Parallel Computing 10 (1989) 83–92
North-Holland

Sufficient conditions for the convergence of asynchronous iterations *

Aydin ÜRESIN and Michel DUBOIS

Department of Electrical Engineering-Systems, University of Southern California, Los Angeles, CA 90089-0781, U.S.A.

Received October 1986
Revised October 1987

Abstract. Asynchronous algorithms have the potential to be more efficient than synchronized algorithms in multiprocessors because the overheads associated with synchronization are removed. The sufficient conditions for the convergence of numerical asynchronous iterations have been established; however, relaxation procedures are also common in non-numerical applications. In this paper, we introduce sufficient conditions for the convergence of asynchronous iterations defined on any set of data, finite or infinite, countable or not. The sufficient conditions are then applied to the scene-labeling problem. A multitasked implementation of the scene-labeling algorithm in a distributed global memory system is detailed. In this implementation, no synchronization nor critical sections are necessary to enforce correctness of execution.

Keywords. Asynchronous algorithms, multiprocessors, sufficient conditions for convergence, scene-labeling problem, implementation on a model architecture.

1. Introduction

The behavior of many systems can be modeled and described by the set of equations

$$x = F(x),$$

where x is an n-dimensional vector of S^n, F is an operator $S^n \to S^n$, and S is any finite or infinite, countable or uncountable set. One way to solve these equations on a computer is through the *fixed point relaxation method* where the $(k+1)$st iterate, x^{k+1}, is calculated from the previous one, using

$$x^{k+1} = F(x^k), \tag{1}$$

given the initial guess, x^0. Convergence conditions of the above iterative process (i.e., the sequence $\{x^k\}$) are well known for the case where $x^0 \in \mathbb{R}^n$, the operator is $F: \mathbb{R}^n \to \mathbb{R}^n$, and $\mathbb{R}$ is the space of real numbers [11]. Furthermore, convergence of (1) has been studied for the discrete case, and the results were published in [13]. In a multiprocessor system, the above task can be accomplished by decomposing the set of equations, $x = F(x)$, as

$$\begin{aligned} x_1 &= F_1(x) \\ &\vdots \\ x_n &= F_n(x) \end{aligned}$$

* This research is supported by an NSF Initiation Grant No. DMC-8505328.

and assigning a processor to each $F_i(x)$, or to several $F_i(x)$'s. One of the issues involved in such a scheme is the synchronization and dependency relations of successive iterations. As can be seen from (1), there is a dependency between x^{k+1} and x^k; hence the $(k+1)$st iteration cannot be started before the completion of iteration k. This in turn seems to require synchronization between two successive iterations.

On the other hand, it has been shown [2,6] that synchronization between processes is a major source of performance degradation in multiprocessor algorithms. The reasons are that the processes have to execute synchronizaton primitives and also that the processes have to wait for the *slowest* among them at a synchronization point [10]. The execution times of different processes between two successive synchronization points are variable because the processes perform different functions or because they perform the same function on different input data sets; also, conflicts for shared resources, such as an interconnection link or a shared-memory module, affect the execution rate of each processor in a random fashion.

Consequently, research has focused on the behavior [4,5] and performance [3,7,8] of the system when the above-mentioned synchronization between x^k and x^{k+1} is removed. The sequence of computations thus obtained is called *asynchronous iteration*. Because of their performance advantage in realistic multiprocessor systems, asynchronous algorithms are considered an important class of parallel algorithms, and as such, they deserve more research [16]. Asynchronous iterations have been formally defined and studied in [4], where convergence conditions are given for the case $x \in \mathbb{R}^n$.

However, many relaxation techniques involve the manipulation of symbolic or discrete-valued data. One example is scene-labeling relaxation, used in image-understanding applications to define the features of a digital picture [14]. In the general formulation of the problem, a set of labels is associated with each object in the picture (objects may be pixels in a digitized picture); labels are related to objects by compatibility relations. Certain labels may be incompatible with some objects; also, for each pair of objects some pairs of labels may not be compatible. The scene-labeling algorithm finds the greatest consistent labeling. 'Consistent' means that the compatibility relations are respected, and 'greatest' means that the maximum number of compatible labels has been associated with each object. The problem of finding a consistent labeling is a generalization of many graph problems. Examples are given in [9].

These considerations have motivated the study of the generalization of asynchronous iterations to cover the case where $x \in S^n$ (S^n defined as above). In the following, we first define the generalization of asynchronous iterative algorithms in which the relaxed variables belong to an arbitrary set S. In this context, the generalized definition of a contracting operator is introduced. Then the convergence condition is given and proved for the general case. The theory is applied to the scene-labeling problem. Some of the results presented here were reported in [15].

2. Convergence of asynchronous iterations

The formal definition of asynchronous iterations presented below is similar to the one given by Baudet [4]. In Baudet's paper, operators from $\mathbb{R}^n$ to $\mathbb{R}^n$ are considered. The following is the generalization of Baudet's definition for the case of an operator F from S^n to S^n, where S is any set, finite or infinite, countable or not, and x is a vector of iterates shared by all processors.

Definition 2.1 (Asynchronous iteration). Let F be an operator from S^n to S^n. An asynchronous iteration corresponding to the operator F and starting with a given vector $x(0)$ is a sequence

$\{x(j)\}$, $j = 0, 1, \ldots$ of vectors of S^n and defined recursively by

$$x_i(j) = \begin{cases} x_i(j-1) & \text{if } i \notin J_j, \\ F_i(x_1(s_1(j)), \ldots, x_n(s_n(j))) & \text{if } i \in J_j, \end{cases}$$

where $J = \{J_j \mid j = 1, 2, \ldots\}$ is a sequence of nonempty subsets of $\{1, 2, \ldots, n\}$, $C = \{s_1(j), \ldots, s_n(j) \mid j = 1, 2, \ldots\}$ is a sequence of elements in $\mathbb{N}^n$, and S is a set.

In addition, J and C are subject to the following conditions, for each $i = 1, \ldots, n$:

(a) $s_i(j) \leqslant j - 1$, $j = 1, 2, \ldots$;

(b) $s_i(j)$, considered as a function of j, tends to infinity as j tends to infinity;

(c) i occurs an infinite number of times in the sets J_j, $j = 1, 2, \ldots$

An asynchronous iteration corresponding to F, starting with $x(0)$, and defined by J and C, will be denoted by $(F, x(0), J, C)$.

We can think of j's as increasing time instances at which at least one component of the previous iterate is updated. Here, the time distances between consecutive instances are not necessarily equal. The set of the components of x updated at j is denoted by J_j. Before performing an update, all the components of x should be fetched from the global memory. The value of x_i that will be used at the jth update is fetched at $s_i(j)$. The only restriction for the fetching of a component is the natural one: Fetching should be completed prior to the corresponding update ($s_i(j) \leqslant j - 1$). Condition (b) guarantees that no iterate is used an infinite number of times to compute the new values of x. As j goes to infinity, eventually more recent iterates will become available. Finally, Condition (c) states that the ith component is updated an infinite number of times.

J and C define the type of asynchronous relaxation. Different schemes are described in [4] and their representations in terms of J and C are given.

Definition 2.2 (Asynchronously contracting operator). An operator F from S^n to S^n is an asynchronously contracting operator on a subset $D = D_1 \times D_2 \times \cdots \times D_n$ of S^n iff there is a sequence $\{A^k = A_1^k \times \cdots \times A_n^k\}$ of subsets of S^n, such that

(a) $A^0 = D$,

(b) $x \in A^p$ implies $F(x) \in A^{p+1}$ for all $p = 0, 1, 2, \ldots$,

(c) $A^{p+1} \subseteq A^p$ for all $p = 0, 1, 2, \ldots$,

(d) $\lim_{k \to \infty} A^k = \{\xi\}$,

where ξ is the point of convergence.

The reason for defining D as the Cartesian product of n sets is to guarantee that the new value of x falls within the domain every time it is updated by a processor. In the following, we give an example of what can happen when this restriction is released. Suppose that D is defined by the following inequalities in $\mathbb{R}^2$:

$$0 \leqslant x_1 \leqslant 2, \qquad 0 \leqslant x_2 \leqslant x_1,$$

and let the value of the function at the point (2, 2) be (1, 1), i.e.,

$$F(2, 2) = (1, 1).$$

If the initial vector, $x(0)$, is chosen as (2, 2) and the computation of the first component is faster than the computation of the second one, the value of x after the first update is (1, 2), which is not a member of D. This means that $x(1)$ is out of the domain, and we cannot tell whether the iteration converges or not, since the behavior of the function out of D is unknown. For similar reasons, A^k is defined as the Cartesian product of n sets.

Theorem 2.3 proves the convergence of asynchronous iterations defined on an arbitrary set S^n when the operator is an asynchronously contracting operator.

Theorem 2.3. *If F is an asynchronously contracting operator on a subset $D = D_1 \times D_2 \times \cdots \times D_n$ of S^n, then an asynchronous iteration $(F, x(0), J, C)$ corresponding to F and starting with a vector $x(0)$ in D converges to a unique fixed point, ξ in D, for any J and C, where ξ is given by Definition* 2.1.

Proof. Consider the sequence of integer pairs, $\{(l_k, u_k)\}$, $k = 0, 1, \ldots$, such that

(a) $l_0 = u_0 = 0$,

(b) $l_k \leqslant u_k \leqslant l_{k+1}$ for all $k = 0, 1, \ldots$,

(c) $J_{l_k} \cup J_{l_k+1} \cup \cdots \cup J_{u_k} = \{1, 2, \ldots, n\}$ for all $k = 1, 2, \ldots$,

(d) $s_i(j) \geqslant u_{k-1}$, when $j \geqslant l_k$ and for all $i = 1, \ldots, n$ and $k = 1, 2, \ldots$.

The lower value of the kth integer pair is denoted as l_k, and the upper one is denoted as u_k. The upper value of a pair comes before the lower value of the next pair (Condition (b)). Condition (c) states that between the iterations l_k and u_k each component is updated at least once. Finally, Condition (d) implies that after the iteration l_k, only those iterations computed at or after the iteration u_{k-1} are used as input.

From Condition (c) of Definition 2.1 we can find a u_k for any given l_k, and there exists a l_{k+1} for any u_k, from Condition (b) of Definition 2.1. Therefore, the above sequence exists for any asynchronous iteration.

Now, we will show that the upper values of the above integer pairs (u_k's) have the property that the sequence of iterates of $(F, x(0), J, C)$ satisfies

$$x(j) \in A^k \quad \text{for } j \geqslant u_k. \tag{2}$$

This will be done by mathematical induction.

(i) In this first step our goal is to prove

$$x(j) \in A^0 \quad \text{for } j \geqslant 0. \tag{3}$$

We already know that (3) is true for $j = 0$, since $x(0)$ is in D. Assume that it is true for $0 \leqslant j < p$. Let z be the vector with components $z_i = x_i(s_i(p))$ for $i = 1, 2, \ldots, n$. From Definition 2.1 we have two cases: either $x_i(p) = x_i(p-1)$ or $x_i(p) = F_i(z)$. In the first case, $x_i(p) \in A_i^0$, since $x_i(p-1) \in A_i^0$ by the above assumption. In the second case, the inequality $s_i(p) < p$ and the assumption implies $z \in A^0$. Then from Definition 2.2 we obtain $F(z) \in A^0$. In both cases we have $x_i(p) \in A_i^0$, which means that $x(p) \in A_1^0 \times A_2^0 \times \cdots \times A_n^0$ and leads to (3).

So far, we have proved (2) for $k = 0$, which is the first step of mathematical induction, and now we need to prove it for $k > 0$.

(ii) Assume that (2) holds for $0 \leqslant k < q$, and again let z be the vector with components $z_i = x_i(s_i(j))$. If $l_q \leqslant j \leqslant u_q$, from the fourth property of the definition of $\{(l_k, u_k)\}$ sequence, $s_i(j) \geqslant u_{q-1}$, and from the assumption we have $z \in A^{q-1}$, which implies that $F(z) \in A^q$. On the other hand, from the third property, for each $i = 1, 2, \ldots, n$, we can find a j_i between l_q and u_q, such that $i \in J_{j_i}$. Therefore, $x_i(j_i) = F_i(z)$ for some j_i in this interval, which means that

$$x_i(j_i) \in A_i^q \quad \text{for } i = 1, 2, \ldots, n \text{ and } l_q \leqslant j_i \leqslant u_q,$$

and since $u_q \geqslant j_i$, we have

$$x_i(j) \in A_i^q, \quad j \geqslant u_q.$$

This shows that (2) holds for $k = q$, and by induction it holds for every integer, k. Since k can be chosen as arbitrarily large, and therefore A^k can be made arbitrarily small, we obtain

$$\lim_{j \to \infty} x(j) = \xi,$$

which is the desired result. □

Theorem 2.3 states that, in order to show that a given asynchronous iteration converges, it is sufficient to verify that the corresponding operator is asynchronously contracting. However, in practice, it may not be easy to verify that Definition 2.2 applies when we deal with real cases. While Definition 2.2 and Theorem 2.3 are applicable to any iteration defined on any set S^n, we would like to start with more easily verifiable conditions for specific and useful classes of asynchronous iterations. When S is the set of reals, the criteria given by Baudet [4] is simpler, and it implies Definition 2.2. The following theorem gives another criteria for convergence. It is applicable to all cases where the synchronized version of the iteration is known to converge.

Before stating the theorem, we need to introduce some additional notations that will be used in the rest of the paper.

Notation. $\leqslant$ is an ordering relation on the elements of S. Let $x, x' \in S^n$. We write $x \leqslant x'$ if $x_i \leqslant x_i'$ for all $i = 1, 2, \ldots, n$. Similarly, $x < x'$ if $x \leqslant x'$ and $x \neq x'$.

Theorem 2.4. *An asynchronous iteration $(\boldsymbol{F}, x^0, J, C)$ corresponding to $\boldsymbol{F}$, which is defined in S^n, converges to a unique fixed point, ξ, if the following conditions hold:*

(a) *there exists an ordering relation, $\leqslant$, on the elements of S, such that if $x \leqslant x'$, then $\boldsymbol{F}(x) \leqslant \boldsymbol{F}(x')$, and also $\boldsymbol{F}(x) \leqslant x$ for $x, x' \in S^n$, and*

(b) *the synchronous iteration corresponding to $\boldsymbol{F}$ and starting with $x^0 \geqslant \xi$ (denoted by the sequence $\{x^k\}$) converges to ξ.*

Proof. First, define

$$D = \{x_1 \mid \xi_1 \leqslant x_1 \leqslant x_1^0\} \times \cdots \times \{x_n \mid \xi_n \leqslant x_n \leqslant x_n^0\},$$

and let

$$A^p = \{x \mid \xi \leqslant x \leqslant x^p\} \quad \text{for all } p = 0, 1, 2, \ldots;$$

then,

$$\{\boldsymbol{F}(x) \mid x \in A^p\} \subseteq \{x \mid \xi \leqslant x \leqslant x^{p+1}\} = A^{p+1}.$$

Therefore,

$$x \in A^p \quad \text{implies} \quad \boldsymbol{F}(x) \in A^{p+1},$$

and

$$A^{p+1} \subseteq A^p.$$

Also, since

$$\lim_{k \to \infty} x^k = \xi,$$

then

$$\lim_{k \to \infty} A^k = \{\xi\}.$$

All the conditions of Definition 2.2 are met. Therefore, $\boldsymbol{F}$ is asynchronously contracting in D, and from Theorem 2.3, any asynchronous iteration $(\boldsymbol{F}, x(0), J, C)$ converges to ξ. □

Another condition for convergence is given below, for the cases where the domain of the operator, $\boldsymbol{F}$, is finite, and each update of the ith component uses the most recent value of the same component.

Theorem 2.5. *An asynchronous iteration $(\boldsymbol{F}, x^0, J, C)$ corresponding to $\boldsymbol{F}$, and which satisfies*

$$s_i(j) = j - 1 \quad \textit{for all } i \in J_j,$$

converges to a unique fixed point, ξ, if there is an ordering relation such that

$$\xi \leqslant \boldsymbol{F}(x) \leqslant x \quad \text{for } \xi \leqslant x \leqslant x^0$$

and

$$x = \boldsymbol{F}(x) \quad \text{iff} \quad x = \xi,$$

where $\boldsymbol{F}$ is defined in S^n, that contains a finite number of elements.

Proof. First, we will show

$$\xi \leqslant x(j) \leqslant x(j-1) \leqslant x^0, \quad j = 1, 2, \ldots \tag{4}$$

by mathematical induction.

(i) The first step is to prove

$$\xi \leqslant x(1) \leqslant x(0) = x^0 \tag{5}$$

that means (4) is true for $j = 1$.

From the definition of $x(j)$ only 2 cases are possible:

(a) $x_i(1) = x_i(0)$, which satisfies the ith component of (5), or

(b) $x_i(1) = F_i(x(0))$, which implies

$$\xi_i \leqslant x_i(1) \leqslant x_i(0)$$

using the property

$$\xi \leqslant \boldsymbol{F}(x) \leqslant x \quad \text{for } \xi \leqslant x \leqslant x^0.$$

For each $i = 1, 2, \ldots, n$ the above arguments are valid and therefore (5) follows.

(ii) Suppose (4) is true for $j = 1, 2, \ldots, q-1$ and let us show it for $j = q$. Again, we have 2 cases for $x_i(q)$:

(a) $x_i(q) = x_i(q-1)$ from which ith component of (4) follows, or,

(b) $x_i(q) \leqslant F_i(z)$, where $z_i = x_i(q-1)$ and $z_k = x_k(s_k(q))$ (for $i \neq k$). From $s_k(q) \leqslant q-1$ and from the hypothesis,

$$\xi \leqslant z \leqslant x^0.$$

Therefore,

$$\xi \leqslant \boldsymbol{F}(z) \leqslant z \leqslant x^0 \quad \text{and} \quad \xi_i \leqslant x_i(q) \leqslant x_i(q-1) \leqslant x_i^0$$

from which we obtain (4).

So far, we have proved that $\{x(j)\}$ is monotically decreasing. Therefore, since S^n contains a finite number of elements, there exists a finite M such that

$$x(j) = \xi' \quad \text{for } j \geqslant M,$$

i.e. $\{x(j)\}$ converges to ξ'.

From the conditions of Definition 2.1, for all i there exists an integer $m_i \geqslant M$ such that

$$x_i(m_i) = F_i(\xi').$$

Therefore,

$$\boldsymbol{F}(\xi') = \xi'$$

which implies $\xi = \xi'$ and $\{x(j)\}$ converges to ξ. □

3. Scene-labeling problem

This section illustrates the application of Theorem 2.5 to the problem of the discrete scene-labeling described in [14].

The following definitions are drawn from [14].

Let $A = \{a_1, \ldots, a_n\}$ be the set of objects to be labeled, and let $\Lambda = \{\lambda_1, \ldots, \lambda_m\}$ be the set of possible labels. For any given object a_i, not every label in Λ may be appropriate. Let Λ_i be the set of labels which are compatible with object a_i, $i = 1, \ldots, n$.

For each pair of objects (a_i, a_j), some labels may be compatible, while others are not. Let $\Lambda_{ij} \subseteq \Lambda_i \times \Lambda_j$ be the set of compatible pairs of labels; thus $(\lambda, \lambda') \in \Lambda_{ij}$ means that it is possible to label a_i with label λ, and a_j with label λ'. If a_i and a_j are independent, then there are no restrictions on the possible pairs of labels that they can have, so that $\Lambda_{ij} = \Lambda_i \times \Lambda_j$.

By a *labeling* $\boldsymbol{L} = (L_1, \ldots, L_n)$ of A, we mean an assignment of a set of labels $L_i \subseteq \Lambda$ to each $a_i \in A$. The labeling is *consistent* if for all i, j we have

$$\big(\{\lambda\} \times L_j\big) \cap \Lambda_{ij} \neq \emptyset \quad \text{for all } \lambda \in L_i.$$

We say that a labeling $\boldsymbol{L} = \{L_1, \ldots, L_n\}$ contains another labeling $\boldsymbol{L}'$ if $L_i' \subseteq L_i$ for $i = 1, \ldots, n$. The *greatest consistent labeling*, $\boldsymbol{L}^\infty$, is a consistent labeling such that any other consistent labeling is contained in $\boldsymbol{L}^\infty$.

According to this model, the discrete relaxation procedure operates as follows. It starts with the initial labeling $\boldsymbol{L}(0) = \{\Lambda_1, \ldots, \Lambda_n\}$. During each step, we eliminate from each L_i all labels λ, such that $(\{\lambda\} \times L_j) \cap \Lambda_{ij} = \emptyset$ for some j. Thus we discard a label λ from object a_i if there exists an object a_j such that no label compatible with λ is assigned to a_j. If for all j's, such that $j = 1, \ldots, n$, and $j \neq i$, there exists a label $\lambda' \in L_j$ and λ' compatible with λ, then we keep the label λ in L_i.

We shall refer to the operation executed at each iteration as Δ. Therefore, for each iteration, k, $\boldsymbol{L}^{k+1} = \Delta(\boldsymbol{L}^k)$ can be decomposed as

$$\begin{aligned} L_1^{k+1} &= \Delta_1\big(L_1^k, \ldots, L_n^k\big) \\ &\vdots \\ L_n^{k+1} &= \Delta_n\big(L_1^k, \ldots, L_n^k\big), \end{aligned}$$

where Δ_i is the operator that discards from L_i any label λ, such that

$$\big(\{\lambda\} \times L_j\big) \cap \Lambda_{ij} = \emptyset \quad \text{for some } j.$$

In other words, Δ_i discards from L_i any label λ, if for at least one a_j there is no label $\lambda' \in L_j$ which is compatible with λ.

Lemma 3.1 (Rosenfeld et al. [14]). $\boldsymbol{L}^\infty \subseteq \Delta(\boldsymbol{L}) \subset \boldsymbol{L}$ *for* $\boldsymbol{L}^\infty \subset \boldsymbol{L} \subseteq \boldsymbol{L}(0)$.

Corollary 3.2. *An asynchronous iteration* $(\Delta, \boldsymbol{L}(0), J, C)$ *such that* $s_i(j) = j - 1$ *for all* $i \in J_j$ *converges to* $\boldsymbol{L}^\infty$.

Proof. Lemma 3.1, together with the fact that

$$\Delta(\boldsymbol{L}) = \boldsymbol{L} \quad \text{iff} \quad \boldsymbol{L} = \boldsymbol{L}^\infty,$$

satisfies the condition of Theorem 2.5. □

In an asynchronous multiprocessor implementation of the scene-labeling algorithm, each processor is assigned a subset of the objects to classify. The relaxation process does not require any synchronization. A process can freely access the set of labels currently associated with objects processed by different processors.

It can also be shown that the condition of Theorem 2.5 is satisfied for the iteration corresponding to the fuzzy model of scene-labeling described in [14].

4. An implementation example

In this section, we shall describe an implementation of the discrete scene-labeling algorithm on multiprocessor computers with distributed global memory. We first briefly explain the model architecture that will be used.

4.1. Model architecture

The multiprocessor architecture onto which we will map the scene-labeling algorithm is similar to the one given in [12]. There are N processing elements (PEs) executing asynchronously. Each PE has a CPU, a private memory, and a part of the global memory. Private memory can only be accessed by the PE that owns it. The part of the global memory that resides in a PE can be directly accessed by that PE via the local bus. However, any other PE can also access it by using the global bus. The advantage of distributing the global memory over the PEs is improved *locality* of accesses and therefore reduced bus traffic: the data that is expected to be most frequently used by a PE can be allocated to that PE's private memory, in which case the communication between the PEs over the global bus is reduced.

4.2. A mapping of the labeling problem onto the model architecture

For the sake of simplicity, we assume in this discussion that only one component, Δ_i, of the operation Δ defined in the previous section is performed in a given processor, PE_i. This is not necessary: more than one component can be assigned to a single processor by simply 'folding' the computation described in the following.

Because of the definition of Δ_i, the Λ_{ij}'s for all j's need to be stored in PE_i. These data are private to PE_i. In our implementation scheme, the label set L_i is also stored in the physical memory associated with PE_i.

In a first, straightforward implementation of Δ_i, PE_i could compare each element in L_i with the other L_j's which are stored in other processors to find out if there is consistency. In this case, the label set L_i must be allocated to the global part of the memory of PE_i. A more efficient scheme is possible from the following considerations.

Suppose that PE_j has discarded the label λ' from L_j during a certain iteration, and that PE_i is informed of this change. From this point on, PE_i uses the new value of L_j (i.e. $L_j - \{\lambda'\}$) to find a label $\lambda \in L_i$, such that

$$\big(\{\lambda\} \times \big(L_j - \{\lambda'\}\big)\big) \cap \Lambda_{ij} \neq \emptyset. \tag{6}$$

Observing that (6) is equivalent to

$$\big(\{\lambda\} \times L_j\big) \cap \big(\Lambda_{ij} - R\big) \neq \emptyset,$$

where $R = \{(x, \lambda') \,|\, (x, \lambda') \in \Lambda_{ij}\}$, leads to the following improvement: PE_i does not need L_j to compute the next iteration of L_i. Instead, updating Λ_{ij} whenever PE_i has the information that λ' is discarded from L_j gives the same result: this is performed by discarding all pairs in Λ_{ij} ending with λ'.

Now, we shall explain the data structure of our implementation. Processor PE_i stores the list of labels, Λ_i, in its private memory. Each label, λ, in Λ_i has a flag associated with it, and it is set if and only if $\lambda \in L_i$. Therefore, deleting a label, λ, from L_i is accompanied by resetting its flag. Λ_{ij}'s for all j's are also stored in PE_i. Each (λ, λ') pair in Λ_{ij} has a pointer associated with it, which points to the flag of the label, λ', that is in PE_j. This pointer is useful to check whether λ' is in L_j or not. If it is not, (λ, λ') is discarded from Λ_{ij}. Note that the only shared

memory elements in this scheme are the flags, and since accessing a shared flag can be done indivisibly (i.e., in one memory access) there is no need for critical sections.

Based on the above considerations, the steps taken by the processor PE_i at each iteration, assuming that the initial loading is done at the beginning, are:

For each $\lambda \in L_i$,

1. Take the first Λ_{ij} in the list, and for each element (λ', λ'') of Λ_{ij},
   (a) If $\lambda' \neq \lambda$, then get the next pair.
   (b) If $\lambda' = \lambda$, then check whether $\lambda'' \in L_j$; i.e., check the flag of λ''. If it is set, then go to Step 2. If it is not, then delete (λ', λ'') from Λ_{ij} and get the next pair in Λ_{ij}.
   (c) It the list, Λ_{ij}, is exhausted (i.e., there is not a pair, (λ', λ'') in Λ_{ij}, such that $\lambda' = \lambda$), then delete λ from L_i, and reset the flag associated with it.
2. Repeat Step 1 for all Λ_{ij}'s.

5. Conclusion

In this paper we have introduced sufficient conditions for the convergence of asynchronous iterations. The condition of Theorem 2.3 is more general than the one given by Baudet and can be applied to discrete asynchronous iterations. Baudet's condition is a special case and is applicable to numerical data. Theorem 2.4 is applicable to all asynchronous iterations on any type of data, provided that the synchronous version of the algorithm converges to a unique solution. Finally, Theorem 2.5 is applicable to any asynchronous iteration on data that can take a finite set of values. This latter case is the case of many symbolic algorithms. We have applied Theorem 2.5 to the scene-labeling relaxation algorithm. The efficiency of the implementation results from the absence of any form of synchronization, even of critical sections. In practice this implies that no processor is ever slowed down because of interactions with other processors.

A difficult issue not addressed in this paper is the rate of convergence. The rate of convergence of asynchronous numerical iterations is very complex to predict [4]. The effectiveness of asynchronous iterations can only be established if the rates of convergence of the synchronized algorithm and of its asynchronous counterpart are similar. This comparison can be done experimentally, and we will have to wait for more experimental work as shared memory multiprocessors become more and more available to the research community. Encouraging results for numerical algorithms have been already published [3,4,8].

Another problem worthy of research is techniques (both software and hardware) to stop the computation. In the implementation described above of the scene-labeling relaxation algorithm, each processor may stop temporarily if no change occurs in its label set in one iteration. This does not mean, however, that the computation is finished. Because of some future label removals in other processors, the processor may have to restart. It is only when all the processors have stopped that the computation can be safely terminated in each processor. This condition can be determined by checking global variables. Access to these global variables should remain asynchronous. While there are many ways to solve this problem, the most efficient technique and the impact of this problem on the efficiency of the computation have not been determined.

References

[1] G.R. Andrews et al., Concepts and notations for concurrent programming, *Comput. Surveys* **15** (1983) 3–43.

[2] T.S. Axelrod, Effects of sychronization barriers on multiprocessor performance, *Parallel Comput.* **3** (1986) 129–140.

[3] R.H. Barlow and D.J. Evans, Synchronous and asynchronous iterative parallel algorithms for linear systems, *Comput. J.* **25** (1982) 56–60.
[4] G.M. Baudet, Asynchronous iterative methods, *J. ACM* **25** (1978) 226–244.
[5] D. Chazan and W. Miranker, Chaotic relaxation, *Linear Alg. Appl.* **2** (1969) 199–222.
[6] M. Dubois and F.A. Briggs, Performance of synchronized iterative processes in multiprocessor systems, *IEEE Trans. Software Engrg.* **8** (1982) 419–431.
[7] M. Dubois and F.A. Briggs, An approximate analytical model for asynchronous processes in multiprocessors, *Proc. 1982 Parallel Processing Conference* (1982) 290–297.
[8] D.J. Evans, Parallel S.O.R. iterative methods, *Parallel Comput.* **1** (1984) 3–18.
[9] R.M. Haralick and L.G. Shapiro, The consistent labeling problem: Part 1, *IEEE Trans. Pattern Anal. Machine Intell.* **1** (1979) 173–184.
[10] H.T. Kung, Synchronized and asynchronous parallel algorithms for multiprocessors, in: J.F. Traub, ed., *Algorithms and Complexity: New Directions and Recent Results* (Academic Press, New York, 1976) 153–200.
[11] J.M. Ortega and W.C. Rheinboldt, *Iterative Solution of Nonlinear Equations in Several Variables* (Academic Press, New York, 1970).
[12] L. Philipson et al., A communication structure for multiprocessor computer with distributed global memory, *Proc. 10th International Symposium on Computer Architecture* (1983) 334–339.
[13] F. Robert, *Discrete Iterations* (Springer, Berlin, 1986).
[14] A. Rosenfeld et al., Scene labeling by relaxation operations, *IEEE Trans. Systems, Man Cybernet.* **6** (1976) 420–433.
[15] A. Üresin and M. Dubois, Generalized asynchronous iteration, *Proc. CONPAR '86: Conference on Algorithms and Hardware for Parallel Processing* (1986) 272–278.
[16] R.G. Voigt, Where are the parallel algorithms?, *Proc. NCC* **54** (1985) 329–334.