November 1988 LIDS–P–1835

PARALLEL AND DISTRIBUTED ITERATIVE ALGORITHMS: A SELECTIVE SURVEY[1]

Dimitri P. Bertsekas[2]
John N. Tsitsiklis[2]

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each processor iterates on a different component of x, at its own pace, using the most recently received (but possibly outdated) information on the remaining components of x. While certain algorithms may fail to converge when implemented asynchronously, a large number of positive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.

Keywords: Iterative methods, asynchronous algorithms, parallel algorithms, distributed algorithms.

---

1. Research supported by the NSF under Grants ECS–8519058 and ECS–8552419, with matching funds from Bellcore Inc. and IBM Inc., and the ARO under Grant DAAL03–86–K–0171.

2. Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, MA 02139.

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 reliable and predictable. *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 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 as well.

There are at least two contexts where distributed computation has played a significant 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 algorithms of the form $x := f(x)$ where $x = (x_1, \ldots, x_n)$ is a vector in $\Re^n$ and $f : \Re^n \mapsto \Re^n$ 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 in-

terprocessor 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, 1989) where proofs of most of the results quoted here can be found.

Iteration (1.1) can be executed *synchronously* whereby all processors perform an iteration, communicate their results to the other processors, and then proceed to the next iteration. In Section 2, we indicate that this 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. In Section 3, we discuss an alternative implementation of iteration (1.1) whereby components are updated one at a time, and we indicate that such implementations can admit considerable parallelism when the iteration mapping f has a sparse structure. Then, in Section 4, we discuss the effects of communication delays on the speed of the computation, assuming that processors communicate using a point–to–point communication network. In Section 5, we indicate that the synchronous execution of iteration (1.1) can have 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 behaviours, ranging from divergence to guaranteed convergence in the face of the worst possible amount of asynchronism and communication delays. We classify the possible behaviours in three broad classes; the corresponding convergence results are surveyed in Sections 6, 7, and 8, respectively. Section 9 contains our conclusions.

2. SYNCHRONOUS ITERATIONS AND SYNCHRONIZATION MECHANISMS

Let $X_1, \ldots, X_p$, be subsets of Euclidean spaces $\Re^{n_1}, \ldots, \Re^{n_p}$, respectively. Let $n = n_1 + \cdots + n_p$, and let $X \subset \Re^n$ be the Cartesian product $X = \prod_{i=1}^{p} X_i$. Accordingly, any $x \in \Re^n$ is decomposed in the form $x = (x_1, \ldots, x_p)$, with each x_i belonging to $\Re^{n_i}$. 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 $f(x) = \big(f_1(x), \ldots, f_p(x)\big)$ for every $x \in X$. We consider an iteration of the form

$$x := f(x), \tag{2.1}$$

and we call f the *iteration mapping* defining the algorithm. We assume that there are p processors, with the ith processor assigned the responsibility of updating the ith component x_i according to the rule $x_i := f_i(x) = f_i(x_1, \ldots, x_p)$. 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 $b \in \Re^n$, 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; Fox *et al.*, 1988) for a discussion of alternative matrix storage schemes.

For the implementation of iteration (2.1) we have just described, 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 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 multihop path in a communication network. The discussion that follows applies to both cases.

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

$$x(t+1) = f\big(x(t)\big), \tag{2.2}$$

where t is an integer–valued variable used to index different iterations, not necessarily representing real time. 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.

(a) Global Synchronization.

Here the processors proceed to the $(t+1)$st iteration 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 message to this effect which is propagated along the spanning tree. Once a processor receives such a message it can proceed to the next phase. (See Fig. 2.1 for an illustration.)

(b) Local synchronization.

Global synchronization can be seen to be rather wasteful in terms of the time required for each

iteration. An alternative is to allow the ith processor to proceed with the $(t+1)$st iteration as soon 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 (joint work with C.H. Papadimitriou, unpublished) that the expected time for K iterations is of the order of $\log p + K\log d$. 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, but can also be viewed as a general purpose synchronization method. 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 $x_j(t)$ had 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.*; Mitra and Mitrani, 1984).

3. GAUSS–SEIDEL ITERATIONS.

Iteration (2.2), in which all of the components of x are simultaneously updated, 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

$$x_i(t+1) = f_i\Big(x_1(t+1), \ldots, x_{i-1}(t+1), x_i(t), \ldots, x_p(t)\Big), \qquad i = 1, \ldots, p. \tag{3.1}$$

Gauss–Seidel iterations are often preferable: they incorporate the newest available information and they tend to converge faster than the corresponding Jacobi iterations. In fact, under certain conditions Jacobi iterations fail to converge while Gauss–Seidel iterations are guaranteed to converge. However, the parallel implementation of iteration (3.1) can be problematic, if no two components can be updated in parallel, and this is the case when the dependency graph describing the structure of the iteration is complete (every component depends on every other component). On the other hand, Gauss–Seidel iterations can be substantially parallelizable when the dependency graph is sparse, as we now illustrate.

Consider the dependency graph of Fig. 3.1. A corresponding Gauss–Seidel iteration is described by

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

and its structure is shown in Fig. 3.2. 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.

Frequently in Gauss–Seidel iterations the order in which the different components are updated is not very important, in which case we are free to choose a particular order. In the context of our example, a different ordering of the components leads to an iteration of the form

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

which is illustrated in Fig. 3.3. 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 is as follows.

Proposition 3.1. (Bertsekas and Tsitsiklis, 1989) 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 are used and so that each subgraph obtained by restricting to the set of nodes with the same color has no directed cycles.

Unfortunately, the coloring problem of Prop. 3.1 is intractable (NP–hard). On the other hand, in several practical situations, and especially when solving partial differential equations or image processing problems, the dependency graph G has a very simple structure and the coloring problem can be solved by inspection. Furthermore, in these contexts, the dependency graph G is often symmetric; that is, the presence of an arc $(i,j) \in A$ also implies the presence of the arc (j,i). If this is the case, the coloring problem of Prop. 3.1 reduces to coloring the nodes of the dependency graph so that no two neighboring nodes have the same color. Even with unstructured dependency graphs, reasonably good colorings can be found using simple heuristics, as is often done; see (Zenios and Mulvey, 1988), for example. 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 very easy to implement, is discussed in (Barbosa, 1986; 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.

4. COMMUNICATION ASPECTS OF SYNCHRONOUS ITERATIONS

4.1. Basic Communication Tasks.

When an iterative algorithm is executed in a network of processors, 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 iterations of the form (2.1). 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 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 positive 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 $O(r)$ time, where r is the diameter of the network, as shown in Fig. 4.1(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 from each node, as in an inner product calculation [see Fig. 4.1(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. 4.1.

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.*, 1988; Bertsekas and Tsitsiklis, 1989). Table 4.1 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.*, 1988) and (Bertsekas and Tsitsiklis, 1989). In most cases these algorithms are optimal in that they solve the problem in the minimum possible number of time steps. It also shown in (Bertsekas and

Tsitsiklis, 1989) 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.

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)	$\Theta(p)$	$\Theta(p)$	$\Theta(p)$	$\Theta(p/\log p)$

Table 4.1: 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.

4.2. The effect of the number of processors on the communication overhead.

In many parallel iterative algorithms, the time required for interprocessor communication is greatly affected by the number of processors used to execute the algorithm. It is generally true that as the number of processors increases, the time spent for communication also increases. Therefore, as we attempt to decrease the solution time of a given problem by using more and more processors, we must contend with increased communication overhead. This might place an upper bound on the size of problems of a given type that we can realistically solve even with an unlimited number of processors.

For a given problem, there are both general and problem–specific reasons why the communication overhead tends to increase with the number of processors. A first reason is the possibility of *pipelining of computation and communication.* If some of the results of computation at a processor can be communicated to other processors while other results are still being computed, the communication overhead will be reduced. Consider for example an iteration of the form

$$x_i(t+1) = f_i\big(x_1(t), \ldots, x_p(t)\big), \qquad i = 1, \ldots, p, \tag{4.1}$$

where each x_i is a vector of dimension k that is assigned to a separate processor i and $n = pk$ is the dimension of the problem. Pipelining of computation and communication is more pronounced when there is a large number of variables assigned to each processor; then the variables that have been already updated within an iteration can be made available to other processors while the updating of other variables is still pending. A second reason is that in many systems, a portion of each message

is used to carry overhead information. The length of this portion is usually fixed and independent of the total length of the message. This means that there is a gain in efficiency when messages are long, since then the overhead per bit of data is diminished. It is clear that the length of the messages can be made longer if the number of variables updated by each processor is larger, since then the values of many variables can be transmitted to other processors as a single message.

Even in the absence of overhead, and of pipelining of computation and communication, the effect of the communications tends to be reduced as the dimension k of the component vectors x_i in the iteration (4.1) is increased. Suppose that processor i uses Eq. (4.1) to update the k–dimensional vector x_i, with knowledge of the other vectors x_j, $j \neq i$. Suppose also that the computation time for each update is $\Theta(nk)$ [as it will be, for example, when the function f_i in Eq. (4.1) is linear without any special sparsity structure]. After updating x_i, processor i must communicate the corresponding k variables to all other processors so that the next iteration can proceed. This can be done via a multinode broadcast, and if a linear array is used for this purpose, the optimal communication time is $\Theta(n)$, assuming that communication of k variables over a single link takes $\Theta(k)$ time. Thus, the ratio

$$\frac{T_{COMM}}{T_{COMP}} = \frac{\text{Communication time per iteration}}{\text{Computation time per iteration}}$$

is $\Theta(1/k)$, and the communication time becomes relatively insignificant as the number k of variables updated by each processor increases. The ratio T_{COMM}/T_{COMP} is independent of the problem dimension n; it only depends on k, that is, the size of the computation task per iteration for each processor.

A further observation from this analysis is that the speedup obtained through parallelization of the iteration (4.1) can be increased as the dimension n of the problem increases. In particular, the computation time per iteration on a serial machine is $\Theta(n^2)$ [it is $\Theta(nk)$ based on our earlier hypothesis and, for a serial machine, we have $p = 1$ and $k = n$], so the speedup using a linear array of p processors, each updating $k = n/p$ variables, becomes

$$\frac{\Theta(n^2)}{\Theta(n) + \Theta(nk)} = \Theta(p),$$

where the $\Theta(n)$ and $\Theta(nk)$ terms correspond to the communication time and the computation time, respectively.

We have thus reached the important conclusion that for iterations of the form (4.1), the communication overhead will not prevent the fruitful utilization of a large number of processors in parallel when the problem is large, even when a linear array (the "least powerful" network) is used for communication. What is needed, as the dimension of the problem increases, is a proportional increase of the number of processors p of the linear array that will keep the number k of variables per processor roughly constant at a level where the communication time is relatively small. Note also that when a hypercube is used in place of a linear array, the optimal multinode broadcast time

is $\Theta\big((pk)/\log p\big)$, so the ratio T_{COMM}/T_{COMP} decreases from $\Theta(1/k)$ to $\Theta\big(1/(k\log p)\big)$. Therefore, as the dimension of the problem increases by a certain factor, the number of processors of the hypercube can be increased by a larger factor while keeping the communication time at a relatively insignificant level, and the attainable speedup increases at a faster rate than with a linear array.

The preceding analysis does not assume any special structure for the iteration (4.1) other than the hypothesis that a single variable update takes $\Theta(n)$ time. The ratio T_{COMM}/T_{COMP} is also small for large k in many other cases where there is special structure. An important example is associated with problems arising from discretization of two–dimensional physical space and with the so called *area–perimeter effect* (see Fig. 4.2). As shown in the figure, the number of variables that have to be communicated by a processor is $\Theta(\sqrt{k})$, and the time taken for communication on a mesh network or a hypercube is $\Theta(\sqrt{k})$. The time taken for each variable update is a constant, and the parallel computation time for each iteration is $\Theta(k)$. The ratio T_{COMM}/T_{COMP} is $\Theta(1/\sqrt{k})$.

The conclusion form the preceding discussion is that with proper selection of the size of the computation task for each processor, the effects of communication can be minimized. Furthermore, as the size of the given problem increases without bound, the speedup can typically also increase without bound by using an appropriate parallel machine. In other words, there is no *a priori* bound on the attainable speedup that is imposed by the communication requirements. The recent prize–winning experimental work of Gustafson *et al.* (1988) supports these conclusions.

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 iteration (2.1) 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) Implementation flexibility: There is no overhead such as the one associated with the global synchronization method. Furthermore, in certain cases, there are even advantages over the local synchronization method as we now discuss. Suppose that the iteration is such that an 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. Suppose now that the algorithm is such that a typical iteration is most 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 very 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 effects 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 mathematicallly by the iteration $x(t+1) = f\big(x(t)\big)$. Thus, even if this iteration is convergent, the corresponding asynchronous iteration could be divergent, and indeed this is sometimes the case. Even if the asynchronous iteration is convergent, such a conclusion often requires rather difficult analysis. 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 through 8.

We now present our model of asynchronous computation. Let the set X and the function f be as decsribed 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

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

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

$$0 \leq \tau_j^i(t) \leq t, \qquad \forall t \geq 0.$$

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

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

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 5.1. (*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 5.2. (*Partial asynchronism*) There exists a positive constant B such that:

(a) For every $t \geq 0$ 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

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

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

The constant B of Assumption 5.2, to be called the *asynchronism measure*, bounds the amount by which the information available to a processor can be outdated. Notice that synchronous execution is the special case of partial asynchronism in which $B = 1$. Notice also that Assumption 5.2(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. It turns out that if we relax Assumption 5.2(c), the convergence of certain asynchronous algorithms is destroyed (Lubachevsky and Mitra, 1986; Bertsekas and Tsitsiklis, 1989). Parts (a) and (b) of Assumption 5.2 are typically satisfied in practice.

Asynchronous algorithms can exhibit three different types of behaviour (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, 1988a, 1988b; Chine, 1988). The following general result is from (Bertsekas, 1983).

Proposition 6.1. Let $X = \prod_{i=1}^{p} X_i \subset \prod_{i=1}^{p} \Re^{n_i}$. Suppose that the mapping $f : X \mapsto X$ has a unique fixed point $x^* \in X$. Furthermore, 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 $x \in X(k)$.
(c) Every sequence $\{x(k)\}$ with the property $x(k) \in X(k)$ for all k, converges to x^*.
Then, under Assumption 5.1 (total asynchronism), the sequence $\{x(t)\}$ generated by the asynchronous iteration (5.1)–(5.2) converges to x^*.

The key idea behind Proposition 6.1 is that eventually $x(t)$ enters and stays in the set $X(k)$; furthermore, due to condition (b) in Prop. 6.1, it eventually moves into the next set $X(k+1)$. Successful application of this result 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 Prop. 6.1 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.

Several authors have also studied asynchronous iterations with zero delays, that is, under the

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

assumption $\tau_j^i(t) = t$ for every $t \in T^i$. See for example, (Robert, Charnay, and Musy, 1975; Robert 1976, 1987, 1988). General necessary and sufficient convergence 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

$$\|x\| = \max_i \frac{\|x_i\|_i}{w_i},$$

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

$$\|f(x) - x^*\| \leq \alpha \|x - x^*\|, \qquad \forall x \in X, \tag{6.1}$$

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

$$X_i(k) = \left\{ x_i \in \Re^{n_i} \mid \|x_i - x_i^*\|_i \leq \alpha^k \|x(0) - x^*\| \right\}. \tag{6.2}$$

It is easily verified that these sets satisfy the conditions of Proposition 6.1 and convergence to x^* follows.

Iteration mappings f with the contraction property (6.1) 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). As a special case, we obtain totally asynchronous convergence of the iteration $\pi := \pi P$ for computing a row vector π with the invariant probabilities of an irreducible, discrete–time, finite–state, Markov chain specified in terms of the stochastic matrix P, provided that one of the components of π is held fixed throughout the algorithm (Bertsekas and Tsitsiklis, 1989). Another special case 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

$$\sum_{j \neq i} |\nabla^2_{ij} F(x)| \leq \nabla^2_{ii} F(x) - \beta, \qquad \forall i, \ \forall x \in X. \tag{6.3}$$

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

Example 6.1. 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 $F(x) = \frac{1}{2}x'Ax$. If $\epsilon > 1$, then the diagonal dominance condition of Eq. (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 Eq. (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 $x_i(2t_0) \approx -2x_i(t_0)/(1+\epsilon) \approx (-2)^2 c/(1+\epsilon)^2$. 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

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

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, 1989). 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 of 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, 1989).

Other studies have dealt with an asynchronous Newton algorithm (Bojanczyk, 1984), an agreement problem (Li and Basar, 1987), diagonally dominant linear programming problems (Tseng, 1987), and a variety of infinite–dimensional problems such as partial differential equations, and variational inequalities (Spiteri, 1984; Miellou and Spiteri, 1985; Spiteri, 1986; 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. 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 and α is the contraction factor of Eq. (6.1). We now consider an asynchronous execution whereby an iteration is performed by each processor i at each time unit and the values of x_j $(j \neq i)$ which are used are outdated by D time units. Concerning the function f, we assume that there exists some scalar β such that $0 < \beta < \alpha$ and

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

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, 1989) 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 $f : \Re^n \mapsto \Re^n$ 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 Prop. 6.1 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. Assuming that the shortest distances are finite, they correspond to the unique fixed point of the monotone mapping $f : \Re^n \mapsto \Re^n$ defined by $f_1(x) = 0$ and

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

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 $O(n^3)$. 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) 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. Some results of this type can be proved analytically, an example being the following. Suppose that there exist positive constants δ, Δ, such that the time between consecutive updates of x_i is bounded below by δ and above by Δ, for each i. Furthermore, assume that the delay of each message is an independent random variable, with exponential distribution and mean one. Then, the expected number of messages exchanged is bounded above by a polynomial in n whose coefficients depend on Δ and δ (Tsitsiklis, 1988).

A number of asynchronous convergence results making essential use of monotonicity conditions are also available for dual relaxation algorithms for linear and nonlinear network flow problems (Bertsekas and Eckstein, 1987, 1988; Bertsekas and El Baz, 1987).

7. PARTIALLY ASYNCHRONOUS ALGORITHMS–I

We now consider partially asynchronous iterations, satisfying Assumption 5.2. 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

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

We then notice that $x(t+1)$ can be determined [cf. Eqs. (5.1)–(5.3)] in terms of $z(t)$; in particular, knowledge of $x(\tau)$, for $\tau \leq t - B$ 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 7.1. (Bertsekas and Tsitsiklis, 1989) Suppose that there exists 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 5.2) we have $d\big(z(t^*)\big) < d\big(z(0)\big)$ and $d\big(z(1)\big) \leq d\big(z(0)\big)$. 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 $d\big(z(t)\big)$ converges to zero at the rate of a geometric progression.

Suppose now that $n_i = 1$ for each i, and consider a mapping $f : \Re^n \mapsto \Re^n$ of the form $f(x) = Ax$ where A is an irreducible stochastic matrix. 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

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

has the properties assumed in Prop. 7.1, 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 7.1 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, 1989) shows that the agreement algorithm need not converge totally asynchronously.

Example 7.1. 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

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

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

(See Fig. 7.1 for an illustration.) Thus,

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

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

Subtracting these two equations we obtain

$$\begin{aligned} |x_2(t_{k+1}) - x_1(t_{k+1})| &= \Big(1 - 2(1/2)^{t_{k+1}-t_k}\Big)|x_2(t_k) - x_1(t_k)| \\ &= (1-\epsilon_k)|x_2(t_k) - x_1(t_k)|, \end{aligned}$$

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 7.1 shows that failure to converge is possible if part (b) of the partial asynchronism Assumption 5.2 fails to hold. There also exist examples demonstrating that parts (a) and (c) of Assumption 5.2 are also necessary for convergence.

Example 7.1 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 7.1, 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, 1989), 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, 1989).

(b) Relaxation algorithms involving nonexpansive mappings with respect to the maximum norm (Tseng *et al.*, 1988; Bertsekas and Tsitsiklis, 1989). 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 $\sum_{j \neq i} |a_{ij}| \leq a_{ii}$, 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 (Bertsekas and Tsitsiklis, 1989).

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

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

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, 1987b; Bertsekas and Tsitsiklis, 1989).

8. PARTIALLY ASYNCHRONOUS ALGORITHMS–II

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 6.1, 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 (Bertsekas and Tsitsiklis, 1989). 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, 1989). 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 $x_i\left(\tau_j^i(t)\right)$ 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 non–quadratic 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, 1989). Similar conclusions are obtained for gradient projection iterations for constrained convex optimization (Bertsekas and Tsitsiklis, 1989).

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).

9. 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.

REFERENCES

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

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

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

Barbosa, V.C., and E.M. Gafni (1987). Concurrency in heavily loaded neighborhood–constrained systems, *Proceedings of the 7th International Conference on Distributed Computing Systems.*

Baudet, G.M. (1978). Asynchronous iterative methods for multiprocessors, *Journal of the ACM, 2,* pp. 226–244.

Bertsekas, D.P. (1982). Distributed dynamic programming, *IEEE Transactions on Automatic Control, AC-27,* pp. 610–616.

Bertsekas, D.P. (1983). Distributed asynchronous computation of fixed points, *Mathematical Programming, 27,* pp. 107–120.

Bertsekas, D.P., and J. Eckstein (1987). Distributed asynchronous relaxation methods for linear network flow problems, *Proceedings of IFAC '87,* Munich, Germany.

Bertsekas, D.P., and J. Eckstein (1988). Dual coordinate step methods for linear network flow problems, Technical Report LIDS–P–1768, Laboratory for Information and Decision Systems, M.I.T., Cambridge, MA.

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

Betsekas, 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 (1988). Optimal communication algorithms for hypercubes, in preparation.

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

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

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

Chine, A. (1988). Etude de la convergence globale ou locale des iterations discretes asynchrones, Technical Report RT35, I.M.A.G., University of Grenoble, France.

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

Dubois, M., and F.A. Briggs (1982). Performance of synchronized iterative processes in multiprocessor systems, *IEEE Transactions on Software Engineering, 8,* pp. 419–431.

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

Gustafson, J. L., G. R. Montry, and R. E. Benner (1988). Development of parallel methods for a 1024–processor hypercube, *SIAM J. Scientific and Statistical Computing, 9,* pp. 609– 638.

Jefferson, D.R. (1985). Virtual time, *ACM Transactions on Programming Languages and Systems, 7,* pp. 404–425.

Kung, H.T. (1976). Synchronized and asynchronous parallel algorithms for multiprocessors, in *Algorithms and Complexity,* J.F. Traub (Ed.), Academic, pp. 153–200.

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

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

Lang, B., J.C. Miellou, and P. Spiteri (1986). Asynchronous relaxation algorithms for optimal control problems, *Mathematics and Computers in Simulation, 28,* pp. 227–242.

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

Li, S., and T. Basar (1987). Asymptotic agreement and convergence of asynchronous stochastic algorithms, *IEEE Transactions on Automatic Control, 32,* pp. 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, *Journal of the ACM, 33,* pp. 130–150.

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

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

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

Miellou, J.C., and P. Spiteri (1985). Un critere de convergence pour des methodes generales de point fixe, *Mathematical Modellling and Numerical Analysis, 19*, pp. 645–669.

Mitra, D. (1987). Asynchronous relaxations for the numerical solution of differential equations by parallel processors, *SIAM J. Scientific and Statistical Computing, 8*, pp. s43–s58.

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

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

Ortega, J.M., and R.G. Voigt (1985). Solution of partial differential equations on vector and parallel computers, *SIAM Review, 27*, pp. 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*, pp. 19–35.

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

Robert, F. (1988). Iterations discretes chaotiques et reseaux d'automates, submitted to *Discrete Applied Mathematics.*

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

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

Spiteri, P. (1984). Contribution a l'etude de grands systemes non lineaires, Doctoral Dissertation, L'Universite de Franche–Comte, Besancon, France.

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

Tajibnapis, W. D. (1977). A correctness proof of a topology information maintenance protocol for a distributed computer network, *Communications of the ACM, 20,* pp. 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.

Tseng, P. (1987). Distributed computation for linear programming problems satisfying certain diagonal dominance condition, Technical Report LIDS–P–1644, Laboratory for Information and Decision Systems, M.I.T., Cambridge, MA.

Tseng, P., D.P. Bertsekas, and J.N. Tsitsiklis (1988). Partially asynchronous parallel algorithms for network flow and other problems, Technical Report LIDS–P–1832, Laboratory for Information and Decision Systems, M.I.T., Cambridge, MA.

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

Tsitsiklis, J.N. (1987). On the stability of asynchronous iterative processes, *Mathematical Systems Theory, 20,* pp. 137–153.

Tsitsiklis, J.N, and D.P. Bertsekas (1986). Distributed asynchronous optimal routing in data networks, *IEEE Transactions on Automatic Control, AC–31*, pp. 325–332.

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

Tsitsiklis, J.N. (1988). The average complexity of the asynchronous Bellman–Ford algorithm, in preparation.

Uresin, A., and M. Dubois (1986). Generalized asynchronous iterations, in *Lecture Notes in Computer Science, 237,* Springer Verlag, 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 (1988b). Parallel asynchronous algorithms for discrete data, Technical Report CRI–88–05, Computer Research Institute, University of Southern California, Los Angeles, California, U.S.A.

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

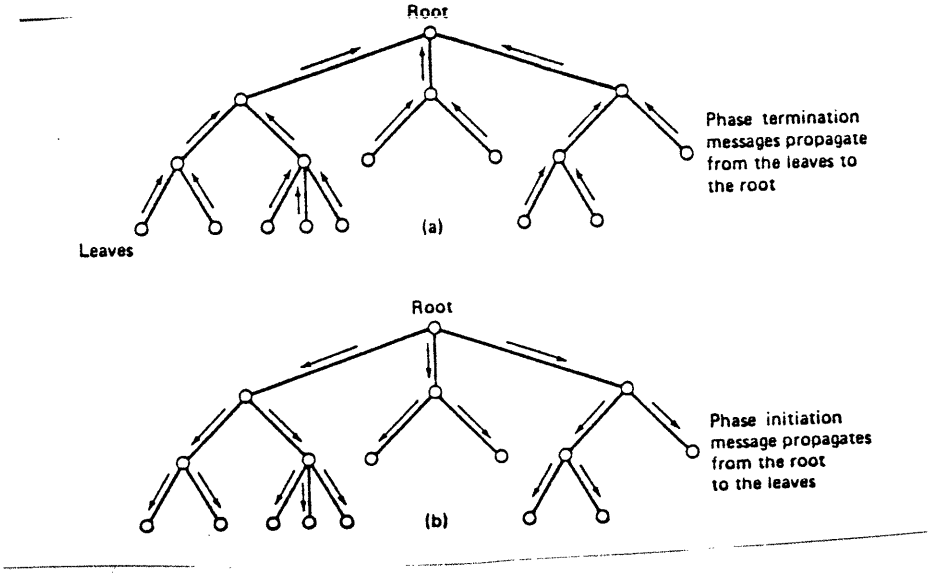

Figure 2.1

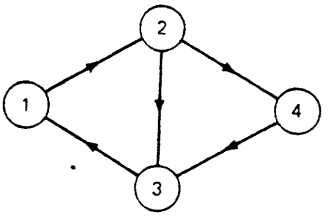

Figure 3.1

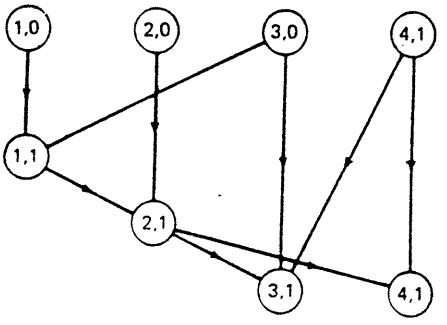

Figure 3.2

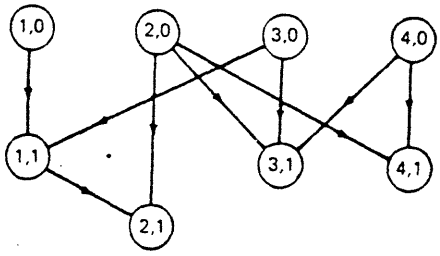

Figure 3.3

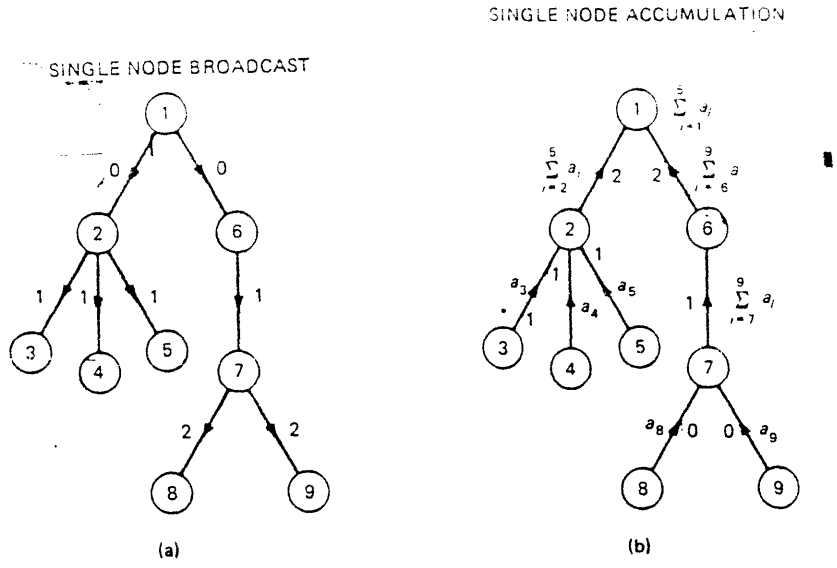

Figure 4.1.

(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.

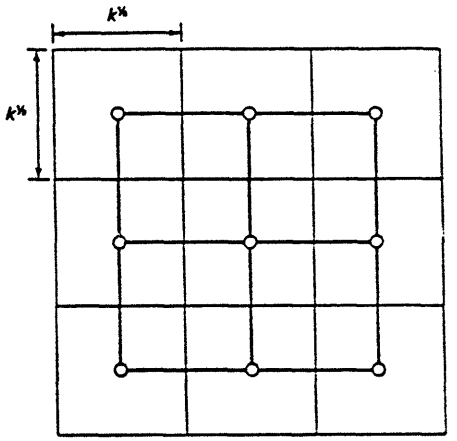

Figure 4.2.

Structure arising from discretization of two–dimensional space. Here the variables are partitioned in rectangles of physical space, and we assume that only neighboring variables interact. Each rectangle contains k variables, and at the end of each iteration, each rectangle must exchange $\Theta(k^{1/2})$ variables with each of its neighboring rectangles.

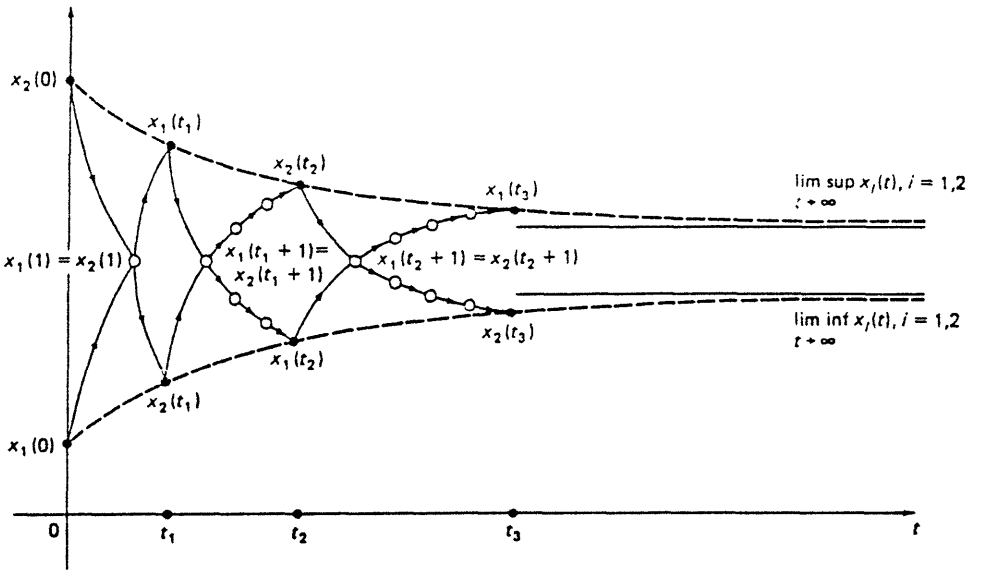

Figure 7.1.