JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **34,** 66–81 (1996)
ARTICLE NO. 0046

Effects of Asynchronism on the Convergence Rate of Iterative Algorithms[1]

AYDIN ÜRESIN[2]

Platinum Software Corporation, 195 Technology, Irvine, California 92718-2402

AND

MICHEL DUBOIS[3]

Department of Electrical Engineering—Systems, University of Southern California, Los Angeles, California 90089-2562

In multiprocessor systems, iterative algorithms can be implemented synchronously or asynchronously. Unfortunately, few guidelines exist to make a choice. In this paper, we compare the execution times of an asynchronous iterative algorithm and of its synchronous counterpart. Synchronization overhead and communication times are neglected in order to focus on the effect of asynchronism on the convergence rate. Under some assumptions, we derive an analytical model. In this model, Q tasks with identical and independently distributed execution time distributions execute on P processors. We show the effects of execution time fluctuations, of the number of processors and tasks, of the scheduling policy and of the amount of coupling among iterate components. The models show that the asynchronous iteration may be up to twice as slow as the synchronous one. This worst case is achieved when the processing time fluctuations are very small, and the coupling among components is strong.

1. INTRODUCTION

The goal of parallel processing is to reduce execution time by running concurrent and cooperating processes on multiple processors. Parallel processes communicate and synchronize by accesses to shared writable data or by message-passing mechanisms. The way in which these communication and synchronization mechanisms are supported has a large impact on system performance.

In this paper we focus on iterative algorithms, in which an operator is applied to a set of data repetitively until convergence. In a multiprocessor, consecutive iterations are separated by a *barrier synchronization*. Barrier synchronization is a common synchronization mechanism often supported by the hardware and which defines a point in the execution of a parallel program such that no process can pass the barrier unless all other processes have done so [1]. Barriers are not needed in asynchronous implementations of iterative algorithms and therefore an asynchronous iteration has the potential of being tolerant of very large communication and synchronization latencies. Unfortunately, the convergence rate of an asynchronous iteration may be less than that of its synchronous counterpart because processors never wait for fresh values of iterate components. The goal of this paper is to evaluate the effects of asynchronism on the convergence rate of iterative algorithms by simulations and analytical models.

Most iterative algorithms, both for numerical and for nonnumerical data, can be described by an operator F repetitively applied to some data x of dimension $n > 1$ starting with an initial value $x(0)$. Such algorithms are represented as

$$x_i(k+1) = F_i(x(k)), \quad k = 0, 1, ..., \tag{1}$$

for all $i = 1, 2, ..., n$, and where $x(k) = (x_1(k), x_2(k), ..., x_n(k))$ is the value of $x = (x_1, x_2, ..., x_n)$ at the kth iteration. In a multiprocessor different components can be computed in parallel by different processors in each iteration. The strict application of (1) results in a *synchronous* iterative algorithm (*synchronous iteration*), in which each processor has to wait for all the new values of all components before starting the next iteration. By contrast, in *asynchronous* iterative algorithms (*asynchronous iterations*), processors are not required to wait for the new values produced by other processors. Rather the components are updated at arbitrary time instances and a new value of the ith component is produced at some time instance t according to

$$x_i(t) = F_i(u^i(t)),$$

where the values of the components of $u^i(t)$ were generated before t. $u^i(t)$ is called *the input at t for the ith component.*

[1] This research was supported by the National Science Foundation under Grant CCR-9222734.

[2] E-mail: auresin@platsoft.com.

[3] E-mail: dubois@paris.usc.edu.

0743-7315/96 $18.00

$x(t)$ denotes the value of x at time t. For notational convenience, we assume that the computations start at $t = 0$ and that the initial data $x(0)$ are generated at $t = -\infty$. Notice that this definition covers a large class of algorithms, including synchronous iterations.

Without any restriction on the timing of computations in an asynchronous iteration, it is not possible to guarantee its convergence. The minimum requirements for the convergence of an asynchronous iteration are as follows.

1. For each time instance t of the execution of an asynchronous iteration, there exists an instance $t' > t$ such that all the components are updated between t and t' (*progress of updates*).

2. For each time instance t of the execution of an asynchronous iteration, let $\tau(t)$ be the largest time instance such that all the updates after t never use input components generated prior to $\tau(t)$. Then the sequence $\tau(t)$ must tend to infinity as t goes to infinity (*progress of inputs*).

Asynchronous iterations satisfying these conditions are called *totally asynchronous* [5]. Such asynchronous iterations only exclude starving computations.

There has been a considerable amount of work in the literature on the convergence conditions for totally asynchronous iterations [2, 4–7, 9, 13–15, 18, 20–23, 25–27]. The most general result can be stated as follows [5, 7, 27].

PROPOSITION 1. *Let $\{X(k)\}$ be a sequence of sets such that*

- $X(k) = X_1(k) \times X_2(k) \times \cdots \times X_n(k)$, *for all k.*
- $\{\xi\} \subseteq \cdots \subset X(k+1) \subset X(k) \subset \cdots \subset X(0) \subseteq X$ *for all k, and furthermore all sequences $\{z(k)\}$ such that $z(k) \in X(k)$ for all k converge to ξ, where ξ is the unique fixed point of F in $X(0)$.*
- $F(x) \in X(k+1)$, *for all k and $x \in X(k)$.*

Then, all asynchronous iterations corresponding to F and starting with some $x(0)$ in $X(0)$ converge to ξ.

We will not give a complete proof here, but for later reference we will state the key intermediate result as a lemma. Let $\{\varphi(k)\}$ be a nondecreasing sequence of time instances starting with zero such that

- All the components are updated within the time interval $(\varphi(k), \varphi(k+1)]$.[4]
- All the updates after $\varphi(k)$ use input components generated after $\varphi(k-1)$.

Also, let $\varphi_i(k)$ be the first time within the interval $(\varphi(k), \varphi(k+1)]$ at which the ith component is updated. Then the following lemma holds.

LEMMA 1. *Under the conditions of Proposition* 1: $t \geq \varphi_i(k) \Rightarrow x_i(t) \in X_i(k+1)$.

This lemma can be proven by mathematical induction. For more details, the reader should refer to [27]. Since all the components are updated within the interval $(\varphi(k), \varphi(k+1)]$, $x(t)$ indefinitely enters $X(k+1)$ after $\varphi(k+1)$, i.e.,

$$t \geq \varphi(k) \Rightarrow x(t) \in X(k) \qquad \text{for all } k > 0. \tag{2}$$

Any sequence $\{(\varphi(k), \varphi(k+1)]\}$ of time intervals satisfying the above property (2) is called a *pseudo-cycle sequence*.

Various models have been used in the literature to compare the performance of asynchronous iterations with their synchronous counterpart. In [6] and in [7, Sect. 6.3.5], two important classes of iteration operators, namely monotonic and contracting operators, were considered in the context of message-passing systems and under the assumption that the messages in the system are received in the same order as they are sent. The analysis is applied to iterations with various degrees of coupling among iterate components. It is concluded that an asynchronous iteration cannot be slower than its synchronous counterpart, provided that the computation of each component takes one time unit and that the synchronous version suffers delays of at least one time unit between iterations due to the communication of recent data. It is not clear how the results of [6] can be applied when the computation times are not constant. In [12], which deals with shared-memory multiprocessors, the convergence issues are ignored but conflicts for memory and for critical sections are modeled. In [25], it is shown that, under a set of conditions, asynchronous distributed algorithms have good communication complexity in the sense that the message traffic they generate is not excessive.

There are many factors affecting the efficiency of multiprocessor algorithms and it is impossible to include them all in the same model. For example, in shared-memory systems, the performance of iterative algorithms is affected by the following factors: overhead caused by the execution of synchronization primitives, processor allocation and scheduling overhead, software lock-out on critical sections, memory access conflicts, and timing of shared memory updates. In this paper, we introduce a model to evaluate the effects of variable computation times on the timing of memory updates and on the convergence rate of asynchronous iterations.

Section 2 describes the basic model of computations of this paper. Sections 3 and 4 refine the model of Section 2 for the cases of strongly coupled and of partially coupled computations, and present the results of simulations and of an analytical model based on tasks with independent and identical distribution of their execution times. Finally, the contributions of the paper are highlighted in Section 5.

2. GENERAL MODEL

In this section, we describe the general computational model of this paper. The notations and assumptions stated here will hold throughout the paper without further mentioning explicitly.

[4] $(a, b]$ denotes the set of real numbers $\{x \mid a < x \leq b\}$.

2.1. Architecture Model

We consider a multiprocessor system with P processors. The set of processors is $P = \{P_0, P_1, ..., P_{P-1}\}$. The components of x are partitioned into Q subsets ($Q \geq P$). Each partition forms a *task* which is an indivisible unit of execution: from its beginning until its end, a task is executed by the same processor without interruption. The set of tasks is $T = \{T_0, T_1, ..., T_{Q-1}\}$. Each task is executed infinitely many times and includes fetching the current value of the shared data, computing the components, and updating the global store. A time interval that covers all these activities of a task T_q is called a *task interval* of T_q, and is denoted TI_q. An asynchronous iterative algorithm (asynchronous iteration) corresponds to an allocation of each task to infinitely many time slots of available processors (Fig. 1). We make the following assumption.

Assumption 1.

- The length of each task interval is finite and in a finite period of time all the components are updated.
- The processors never stay idle except at the synchronization points between the iterations. The execution time of the synchronization primitives and the overhead due to other types of synchronization (if any) such as critical sections are ignored.
- The task intervals of the same task never overlap in time.

Although no particular architecture is specified, this model is more useful for shared memory systems. The first condition is required to enforce total asynchronism. The second condition eliminates the kind of overhead which is associated with the synchronous iteration. The major restriction is the third (nonredundancy) condition. Although the results are derived under this condition, they are still good approximations when a small amount of redundancy is allowed.

2.2. Condition on the Iterations

Let F be the iteration operator, let $x(0)$ be the initial vector, and let $X(M)$ be the solution space for every iteration in this paper.

Assumption 2. $\{X(k)\}$ satisfies the conditions of Proposition 1, and furthermore, the synchronous iteration starting with $x(0)$ takes exactly M iterations.

We now show that the class of iteration operators satisfying Assumption 2 includes nonexpansive ($F(x) \leq x$) and monotone ($x \leq y$ implies $F(x) \leq F(x)$) operators, where inequalities are componentwise. Without loss of generality, let M be the number of synchronous iterations and define $X(k) = \{x \mid F^k(x(0)) \geq x \geq F^M(x(0))\}$, $k < M$. The claim follows immediately. References [7, 27] include several problems that can be solved by such iterations, including shortest path, constraint satisfaction and transitive closure.

Let $\{\varphi_{\min}(k)\}$ be the minimum pseudo-cycle sequence corresponding to an asynchronous iteration A. In other words, for all pseudo-cycle sequences $\varphi(k)$ and for all k, $\varphi_{\min}(k) \leq \varphi(k)$. The execution time of an iteration is the time it takes for the iterates to enter $X(M)$ indefinitely. From the definition of pseudo-cycle sequence, the execution time is M minimum pseudo-cycles in the asynchronous case and, from Assumption 2, it is also M iterations in the synchronous case. Without Assumption 2 on F the results of this paper would not always hold because, in general, the synchronous version might "luckily" hit the solution in an arbitrarily small number of iterations.

Let $\phi_{\min}$ and I denote the average pseudo-cycle time of $\{\varphi_{\min}(k)\}$ and the average iteration time of the synchronous version, respectively. These averages are taken over all the pseudo-cycles (or iterations) of one execution. The ratio between the execution times of the asynchronous and synchronous versions of the same algorithm is called the *slowdown factor* and is denoted by S. Then $S = \phi_{\min}/I$.

The value of $\phi_{\min}$ depends on the coupling among iterate components. In the following section, we analyze the case of strong coupling.

3. STRONGLY COUPLED ITERATION OPERATORS

For a given iteration operator F, the worst we can expect is that, in each pseudo-cycle, each task needs the outcomes of all the tasks executed in the previous pseudo-cycle in order to make any progress toward the solution. This is the situation where the "coupling" among the tasks is the strongest possible.

3.1. Description of the Model

We define *strong coupling* by the following assumptions. The first assumption restricts the computational model and

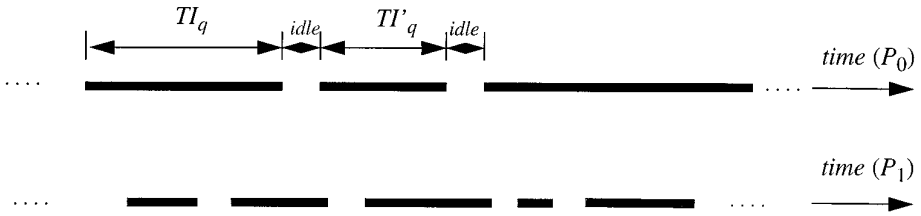

FIG. 1. An asynchronous iteration.

the second describes the condition on the iteration operator F.

Assumption 3. The components computed in each task interval TI_q are released only at the end of TI_q. Furthermore, the values of input components used for these computations are those available at the start of TI_q.

The *age* $A_i(x_i)$ of component x_i is defined as the largest integer such that $x_i \in X_i(A_i(x_i))$ and the age $A(x)$ of x is the minimum age of all its components. When a task is allocated to several components, the age of the task is also defined as the minimum age of its components.

Assumption 4. For all x and i the iteration operator F satisfies

$$A_i(F_i(x)) = \min_j \{A_j(x_j)\} + 1 = A(x) + 1.$$

For strongly coupled iterations, we define the following sequence $\{\varphi'(k)\}$.

DEFINITION 1. $\{\varphi'(k)\}$ is the increasing sequence of time instances starting with zero such that for all k, the time interval $(\varphi'(k), \varphi'(k+1)]$ is the smallest interval that covers at least one task interval of each task.

Let ϕ' be the average pseudo-cycle length of $\{\phi'(k)\}$. The following proposition shows that under the above assumptions, ϕ' is the exact estimate of $\phi_{\min}$.

PROPOSITION 2. $t < \varphi'(k) \Rightarrow x(t) \notin X(k)$, *for all* $k > 0$.

Proof. We can prove this claim by induction. Since there exists a component x_i which is not updated before $\varphi'(1)$, the age of x_i and therefore the age of x is 0 prior to $\varphi'(1)$. This proves the proposition for $k = 1$. Suppose that it also holds for $k = 1, 2, \ldots, l$. Consider the first task interval TI_q on a processor P_p covered by the $(l+1)$th pseudo-cycle of $\{\varphi'(k)\}$. Because of Lemma 1 each component x_i updated in TI_q indefinitely enters $X_i(l+1)$ right after TI_q. If components x_i updated in TI_q are also updated right before TI_q within the $(l+1)$th pseudo-cycle, the input for this update is generated earlier than $\varphi'(l)$; therefore it is not in $X(l)$ and the updated component x_i at this instance cannot be in $X_i(l+1)$. This means that x does not enter $X(l+1)$ before the processors execute at least one instance of each task in the lth pseudo-cycle. This proves the proposition for $k = l+1$ and the claim follows. ■

Figure 2a shows an iteration of a synchronous algorithm for five tasks and three processors. A pseudo-cycle of a corresponding asynchronous iteration is in Fig. 2b. We observe that in the synchronous case all the processors restart right after the previous iteration, whereas in the asynchronous case only one processor restarts right after the previous pseudo-cycle (P_0 in the figure). From Proposition 2, the tasks starting before the kth pseudo-cycle and completing within the kth pseudo-cycle do not increment the ages of their components and they must be considered "wasted."

We can identify three parts in the total processor time $\phi(k) \cdot P$ spent in the kth pseudo-cycle of an asynchronous iteration, as follows.

- Part 1: $\phi_1(k) \cdot P$. Let $d_p(k)$ be the time from the beginning of a pseudo-cycle k until the first processor initiation on P_p. Then, $\phi_1(k) \cdot P$ is defined as the sum of all $d_p(k)$'s. (For example, in Fig. 2b, it is equal to $d_1(k) + d_2(k)$). Note that there are up to $(P-1)$ nonzero $d_p(k)$'s.
- Part 2: $\phi_2(k) \cdot P$. This is the total "useful" work in the pseudo-cycle including the first full execution of each task. (For example, in Fig. 2b, it is the total processor time covered by T_0 through T_4).

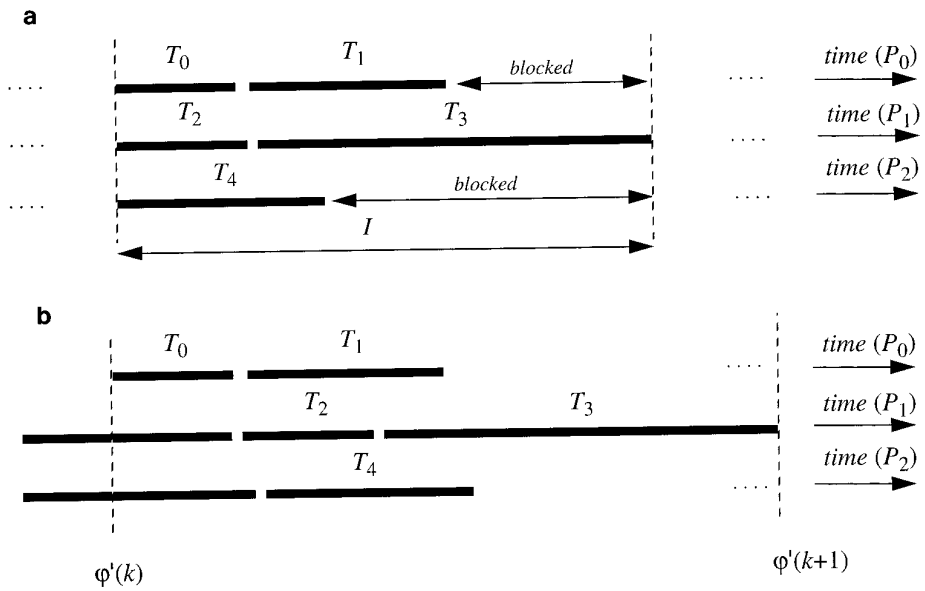

FIG. 2. Iteration time and pseudo-cycle time.

• Part 3: $\phi_3(k) \cdot P$. This is defined as the sum of all $e_p(k)$'s, where $e_p(k)$ is the time wasted by P_p after the completion of its useful work in the kth pseudo-cycle.

In the rest of the paper, ϕ_1, ϕ_2, ϕ_3, d_p, and e_p denote the average quantities taken over k. Each of the three components of the pseudo-cycle must be estimated either analytically or by simulations.

3.2. Probabilistic Analysis

In this section, we use the approach in [19] to derive analytical estimates of ϕ', I, and S. For the analytical derivations, the following restriction on the distribution of the task intervals must hold and will be assumed throughout the section.

Assumption 5. The task interval lengths are independent and identically distributed (i.i.d.) random variables drawn from the same distribution function with increasing failure rate (IFR) with mean μ and variance σ^2.

A distribution function $G(x)$ is said to be IFR if $G(0) = 0$ (i.e., it is the distribution function of a positive random variable) and if, for all $x_0 > 0$, $[1 - G(x + x_0)]/[1 - G(x)]$ is monotone decreasing in x. When G has a density g then this condition is equivalent to $g(x)/[1 - G(x)]$ being monotone increasing in x. IFR distributions include exponential, gamma with $\mu/\sigma \geq 1$, Weibull, truncated normal (normal constrained to be positive), uniform on the interval $(0, A)$ for any $A > 0$, and constant $= A$ for any $A > 0$.

In practice a task interval is made of a succession of operations. The length of the task interval is the sum of the (random) times taken by each individual operation. In computations that are not data dependent, such as iterative solutions of linear systems of equations, the distribution of the task interval lengths tends to a truncated normal distribution as a consequence of the central limit theorem. (We can consider each task interval as a batch of small tasks as in [19].) When the fluctuations are dependent on the data, Assumption 5 may be difficult to justify. It may not even be appropriate to assume that task interval lengths are random.

3.2.1. Synchronous Case

When $Q = P$, the average synchronous iteration time is the mean of the largest order statistics among P random variables [10]. In general, let us denote the expected value of the maximum of P independent variates (the Pth order statistic) each with cumulative distribution function $G(x)$ by $X^{(1)}_{P:P}$ (the superscript will be clarified later). Then, $X^{(1)}_{P:P} = E(\text{Max}(x_1, x_2, ..., x_P))$.

Assuming, to simplify, that $G(x)$ is continuous (the argument also holds in the case of a discrete distribution), the cumulative distribution function of the Pth order statistic is $F_P(t) = \text{Prob}\{\text{all } x_i \leq t\} = G^P(t)$ and since x is nonnegative, $X^{(1)}_{P:P} = \int_0^\infty t dG(t) = \int_0^\infty [1 - G^P(t)]\, dt$.

Consider a family of distributions $G(t)$ with mean μ and variance σ^2 such that $G(t) = G_0([t - \mu]/\sigma)$. Then, $X^{(1)}_{P:P} = \int_0^\infty [1 - G_0^P([t - \mu]/\sigma)]\, dt = \mu + \sigma \int_0^\infty [1 - G_0^P(y)]\, dy$ and

$$X^{(1)}_{P:P} = \mu + \sigma \times O_{P:P} \tag{3}$$

where $O_{P:P}$ is the Pth order statistics [10] among P samples drawn from the interval length distribution with mean 0 and variance 1.

For a *uniform* distribution, $O_{P:P} = \sqrt{3} \times [P - 1]/[P + 1] \approx \sqrt{3}$ for large values of P. For an *exponential* distribution with a positive offset, $O_{P:P} = \sum_{1<k\leq P} (1/k) \approx \log P - 0.42$ for large values of P. For a *normal* distribution, no analytical formula exists but $O_{P:P}$'s can be found in tables or by recurrence relations [10]. Some bounds exist; in particular, for very large values of P, $O_{P:P} \approx \sqrt{2 \log P}$.

Since the normal distribution can take negative values we need to truncate the distribution at 0 in order to avoid generating negative task interval lengths. Unfortunately, the family of distributions generated by truncating a normal distribution with different values of μ and σ^2 does not satisfy the condition leading to (3). $O_{P:P}$ or $X^{(1)}_{P:P}$ must be estimated with a simple simulation program repetitively generating P samples from the distribution and taking the average of the maximum values.

When $Q > P$, the exact value of the average synchronous iteration time cannot be obtained analytically in general. We use a similar approach as in [19] to obtain an approximation. We divide the iteration time into two parts: T_1 and T_2. T_1 is the time between the start of the iteration and the start of the last task interval in that iteration and T_2 is the rest of the iteration time until the end of the last task interval. Since all processors are busy until T_1, since the distribution is IFR, and since the total work in each iteration must be less than the work done up to T_1 plus $P\mu$ we have that $Q\mu \leq P\mu + PE(T_1)$ and therefore $[(Q - P)/P]\, \mu \leq E(T_1)$. On the other hand, $E(T_2) \leq X^{(1)}_{P:P}$. We approximate the expected value of the synchronous iteration time by

$$E(I) \approx \frac{Q - P}{P} \mu + X^{(1)}_{P:P}. \tag{4}$$

Given a distribution of the interval lengths, (4) is correct for the case $Q = P$, but is an approximation otherwise [11, 24]. The first term of (4) is a low estimate of the time until the start of the last task, and the second term is a high estimate of the time between the start of the last task and the completion of the last task. Depending on which term is dominant (4) may therefore underestimate or overestimate the time taken by a synchronous iteration. We have run a large number of simulations [28] and have stressed the model with very large values of P, Q and

c_v, the coefficient of variation.[5] Some of these results are reported in the figures; (4) is always within 10% of the simulated synchronous iteration times.

3.2.2. Asynchronous Case

Since different schedulings of the task yield different asynchronous outcomes, we restrict the scheduling in the analytic model and in most simulations, as follows.

Assumption 6. Age Scheduling: When a processor becomes free at any time instance τ in a pseudo-cycle k, it always selects for the next execution a task T_q, which has not yet started in the kth pseudo-cycle, unless all the tasks have been started.

Age scheduling can be implemented by time-stamping each task descriptor in the scheduling queue with the time when the latest execution of the task was started and by assigning a higher priority level to the tasks with lower timestamps. This scheduling strategy ensures that each execution contributes to increasing the age of the iterate vector. It optimizes the speed of the asynchronous iteration and is very easy to enforce in a practical situation.

Under age scheduling, a pseudo-cycle is the time it takes to complete Q tasks. Remember that $d_p(k)$ is the time from the beginning of a pseudo-cycle k until the next task interval on P_p. The IFR condition assures that d_p is stochastically bounded by the average time it takes to complete a whole task [19]. Since there are up to $(P-1)d_p(k)$'s for each k, an upper bound on the overhead at the beginning of a pseudo-cycle is given by $(P-1)$ task executions. Therefore, a pseudo-cycle takes no more than $(Q + P - 1)$ task executions. Using the same decomposition into T_1 and T_2 as above for the synchronous case, we obtain

$$\begin{aligned} E(\phi') &\approx \frac{P-1}{P}\mu + \frac{Q}{P}\mu + (X^{(1)}_{P\cdot P} - \mu) \\ &= \frac{Q-1}{P}\mu + X^{(1)}_{P\cdot P}. \end{aligned} \tag{5}$$

This formula is an approximation for any value of P and Q. The first term is the overhead due to the use of outdated information in component updates, the second term is the useful work, and the third term is caused by fluctuations in the computation times.

Although approximate, (5) yields pseudo-cycle times that are well within 10% of the simulated times [28], except when $Q = P$ and when the coefficient of variation of the interval length distribution is very high. The reason is that the task interval starting the pseudo-cycle can be very short so that the processor allocated to it does useless work until T_1. Therefore the estimate for $E(T_1)$ is too low, with an error larger than 10%, and the error increases with the coefficient of variation and the number of processors.

[5] The coefficient of variation is the ratio between the standard deviation and the mean. We denote it by c_v. Therefore, $c_v = \sigma/\mu$.

When $Q = P$ and the coefficient of variation is high, we use a different approximation than (5), as follows. The useful work in each pseudo-cycle is the task of the processor starting the pseudo-cycle plus one task from every other processor which must first complete its task in progress at the beginning of the pseudo-cycle. Because of the IFR condition, the time taken by each processor to complete its useful work in the pseudo-cycle is upper bounded by the time taken by two consecutive task intervals. Therefore

$$E(\phi') \le X^{(2)}_{P\cdot P}, \tag{6}$$

where the superscript (2) indicates that the order statistic is for the variate equal to the sum of two random samples drawn from the distribution of task interval lengths.

3.2.3. Slowdown Factor

Combining (4) and (5) we obtain

$$S \approx 1 + \frac{(P-1)\mu}{(Q-P)\mu + PX^{(1)}_{P\cdot P}} \le 1 + \frac{P-1}{Q}. \tag{7}$$

This approximation on the slowdown factor is always greater than 1 but less than 2. It decreases as σ increases and its maximum value of $1 + (P-1)/Q$ is reached for small values of the fluctuation, so that $Q\mu \gg P(X^{(1)}_{P\cdot P} - \mu)$. For large fluctuations, the slowdown factor is close to 1.

The upper bound in (7) is an upper bound on an approximation. In fact, for $Q > P$, it is not possible to prove that the upper bound on the slowdown is 2, because of the lack of a good lower bound for the synchronous iteration time. However, for $Q = P$ (an important case in practice), we know the exact value of the synchronous iteration time.

From (6), we then conclude

$$S \le \frac{X^{(2)}_{P\cdot P}}{X^{(1)}_{P\cdot P}} \le 2. \tag{8}$$

That $X^{(2)}_{P\cdot P} \le 2X^{(1)}_{P\cdot P}$ can be easily understood from the following argument. The simulation to obtain $X^{(1)}_{P\cdot P}$ draws consecutive batches of P samples and selects the maximum value in each batch. The simulation to obtain $X^{(2)}_{P\cdot P}$ draws two batches of P samples from the same distribution and adds them pairwise. Clearly, the sum of the maxima in two consecutive batches of P samples cannot be less than the maximum of the values obtained by adding the samples in the two batches pair-wise.

3.3. Simulations

We have run simulations to verify the accuracy of (4), (5), (6), (7), and (8). The simulations were run for three stochastic distributions of the task interval lengths [28]. The first distribution is a truncated normal: we draw a sample from a normal with mean 1 and variance σ; if the sample is less than zero, we set it to zero. To stress the

model, we have used very high coefficients of variation of the original normal distributions, from 0.01 to 100. The second distribution is a uniform distribution over [0, 2) with no offset. The third distribution is an exponential with no offset and with mean equal to 1. The number of processors was varied from $P = 2$ to $P = 2048$.

The simulations use Assumptions 3 and 4 (strong coupling). For $Q = P$, the tasks are allocated statically to processors. For $Q > P$, the tasks are allocated dynamically, according to age scheduling. Every time a processor is released, the set of tasks is scanned to find the current value of age $A(x)$ (the minimum value of all the tasks' ages). One of the tasks with the lowest age value is allocated to the processor. When the task interval is completed on the processor the age of the task is set to its age at the start of the interval plus one. We run the simulations for 1,000 consecutive iterations (synchronous case) and for 1,000 pseudo-cycles (asynchronous case). These simulations are very computation intensive (of the order of Q^2) so that simulations with very large number of tasks are too slow.

We have also derived the analytical models. When $Q > P$, we always use the models of (4), (5), and (7); $X_{P:P}^{(1)}$ is found by the formulas for the exponential and uniform distributions whereas for the truncated normal distributions we derive the order statistic through simulation. We use the same model for $Q = P$ and for the truncated normal distributions with low coefficients of variation ($c_v = 0.01$, 0.1, or 0.3). For $Q = P$ we use the models of (4), (6), and (8) for the uniform, exponential, and truncated normal distributions with high coefficient of variation ($c_v = 1$, 5, 10, or 100). In these models, $X_{P:P}^{(2)}$ is obtained by taking the average of the maxima in 1,000 batches of P samples where each sample is the sum of two independent samples drawn from the interval length distribution (the complexity of this computation of the order of P.)

Figure 3 shows that the estimated pseudo-cycle time predicted by the model is very close to the ones obtained by simulation and Fig. 4 displays the slowdown factor for the same cases.

The simulation results agree with the conjecture that the slowdown is upper-bounded by $1 + P/Q$. The model is not as good as for the pseudo-cycle times because the error of the slowdown factor sometimes accumulates the error made in the estimation of both the synchronous iteration time and the asynchronous pseudo-cycle time when $Q > P$. Large errors (15%) are observed for $Q = 128$ and $c_v > 1$ [28]. Such large coefficients of variation (which imply that the fluctuation can be more than three times the mean) are unlikely in practice. The upper bound on the slowdown factor decreases with the number of tasks and with the coefficient of variation. As Q/P increases, the behaviors of the synchronous and asynchronous iterations tend to converge and the slowdown factor goes down to 1. The worst-case value for the slowdown factor is close to 2, for $P = Q$ and $c_v \approx 0$. This is because the pseudo-cycle time is very sensitive to small fluctuations of the task interval lengths whereas the synchronous iteration time is not.

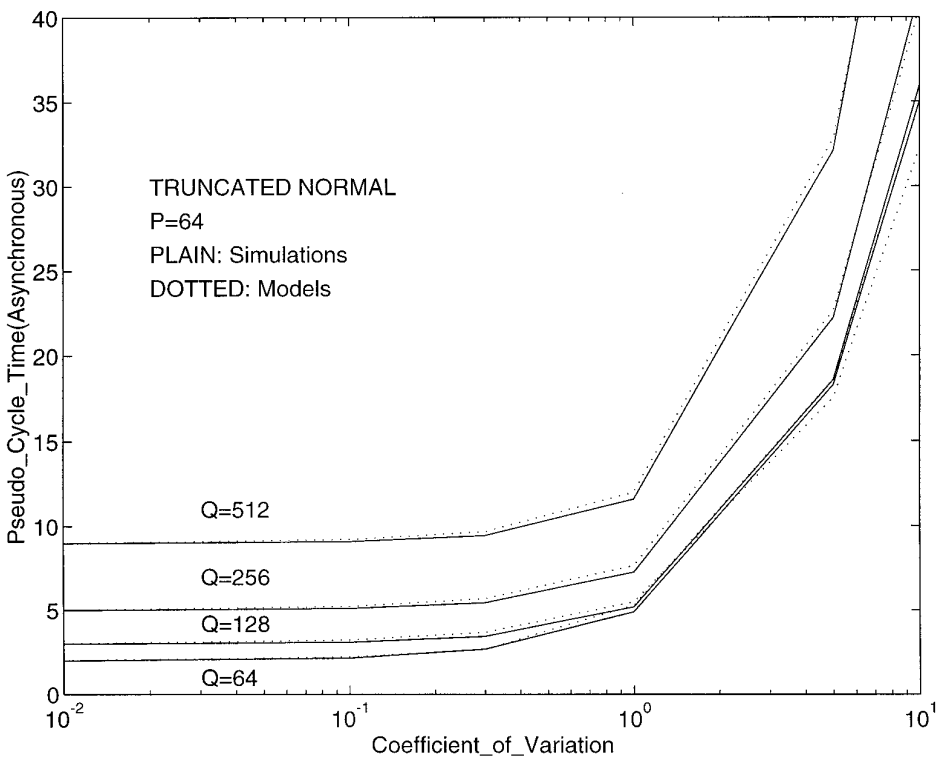

FIG. 3. Pseudo-cycle time for $P = 64$ and truncated normal distribution.

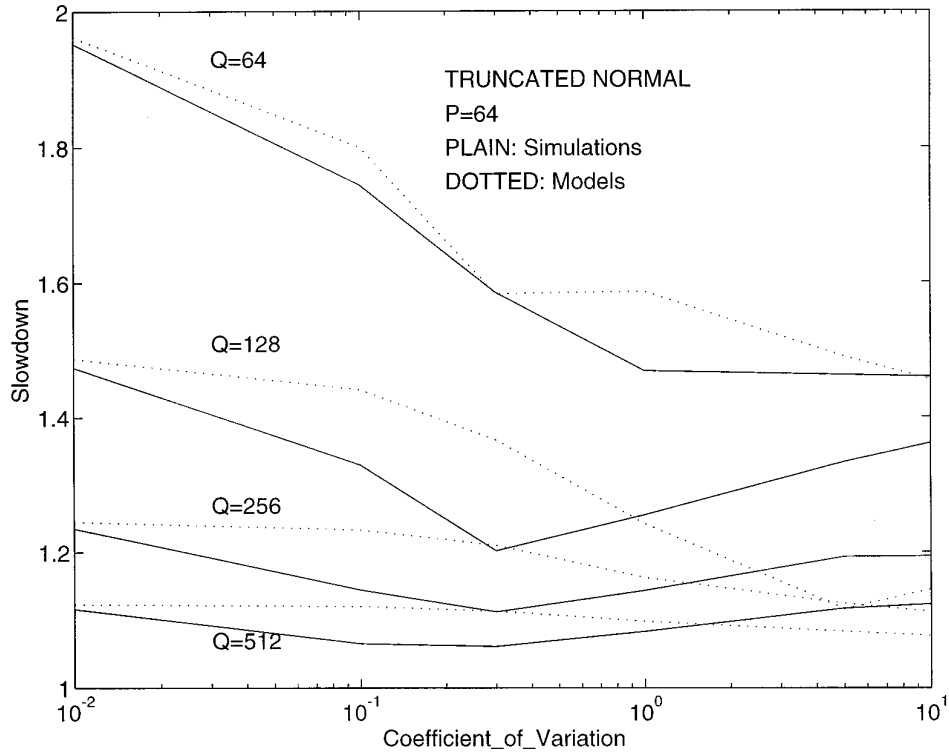

FIG. 4. Slowdown factor for $P = 64$ and normal distribution.

Figure 5 shows the slowdown factor as a function of the number of processors and for different distributions for the case of $Q = P$. The simulations (up to 2K processors) are compared to the models (up to 64K processors). We see that the model does not always agree with the simulations for small numbers of processors. The model is worse for the uniform distribution and best for the exponential distribution. Significant differences in the trends of the

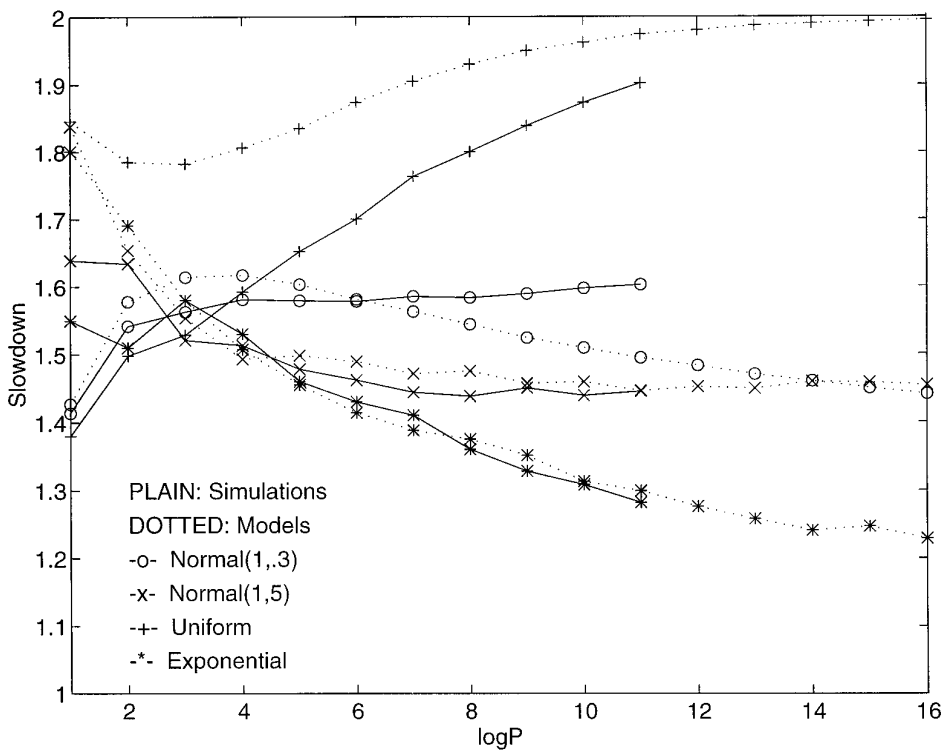

FIG. 5. Slowdown factor as a function of P ($Q = P$).

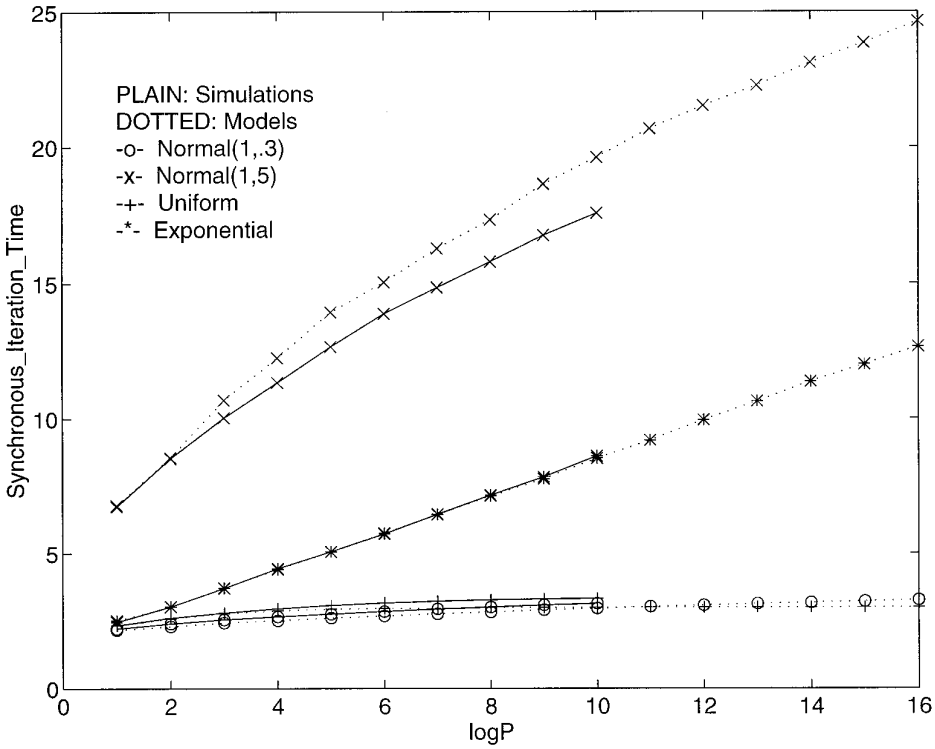

FIG. 6. Synchronous iteration times as a function of P ($Q = 2P$).

curves exist among the three distributions, which indicates that a single model based on a distribution-free parameter such as the coefficient of variation cannot explain all behaviors. All curves tend towards a constant less than two as the number of processors increases.

As shown in Figs. 6 and 7, the synchronous iteration time and the pseudo-cycle time are predicted very accurately by the models for the case where $Q = 2P$. From these figures, we see that for distributions with lower coefficients of variations (such as the uniform and the truncated normal with

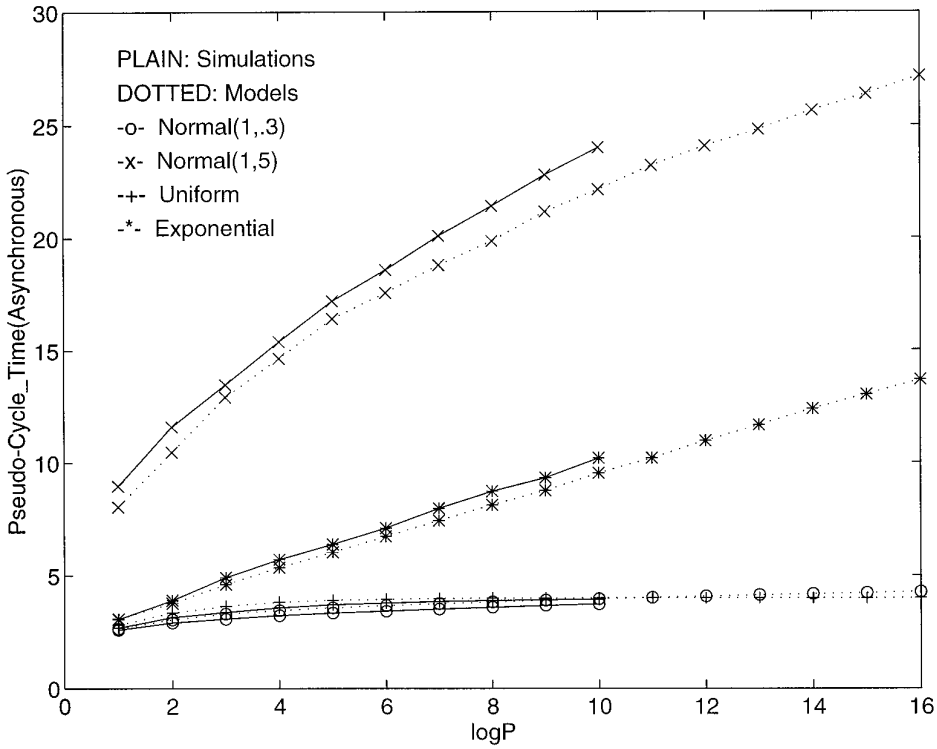

FIG. 7. Pseudo-cycle times as a function of P ($Q = 2P$).

$c_v = 0.3$), the fluctuation has very little effect on the synchronous iteration time and on the pseudo-cycle times. For distributions with large coefficients of variation (such as the exponential and the truncated normal with $c_v = 5$) the effect of the fluctuation dominates. The computation times in the synchronous and asynchronous cases increase at a logarithmic rate with P. However, the slowdown factors (not shown) remain roughly constant as a function of P, within a band between 1.15 and 1.35. Because the range is so narrow, the models are not very successful at capturing the small differences between the slowdown factors.

3.4. Effect of the Number of Tasks

As the number of tasks increases for a given number of processors, the difference between the time taken by the synchronous and asynchronous iterations narrows. Moreover, integer values of Q/P correspond to good load balancing, in which case the efficiency of the synchronous iteration is best. Figure 8 shows that this effect is particularly visible at very low coefficients of variation.

3.5. Effect of the Task Scheduling Policy

In all the simulations above, the next task to run is the one which currently does not run and has the lowest age, according to Assumption 6 (age scheduling). In practice, the task scheduling algorithm may be different. It is not difficult to come up with scheduling strategies which could considerably slow down or even stall completely the convergence of the asynchronous iteration.

Besides age scheduling, we have run the simulation for FIFO and static policies. FIFO is different from age scheduling because the priorities in the scheduling queue are based on the time when tasks join the queue (instead of the time when they leave the queue). In the static scheduling policy, each processor is assigned Q/P tasks and executes them in turn. This policy has the lowest runtime scheduling overhead. As shown in Fig. 9, differences in scheduling have little effect on the convergence rate of the asynchronous algorithm when the coefficient of variation of the execution times is low. However, for large coefficients of variations, the differences among scheduling strategies become significant.

4. PARTIALLY COUPLED ITERATION OPERATORS

So far, we have assumed that the execution of a task depends on the outcome of *all* the tasks executed in the previous pseudo-cycle in order to make any progress towards the solution. In general, this is very pessimistic because there are rarely dependencies between all pairs of tasks.

4.1. General Model

Let $S_q(k)$ be the set of tasks on which the computation of T_q depends in the kth pseudo-cycle in order to increment its age (including T_q itself).

Assumption 7. For all x and i the iteration operator F satisfies

$$A_i(F_i(x(k))) = \min_{j \in S_i(k)} \{A_j(x_j(k))\} + 1.$$

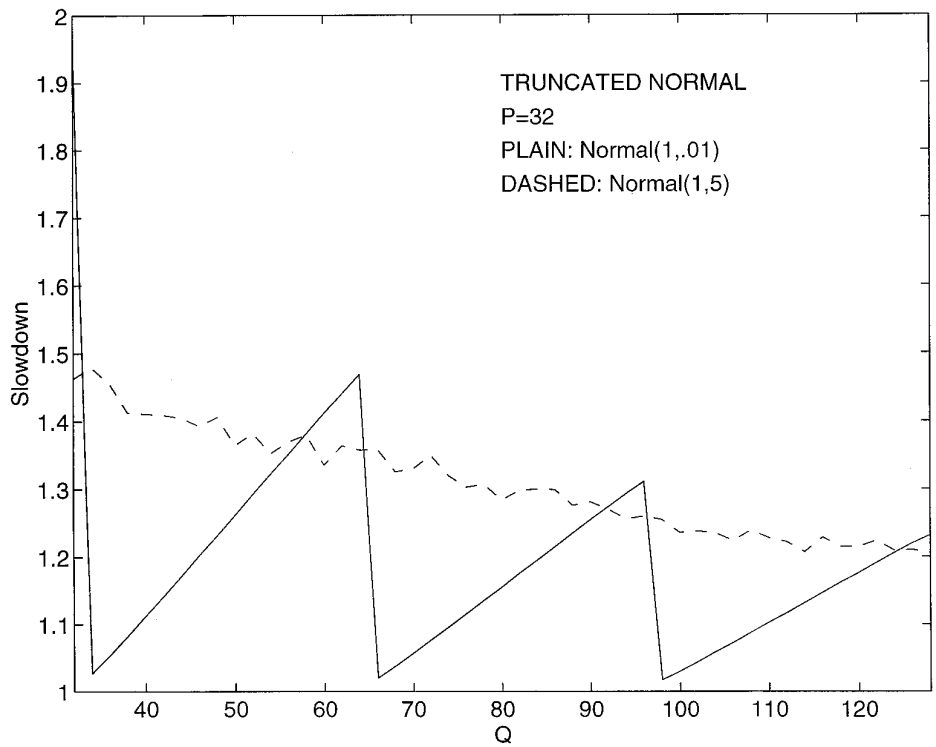

FIG. 8. Slowdown factor as a function of Q ($P = 32$).

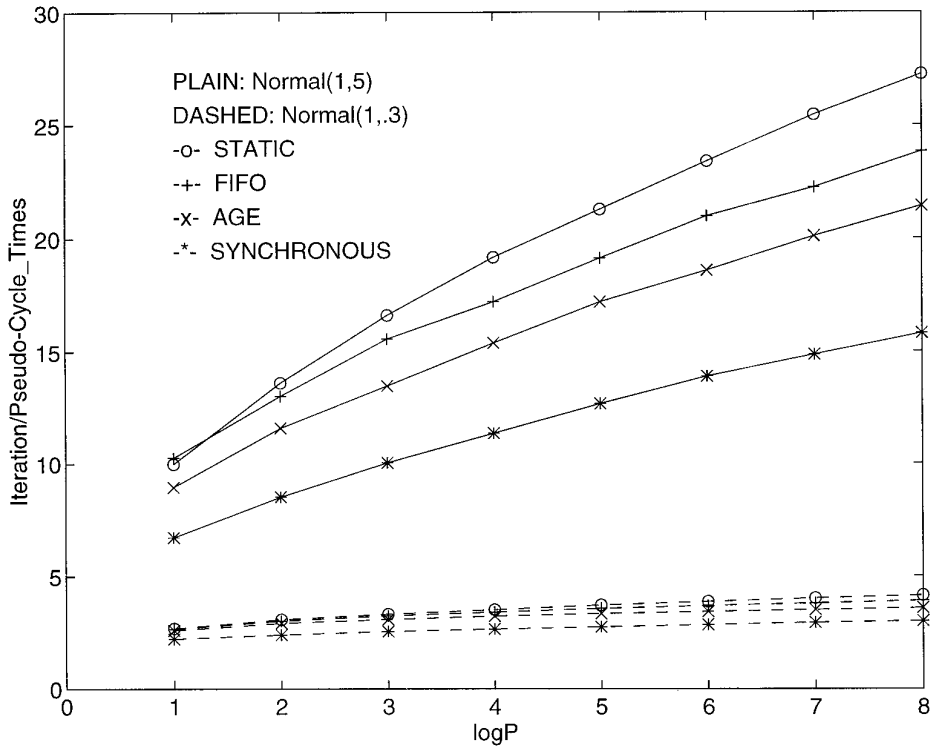

FIG. 9. Effects of scheduling.

Taking this limited coupling into account, we can slightly modify the definition of the pseudo-cycle sequence to obtain lower upper bounds.

DEFINITION 2. For all q, $\{\varphi''^{(q)}(k)\}$ is defined as a nondecreasing sequence of time instances, starting with the initial time, such that the interval $(\beta''^{(q)}(k), \varphi''^{(q)}(k+1)]$ is the smallest time interval that covers one task interval of T_q, where

$$\beta''^{(q)}(k) = \operatorname*{Max}_{j \in S_q(k)} \{\varphi''^{(j)}(k)\}$$

The pseudo-cycle sequence $\{\varphi''(k)\}$ is defined as $\varphi''(k) = \mathrm{Max}_q\{\varphi''^{(q)}(k)\}$

After $\varphi''^{(q)}(M)$, the iterates updated by task T_q indefinitely enter domain $X(M)$ and after $\varphi''(M)$ the iteration has converged, whereas the algorithm with barrier synchronization takes M iterations to enter domain $X(M)$. $\varphi'^{(q)}(k)$ is a special case of $\varphi''^{(q)}(k)$ when $S_q(k)$ includes all tasks.

If the time between $\beta''^{(q)}(k)$ and the beginning of the first execution of T_q after $\beta''^{(q)}(k)$ is denoted by $r^{(q)}(k)$ and if $g^{(q)}(k)$ is the duration of this first execution of T_q, then

$$\varphi''^{(q)}(k+1) = \operatorname*{Max}_{j \in S_q(k)} \{\varphi''^{(j)}(k)\} + r^{(q)}(k) + g^{(q)}(k). \quad (9)$$

When task interval lengths fluctuate, the convergence rate of the asynchronous iteration will be very high provided the average distance $r^{(q)}(k)$ is kept large so that task intervals of tasks in $S_q(k)$ have very small probability to overlap with the following task interval of task T_q. This is the case, for example, when $Q > P$.

EXAMPLE 1. Consider the computation in Fig. 10. There are three data iterates and each iterate x_i is statically assigned to P_i. Each task interval length is constant and

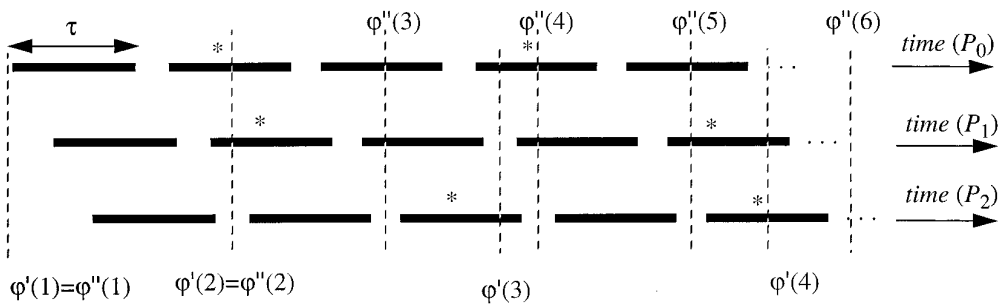

FIG. 10. Example of partial coupling.

equal to τ. Assuming the worst-case scenario, F_0 cannot advance the iterate vector towards convergence without a recent value of x_2, and the second task interval of the first component (x_0) does not contribute to the pseudo-cycle from $\varphi'(1)$ to $\varphi'(2)$. This is because the definition of $\varphi'(k)$ assumes the worst case in which F_0 cannot advance the iterate towards convergence without a recent value of x_2. In this way, 2 out of 5 global computation intervals of each component are wasted and the average pseudo-cycle time is $(5/3)\tau$. The wasted tasks are marked by * in the figure. Now, suppose that the dependencies among the components are restricted as follows:

$$x_0 = F_0(x_0)$$
$$x_1 = F_1(x_0, x_1)$$
$$x_2 = F_2(x_0, x_1, x_2).$$

Since F_0 depends only on itself, the computation of x_0 in a pseudo-cycle may overlap with the computation of x_2 in the previous pseudo-cycle. As a result, every computation of x_0 makes progress towards the solution. Similarly, every computation of x_1 and x_2 also makes progress. Consequently, the average minimum pseudo-cycle time is upper bounded by τ. The corresponding sequence $\{\varphi''(k)\}$ is shown in the figure.

Example 1 suggests that $\{\varphi''(k)\}$ sometimes yields a much smaller bound on the execution time than $\{\varphi'(k)\}$, and it can be concluded that, when the coupling is weak, the pseudo-cycle time of an asynchronous implementation may be much less than the iteration time of its synchronous counterpart. To see this in general, define $\varphi'^{(q)}(k)$ as the value of $\varphi''^{(q)}(k)$ when $S^{(q)}(k)$'s are replaced in (9) by the set T of all tasks. Clearly, $\varphi'(k)$ can be written as

$$\varphi'(k) = \operatorname*{Max}_q \{\varphi'^{(q)}(k)\}.$$

Since $\varphi''^{(q)}(k) \le \varphi'^{(q)}(k)$, $\varphi'(k)$ cannot be less than $\varphi''(k)$. Furthermore, from (9), $\varphi''^{(q)}(k)$ decrease as $S^{(q)}(k)$ gets smaller. It can be concluded that when the coupling is weak, asynchronous implementations are likely to perform better than their synchronous counterpart.

4.2. Self-Coupled Iteration Operators

An optimistic approach to estimate the execution time of an asynchronous iteration is to assume that every task execution makes progress towards the solution. This approach was taken in [3, 11, 12] and is equivalent to assuming that each task is only coupled with itself, i.e., $S^{(q)}(k) = \{q\}$, for all q and k. Let M be the total number of iterations to reach the solution. Then, an asynchronous execution contains exactly $Q \times M$ task intervals. Using the same approach as for the derivations of (5) and (6) we obtain

$$M \times E(\phi) \approx \frac{M \times Q - 1}{P} \times \mu + X^{(1)}_{P:P} \approx \frac{M \times Q}{P} \times \mu.$$

4.3. Performance Comparison

Figure 11 compares the slowdown factor for strongly, partially, and self-coupled iterations using simulations. In

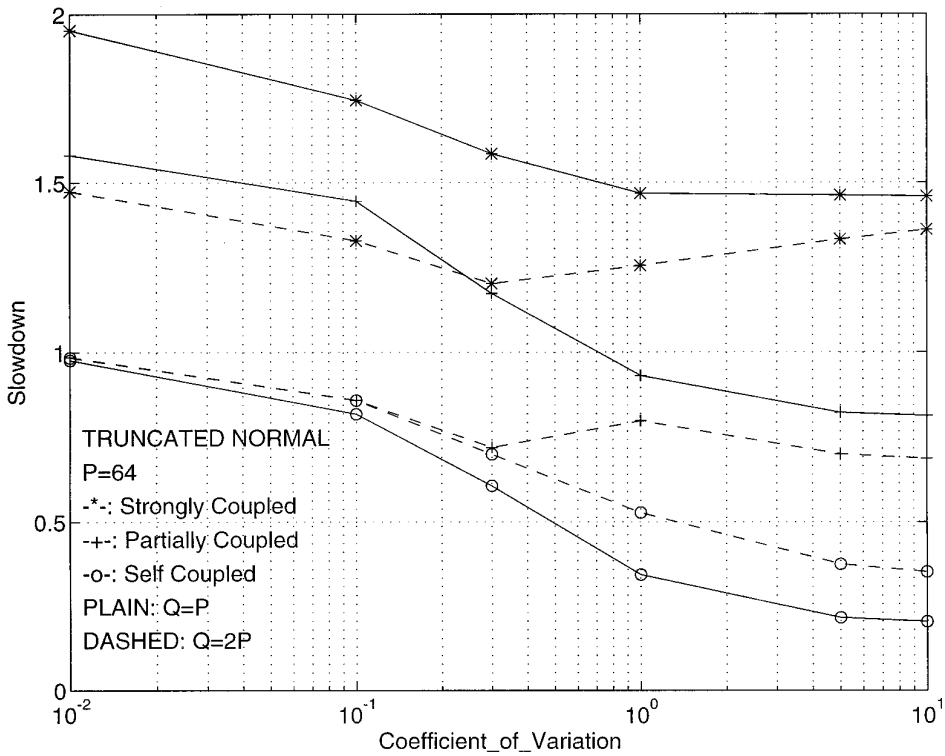

FIG. 11. Comparison of iterations with various degree of coupling ($P = 64$).

all cases we assume that iterates are read at the start of a task interval and are released at the end of a task interval. For the self-coupled iterations, the age of a task is always incremented by one at the end of each task interval. For the partially coupled iteration, we assume that each task T_q depends on components produced by its two neighbors, $T_{(q-1)}$ and $T_{(q+1)}$. Therefore, the age of the iterates updated by T_q is computed at the beginning of the task interval as the minimum of the ages of T_q, $T_{(q-1)}$, and $T_{(q+1)}$ and is incremented at the end of the task interval. We always schedule first one of the tasks with the lowest age (age scheduling).

When $Q = P$, the slowdown of the partially coupled iteration is lower than 1 for $c_v > 0.7$. When $Q = 2P$, the slowdown of the partially coupled iteration is less than one, for all coefficients of variation, and for $c_v < 0.3$, it is as good as for the self-coupled iteration. The self-coupled iteration is always the most efficient and its slowdown is always less than one.

4.4. Discussion

In the case where the sets $S^{(q)}(k)$'s are constant in the whole data domain and are detectable by simple dependency analysis, it is not necessary to use a global barrier synchronization in the synchronous algorithm. Furthermore, if the sets $S^{(q)}(k)$ are small, it is unfair to compare an asynchronous iteration to its synchronous version with global barrier synchronization. Rather, in the synchronous computation, a task T_q should only wait for the completion of the tasks in the set $S^{(q)}(k)$ in order to start its next execution, as advocated in [16]. One problem with this approach is the complexity of synchronizing and scheduling tasks to take advantage of this opportunity.

Another problem arises when the sets $S^{(q)}(k)$'s are small but dynamically changing during the course of the computation. For example, in the consistent labeling problem, dependencies among components are data dependent [17, 26]. In such a situation, the sets $S^{(q)}(k)$ are not known a priori and it is not possible to use the restricted form of synchronization.

We can conclude that the elimination of synchronization points may improve the performance of iterative algorithms significantly when the sets $S^{(q)}(k)$ are small. Unfortunately, we do not yet know how to measure and estimate the sets $S^{(q)}(k)$ in all cases. The worst case is strong coupling. The best case is self coupling. A particular iteration will fall in between these two extremes. As shown in Fig. 11, the slowdown factor of all partially coupled asynchronous iterations with an operator satisfying Assumption 2 and under age scheduling is in the band delimited by the curves for the simulations under strong and self coupling.

4.5. Colorable Iteration Operators

Definition 2 does not rule out the possibility of pseudo-cycles of length zero. We illustrate this possibility by a popular scheme to order component updates, called red–black ordering, in the following example.

EXAMPLE 2. In red–black ordering, each component has a color, red or black, and the computation of the red (black) components depends only on the values of the black (red) components. This computational scheme is shown in Fig. 12 for three processors and six tasks. Three of the tasks update red components and the other three tasks update black components; they are labeled by *r* or *b* (respectively) in the figure. All task intervals are constant and equal to τ. $\varphi''^{(r)}(k)$ ($\varphi''^{(b)}(k)$) is equal to $\varphi''^{(q)}(k)$ given in Definition 2 for red (black) tasks. The first phase computes the red components and makes one unit of progress. The second phase updates the black components using the latest value of the red components and therefore also makes one unit of progress. Effectively, one unit of progress is made in each time period of τ, which means that the average pseudo-cycle time is τ, half the time given by the analysis in Section 3. (According to the definition of $\{\varphi'(k)\}$ a pseudo-cycle should cover all the tasks and in this example should have a length of 2τ).

We can generalize the example as follows. Let $S = \{T_0, T_1, ..., T_{R-1}\}$ be a partition of the set of all tasks T such that the tasks in T_i are only coupled with the tasks in $T_{(i-1) \bmod R}$, for all $0 \le i \le R - 1$. Iteration operators for which the components can be partitioned in this fashion are called *colorable* and the tasks in T_C have color C. In the case of colorable operators, it is sufficient that the kth pseudo-cycle covers the tasks with color k mod R. For such pseudo-cycle sequences all tasks with color k mod R make at least k units of progress in k pseudo-cycles. This partitioning of the set of tasks into R subsets in effect reduces

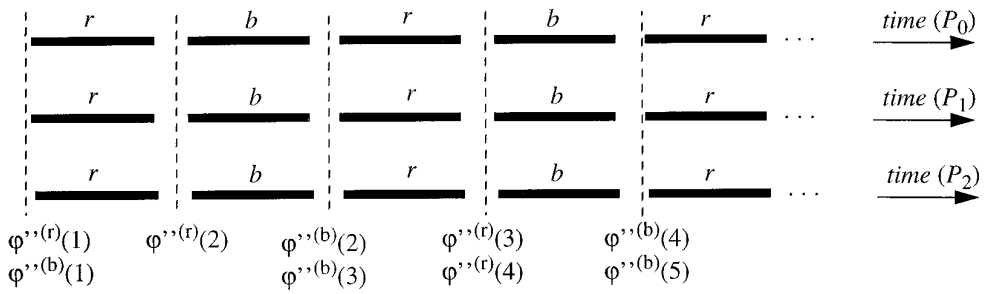

FIG. 12. Red–black ordering.

the number of tasks a pseudo-cycle has to cover. An average pseudo-cycle has to cover Q/R tasks instead of Q.

4.6. Global and Local Computations

In some cases where the dependency among the components is known a priori, it is possible to rearrange the computations so that the critical components that will be used by other tasks are computed and released as soon as possible, before the components that will not be used by other tasks. This reduces the number of tasks that are considered to be wasted.

EXAMPLE 3. Consider the discretized approximation of the Laplace equation

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0.$$

Discretization yields a rectilinear grid with boundary conditions. Each point of the grid is updated successively using the following iteration formula:

$$u_{i,j} = \tfrac{1}{4} \times [u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1}].$$

The next value of each point is computed by taking the average of its four neighbors. We can partition the grid into rectangular regions and order the components such that each task first computes and updates the boundary points of its region (global computation phase), and then processes the interior points (local computation phase).

Since the local computation phase of a task does not interact with other tasks, it should be allowed to overlap with the previous or the next pseudo-cycles. We define the following sequence $\{\varphi'''^{(q)}(k)\}$.

DEFINITION 3. For all q, $\{\varphi'''^{(q)}(k)\}$ is defined as a nondecreasing sequence of time instances, starting with the initial time, such that the interval $(\beta'''^{(q)}(k), \varphi'''^{(q)}(k+1)]$ is the smallest time interval that covers one *global* computation phase of T_q, where

$$\beta'''^{(q)}(k) = \max_{j \in S_q(k)} \{\varphi'''^{(j)}(k)\}$$

and $S_q(k)$ is defined as in Section 4.1. The pseudo-cycle sequence $\{\varphi'''(k)\}$ is defined as $\varphi'''(k) = \max_q\{\varphi'''^{(q)}(k)\}$.

In other words, local computation phases can be considered as idle times in which no significant global work is done. Obviously, $\varphi'''^{(q)} \leq \varphi''^{(q)}(k)$ for all k. It is also clear that for a given task scheduling and task interval lengths, reducing the size of the global computation phases can only reduce $\varphi'''^{(q)}$ and therefore the execution time.

5. CONCLUSION

We now summarize the results derived in this paper. It should be emphasized that we have made a basic assumption on the iteration operator F; it is impossible to obtain results on the convergence rate of neither asynchronous nor synchronous executions for arbitrary iteration operators.

Our computational model allows for the evaluation of asynchronous algorithms in which Q, the number of tasks, may be greater or equal to P, the number of processors. When $Q = P$, the allocation of tasks to processors is static and each processor executes a different task. When $Q > P$ the scheduling of tasks is dynamic and it is possible to design scheduling strategies that will make the convergence of the asynchronous algorithm arbitrarily slow. In our study we have concentrated on the scheduling strategy that maximizes the convergence rate of the asynchronous implementation and which we have called *age scheduling*.

We have developed a simulation approach to evaluate the convergence speed of the asynchronous iteration, under various conditions of coupling among iterate components, from strong coupling to self coupling. Under strong coupling, the asynchronous iteration cannot converge faster than its synchronous counterpart whereas under self coupling it always does. When the coupling is intermediate between self and strong, the convergence rate of the asynchronous iteration may be better, depending on the size of the fluctuations of the task execution times.

To provide insight into the simulation data and to extrapolate the simulation results, we have also developed some analytical models which assume that task execution times are random and independent and drawn from the same IFR distribution. Under these conditions we proved that the asynchronous iteration under strong coupling and with age scheduling is at most twice slower than its synchronous counterpart when $Q = P$. When $Q > P$, we could not prove the same property. However, an approximate model validated in the paper by extensive simulations shows that the asynchronous algorithm can be up to $1 + P/Q$ times slower than its synchronous version. This upper bound is reached when the fluctuations of the task execution times are very small. Therefore, in the important case where each processor is statically allocated to one task, where all tasks are roughly equal in size (good static load balance), and where iterates are strongly coupled, the models in this paper predict that a good speedup through asynchronism is hard to obtain. The reasons are that, for small fluctuations, the penalty of synchronization is low, whereas the convergence rate of the asynchronous version is at its worst.

The simulations covered uniform, exponential, and truncated normal distributions for a wide range of coefficients of variation and of numbers of processors. When P increases we have observed that the ratio of the convergence rates of the asynchronous and synchronous algorithms under strong coupling converges to a constant between 1 and 2 which varies with the execution time distribution. This is a very interesting observation: As P increases, the overhead of the synchronization protocol among processors is likely to dominate and, provided the ratio of the conver-

gence rates does not increase, there should be a system size for which the asynchronous algorithm is better than the synchronous one, even under worst-case conditions.

We have shown that an asynchronous iteration executes faster by delaying the fetching of input data for component updates and by releasing the updated data earlier. As a consequence, we can design more efficient algorithms by ordering the component updates in a task, such that the components needed by other tasks are updated earlier than the ones that are only used locally.

When the dependency among tasks is weak and predictable, we do not need barrier synchronization to implement a synchronous iteration. In this case, only the dependent tasks need to be synchronized. A synchronous implementation with this type of synchronization seems to be a better choice than the asynchronous version. However, when the coupling is not predictable, barrier synchronization is unavoidable in a synchronous implementation. Therefore, the weak and unpredictable coupling exploits the performance advantage of an asynchronous implementation the most, which executes faster with decreasing coupling.

To further investigate the convergence properties of asynchronous iterations will require more experimental work. Many implementations on various machines have been reported in the literature already. Experiments where the number of tasks executed in the asynchronous iteration were less than the number of tasks in the synchronous are relatively rare. A notable exception was found in [8]. In this paper spectacular speedups—which cannot be explained by synchronization overheads alone—are reported for some (nonmonotonic) operators, even when coupling among iterates is strong. We are currently investigating these cases more closely.

The type of information needed to understand the fundamentals behind the convergence rate of asynchronous algorithms is difficult to obtain from an implementation on an actual machine. Besides models such as the one developed in this paper, we believe that a good way to understand asynchronism is to run the algorithms on architectural simulators, in which we can easily observe all timings and change the architectural parameters. The understanding provided by such simulations would allow one to make a decision as to the best synchronization scheme to use in a particular situation. This is one path that our future research in this field will take.

REFERENCES

1. Axelrod, T. S. Effects of synchronization barriers on multiprocessor performance. *Parallel Comput.* **3** (1986), 129–140.
2. Baudet, G. M. Asynchronous iterative methods for multiprocessors. *J. Assoc. Comput. Mach.* **25**(2) (Apr. 1978), 226–244.
3. Brochard, L., Prost, J.-P., and Fauire, F. Synchronization and load unbalance effects of parallel iterative algorithms. *Proc. Int. Conf. on Parallel Processing (ICPP).* 1989, pp. 153–160.
4. Bertsekas, D. P. Distributed dynamic programming. *IEEE Trans. Automatic Control* **AC-27,** (1982), 610–616.
5. Bertsekas, D. P. Distributed asynchronous computation of fixed points. *Math. Programming* **27** (1983), 107–120.
6. Bertsekas, D. P., and Tsitsiklis, J. N. Convergence rate and termination of asynchronous iterative algorithms. *Proc. Int. Conf. on Supercomputing.* 1989, pp. 461–470.
7. Bertsekas, D. P., and Tsitsiklis, J. N. *Parallel and Distributed Computation.* Prentice-Hall, New York, 1989.
8. Bull, J. M., and Freeman, T. L. Numerical performance of an asynchronous Jacobi iteration. *Second Joint International Conference on Vector and Parallel Processing, CONPAR92-VAPP-V.* Springer-Verlag, 1992, pp. 361–366.
9. Chazan, D., and Miranker, W. Chaotic relaxation. *Linear Algebra Appl.* **2** (1969), 199–222.
10. David, H. A. *Order Statistics.* Wiley, New York, 1970.
11. Dubois, M., and Briggs, F. A. Performance of synchronized iterative processes in multiprocessor systems. *IEEE Trans. Software Engrg.* **8**(4) (July 1982), 419–431.
12. Dubois, M., and Briggs, F. A. The runtime efficiency of parallel asynchronous algorithms. *IEEE Trans. Comput.* **40**(11) (Nov. 1991), 1260–1266.
13. El Baz, D. M-functions and parallel asynchronous algorithms. *SIAM J. Numer. Anal.* **27,** 136–140.
14. El Tarazi, M. N. Some convergence results for asynchronous algorithms. *Numer. Math.* **39** (1982), 325–340.
15. Frommer, A. On asynchronous iterations in partially ordered spaces. *Numer. Funct. Anal. Optimization* **12**(3 & 4) (1991), 315–325.
16. Greenbaum, A. Synchronization costs on multiprocessors. *Parallel Comput.* **10,** (1989), 3–14.
17. Haralick, R. M., and Shapiro, L. G. The consistent labeling problem, I. *IEEE Trans. Pattern Anal. Mach. Intelligence* **1**(2) (Apr. 1979), 173–184.
18. Kung, H. T. Synchronized and asynchronous algorithms for multiprocessors. In Traub, J. F. (Ed.), *Algorithms and Complexity: New Directions and Recent Results.* Academic Press, New York, 1976.
19. Kruskal, C. P., and Weiss, A. Allocating independent subtasks on parallel processors. *Proc. Int. Conf. on Parallel Processing (ICPP).* (1984), pp. 236–240.
20. Miellou, J. C. Algorithmes de relaxation chaotiques a retards. *R.A.I.R.O.* **1** (1975), 55–82.
21. Miellou, J. C. Iterations chaotiques a retards, etude de la convergence dans le cas d'espaces partiellement ordonnes. *C.R. Acad. Sci. Paris* **280** (1975), 233–236.
22. Miellou, J. C. Asynchronous iterations and order intervals. In Cosnard, M. (Ed.), *Parallel Algorithms and Architectures.* North-Holland, Amsterdam, 1986, pp. 85–96.
23. Miellou, J. C., and Spiteri, P. Un critere de convergence pour des methodes generales de point fixe. *R.A.I.R.O.* Modél. Math. Anal. Numér. **19** (1985), 645–669.
24. Robinson, J. T. Some analysis techniques for asynchronous multiprocessor algorithms. *IEEE Trans. Software Engrg.* **5**(1) (Jan. 1979), 24–31.
25. Tsitsiklis, N. T., and Stamoulis, G. D. On the average communication complexity of asynchronous distributed algorithms. Technical Report LIDS-P-1986, Laboratory for Information and Decision Systems, 1990.
26. Üresin, A., and Dubois, M. Sufficient conditions for the convergence of asynchronous iterations. *Parallel Comput.* **10** (1989), 83–92.
27. Üresin, A., and Dubois, M. Parallel asynchronous algorithms for discrete data. *Assoc. Comput. Mach.* **37**(3) (1990), 588–606.
28. Üresin, A., and Dubois, M. Effects of asynchronism on the convergence rate of a class of iterations. Technical Report CENG 95-05, U.S.C., 1995.

AYDIN ÜRESIN obtained his BS in electrical engineering from Istanbul Technical University, Turkey, and the MS and PhD degrees in computer engineering from the University of Southern California. From 1990 to 1994, he was an assistant professor at York University, Canada, and at Istanbul Technical University, Turkey. Currently he is involved in software development for business applications at Platinum Software Corporation in Irvine, California.

MICHEL DUBOIS is an associate professor in the Department of Electrical Engineering at the University of Southern California. Before joining U.S.C. in 1984, he was a research engineer at the Central Research Laboratory of Thomson-CSF in Orsay, France. His main interests are computer architecture and parallel processing. He currently leads the RPM project, a project funded by the National Science Foundation. RPM (Rapid Prototyping engine for Multiprocessors) is built with FPGAs (Field Programmable Gate Arrays) and is a hardware platform to implement and compare various multiprocessor systems. Dr. Dubois holds a PhD from Purdue University, an M.S. degree from the University of Minnesota, and an engineering degree from the Faculte Polytechnique de Mons in Belgium, all in electrical engineering. He is a member of the ACM and a senior member of the Computer Society of the IEEE.

Received November 17, 1993; revised February 10, 1995; accepted November 20, 1995.