergent, the corresponding asynchronous iteration could be divergent, and indeed this is sometimes the case. Even if the convergence of the asynchronous iteration can be established, the corresponding analysis is often difficult. Nevertheless, there is a large number of results stating that certain classes of important algorithms retain their desirable convergence properties in the face of asynchronism: they will be surveyed in Sections 6–8. Another difficulty relates to the fact that an asynchronous algorithm may have converged (within a desired accuracy) but the algorithm does not terminate because no processor is aware of this fact. We address this issue in Section 9.

We now present our model of asynchronous computation. Let the set X and the function f be as described in Section 2. Let t be an integer variable used to index the events of interest in the computing system. Although t will be referred to as a time variable, it may have little relation with "real time". Let $x_i(t)$ be the value of x_i residing in the memory of the ith processor at time t. We assume that there is a set of times T^i at which x_i is updated. To account for the possibility that the ith processor may not have access to the most recent values of the components of x, we assume that

$$x_i(t+1)=f_i(x_1(\tau_1^i(t)),\ldots,x_n(\tau_n^i(t))),\quad \forall t\in T^i, \tag{5.1}$$

where $\tau_j^i(t)$ are times satisfying

$$0\le \tau_j^i(t)\le t,\quad \forall t\ge 0.$$

At all times $t\notin T^i$, $x_i(t)$ is left unchanged and

$$x_i(t+1)=x_i(t),\quad \forall t\notin T^i. \tag{5.2}$$

We assume that the algorithm is initialized with some $x(0)\in X$.

The above mathematical description can be used as a model of asynchronous iterations executed by either a message-passing distributed system or a shared-memory parallel computer. For an illustration of the latter case, see Fig. 7.

The difference $t-\tau_j^i(t)$ is equal to zero for a synchronous execution. The larger this difference is, the larger is the amount of asynchronism in the algorithm. Of course, for the algorithm to make any progress at all we should not allow $\tau_j^i(t)$ to remain forever small. Furthermore, no processor should be allowed to drop out of the computation and stop iterating. For this reason, certain assumptions need to be imposed. There are two different types of assumptions which we state below.

Assumption 1. (Total asynchronism). The sets T^i are infinite and if $\{t_k\}$ is a sequence of elements of T^i which tends to infinity, then $\lim_{k\to\infty}\tau_j^i(t_k)=\infty$ for every j.

Assumption 2. (Partial asynchronism). There exists a positive constant B such that:

(a) For every $t\ge 0$ and every i, at least one of the elements of the set $\{t,t+1,\ldots,t+B-1\}$ belongs to T^i.

(b) There holds

$$t-B<\tau_j^i(t)\le t,\quad \forall i,j,\ \forall t\in T^i. \tag{5.3}$$

(c) There holds $\tau_i^i(t)=t$, for all i and $t\in T^i$.

The constant B of Assumption 2, to be called the *asynchronism measure*, bounds the amount by which the information available to a processor can be outdated. Notice that a Jacobi-type synchronous iteration is the special case of partial asynchronism in which $B=1$. Notice also that Assumption (c) states that the information available to processor i regarding its own component is never outdated. Such an assumption is natural in most contexts, but could be violated in certain types of shared memory parallel computing systems if we allow more than one processor to update the same component of x. It turns

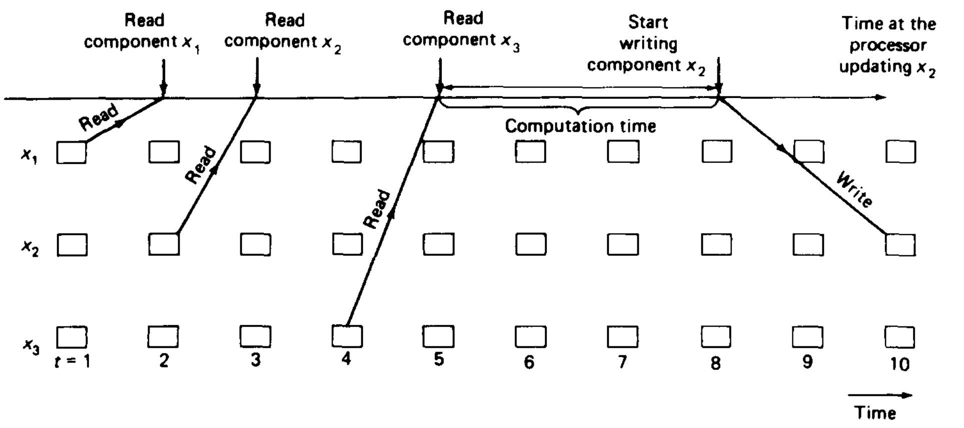


FIG. 7. Illustration of a component update in a shared memory multiprocessor. Here x_2 is viewed as being updated at time $t=9$ ($9\in T^2$), with $\tau_1^2(9)=1$, $\tau_2^2(9)=2$, and $\tau_4^2(9)=4$. The updated value of x_2 is entered at the corresponding register at $t=10$. Several components can be simultaneously in the process of being updated, and the values of $\tau_j^i(t)$ can be unpredictable.

out that if we relax Assumption 2(c), the convergence of certain asynchronous algorithms is destroyed (Lubachevsky and Mitra, 1986; Bertsekas and Tsitsiklis, 1989b, p. 506 and p. 517). Parts (a) and (b) of Assumption 2 are typically satisfied in practice.

Asynchronous algorithms can exhibit three different types of behavior (other than guaranteed divergence):

(a) Convergence under total asynchronism.
(b) Convergence under partial asynchronism, for every value of B, but possible divergence under totally asynchronous execution.
(c) Convergence under partial asynchronism if B is small enough, and possible divergence if B is large enough.

The mechanisms by which convergence is established in each one of the above three cases are fundamentally different and we address them in the subsequent three sections, respectively.

6. TOTALLY ASYNCHRONOUS ALGORITHMS

Totally asynchronous convergence results have been obtained† by Chazan and Miranker (1969) for linear iterations, Miellou (1975a), Baudet (1978), El Tarazi (1982), Miellou and Spiteri (1985) for contracting iterations, Miellou (1975b) and Bertsekas (1982) for monotone iterations, and Bertsekas (1983) for general iterations. Related results can be also found in Uresin and Dubois (1986, 1988, 1990). The following general result is from Bertsekas (1983).

Proposition 3. Let $X = \prod_{i=1}^{p} X_i \subset \prod_{i=1}^{p} \Re^{n_i}$. Suppose that for each $i \in \{1, \ldots, p\}$, there exists a sequence $\{X_i(k)\}$ of subsets of X_i such that:

(a) $X_i(k+1) \subset X_i(k)$, for all $k \geq 0$.
(b) The sets $X(k) = \prod_{i=1}^{p} X_i(k)$ have the property $f(x) \in X(k+1)$, for all $x \in X$.
(c) Every limit point of a sequence $\{x(k)\}$ with the property $x(k) \in X(k)$ for all k, is a fixed point of f.

Then, under Assumption 1 (total asynchronism), and if $x(0) \in X(0)$, every limit point of the sequence $\{x(t)\}$ generated by the asynchronous iteration (5.1)–(5.2) is a fixed point of f.

Proof. We show by induction that for each $k \geq 0$, there is a time t_k such that:

(a) $x(t) \in X(k)$ for all $t \geq t_k$.
(b) For all i and $t \in T^i$ with $t \geq t_k$, we have $x^i(t) \in X(k)$, where

$$x^i(t) = (x_1(\tau_1^i(t)), x_2(\tau_2^i(t)), \ldots, x_n(\tau_n^i(t))), \quad \forall t \in T^i.$$

[In words: after some time, all solution estimates will be in $X(k)$ and all estimates used in iteration (5.1) will come from $X(k)$.]

† Actually, some of these papers only consider partially asynchronous iterations, but their convergence results readily extend to cover the case of total asynchronism.

The induction hypothesis is true for $k = 0$, since the initial estimate is assumed to be in $X(0)$. Assuming it is true for a given k, we will show that there exists a time t_{k+1} with the required properties. For each $i = 1, \ldots, n$, let t^i be the first element of T^i such that $t^i \geq t_k$. Then by condition (b) in the statement of the proposition, we have $f(x^i(t^i)) \in X(k+1)$ and

$$x_i(t^i + 1) = f_i(x^i(t^i)) \in X_i(k+1).$$

Similarly, for every $t \in T^i$, $t \geq t^i$, we have $x_i(t+1) \in X_i(k+1)$. Between elements of T^i, $x_i(t)$ does not change. Thus,

$$x_i(t) \in X_i(k+1), \quad \forall t \geq t^i + 1.$$

Let $t_k' = \max_i \{t^i\} + 1$. Then, using the Cartesian product structure of $X(k)$ we have

$$x(t) \in X(k+1), \quad \forall t \geq t_k'.$$

Finally, since by Assumption 1, we have $\tau_j^i(t) \to \infty$ as $t \to \infty$, $t \in T^i$, we can choose a time $t_{k+1} \geq t_k'$ that is sufficiently large so that $\tau_j^i(t) \geq t_k'$ for all i, j and $t \in T^i$ with $t \geq t_{k+1}$. We then have, $x_j(\tau_j^i(t)) \in X_j(k+1)$, for all $t \in T^i$ with $t \geq t_{k+1}$ and all $j = 1, \ldots, n$, which implies that

$$x^i(t) = (x_1(\tau_1^i(t)), x_2(\tau_2^i(t)), \ldots, x_n(\tau_n^i(t))) \in X(k+1).$$

The induction is complete. Q.E.D.

The key idea behind Proposition 3 is that eventually $x(t)$ enters and stays in the set $X(k)$; furthermore, due to condition (b) in Proposition 3, it eventually moves into the next set $X(k+1)$. The most restrictive assumption in the proposition is the requirement that each $X(k)$ is the Cartesian product of sets $X_i(k)$. Successful application of Proposition 3 depends on the ability to properly define the sets $X_i(k)$ with the required properties. This is possible for two general classes of iterations which will be discussed shortly.

Notice that Proposition 3 makes no assumptions on the nature of the sets $X_i(k)$. For this reason, it can be applied to problems involving continuous variables, as well as discrete iterations involving finite-valued variables. Furthermore, the result extends in the obvious way to the case where each $X_i(k)$ is a subset of an infinite-dimensional space (instead of being a subset of $\Re^{n_i}$) or to the case where f has multiple fixed points.

Interestingly enough, the sufficient conditions for asynchronous convergence provided by Proposition 3, are also known to be necessary for two special cases: (i) if $n_i = 1$ for each i and the mapping f is linear (Chazan and Miranker, 1969), and (ii) if the set X is finite (Uresin and Dubois, 1990).

Several authors have also studied asynchronous iterations with zero delays, that is, under the assumption $\tau_j^i(t) = t$ for every $t \in T^i$: see for example Robert *et al.* (1975); Robert (1976, 1987, 1988). Note that this is a special case of our asynchronous model, but is more general than the synchronous Jacobi and Gauss-Seidel iterations of Section 2, because the sets T^i are allowed to be arbitrary. General necessary and sufficient convergence conditions for the zero-delay case can be found in Tsitsiklis (1987) where it is shown that asynchronous convergence is guaranteed if and only if there exists a Lyapunov-type function which testifies to this.

6.1. *Maximum norm contractions*

Consider a norm on $\Re^n$ defined by

$$\|x\| = \max_i \frac{\|x_i\|_i}{w_i}, \qquad (6.1)$$

where $x_i \in \Re^{n_i}$ is the ith component of x, $\|\cdot\|_i$ is a norm on $\Re^{n_i}$, and w_i is a positive scalar, for each i. Suppose that f has the following contraction property: there exists some $\alpha \in [0, 1)$ such that

$$\|f(x) - x^*\| \le \alpha\,\|x - x^*\|, \quad \forall x \in X, \qquad (6.2)$$

where x^* is a fixed point of f. Given a vector $x(0) \in X$ with which the algorithm is initialized, let

$$X_i(k) = \{x_i \in \Re^{n_i} \mid \|x_i - x_i^*\|_i \le \alpha^k\,\|x(0) - x^*\|\}.$$

It is easily verified that these sets satisfy the conditions of Proposition 3 and convergence to x^* follows.

Iteration mappings f with the contraction property (6.2) are very common. We list a few examples:

(a) Linear iterations of the form $f(x) = Ax + b$, where A is an $n \times n$ matrix such that $\rho(|A|) < 1$. Here, $|A|$ is the matrix whose entries are the absolute values of the corresponding entries of A, and $\rho(|A|)$, the *spectral radius* of $|A|$, is the largest of the magnitudes of the eigenvalues of $|A|$ (Chazan and Miranker, 1969). This result follows from a corollary of the Perron–Frobenius theorem that states that $\rho(|A|) < 1$ if and only if A is a contraction mapping with respect to a weighted maximum norm of the form (6.1), for a suitable choice of the weights. As a special case, we obtain totally asynchronous convergence of the iteration $\pi := \pi P$ for computing a row vector π consisting of the invariant probabilities of an irreducible, discrete-time, finite-state, Markov chain. Here, P is the transition probability matrix of the chain and one of the components of π is held fixed throughout the algorithm (Bertsekas and Tsitsiklis, 1989b). Another special case, the case of periodic asynchronous iterations, is considered in Donnelly (1971). Let us mention here that the condition $\rho(|A|) < 1$ is not only sufficient but also necessary for totally asynchronous convergence (Chazan and Miranker, 1969).

(b) Gradient iterations of the form $f(x) = x - \gamma\nabla F(x)$, where γ is a small positive stepsize parameter, $F: \Re^n \mapsto \Re$ is a twice continuously differentiable cost function whose Hessian matrix is bounded and satisfies the diagonal dominance condition

$$\sum_{j \ne i} |\nabla_{ij}^2 F(x)| \le \nabla_{ii}^2 F(x) - \beta, \quad \forall i, \forall x \in X. \qquad (6.3)$$

Here, β is a positive constant and $\nabla_{ij}^2 F$ stands for $(\partial^2 F)/(\partial x_i\, \partial x_j)$ (Bertsekas, 1983; Bertsekas and Tsitsiklis, 1989b).

Example 1. Consider the iteration $x := x - \gamma Ax$, where A is the positive definite matrix given by

$$A = \begin{bmatrix} 1+\epsilon & 1 & 1 \\ 1 & 1+\epsilon & 1 \\ 1 & 1 & 1+\epsilon \end{bmatrix},$$

and γ, ϵ are positive constants. This iteration can be viewed as the gradient iteration $x := x - \gamma\nabla F(x)$ for minimizing the quadratic function $F(x) = \frac{1}{2}x'Ax$ and is known to converge synchronously if the stepsize γ is sufficiently small. If $\epsilon > 1$, then the diagonal dominance condition of (6.3) holds and totally asynchronous convergence follows, when the stepsize γ is sufficiently small. On the other hand, when $0 < \epsilon < 1$, the condition of (6.3) fails to hold for all $\gamma > 0$. In fact, in that case, it is easily shown that $\rho(|I - \gamma A|) > 1$ for every $\gamma > 0$, and totally asynchronous convergence fails to hold, according to the necessary conditions quoted earlier. An illustrative sequence of events under which the algorithm diverges is the following. Suppose that the processors start with a common vector $x(0) = (c, c, c)$ and that each processor executes a very large number t_0 of updates of its own component without informing the others. Then, in effect, processor 1 solves the equation $0 = (\partial F/\partial x_1)(x_1, c, c) = (1+\epsilon)x_1 + c + c$, to obtain $x_1(t_0) \approx -2c/(1+\epsilon)$, and the same conclusion is obtained for the other processors as well. Assume now that the processors exchange their results at time t_0 and repeat the above described scenario. We will then obtain $x_i(2t_0) \approx -2x_i(t_0)/(1+\epsilon) \approx (-2)^2c/(1+\epsilon)^2$. Such a sequence of events can be repeated *ad infinitum*, and it is clear that the vector $x(t)$ will diverge if $\epsilon < 1$.

(c) The projection algorithm (as well as several other algorithms) for variational inequalities. Here, $X = \prod_{i=1}^{p} X_i \subset \Re^n$ is a closed convex set, $f: X \mapsto \Re^n$ is a given function, and we are looking for a vector $x^* \in X$ such that

$$(x - x^*)'f(x^*) \ge 0, \quad \forall x \in X.$$

The projection algorithm is given by $x := [x - \gamma f(x)]^+$, where $[\cdot]^+$ denotes orthogonal projection on the set X. Totally asynchronous convergence to x^* is obtained under the assumption that the mapping $x \mapsto x - \gamma f(x)$ is a maximum norm contraction mapping, and this is always the case if the Jacobian of f satisfies a diagonal dominance condition (Bertsekas and Tsitsiklis, 1989b). Special cases of variational inequalities include constrained convex optimization, solution of systems of nonlinear equations, traffic equilibrium problems under a user-optimization principle, and Nash games. Let us point out here that an asynchronous algorithm for solving a traffic equilibrium problem can be viewed as a model of a traffic network in operation whereby individual users optimize their individual routes given the current condition of the network. It is natural to assume that such user-optimization takes place asynchronously. Similarly, in a game theoretic context, we can think of a set of players who asynchronously adapt their strategies so as to improve their individual payoffs, and an asynchronous iteration can be used as a model of such a situation.

(d) Waveform relaxation methods for solving a system or ordinary differential equations under a weak coupling assumption (Mitra, 1987), as well as for two-point boundary value problems (Lang *et al.*, 1986; Spiteri, 1984; Bertsekas and Tsitsiklis, 1989b).

Other studies have dealt with an asynchronous Newton algorithm (Bojanczyk, 1984), an agreement problem (Li and Basar, 1987), diagonally dominant linear programming problems (Tseng, 1990), and a variety of infinite-dimensional problems such as partial differential equations, and variational inequalities (Spiteri, 1984, 1986; Miellou and Spiteri, 1985; Anwar and El Tarazi, 1985).

In the case of maximum norm contraction mappings, there are some convergence rate estimates available which indicate that the asynchronous iteration converges faster than its synchronous counterpart, especially if the coupling between the different components of x is relatively weak. Let us suppose that an update by a processor takes one time unit and that the communication delays are always equal to D time units, where D is a positive integer. With a synchronous algorithm, there is one iteration every $D+1$ time units and the "error" $\|x(t)-x^*\|$ can be bounded by $C\alpha^{t/(D+1)}$, where C is some constant [depending on $x(0)$] and α is the contraction factor of (6.2). We now consider an asynchronous execution whereby, at each time step, an iteration is performed by each processor i and the result is immediately transmitted to the other processors. Thus the values of x_j $(j \neq i)$ which are used by processor i are always outdated by D time units. Concerning the function f, we assume that there exists some scalar β such that $0<\beta<\alpha$ and

$$\|f_i(x)-x_i^*\|_i \leq \max\left\{\alpha\,\|x_i-x_i^*\|_i,\ \beta\max_{j\neq i}\|x_j-x_j^*\|_j\right\}, \quad \forall i. \quad (6.4)$$

It is seen that a small value of β corresponds to a situation where the coupling between different components of x is weak. Under condition (6.4), the convergence rate estimate for the synchronous iteration cannot be improved, but the error $\|x(t)-x^*\|$ for the asynchronous iteration can be shown (Bertsekas and Tsitsiklis, 1989b) to be bounded above by $C\rho^t$, where C is some constant and ρ is the positive solution of the equation $\rho = \max\{\alpha, \beta\rho^{-D}\}$. It is not hard to see that $\rho < \alpha^{1/(D+1)}$ and the asynchronous algorithm converges faster. The advantage of the asynchronous algorithm is more pronounced when β is very small (very weak coupling) in which case ρ approaches α. The latter is the convergence rate that would have been obtained if there were no communication delays at all. We conclude that, for weakly coupled problems, asynchronous iterations are slowed down very little by communication delays, in sharp contrast with their synchronous counterparts.

6.2. *Monotone mappings*

Consider a function $f:\mathfrak{R}^n \mapsto \mathfrak{R}^n$ which is continuous, monotone [that is, if $x \leq y$ then $f(x) \leq f(y)$], and has a unique fixed point x^*. Furthermore, assume that there exist vectors u, v, such that $u \leq f(u) \leq f(v) \leq v$. If we let f^k be the composition of k copies of f and $X(k) = \{x \mid f^k(u) \leq x \leq f^k(v)\}$, then Proposition 3 applies and establishes totally asynchronous convergence. The above stated conditions on f are satisfied by the iteration mapping corresponding to the successive approximation (value iteration) algorithm for discounted and certain undiscounted infinite horizon dynamic programming problems (Bertsekas, 1982).

An important special case is the asynchronous Bellman–Ford algorithm for the shortest path problem. Here we are given a directed graph $G=(N,A)$, with $N=\{1,\ldots,n\}$ and for each arc $(i,j)\in A$, a weight a_{ij} representing its length. The problem is to compute the shortest distance x_i from every node i to node 1. We assume that every cycle not containing node 1 has positive length and that there exists at least one path from every node to node 1. Then, the shortest distances correspond to the unique fixed point of the monotone mapping $f:\mathfrak{R}^n \mapsto \mathfrak{R}^n$ defined by $f_1(x)=0$ and

$$f_i(x) = \min_{\{j \mid (i,j)\in A\}} (a_{ij}+x_j), \quad i \neq 1.$$

The Bellman–Ford algorithm consists of the iteration $x := f(x)$ and can be shown to converge asynchronously (Tajibnapis, 1977; Bertsekas, 1982). We now compare the synchronous and the asynchronous versions. We assume that both versions are initialized with $x_i = \infty$ for every $i \neq 1$, which is the most common choice. The synchronous iteration is known to converge after at most n iterations. However, assuming that the communication delays from processor i to j are fixed to some constant D_{ij}, and that the computation time is negligible, it is easily shown that the asynchronous iteration is guaranteed to terminate earlier than the synchronous one.

Notice that the number of messages exchanged in the synchronous Bellman–Ford algorithm is at most n^3. This is because there are at most n stages and at most n messages are transmitted by each processor at each stage. Interestingly enough, with an asynchronous execution, and if the communication delays are allowed to be arbitrary, some simple examples (due to E. M. Gafni and R. G. Gallager; see Bertsekas and Tsitsiklis, 1989b) show that the number of messages exchanged until termination could be exponential in n, even if we restrict processor i to transmit a message only when the value of x_i changes. This could be a serious drawback but experience with the algorithm indicates that this worst case behavior rarely occurs and that the average number of messages exchanged is polynomial in n. It also turns out that the expected number of messages is polynomial in n under some reasonable probabilistic assumptions on the execution of the algorithm (Tsitsiklis and Stamoulis, 1990).

A number of asynchronous convergence results making essential use of monotonicity conditions are also available for relaxation and primal–dual algorithms for linear and nonlinear network flow problems (Bertsekas, 1986; Bertsekas and Eckstein, 1987, 1988; Bertsekas and El Baz, 1987; Bertsekas and Castanon, 1989, 1990). Experiments showing faster convergence for asynchronous over synchronous relaxation methods for assignment problems using a shared memory machine are given in Bertsekas and Castonon (1989).

We finally note that, under the monotonicity assumptions of this subsection, the convergence rate of an asynchronous iteration is guaranteed to be at least as good as the convergence rate of a corresponding synchronous iteration, under a fair comparison (Bertsekas and Tsitsiklis, 1989a).

7. PARTIALLY ASYNCHRONOUS ALGORITHMS—I

We now consider iterations satisfying the partial asynchronism Assumption 2. Since old information is "purged" from the algorithm after at most B units, it is natural to describe the "state" of the algorithm at time t by the vector $z(t) \in X^B$ defined by

$$z(t) = (x(t), x(t-1), \ldots, x(t-B+1)).$$

We then notice that $x(t+1)$ can be determined [cf. (5.1)–(5.3)] in terms of $z(t)$; in particular, knowledge of $x(\tau)$, for $\tau \le t-B$ is not needed. We assume that the iteration mapping f is continuous and has a nonempty set $X^* \subset X$ of fixed points. Let Z^* be the set of all vectors $z^* \in X^B$ of the form $z^* = (x^*, x^*, \ldots, x^*)$, where x^* belongs to X^*. We present a sometimes useful convergence result, which employs a Lyapunov-type function d defined on the set X^B.

Proposition 4. (Bertsekas and Tsitsiklis, 1989b) Suppose that there exist a positive integer t^* and a continuous function $d: X^B \mapsto [0, \infty)$ with the following properties: For every initialization $z(0) \notin Z^*$ of the iteration and any subsequent sequence of events (conforming to Assumption 2) we have $d(z(t^*)) < d(z(0))$ and $d(z(1)) \le d(z(0))$. Then every limit point of a sequence $\{z(t)\}$ generated by the partially asynchronous iteration (5.1)–(5.2) belongs to Z^*. Furthermore, if $X = \Re^n$, if the function d is of the form $d(z) = \inf_{z^* \in Z^*} \|z - z^*\|$, where $\|\cdot\|$ is some vector norm, and if the function f is of the form $f(x) = Ax + b$, where A is a $n \times n$ matrix and b is a vector in $\Re^n$, then $d(z(t))$ converges to zero at the rate of a geometric progression.

For an interesting application of the above proposition, consider a mapping $f: \Re^n \mapsto \Re^n$ of the form $f(x) = Ax$ where A is an irreducible stochastic matrix, and let $n_i = 1$ for each i. In the corresponding iterative algorithm, each processor maintains and communicates a value of a scalar variable x_i and once in a while forms a convex combination of its own variable with the variables received from other processors according to the rule

$$x_i := \sum_{j=1}^{n} a_{ij} x_j.$$

Clearly, if the algorithm converges then, in the limit, the values possessed by different processors are equal. We will thus refer to the asynchronous iteration $x := Ax$ as an *agreement* algorithm. It can be shown that, under the assumption of partial asynchronism, the function d defined by

$$d(z(t)) = \max_i \max_{t-B<\tau\le t} x_i(\tau) - \min_i \min_{t-B<\tau\le t} x_i(\tau) \quad (7.1)$$

has the properties assumed in Proposition 4, provided that at least one of the diagonal entries of A is positive. In particular, if the processors initially disagree, the "maximum disagreement" [cf. (7.1)] is reduced by a positive amount after at most $2nB$ time units (Tsitsiklis, 1984). Proposition 4 applies and establishes geometric convergence to agreement. Furthermore, such partially asynchronous convergence is obtained no matter how big the value of the asynchronism measure B is, as long as B is finite.

The following example (Bertsekas and Tsitsiklis, 1989b) shows that the agreement algorithm need not converge totally asynchronously.

Example 2. Suppose that

$$A = \begin{bmatrix} 1/2 & 1/2 \\ 1/2 & 1/2 \end{bmatrix}.$$

Here, the synchronous iteration $x(t+1) = Ax(t)$ converges in a single step to the vector $x = (y, y)$, where $y = (x_1 + x_2)/2$. Consider the following totally asynchronous scenario. Each processor updates its value at each time step. At certain times $t_1, t_2, \ldots$, each processor transmits its value which is received with zero delay and is immediately incorporated into the computations of the other processor. We then have

$$x_1(t+1) = \frac{x_1(t)}{2} + \frac{x_2(t_k)}{2}, \quad t_k \le t < t_{k+1},$$

$$x_2(t+1) = \frac{x_1(t_k)}{2} + \frac{x_2(t)}{2}, \quad t_k \le t < t_{k+1}.$$

(See Fig. 8 for an illustration.) Thus,

$$x_1(t_{k+1}) = (1/2)^{t_{k+1}-t_k} x_1(t_k) + (1 - (1/2)^{t_{k+1}-t_k}) x_2(t_k),$$

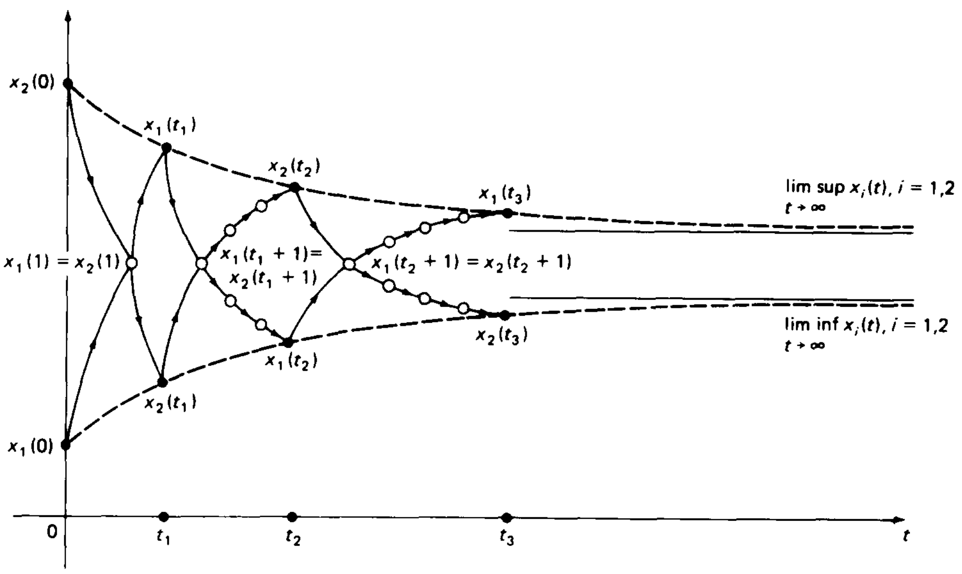
$$x_2(t_{k+1}) = (1/2)^{t_{k+1}-t_k} x_2(t_k) + (1 - (1/2)^{t_{k+1}-t_k}) x_1(t_k).$$

Subtracting these two equations we obtain

$$|x_2(t_{k+1}) - x_1(t_{k+1})| = (1 - 2(1/2)^{t_{k+1}-t_k})\, |x_2(t_k) - x_1(t_k)|$$
$$= (1 - \epsilon_k)\, |x_2(t_k) - x_1(t_x)|,$$

where $\epsilon_k = 2(1/2)^{t_{k+1}-t_k}$. In particular, the disagreement $|x_2(t_k) - x_1(t_k)|$ keeps decreasing. On the other hand, convergence to agreement is not guaranteed unless $\prod_{k=1}^{\infty} (1 - \epsilon_k) = 0$ which is not necessarily the case. For example, if we choose the differences $t_{k+1} - t_k$ to be large enough so that $\epsilon_k < k^{-2}$, then we can use the fact $\prod_{k=1}^{\infty} (1 - k^{-2}) > 0$ to see that convergence to agreement does not take place.

Example 2 shows that failure to converge is possible if part (b) of the partial asynchronism Assumption 2



$x_2(0)$
$x_1(t_1)$
$x_2(t_2)$
$x_1(t_3)$
$\limsup_{t\to\infty} x_i(t), i = 1,2$
$x_1(1) = x_2(1)$
$x_1(t_1+1) = x_2(t_1+1)$
$x_1(t_2+1) = x_2(t_2+1)$
$x_2(t_3)$
$\liminf_{t\to\infty} x_i(t), i = 1,2$
$x_1(t_2)$
$x_2(t_1)$
$x_1(0)$
0
t_1
t_2
t_3
t

FIG. 8. Illustration of divergence in Example 2.

fails to hold. There also exist examples demonstrating that parts (a) and (c) of Assumption 2 are also necessary for convergence.

Example 2 illustrates best the convergence mechanism in algorithms which converge partially asynchronously for every B, but not totally asynchronously. The key idea is that the distance from the set of fixed points is guaranteed to "contract" once in a while. However, the contraction factor depends on B and approaches 1 as B gets larger. (In the context of Example 2, the contraction factor is $1 - \epsilon_k$ which approaches 1 as $t_{k+1} - t_k$ is increased to infinity.) As time goes to infinity, the distance from the set of fixed points is contracted an infinite number of times but this guarantees convergence only if the contraction factor is bounded away from 1, which then necessitates a finite but otherwise arbitrary bound on B.

Partially asynchronous convergence for every value of B has been established for several variations and generalizations of the agreement algorithm (Tsitsiklis, 1984; Bertsekas and Tsitsiklis, 1989b), as well as for a variety of other problems:

(a) The iteration $\pi := \pi P$ for the computation of a row vector π of invariant probabilities, associated with an irreducible stochastic matrix P with a nonzero diagonal entry (Lubachevsky and Mitra, 1986). This result can be also obtained by letting $x_i = \pi_i/\pi_i^*$, where π^* is a positive vector satisfying $\pi^* = \pi P$, and by verifying that the variables x_i obey the equations of the agreement algorithm (Bertsekas and Tsitsiklis, 1989b).

(b) Relaxation algorithms involving nonexpansive mappings with respect to the maximum norm (Tseng *et al.*, 1990; Bertsekas and Tsitsiklis, 1989b). Special cases include dual relaxation algorithms for strictly convex network flow problems and linear iterations for the solution of linear equations of the form $Ax = b$, where A is an irreducible matrix satisfying the weak diagonal dominance condition $\sum_{j\neq i} |a_{ij}| \le a_{ii}$, for all i.

(c) An asynchronous algorithm for load balancing in a computer network whereby highly loaded processors transfer fractions of their load to their lightly loaded neighbors, until the load of all processors becomes the same (Bertsekas and Tsitsiklis, 1989b).

In all of the above cases, partially asynchronous convergence has been proved for all values of B, and examples are available which demonstrate that totally asynchronous convergence fails.

We close by mentioning a particular context in which the agreement algorithm could be of use. Consider a set of processors who obtain a sequence of noisy observations and try to estimate certain parameters by means of some iterative method. This could be a stochastic gradient algorithm (such as the ones arising in recursive system identification) or some kind of a Monte Carlo estimation algorithm. All processors are employed for the estimation of the same parameters but their individual estimates are generally different because the noises corrupting their observations can be different. We let the processors communicate and combine their individual estimates in order to average their individual noises, thereby reducing the error variance. We thus let the processors execute the agreement algorithm, trying to agree on a common estimate, while simultaneously obtaining new observations which they incorporate into their estimates. There are two opposing effects here: the agreement algorithm tends to bring their estimates closer together, while new observations have the potential of increasing the difference of their estimates. Under the partial asynchronism assumption, the agreement algorithm tends to converge geometrically. On the other hand, in several stochastic algorithms (such as the stochastic approximation iteration

$$x := x - \frac{1}{t}(\nabla F(x) + w),$$

where w represents observation noise) the stepsize $1/t$

decreases to zero as time goes to infinity. We then have, asymptotically, a separation of time scales: the stochastic algorithm operates on a slower time scale and therefore the agreement algorithm can be approximated by an algorithm in which agreement is instantly established. It follows that the asynchronous nature of the agreement algorithm cannot have any adverse effect on the convergence of the stochastic algorithm. Rigorous results of this type can be found in Tsitsiklis (1984); Tsitsiklis *et al.* (1986); Kushner and Yin (1987a, b); Bertsekas and Tsitsiklis (1989b).

8. PARTIALLY ASYNCHRONOUS ALGORITHMS—II

We now turn to the study of partially asynchronous iterations that converge only when the stepsize is small. We illustrate the behavior of such algorithms in terms of a prototypical example.

Let A be an $n \times n$ positive definite symmetric matrix and let b be a vector in $\Re^n$. We consider the asynchronous iteration $x := x - \gamma(Ax - b)$, where γ is a small positive stepsize. We define a cost function $F: \Re^n \mapsto \Re$ by $F(x) = \frac{1}{2}x'Ax - x'b$, and our iteration is equivalent to the gradient algorithm $x := x - \gamma \nabla F(x)$ for minimizing F. This algorithm is known to converge synchronously provided that γ is chosen small enough. On the other hand, it was shown in Example 1 that the gradient algorithm does not converge totally asynchronously. Furthermore, a careful examination of the argument in that example reveals that for every value of γ there exists a B large enough such that the partially asynchronous gradient algorithm does not converge. Nevertheless, if γ is fixed to a small value, and if B is not excessively large (we roughly need $B \leq C/\gamma$, where C is some constant determined by the structure of the matrix A), then the partially asynchronous iteration turns out to be convergent. An equivalent statement is that for every value of B there exists some $\gamma_0 > 0$ such that if $0 < \gamma < \gamma_0$ then the partially asynchronous algorithm converges (Tsitsiklis *et al.*, 1986; Bertsekas and Tsitsiklis, 1989b). The rationale behind such a result is the following. If the information available to processor i on the value of x_j is outdated by at most B time units, then the difference between the value $x_i(\tau_j^i(t))$ possessed by processor i and the true value $x_j(t)$ is of the order of γB, because each step taken by processor j is of the order of γ. It follows that for γ very small the errors caused by asynchronism become negligible and cannot destroy the convergence of the algorithm.

The above mentioned convergence result can be extended to more general gradient-like algorithms for nonquadratic cost functions F. One only needs to assume that the iteration is of the form $x := x - \gamma s(x)$, where $s(x)$ is an update direction with the property $s_i(x) \nabla_i F(x) \geq K |\nabla_i F(x)|^2$, where K is a positive constant, together with a Lipschitz continuity condition on ∇F, and a boundedness assumption of the form $\|s(x)\| \leq L \|\nabla F(x)\|$ (Tsitsiklis *et al.*, 1986; Bertsekas and Tsitsiklis, 1989b). Similar conclusions are obtained for gradient projection iterations for constrained convex optimization (Bertsekas and Tsitsiklis, 1989b).

An important application of asynchronous gradient-like optimization algorithms arises in the context of optimal quasistatic routing in data networks. In a common formulation of the routing problem one is faced with a convex nonlinear multicommodity network flow problem (Bertsekas and Gallager, 1987) that can be solved using gradient projection methods. It has been shown that these methods also converge partially asynchronously, provided that a small enough stepsize is used (Tsitsiklis and Bertsekas, 1986). Furthermore, such methods can be naturally implemented on-line by having the processors in the network asynchronously exchange information on the current traffic conditions in the system and perform updates trying to reduce the measure of congestion being optimized. An important property of such an asynchronous algorithm is that it adapts to changes in the problem being solved (such as changes on the amount of traffic to be routed through the network) without a need for aborting and restarting the algorithm. Some further analysis of the asynchronous routing algorithm can be found in Tsai (1986, 1989) and Tsai *et al.* (1986).

9. TERMINATION OF ASYNCHRONOUS ITERATIONS

In practice, iterative algorithms are executed only for a finite number of iterations, until some termination condition is satisfied. In the case of asynchronous iterations, the problem of determining whether termination conditions are satisfied is rather difficult because each processor possesses only partial information on the progress of the algorithm.

We now introduce one possible approach for handling the termination problem for asynchronous iterations. In this approach, the problem is decomposed into two parts:

(a) An asynchronous iterative algorithm is modified so that it terminates in finite time.
(b) A special procedure is used to detect termination in finite time after it has occured.

In order to handle the termination problem, we have to be a little more specific about the model of interprocessor communication. While the general model of asynchronous iterations introduced in Section 5 can be used for both shared memory and message-passing parallel architectures, we adopt here a more explicit message-passing model. In particular, we assume that each processor j sends messages with the value of x_j to every other processor i. Processor i keeps a buffer with the most recently received value of x_j. We denote the value in this buffer at time t by $x_j^i(t)$. This value was transmitted by processor j at some earlier time $\tau_j^i(t)$ and therefore $x_j^i(t) = x_j(\tau_j^i(t))$. We also assume the following:

Assumption 3. (a) If $t \in T^i$ and $x_i(t+1) \neq x_i(t)$, then processor i will eventually send a message to every other processor.

(b) If a processor i has sent a message with the value of $x_i(t)$ to some other processor j, then processor i will send a new message to processor j only after the value of x_i changes (due to an update by processor i).

(c) Messages are received in the order that they are transmitted.

(d) Each processor sends at least one message to every other processor.

Assumption 3(d) is only needed to get the algorithm started. Assumption 3(b) is crucial and has the following consequences. If the value of $x(t)$ settles to some final value, then there will be some time t^* after which no messages will be sent. Furthermore, all messages transmitted before t^* will eventually reach their destinations and the algorithm will eventually reach a quiescent state where none of the variables x_i changes and no message is in transit. We can then say that the algorithm has terminated.

More formally, we view termination as equivalent to the following two properties:

(i) No message is in transit.
(ii) An update by some processor i causes no change in the value of x_i.

Property (ii) is a collection of local termination conditions. There are several algorithms for termination detection when a termination condition can be decomposed as above (Dijkstra and Scholten, 1980; Bertsekas and Tsitsiklis, 1989b). Thus termination detection causes no essential difficulties, under the assumption that the asynchronous algorithm terminates in finite time.

We now turn to the more difficult problem of converting a convergent asynchronous iterative algorithm into a finitely terminating one. If we were dealing with the synchronous iteration $x(t+1) = f(x(t))$, it would be natural to terminate the algorithm when the condition $\|x(t+1) - x(t)\| \le \epsilon$ is satisfied, where ϵ is a small positive constant reflecting the desired accuracy of solution, and where $\|\cdot\|$ is a suitable norm. This suggests the following approach for the context of asynchronous iterations. Given the iteration mapping f and the accuracy parameter ϵ, we define a new iteration mapping $g : X \mapsto X$ by letting

$$g_i(x) = \begin{cases} f_i(x) & \text{if } \|f_i(x) - x_i\| \ge \epsilon, \\ x_i, & \text{otherwise.} \end{cases}$$

We will henceforth assume that the processors are executing the asynchronous iteration $x := g(x)$. The key question is whether this new iteration is guaranteed to terminate in finite time. One could argue as follows. Assuming that the original iteration $x := f(x)$ is guaranteed to converge, the changes in the vector x will eventually become arbitrarily small, in which case we will have $g(x) = x$ and the iteration $x := g(x)$ will terminate. Unfortunately, this argument is fallacious, as demonstrated by the following example.

Example 3. Consider the function $f := \Re^2 \mapsto \Re^2$ defined by

$$f_1(x) = \begin{cases} -x_1, & \text{if } x_2 \ge \epsilon/2, \\ 0, & \text{if } x_2 < \epsilon/2. \end{cases}$$

$$f_2(x) = x_2/2.$$

It is clear that the asynchronous iteration $x := f(x)$ converges to $x^* = (0, 0)$: in particular, x_2 is updated according to $x_2 := x_2/2$ and tends to zero; thus, it eventually becomes smaller than $\epsilon/2$. Eventually processor 1 receives a value of x_2 smaller than $\epsilon/2$ and a subsequent update by the same processor sets x_1 to zero.

Let us now consider the iteration $x := g(x)$. If the algorithm is initialized with x_2 between $\epsilon/2$ and ϵ, then the value of x_2 will never change, and processor 1 will keep executing the nonconvergent iteration $x_1 := -x_1$. Thus, the asynchronous iteration $x := g(x)$ is not guaranteed to terminate.

The remainder of this section is devoted to the derivation of conditions under which the iteration $x := g(x)$ is guaranteed to terminate. We introduce some notation. Let I be a subset of the set $\{1, \ldots, p\}$ of all processors. For each $i \in I$, let there be given some value $\theta_i \in X_i$. We consider the asynchronous iteration $x := f^{I,\theta}(x)$, which is the same as the iteration $x := f(x)$ except that any component x_i, with $i \in I$, is set to the value θ_i. Formally, the mapping $f^{I,\theta}$ is defined by letting $f_i^{I,\theta}(x) = f_i(x)$, if $i \notin I$, and $f_i^{I,\theta}(x) = \theta_i$, if $i \in I$.

Proposition 5. (Bertsekas and Tsitsiklis, 1989a) Let Assumption 3 hold. Suppose that for any $I \subset \{1, \ldots, p\}$ and for any choice of $\theta_i \in X_i$, $i \in I$, the asynchronous iteration $x := f^{I,\theta}(x)$ is guaranteed to converge. Then, the asynchronous iteration $x := g(x)$ terminates in finite time.

Proof. Consider the asynchronous iteration $x := g(x)$. Let I be the set of all indices i for which the variable $x_i(t)$ changes only a finite number of times. For each $i \in I$, let θ_i be the limiting value of $x_i(t)$. Since f maps X into itself, so does g. It follows that $\theta_i \in X_i$ for each i. For each $i \in I$, processor i sends a positive but finite number of messages [Assumptions 3(d) and (b)]. By Assumption 3(a), the last message sent by processor i carries the value θ_i and by Assumption 3(c) this is also the last message received by any other processor. Thus, for all t large enough, and for all j, we will have $x_i^j(t) = x_i(\tau_i^j(t)) = \theta_i$. Thus, the iteration $x := g(x)$ eventually becomes identical with the iteration $x := f^{I,\theta}(x)$ and therefore converges. This implies that the difference $x_i(t+1) - x_i(t)$ converges to zero for any $i \notin I$. On the other hand, because of the definition of the mapping g, the difference $x_i(t+1) - x_i(t)$ is either zero, or its magnitude is bounded below by $\epsilon > 0$. It follows that $x_i(t+1) - x_i(t)$ eventually settles to zero, for every $i \notin I$. This shows that $i \in I$ for every $i \notin I$; we thus obtain a contradiction unless $I = \{1, \ldots, p\}$, which proves the desired result. Q.E.D.

We now identify certain cases in which the main assumption in Proposition 5 is guaranteed to hold. We consider first the case of monotone iterations and we assume that the iteration mapping f has the properties introduced in Section 6.2. For any I and $\{\theta_i \mid i \in I\}$, the mapping $f^{I,\theta}$ inherits all of the properties of f, except that $f^{I,\theta}$ is not guaranteed to have a unique fixed point. If this latter property can be independently verified, then the asynchronous iteration $x := f^{I,\theta}(x)$ is guaranteed to converge, and Proposition 3 applies. Let us simply say here that this property can be indeed verified for several interesting problems.

Let us now consider the case where f satisfies the contraction condition $\|f(x) - x^*\| \le \alpha \|x - x^*\|$ of (6.2). Unfortunately, it is not necessarily true that the mappings $f^{I,\theta}$ also satisfy the same contraction condition. In fact, the mappings $f^{I,\theta}$ are not even guaranteed to have a fixed point. Let us strengthen the contraction condition of (6.2) and assume that

$$\|f(x) - f(y)\| \le \alpha \|x - y\|, \quad \forall x, y \in \mathcal{R}^n, \qquad (9.1)$$

where $\|\cdot\|$ is the weighted maximum norm of (6.1) and $\alpha \in [0, 1)$. We have $f^{I,\theta}(x) - f^{I,\theta}(y) = \theta_i - \theta_i = 0$ for all $i \in I$. Thus,

$$\begin{aligned}\|f^{I,\theta}(x) - f^{I,\theta}(y)\| &= \max_{i \notin I} \frac{1}{w_i} \|f_i(x) - f_i(y)\|_i \\ &\le \max_{i} \frac{1}{w_i} \|f_i(x) - f_i(y)\|_i \\ &= \|f(x) - f(y)\| \le \alpha \|x - y\|.\end{aligned}$$

Hence, the mappings $f^{I,\theta}$ inherit the contraction property (9.1). As discussed in Section 6, this property guarantees asynchronous convergence and therefore Proposition 5 applies again.

We conclude that the modification $x := g(x)$ of the asynchronous iteration $x := f(x)$ is often, but not always, guaranteed to terminate in finite time. It is an interesting research question to devise economical termination procedures for the iteration $x := f(x)$ that are always guaranteed to work. The snapshot algorithm of Chandy and Lamport (1985) [see (Bertsekas and Tsitsiklis, 1989, Section 8.2)] seems to be one option.

10. CONCLUSIONS

Iterative algorithms are easy to parallelize and can be executed synchronously even in inherently asynchronous computing systems. Furthermore, for the regular communication networks associated with several common parallel architectures, the communication requirements of iterative algorithms are not severe enough to preclude the possibility of massive parallelization and speedup of the computation. Iterative algorithms can also be executed asynchronously, often without losing the desirable convergence properties of their synchronous counterparts, although the mechanisms that affect convergence can be quite different for different types of algorithms. Such asynchronous execution may offer substantial advantages in a variety of contexts.

At present, there is very strong evidence suggesting that asynchronous iterations converge faster than their synchronous counterparts. However, this evidence is principally based on analysis and simulations. There is only a small number of related experimental works using shared memory machines. These works support the conclusions of the analysis but more testing with a broader variety of computer architectures is needed to provide a comprehensive picture of the practical behavior of asynchronous iterations. Furthermore, the proper implementation of asynchronous algorithms in real parallel machines can be quite challenging and more experience is needed in this area. Finally, much remains to be done to enlarge the already substantial class of problems for which asynchronous algorithms can be correctly applied.

REFERENCES

Anwar, M. N. and N. El Tarazi (1985). Asynchronous algorithms for Poisson's equation with nonlinear boundary conditions. *Computing*, **34**, 155–168.

Awerbuch, B. (1985). Complexity of network synchronization. *J. ACM*, **32**, 804–823.

Barbosa, V. C. (1986). Concurrency in systems with neighborhood constraints. Doctoral Dissertation, Computer Science Dept., U.C.L.A., Los Angeles, CA, U.S.A.

Barbosa, V. C. and E. M. Gafni (1987). Concurrency in heavily loaded neighborhood-constrained systems. *Proc. 7th Int. Conf. on Distributed Computing Systems.*

Baudet, G. M. (1978). Asynchronous iterative methods for multiprocessors. *J. ACM*, **2**, 226–244.

Bertsekas, D. P. (1982). Distributed dynamic programming, *IEEE Trans. Aut. Control*, **AC-27**, 610–616.

Bertsekas, D. P. (1983). Distributed asynchronous computation of fixed points. *Math. Programm.* **27**, 107–120.

Bertsekas, D. P. (1986). Distributed asynchronous relaxation methods for linear network flow problems. Technical Report LIDS-P-1606, Laboratory for Information and Decision Systems, M.I.T., Cambridge, MA.

Bertsekas, D. P. and D. A. Castanon (1989). Parallel synchronous and asynchronous implementations of the auction algorithm. Technical Report TP-308, Alphatech Inc, Burlington, MA. Also *Parallel Computing* (to appear).

Bertsekas, D. P. and D. A. Castanon (1990). Parallel asynchronous primal-dual methods for the minimum cost flow problem. *Math. Programming* (submitted).

Bertsekas, D. P. and J. Eckstein (1988). Dual coordinate step methods for linear network flow problems. *Math. Programming*, **42**, 203–243.

Bertsekas, D. P. and J. Eckstein (1987). Distributed asynchronous relaxation methods for linear network flow problems. *Proc IFAC '87*, Munich, F.R.G.

Bertsekas, D. P. and D. El Baz (1987). Distributed asynchronous relaxation methods for convex network flow problems. *SIAM J. Control Optimiz.*, **25**, 74–84.

Bertsekas, D. P. and R. G. Gallager (1987). *Data Networks*. Prentice Hall, Englewood Cliffs, NJ.

Bertsekas, D. P., C. Ozveren, G. Stamoulis, P. Tseng and J. N. Tsitsiklis (1989). Optimal communication algorithms for hypercubes, Technical Report LIDS-P-1847, Laboratory for Information and Decision Systems, M.I.T., Cambridge, MA. Also *J. Parallel Distrib. Comput.* (to appear).

Bertsekas, D. P. and J. N. Tsitsiklis (1989a). Convergence Rate and Termination of Asynchronous Iterative Algorithms, *Proc. 1989 Int. Conf. on Supercomputing*, Irakleion, Greece, 1989, pp. 461–470.

Bertsekas, D. P. and J. N. Tsitsiklis (1989b). *Parallel and Distributed Computation: Numerical Methods*, Prentice-Hall, Englewood Cliffs, NJ.

Bojanczyk, A. (1984). Optimal asynchronous Newton method for the solution of nonlinear equations. *J. ACM*, **32**, 792–803.

Chandy, K. M. and L. Lamport (1985). Distributed snapshots: determining global states of distributed systems, *ACM Trans. Comput. Syst.*, **3**, 63–75.

Chazan, D. and W. Miranker (1969). Chaotic relaxation. *Linear Algebra and its Applications*, **2**, 199–222.

Donnelly, J. D. P. (1971). Periodic chaotic relaxation, *Linear Algebra and its Applications*, **4**, 117–128.

Dijkstra, E. W. and C. S. Scholten (1980). Termination detection for diffusing computations. *Inform. Process. Lett.*, **11**, 1–4.

Dubois, M. and F. A. Briggs (1982). Performance of synchronized iterative processes in multi-processor systems, *IEEE Trans. Software Engng*, **8**, 419–431.

El Tarazi, M. N. (1982). Some convergence results for asynchronous algorithms. *Numerische Mathematik*, **39**, 325–340.

Fox, G., M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker (1988). *Solving Problems on Concurrent Processors, Vol. 1*, Prentice-Hall, Englewood Cliffs, NJ.

Jefferson, D. R. (1985). Virtual time. *ACM Trans. Programm. Languages Syst.*, **7**, 404–425.

Johnsson, S. L. and C. T. Ho (1989). Optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Comput.*, **38**, 1249–1268.

Kung, H. T. (1976). Synchronized and asynchronous parallel algorithms for multiprocessors. In *Algorithms and Complexity*. Academic Press, New York, pp. 153–200.

Kushner, H. J. and G. Yin (1987a). Stochastic approximation algorithms for parallel and distributed processing. *Stochastics*, **22**, 219–250.

Kushner, H. J. and G. Yin (1987b). Asymptotic properties of distributed and communicating stochastic approximation algorithms. *SIAM J. Control Optimiz.*, **25**, 1266–1290.

Lang, B., J. C. Miellou and P. Spiteri (1986). Asynchronous relaxation algorithms for optimal control problems. *Math. Comput. Simult.*, **28**, 227–242.

Lavenberg, S., R. Muntz and B. Samadi (1983). Performance analysis of a rollback method for distributed simulation. In A. K. Agrawala and S. K. Tripathi (Eds.), *Performance* 83. North Holland, Amsterdam, pp. 117–132.

Li, S. and T. Basar (1987). Asymptotic agreement and convergence of asynchronous stochastic algorithms. *IEEE Trans. Aut. Control*, **32**, 612–618.

Lubachevsky, B. and D. Mitra (1986). A chaotic asynchronous algorithm for computing the fixed point of a nonnegative matrix of unit spectral radius. *J. ACM*, **33**, 130–150.

McBryan, O. A. and E. F. Van der Velde (1987). Hypercube algorithms and implementations, *SIAM J. Scientific Statist. Comput.*, **8**, s227–s287.

Miellou, J. C. (1975a). Algorithmes de relaxation chaotique a retards. *R.A.I.R.O.*, **9**, R-1, 55–82.

Miellou, J. C. (1975b). Iterations chaotiques a retards, études de la convergence dans le cas d'èspaces partiellement ordonnes. *Comptes Rendus, Academie de Sciences de Paris*, **280**, *Serie A*, 233–236.

Miellou, J. C. and P. Spiteri (1985). Un critère de convergence pour des méthodes generales de point fixe. *Math. Modelling Numer. Anal.*, **19**, 645–669.

Mitra, D. (1987). Asynchronous relaxations for the numerical solution of differential equations by parallel processors, *SIAM J. Sci. Statist. Comput.*, **8**, s43–s58.

Mitra, D. and I. Mitrani (1984). Analysis and optimum performance of two message-passing parallel processors synchronized by rollback. In E. Gelenbe (Ed.), *Performance* '84. North Holland, Amsterdam, 35–50.

Nassimi, D. and S. Sahni (1980). An optimal routing algorithm for mesh-connected parallel computers, *J. ACM*, **27**, 6–29.

Ortega, J. M. and R. G. Voigt (1985). Solution of partial differential equations on vector and parallel computers, *SIAM Review*, **27**, 149–240.

Ozveren, C. (1987). Communication aspects of parallel processing, Technical Report LIDS-P-1721, Laboratory for Information and Decision Systems, MIT, Cambridge, MA.

Robert, F. (1976). Contraction en norme vectorielle: convergence d'iterations chaotiques pour des equations non lineaires de point fixe a plusieurs variables. *Linear Algebra and its Applications*, **13**, 19–35.

Robert, F. (1987). Iterations discretes asynchrones, Technical Report 671M, I.M.A.G., University of Grenoble, France.

Robert, F., M. Charnay and F. Musy (1975). Iterations chaotiques serie-parallele pour des equations non-lineaires de point fixe, *Aplikace Matematicky*, **20**, 1–38.

Saad, Y. and M. H. Schultz (1987). Data Communication in Hypercubes, Research Report YALEU/DCS/RR-428, Yale University, New Haven, CN.

Smart, D. and J. White (1988). Reducing the parallel solution time of sparse circuit matrices using reordered Gaussian elimination and relaxation. *Proc.* 1988 *ISCAS*, Espoo, Finland.

Spiteri, P. (1984). Contribution a l'étude de grands systèmes non lineaires. Doctoral Dissertation, L'Universite de Franche-Comte, Besançon, France.

Spiteri, P. (1986). Parallel asynchronous algorithms for solving boundary value problems. In M. Cosnard *et al.* (Eds.), *Parallel Algorithms and Architectures*, North Holland, Amsterdam, pp. 73–84.

Tajibnapis, W. D. (1977). A correctness proof of a topology information maintenance protocol for a distributed computer network. *Commun. ACM*, **20**, 477–485.

Tsai, W. K. (1986). Optimal quasi-static routing for virtual circuit networks subjected to stochastic inputs. Doctoral Dissertation, Dept of Electrical Engineering and Computer Science, M.I.T., Cambridge, MA.

Tsai, W. K. (1989). Convergence of gradient projection routing methods in an asynchronous stochastic quasi-static virtual circuit network. *IEEE Trans. Aut. Control*, **34**, 20–33.

Tsai, W. K., J. N. Tsitsiklis and D. P. Bertsekas (1986). Some issues in distributed asynchronous routing in virtual circuit data networks. *Proc. 25th IEEE Conf. on Decision and Control*, Athens, Greece. 1335–1337

Tseng, P. (1990). Distributed computation for linear programming problems satisfying a certain diagonal dominance condition. *Math. Operat. Res*, **15**, 33–48.

Tseng, P., D. P. Bertsekas and J. N. Tsitsiklis (1990). *SIAM J. Control Optimiz.* **28**. 678–710

Tsitsiklis, J. N. (1984). Problems in decentralized decision making and computation. Ph.D. thesis, Dep of Electrical Engineering and Computer Science, M.I.T., Cambridge, MA.

Tsitsiklis, J. N. (1987). On the stability of asynchronous iterative processes. *Math. Syst.*, **20**, 137–153.

Tsitsiklis, J. N. (1989). A comparison of Jacobi and Gauss–Seidel parallel iterations. *Applied Math. Lett.*, **2**, 167–170.

Tsitsiklis, J. N. and D. P. Bertsekas (1986). Distributed asynchronous optimal routing in data networks, *IEEE Trans. Aut. Control*, **AC-31**, 325–332.

Tsitsiklis, J. N., D. P. Bertsekas and M. Athans (1986). Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *IEEE Trans. Aut. Control*, **AC-31**, 803–812.

Tsitsiklis, J. N. and G. D. Stamoulis (1990). On the average communication complexity of asynchronous distributed algorithms, Technical Report LIDS-P-1986, Laboratory for Information and Decision Systems, M.I.T., Cambridge, MA.

Uresin, A. and M. Dubois (1986). Generalized asynchronous iterations. In *Lecture Notes in Computer Science*, **237**, Springer, Berlin, pp. 272–278.

Uresin, A. and M. Dubois (1988a). Sufficient conditions for the convergence of asynchronous iterations. Technical Report, Computer Research Institute, University of Southern California, Los Angeles, California, U.S.A.

Uresin, A. and M. Dubois (1990). Parallel asynchronous algorithms for discrete data. *J. ACM*, **37**, 588–606.

Zenios, S. A. and R. A. Lasken (1988). Nonlinear network optimization on a massively parallel Connection Machine. *Annals of Operations Research*, **14**, 147–165.

Zenios, S. A. and J. M. Mulvey (1988). Distributed algorithms for strictly convex network flow problems, *Parallel Computing*, **6**, 45–56.