WORST-CASE ANALYSIS OF ASYNCHRONOUS ITERATIVE ALGORITHMS

TECHNICAL REPORT NO. CENG 89-17

AYDIN URESIN MICHEL DUBOIS

JUNE 1989

This work was performed under a Grant from AT&T

WORST-CASE ANALYSIS OF ASYNCHRONOUS ITERATIVE ALGORITHMS

Aydın Üresin and Michel Dubois

Electrical Engineering-Systems

University of Southern California

Los Angeles, CA 90089-0781, U.S.A.

Phone: (213) 743-8080

E-mail: dubois@priam.usc.edu

Abstract

Iterative algorithms are widely used in both numeric and symbolic computing. The implementation of these algorithms in shared-memory multiprocessors can be synchronous or asynchronous. Since synchronization is a major source of performance degradation in such systems, asynchronous algorithms can benefit from reduced synchronization requirements. Other factors affecting the performance are the contention for shared resources, both software and hardware, and the timing of shared data accesses. In this paper, we present a model useful to analyze the performance of asynchronous iterative algorithms considering the latter factor.

Key words and phrases- iterative algorithms, asynchronous algorithms, shared-memory multiprocessors, synchronization, analysis of algorithms.

1 Introduction

Most iterative algorithms, both for numerical and nonnumerical data, are characterized by an operator F successively applied to some data x starting with the initial value of the data $x(0)$ and can be represented by the following iteration formula :

$$x(k+1) = F(x(k)) \ , \qquad k = 0, 1, \ldots \tag{1}$$

where $x(k)$ is the value of x at the k-th iteration. If x is a vector, i.e., multidimensional data, the computation of its components at each iteration can be performed in parallel. Shared-memory multiprocessors can be used for this purpose [8]. The vector x resides in the shared memory and the components of F are computed by different processors. The computations can be *synchronous* or *asynchronous* [4], [9]. Strict application of the above equation results in a synchronous iterative algorithm, where each processor has to wait for all the other components to be updated before starting the next iteration. In asynchronous iterative algorithms (asynchronous iterations), on the other hand, processors are allowed to continue their computation after an update, and are not required to wait for other processors.

Naturally, the designer of an asynchronous iterative algorithm should guarantee its correctness, i.e., the algorithm should converge to the desired result regardless of the timing of shared memory accesses in individual processors. There has been a considerable amount of work in the literature on the convergence conditions for asynchronous iterations [2]-[5], [10], [12].

Besides correctness efficiency is also a major concern regarding any algorithm and in the case of an asynchronous iteration the greatest interest is in its speed relative to the synchronous version of the same algorithm. In particular, we would like to know the performance effect obtained by simply removing the synchronization points at the end of each iteration of an iterative algorithm.

In shared-memory systems, the performance of iterative algorithms is affected by the following factors:

- Overhead caused by the execution of synchronization primitives (serializing bottleneck),
- Processor allocation and scheduling overheads,
- Software lock-out on critical sections,
- Memory access conflicts, and
- Timing of shared memory updates: some efficiency is lost because of explicit waits in the synchronous case and of implicit delays in utilizing the most recent data in the asynchronous case.

The most deleterious synchronization primitive is the *barrier synchronization* which defines a logical point in the control flow of a parallel algorithm at which *all* processes must arrive before any one of them is allowed to proceed further, and it has been shown in [1] that it may drastically reduce the efficiency for large numbers of processors. The effects of synchronization and of memory access conflicts on the efficiency of synchronous iterative parallel algorithms have also been studied in [6] and [11].

In this paper, we present a performance model for shared-memory multiprocessors, ignoring all the overheads associated with the communication and the memory accesses. The main intention is to understand the effect of the ordering of shared data accesses on the behavior of asynchronous algorithms, in isolation from all the other factors. The model is suitable for shared-memory systems, rather than message-passing systems; it does not allow processors to continue computations after a request for data access is sent. In this respect, our approach is fundamentally different from the approach used in Chapter 6 of [4], for the performance comparison of asynchronous and synchronous iterations. The latter analysis is more useful for message-passing systems where a processor does not need to wait for the completion of a data access after it sends the request for the access.

In Section 2 we first describe a general computational model for asynchronous iterations and we justify the model for monotonic iteration operators which appear in most applications. In Section 3, we study the case where the task execution times are upper and lower bounded and dynamic scheduling is adopted. We show that the ultimate slowdown factor of an asynchronous iterative algorithm A_a, which is the ratio between the speed of the synchronous version A_s of A_a and the speed of A_a, is upper bounded by a constant. In Section 4, we study another important special case where there are as many processors available as we want, and the computation time for each component is constant. We investigate the conditions under which the slowdown factor is close to 1 and show how some of the results are sensitive to small fluctuations in the timing. We further restrict the computational model by making the computation times of all the components equal. We consider a special case where the computations are "skewed" because of the initial forking of the tasks. For all these cases the smallest upper bound we get for the slowdown factor is 1. In Section 5, we explain that this should not lead to the misinterpretation that the slowdown factor cannot be less than 1. The reason is that we always make the implicit assumption that there exists the strongest possible coupling between the components of the shared data, which is rather pessimistic. We discuss how we can obtain more realistic upper bounds by taking into account the "coupling" between the components. How to measure and estimate the coupling, in the context of our model, is still an open problem.

2 General Model

Model 1 described in this section is the most general model in this paper.

Model 1 There are P processors and the components of the global data x is partitioned into Q subsets ($Q \geq P$). The set of all the processors in the system is $\mathcal{P} = \{PE_0, PE_1, \cdots, PE_p, \cdots, PE_{P-1}\}$. Each partition corresponds to 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 all the tasks is $\mathcal{T} = \{T_0, T_1, \cdots, T_q, \cdots, T_{Q-1}\}$ each of which is to be executed infinitely many times. All the activities regarding the update of each component in a partition, including fetching the current value of the shared data, the computation of the components and the actual updates of these components in the global store are performed in the associated task. A time interval that covers all these activities of a task T_q is called a *task interval* of T_q. It is assumed that a task interval has two phases: *the global computation phase* and *the local computation phase*. In the second phase the task does not interact with other tasks, i.e., it does not access any component shared with other tasks. All the interactions occur in the global computation phase. An asynchronous iterative algorithm (asynchronous iteration) corresponds to an allocation of each task to infinitely many time slots of available processors. No assumption is made on the ordering of the task intervals. It is only assumed that the length of each task interval is finite, that in a finite period of time, all the components are updated at least once (non-starvation), and that task intervals of the same task never overlap in time (non-overlapping). It is also assumed that the processors do not stay idle except at the synchronization points, and at the data accesses. $\square$

Figure 1 displays the above defined intervals: TI_q, GP_q and LP_q denote the task interval, the global computation phase and the local computation phase of task T_q, respectively. Although

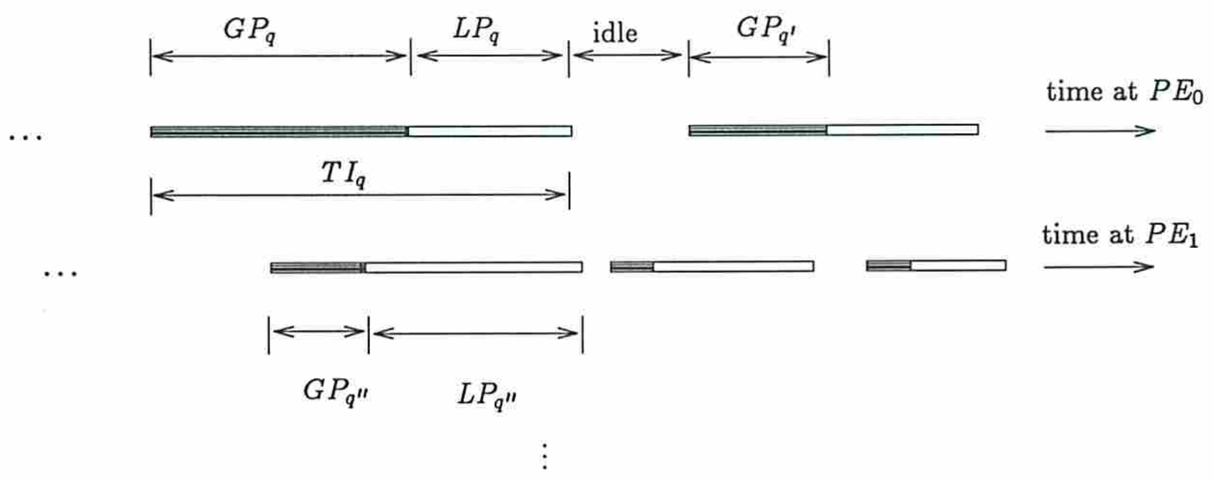

Figure 1: Model 1

the model is general, it is more useful for shared-memory systems. The major restriction is the non-overlapping condition. This rules out the possibility of the overlapping of computation and communication phases of the same task which may be the case in message-passing systems.

Example 1 Consider the discretized approximation of the Laplace equation

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

Discretization is given by a rectilinear grid, with boundary conditions. Each point of the grid is iterated successively using the following iteration formula:

$$u_{i,j} := \frac{1}{4} \left[u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} \right]$$

i.e., the next value of each point is computed by taking the average of the neighboring points. 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 corresponding region (global computation phase), and then processes the interior points (local computation phase).

□

Obviously, for the convergence of an asynchronous iteration, some conditions on the iteration operator F are required. In references [2]-[5] and [12] this issue is studied in detail. In the following we will give a summary as required in the context of this paper. Theorem 1 gives the convergence conditions in terms of a metric $d(\cdot)$ defined in the domain of F. Before the theorem we need some definitions. In these definitions and the theorem, for all real vectors u, v, we will write $u \leq v$, if for all $i = 1, 2, \ldots n$, $u_i \leq v_i$.

Definition 1 Let $X = X_1 \times X_2 \times \cdots \times X_n$ be the domain in which the operator $F = F_1 \times F_2 \times \cdots \times F_n$ is defined, which has a unique fixed point ξ, i.e., $F(\xi) = \xi$. A *metric* $d(\cdot) = d_1(\cdot) \times d_2(\cdot) \times \cdots \times d_n(\cdot)$ is a function such that for all $i = 1, \cdots, n$, $d_i(\cdot) : X_i \to R_+$, and $d_i(\xi) = 0$, where R_+ is the set of nonnegative real numbers. $\square$

In other words, $d_i(\cdot)$ defines an ordering on the elements of X_i and with respect to this ordering, the infimum is ξ_i. We say that x is at least as close to the solution as y if $d(x) \leq d(y)$.

Definition 2 A sequence $\{x(k)\}$ the elements of which take values from X is said to *converge* to $\xi \in X$ w.r.t. $d(\cdot)$ if $\{d(x(k))\}$ converges to the null vector $(0, 0, \ldots, 0)$, where ξ is the unique fixed point in X such that $d(\xi)$ is null. $\square$

The reason for the above definition is the need for the notion of convergence in nonnumeric domains. Although not formalized, what is meant by the convergence of a real vector to the null vector is the convergence of each individual component to zero.

Definition 3 Let $\{x(k)\}$ be the sequence of global data values in a synchronous iteration corresponding to F and starting with $x(0)$, i.e, the sequence defined by (1). *A cycle sequence* $\{\phi(k)\}$ of an asynchronous iteration corresponding to F and starting with $x(0)$, if there exists any, is a sequence of nondecreasing time instances such that for all k and for all $\tau \geq \phi(k)$, $\quad d(x_\tau) \leq d(x(k))$, where x_τ is the value of x at the time instance τ, during the execution of the asynchronous iteration. The interval $[\phi(k), \phi(k+1)]$ is called the k-th cycle and its length is denoted by $c(k)$. The minimum possible cycle sequence is called the *min-cycle sequence* and is denoted by $\{\phi_{min}(k)\}$, i.e., for all cycle sequences $\{\phi(k)\}$ and for all k, $\phi_{min}(k) \leq \phi(k)$. $\square$

$\phi(k)$ is the time taken by the asynchronous computation to make at least the same "progress" towards the solution as the first k iterations of the synchronous algorithm. The progress equivalent to the one made by 1 iteration of the synchronous algorithm will be called "1 unit of progress". The average min-cycle time, defined as the average value of $\phi_{min}(k+1) - \phi_{min}(k)$ is equal to the total execution time divided by the total number of iterations required to reach the solution. For this reason, comparing the speed of an asynchronous iteration with its synchronous version is the same as comparing the average min-cycle time of the asynchronous iteration with the average iteration time of the synchronous version. Unfortunately, the min-cycle sequence is difficult to estimate exactly using the available parameters. We can, however, identify some other cycle sequences which are not minimum. Such a sequence $\{\varphi'(k)\}$ is defined in the following. In Theorem 1 it is shown that $\{\varphi'(k)\}$ is in fact a cycle sequence.

Definition 4 $\{\varphi'(k)\}$ is an increasing sequence of the time instances starting with $\varphi'(0) = \phi(0)$ and such that for all k, the time interval $[\varphi'(k), \varphi'(k+1)]$ covers at least one task interval of each task. $\square$

It should be emphasized that $\{\varphi'(k)\}$ is only a pessimistic estimate of the cycle sequence. For a given iteration operator F, the worst we can expect is that in each cycle, each task needs the outcome of all the tasks executed in the previous cycle, in order to make any progress towards the solution. This is the situation when there exists maximum possible "coupling" between the tasks. For the class of iteration operators with maximum coupling, $\{\varphi'(k)\}$ is the exact estimate of the min-cycle sequence if the length of the local computation intervals are zero (i.e., if the computed values are released only at the end of each task execution). In Section 4, we will modify $\{\varphi'(k)\}$ so that it allows nonzero lengths of local computation intervals, and will derive some results after the modification. As discussed in Section 5, on the other extreme, there may be no dependency between the tasks, in which case $\{\varphi'(k)\}$ is not exact, but only upper bounds $\{\phi_{min}(k)\}$.

Theorem 1 *Under Model 1, an asynchronous iteration corresponding to F and starting from $x(0) \in X$ converges to ξ if its synchronous counterpart converges to ξ, w.r.t. the metric function $d(\cdot)$, where*

- *X is closed under F,*
- *$d(F(u)) \leq d(u)$, for all $u \in X$,*
- *$d(F(u)) \leq d(F(v))$, for all $u, v \in X$ such that $d(u) \leq d(v)$.*

Sketch of proof. A similar theorem was proved in [3], [4] and [12]. For completeness, we only give a sketch of the proof. We will simply show, by induction, that for all k, the value x_τ of the global data x at time instance τ indefinitely satisfies: $d(x_\tau) \leq d(x(k))$ after $\varphi'(k)$, which means that $\varphi'(k)$ is a cycle sequence. From Definition 3, the theorem immediately follows.

The basis clause of the induction is straightforward. From the second condition, it follows that updating a data component can not make the component move away from the solution. Therefore, the value of the data is always at least as close to the solution as $x(0)$.

Suppose that after $\varphi'(l)$ the value of the data is at least as close to the solution as $x(l)$, and consider the first task interval of a task T_q that starts after $\varphi'(l)$. Since at the beginning of the task interval the distance ($d(\cdot)$) of the data is not larger than that of $x(l)$, from the third condition, each component i updated by this task is at least as close to the solution as $x_i(l+1)$, after the interval, because

$$d(x(0)) \geq d(x(1)) \geq \cdots \geq d(x(l)) \geq d(x(l+1)) \geq \cdots \qquad \square$$

It can be shown that the discretized solution of the Laplace equation given in Example 1 satisfies the conditions of the above theorem [2].

3 Dynamic Allocation

In this section we will consider a computational model, in which $P < Q$ and the tasks are scheduled dynamically. Under this model, we will derive an upper bound on the slowdown factor of an asynchronous iterative algorithm A_a, which is the ratio between the speed of the synchronous algorithm A_s corresponding to A_a and the speed of A_a.

The cycle sequence referred to in this section is the same as in Definition 4, i.e., $\{\phi(k)\} \equiv \{\varphi'(k)\}$. C_{min} and C denote the average min-cycle time and the average cycle time of an asynchronous iteration A_a. I is the average iteration time (min-cycle time) of the synchronous version A_s of A_a. For all these parameters the averages are taken over all the cycles of A_a and A_s. Let M be the number of iterations required by the synchronous version. Then the slowdown factor S satisfies

$$S = \frac{M \cdot C_{min}}{M \cdot I} \leq \frac{C}{I}$$

Figure 2 shows an iteration of a synchronous algorithm for 5 tasks and 3 processors; a cycle of the corresponding asynchronous algorithm is demonstrated by Figure 3. We immediately notice that in the synchronous case all the processors restart right after the previous iteration, while in the

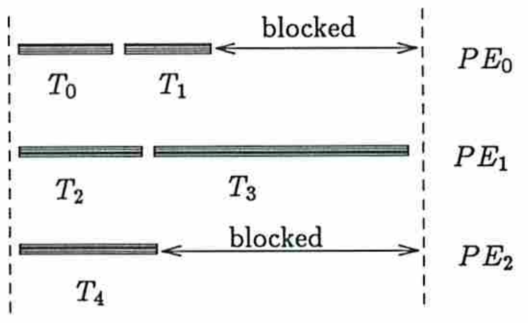

Figure 2: Synchronous iteration

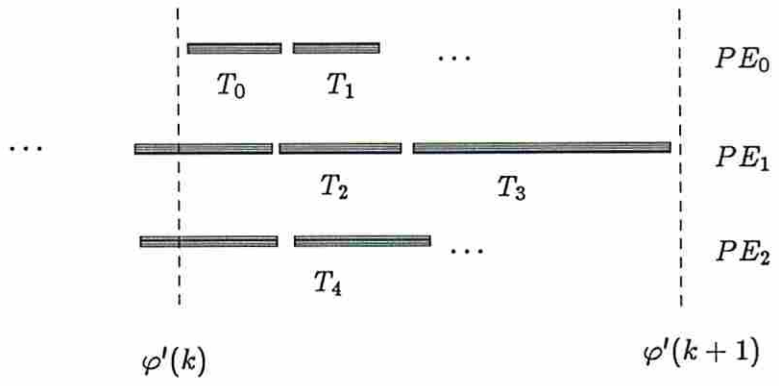

Figure 3: Asynchronous iteration

asynchronous case, only one processor restarts right after the previous cycle (PE_0 in the figure). For the remaining $(P-1)$ processors, after the previous cycle is over, a period of time is wasted before the next task is initiated on each processor. This period corresponds to the upper bound on the implicit delay in utilizing the most recent data and may cover up to one whole task interval.

The computational model of this section can be described as follows.

Model 2 The task interval length of each task T_q is lower and upper bounded by t_{qmin} and t_{qmax} respectively. With no loss of generality, we assume $t_{0max} \geq t_{1max} \geq t_{2max} \geq \cdots$. The scheduling policy satisfies the following requirement: when a processor becomes available at any time instance τ in a cycle k, it always selects a task T_q, for the next execution, which has not yet started in the

k-th cycle, unless all the tasks have been started. Furthermore,

$$\frac{t_{max}}{t_{min}} < \frac{Q}{P},$$

where $t_{max} = \max_q\{t_{qmax}\}$ and $t_{min} = \min_q\{t_{qmin}\}$. $\square$

$\overline{t_{min}}$, $\overline{t_{max}}$ and t_D are defined as follows:

$$\begin{aligned} Q \cdot \overline{t_{min}} &= \sum_{q=0}^{Q-1} t_{qmin}, \\ Q \cdot \overline{t_{max}} &= \sum_{q=0}^{Q-1} t_{qmax}, \\ (P-1) \cdot t_D &= \sum_{q=0}^{P-2} t_{qmax}. \end{aligned}$$

Obviously,

$$t_{min} \le \overline{t_{min}} \le \overline{t_{max}} \le t_D \le t_{max}.$$

We now explain how we can satisfy the restriction of Model 2 on the scheduling policy.[1] In the discussion, τ will represent any time instance in the k-th cycle at which any processor becomes available for the next task execution. From the above statement of the scheduling condition, it follows that we need a policy that gives priority to the task whose previous execution has started the earliest. This guarantees that the processor available at τ selects a task T_q if T_q has not yet started in the k-th cycle and also if T_q is not currently executed by another processor. However, this is not enough. To see this, suppose that there are $L < P$ tasks which started before the k-th cycle and are still being executed at τ. The condition on the scheduling would be violated if all the other $(Q - L)$ tasks had been started before τ in the k-th cycle. We need to guarantee that this anomaly does not occur, i.e., the number of tasks that can be initiated in the period from the

[1]The reader can verify that this constraint cannot be satisfied by a global First In First Out process queue.

beginning of the k-th cycle until τ is less than $(Q - L)$. This period is upper bounded by t_{max}, since some tasks working at τ have been started before the k-th cycle. Therefore, the upper bound on the task initiations in the same period, on $(P - L)$ processors is $\frac{t_{max}}{t_{min}}(P - L)$. We need this number to be less than $(Q - L)$. Therefore, we require

$$\frac{t_{max}}{t_{min}} < \frac{Q - L}{P - L}.$$

This holds for all $L = 1, 2, \ldots, P - 1$ if

$$\frac{t_{max}}{t_{min}} < \frac{Q}{P}$$

which explains the restriction in the model. Notice that the case $P = Q$ is not covered. However, this case will be analyzed separately in the next section. Simulation shows that the result derived below is a good approximation even with the lack of this restriction, because the above described anomaly does not occur very often.

We now estimate an upper bound on the total processor time $C \cdot P$ spent in a cycle by splitting it into three parts: $C_1 \cdot P$, $C_2 \cdot P$ and $C_3 \cdot P$. Let d_p be the time from the beginning of a cycle k until the first processor initiation on PE_p. Then, $C_1 \cdot P$ is defined as the sum of all d_p's. Since there are up to $(P - 1)$ nonzero d_p's

$$C_1 \cdot P \leq \sum_{q=0}^{P-2} t_{qmax} = (P - 1)\, t_D.$$

The second part is the total "useful" work in the cycle which covers the first full execution of each task:

$$C_2 \cdot P \leq \sum_{q=0}^{Q-1} t_{qmax} = Q \cdot \overline{t_{max}}.$$

Finally, $C_3 \cdot P$ is the sum of all e_p's, where e_p is the time from PE_p completes its useful work until the completion of the cycle. Notice that one of the task completions also determines the cycle completion. From the condition on the scheduling, before the beginning of this task, it is not possible that another processor completes its useful work. Therefore,

$$C_3 \cdot P \le (P-1)\, t_{max}$$

and

$$C \cdot P \;=\; C_1 \cdot P + C_2 \cdot P + C_3 \cdot P \;\le\; Q\,\overline{t_{max}} \;+\; (P-1)\,(t_D + t_{max}).$$

The iteration time of the corresponding synchronous algorithm is

$$I \cdot P \;\ge\; Q\,\overline{t_{min}}.$$

As a result,

$$S \;\le\; \frac{\overline{t_{max}}}{\overline{t_{min}}} + \frac{P-1}{Q}\,\frac{(t_D + t_{max})}{\overline{t_{min}}} \;\le\; \frac{\overline{t_{max}} + t_D + t_{max}}{\overline{t_{min}}}$$

The nice conclusion is that the ultimate upper bound does not depend on the number of processors and on the problem size. This translates into the fact that the time complexity of an asynchronous algorithm cannot be greater than that of its synchronous version.

The result can obviously be applied to the case when all the task interval lengths are equal and constant (say t). However, in this case, we can obtain a smaller upper bound by noticing that in the asynchronous case, a period of $\lceil \frac{Q+P-1}{P} \rceil\, t$ is guaranteed to cover Q tasks provided that the starting instance of the period coincides with at least one task initiation. If it coincides with all P task initiations, as in the synchronous case, then the period required to cover Q tasks is exactly $\lceil \frac{Q}{P} \rceil\, t$. Therefore,

$$S \le \frac{\lceil \frac{Q+P-1}{P} \rceil}{\lceil \frac{Q}{P} \rceil} \le \frac{\lceil \frac{Q+P}{P} \rceil}{\lceil \frac{Q}{P} \rceil} = 1 + \frac{1}{\lceil \frac{Q}{P} \rceil} \le 1 + \frac{P}{Q}.$$

It is clear that the above analysis is pessimistic, and does not reflect the average behavior of Model 2. To see this, we have simulated Model 2. For each run of the simulation $P = 64$ and

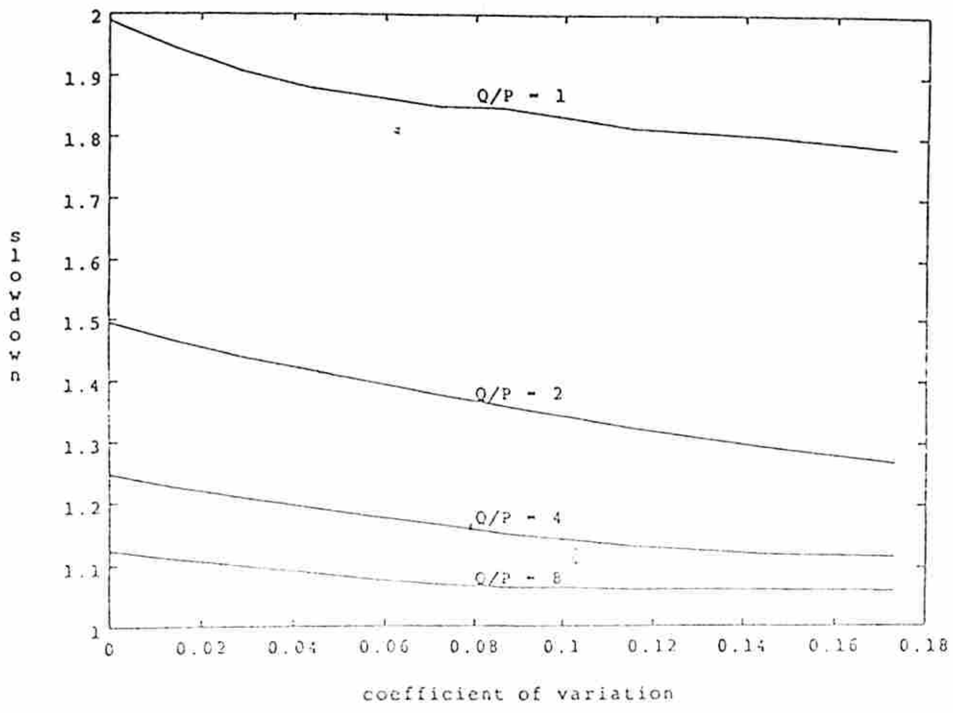

Figure 4: Slowdown factor for uniform distribution ($P = 64$)

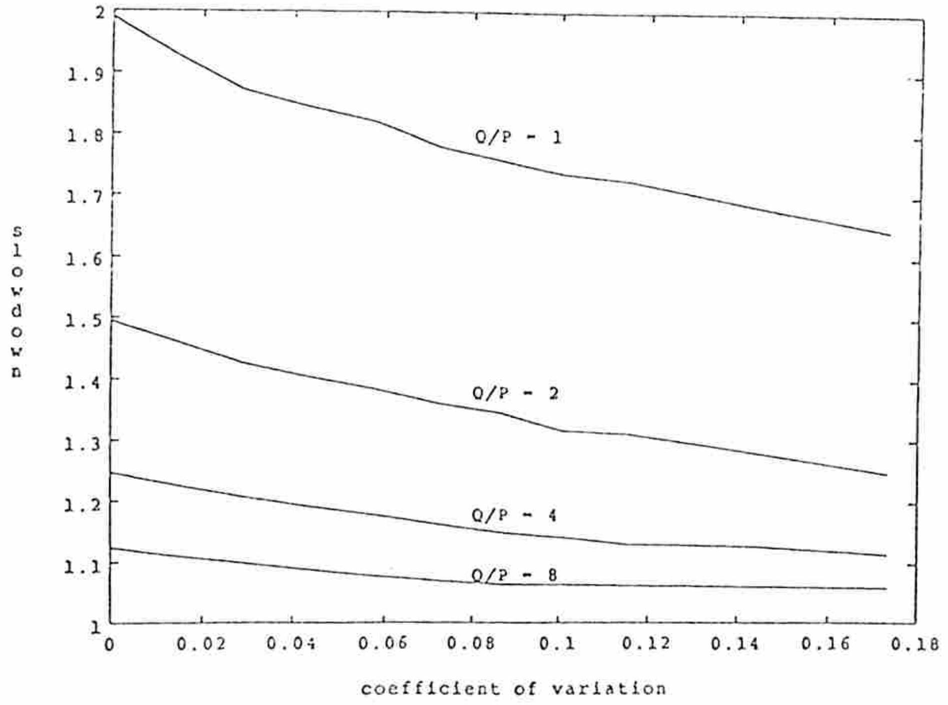

Figure 5: Slowdown factor for normal distribution ($P = 64$)

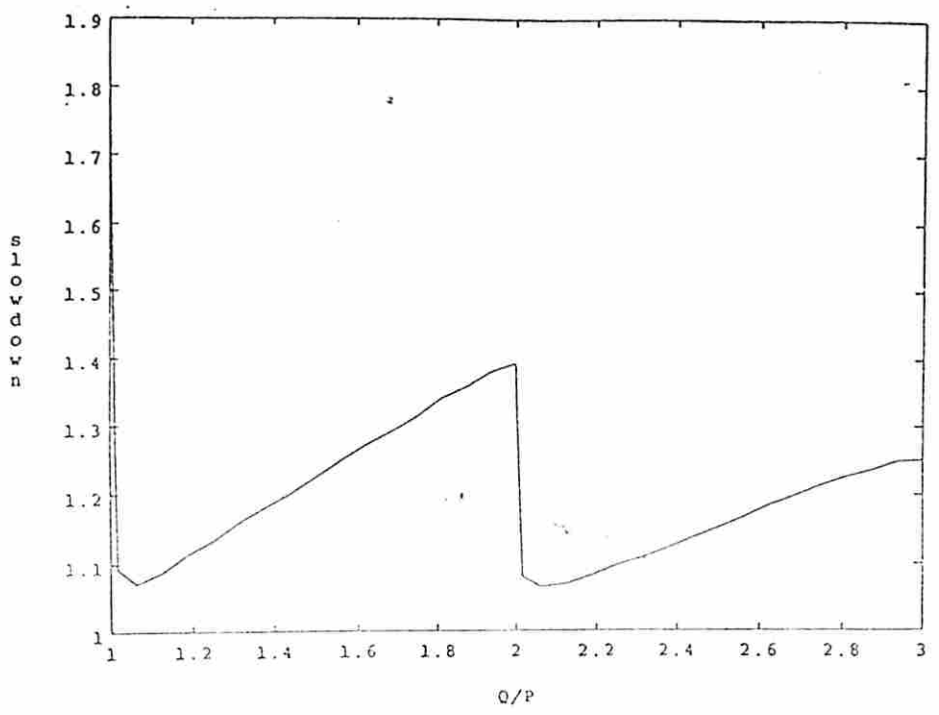

Figure 6: Slowdown factor vs. Q/P for uniform distribution between 90 and 110 ($P = 64$)

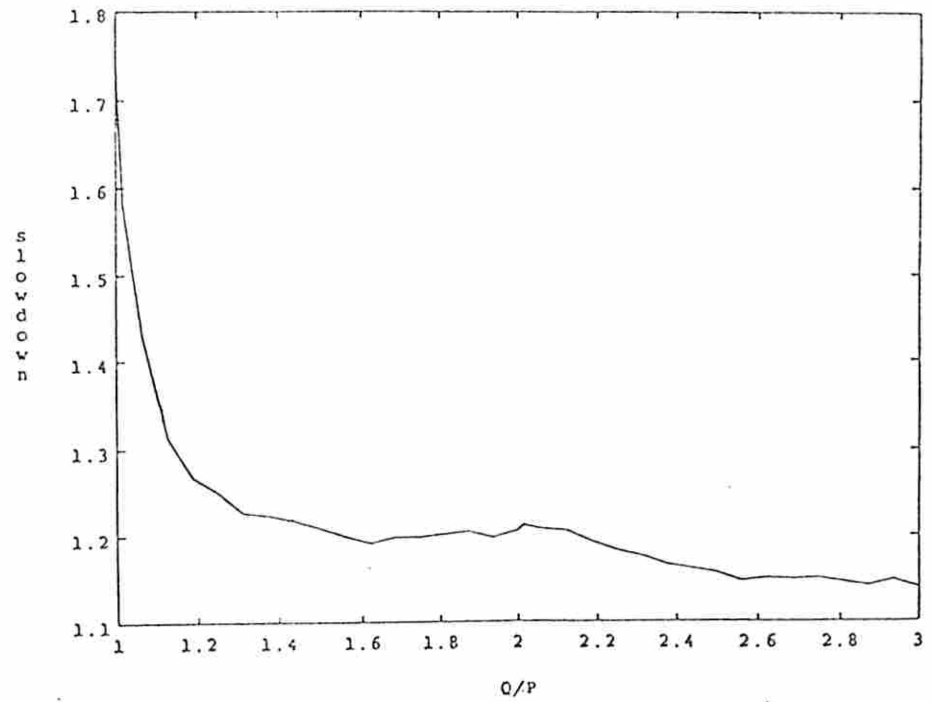

Figure 7: Slowdown factor vs. Q/P for uniform distribution between 50 and 150 ($P = 64$)

the task intervals are drawn from the same distribution with mean 100. The results are plotted in Figures 4-7. From Figures 4-5, we notice that the slowdown factor is always less than $1 + \frac{P}{Q}$. This can be explained by the intuitive fact that, most of the time, a time period of $\frac{Q+P}{P} t_{av}$ covers at least Q tasks, where t_{av} is the average task interval length. It is also seen that the slowdown factor decreases with increasing variance. This means that larger fluctuations decrease the advantage of synchronous algorithms, because more processor time is wasted at synchronization points. Another conclusion of Figures 4 and 5 is that the results are not very different for uniform and normal distributions.

Figures 6 and 7 show what happens when Q/P is not an integer. Integer values of Q/P correspond to good load balancing in which case the advantage of the synchronous algorithm is maximized. This explains the peaks at integer values of Q/P. In Figure 7 the peaks are diminished, because task intervals are wildly changing between 50 and 150.

From these figures, we can deduce that the efficiency of asynchronous algorithms with respect to their synchronous counterparts increases as the fluctuations of the task compute time increase.

4 Static Allocation

In this section, we first define a cycle sequence $\{\varphi''(k)\}$ which is smaller than $\{\varphi'(k)\}$. Using this, we will discuss how the slowdown factor can be decreased, in the case where there are as many processors available as tasks and the task interval lengths are constant.

According to the definition of $\{\varphi'(k)\}$, a cycle should contain a task interval of each task which does not overlap with the next and the previous cycles. However, 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 cycles. Therefore, the following defined sequence $\{\varphi''(k)\}$ is also a cycle sequence.

Definition 5 $\{\varphi''(k)\}$ is an increasing sequence of time instances such that for all k, $[\varphi''(k), \varphi''(k+1)]$ covers at least one *global* computation phase of each task. □

Another way to view this is that local computation phases can be considered as idle times in which no significant global work is done. Obviously, $\{\varphi''(k)\}$ is a smaller estimate than $\{\varphi'(k)\}$. It is also clear that for a given task scheduling and task interval lengths, decreasing the global computation phases can only decrease $\{\varphi''(k)\}$ and therefore the upper bound on the execution time. In the following, we have a closer look at $\{\varphi''(k)\}$.

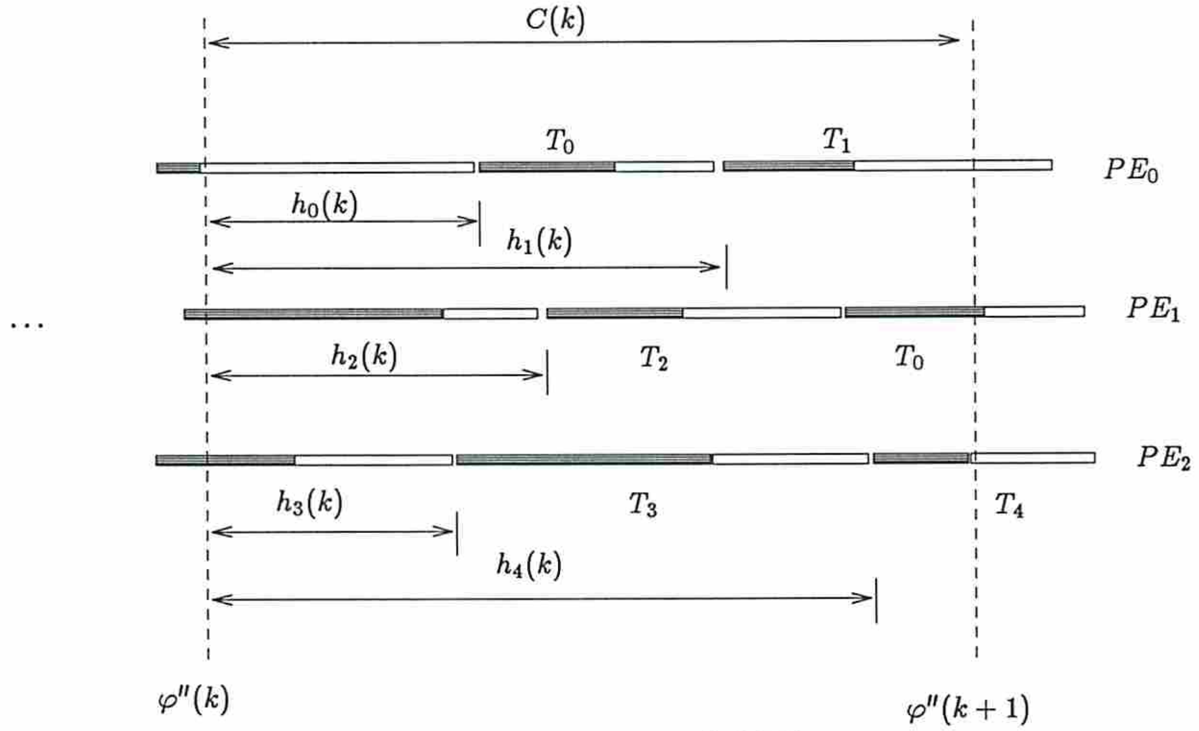

Figure 8: The k-th cycle of $\{\varphi''(k)\}$

Let $h_q(k)$ be the time difference between $\varphi''(k)$ and the starting instance of the first global computation phase of the task T_q and $g_q(k)$ be the length of this phase (See Figure 8 which shows a case for 5 tasks and 3 processors). Then,

$$\varphi''(k+1) = \varphi''(k) + \max_q\{h_q(k) + g_q(k)\}.$$

Let $c(k) = \varphi''(k+1) - \varphi''(k)$ be the k-th cycle length of $\{\varphi''(k)\}$. Then,

$$c(k) = \max_q\{h_q(k) + g_q(k)\}. \tag{2}$$

Now, we state the computational model of this section.

Model 3 Besides the conditions of Model 1, we assume that for all q, the task intervals and the global and local computation phases of task T_q have constant lengths which are denoted by t_q, g_q and l_q, respectively. There are as many processors as tasks ($P = Q$). □

Although not explicitly stated, static allocation can easily be assumed for this model, where each task is executed by the same processor throughout the computation. The reason is that at the time a processor PE_p finishes executing a task T_q, the other tasks are being executed by the other processors. The only available task to be executed next by PE_p is T_q. Even though more than one task may complete exactly at the same time and processors may switch tasks, this interchange does not change the timing of events, because all processors run at the same speed.

An important result on the upper bound for the execution time can be stated for Model 3. Since $h_q(k)$ varies between 0 and $t_q = g_q + l_q$, from (2)

$$c(k) \leq \max_q\{2\,g_q + l_q\}. \tag{3}$$

Note that the min-cycle time of the synchronous version takes t_{max} time. Then,

$$S \leq \frac{C'}{I} \leq \frac{\max_q\{2\,g_q + l_q\}}{t_{max}},$$

where C' is the average cycle time with respect to $\{\varphi''(k)\}$. Since

$$\max_q\{2\,g_q + l_q\} \leq \max_q\{2\,g_q + 2\,l_q\} = 2\,t_{max},$$

$$S \leq 2.$$

The following proposition follows.

Proposition 1 *Under Model 3, any asynchronous iteration is at most 2 times slower than its synchronous counterpart.* □

From (3), the slowdown factor is 1 when for all q, $g_q \ll l_q$. Another related observation is not a consequence of (3) and can be stated as follows. If t_{max} is not smaller than any $2\,g_q + l_q$, for $q \neq 0$, then any task interval of the slowest task can always cover at least one global computation phase of another task. In this case, the slowest task determines every cycle. These results are summarized below:

Proposition 2 *Under Model 3, any asynchronous iteration is not slower than its synchronous version if*

- $g_q \ll l_q$, *for all* q, *or*
- $t_{max} \geq 2\,g_q + l_q$, *for all* $q \neq 0$. □

A special case worthwhile to consider is that where $g = g_0 = g_1 = \cdots$, $l = l_0 = l_1 = \cdots$ and $t = g + l$. If all the tasks initially start at the same time ($h_q(0) = 0$, for all q), then this case is equivalent to the synchronous version (Figure 9). Therefore, ignoring the synchronization costs, the performance of this scheme is the same as its synchronous counterpart. However, because of the initial forking of the tasks, or for other reasons, it may be expected that, at least after some

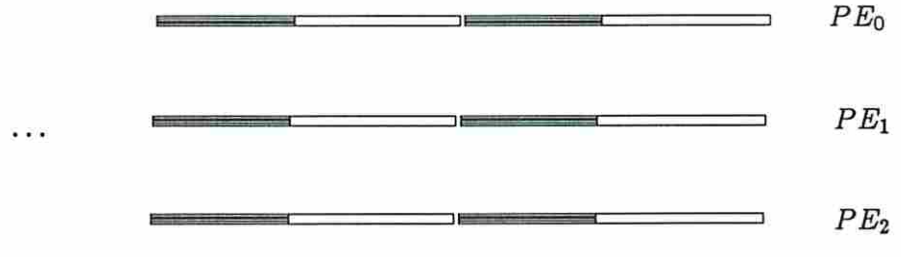

Figure 9: The computation is not skewed

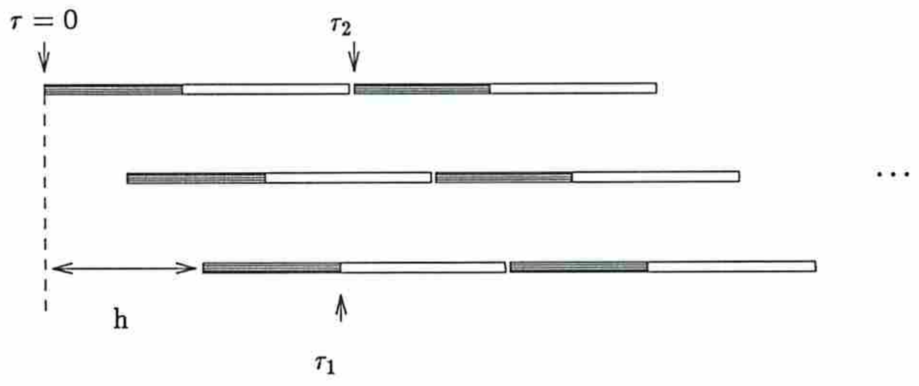

Figure 10: The computation is skewed

transient period, task intervals will be skewed (Figure 10). Suppose we neglect the transient period and let the skew distance $h \leq t$ be defined as the time difference between the initial starting points of the tasks that start the earliest and the latest (Figure 10). Also, let the earliest task start at $\tau = 0$. Then the latest task interval will complete its global computation phase at $\tau_1 = h + g$ and the earliest task will start its second global computation at $\tau_2 = l + g$. If $\tau_1 \leq \tau_2$ the earliest task interval will always cover at least one global computation phase of each task, and the following proposition can be concluded.

Proposition 3 *Under Model 3, any asynchronous iteration satisfying* $g = g_0 = g_1 = \cdots$ *and* $l = l_0 = l_1 = \cdots$ *is not slower than its synchronous version, if* $l \geq h$ *, where* h *is the skew distance.* □

It should be noted that the above result is very sensitive to small fluctuations in the computation time, when h and l are close. Consider an asynchronous iteration that can be represented approximately by Model 3, but the timing of events can be deviated from Model 3 by as much as $\pm\epsilon$ time units, where $\epsilon > 0$. Such a deviation can be caused by shared memory conflicts, for example. For this asynchronous iteration, it is no longer guaranteed that $\tau_2 \geq \tau_1$, when $h = l$, because in the worst case, $\tau_1 = h + g + \epsilon$ and $\tau_2 = l + g - \epsilon$. Since the second round of the earliest task starts before the first round of the latest task, the earliest task may not be able to use, in the second round, the "fresh" component values which the latest task has provided and it may happen that the second round of the earliest task may be totally wasted. Therefore, the cycle time may be up to $2\,t_{max}$. As it is formalized below, to guarantee $\tau_1 \leq \tau_2$ we should have $l \geq h + 2\,\epsilon$. In the case where all the computations are global ($l = 0$) this never holds.

Proposition 4 *Any asynchronous iteration, for* $g = g_0 = g_1 = \cdots$, $l = l_0 = l_1 = \cdots$, *with a skew* *distance h and which is deviated from Model 3 by at most $\epsilon > 0$ time units, is not slower than its synchronous version if $l \geq h + 2\,\epsilon$.* □

Similarly, the second result of Proposition 2 is also sensitive to the small fluctuations in the timing.

5 Cycle Definition Revisited

The above presented upper bounds on the execution times of asynchronous algorithms are at least as large as those of their synchronous counterparts. This should not mean, however, that an asynchronous iteration cannot be faster than its counterpart with barrier synchronization, even when the synchronization overhead is ignored. The reason is that we assumed the worst case in defining the cycle time, where the execution of a task depends critically on the outcome of *all* the tasks in the previous cycle, in order to make any progress towards the solution. This is not true, when there is not a dependency between each pair of tasks. Let S_q be the set of tasks that the computation of T_q depends on. Taking this dependency into account, we can slightly modify the definition of the cycle sequence, to obtain smaller upper bounds.

Definition 6 For all q, $\{\varphi_q(k)\}$ is defined as a nondecreasing sequence of time instances, starting with $\varphi_q(0) = \phi(0)$, such that the interval $[\tau_q(k), \varphi_{q+1}(k+1)]$ covers at least one global computation phase of the task T_q, where

$$\tau_q(k) = \max_{j \in S_q}\{\varphi_j(k)\}. \qquad \square$$

After $\varphi_q(M)$, the task T_q reaches the solution, whereas the version of the algorithm with barrier synchronization takes M iterations to reach the solution. We will not exhibit the proof since it is very similar to the proof of Theorem 1.

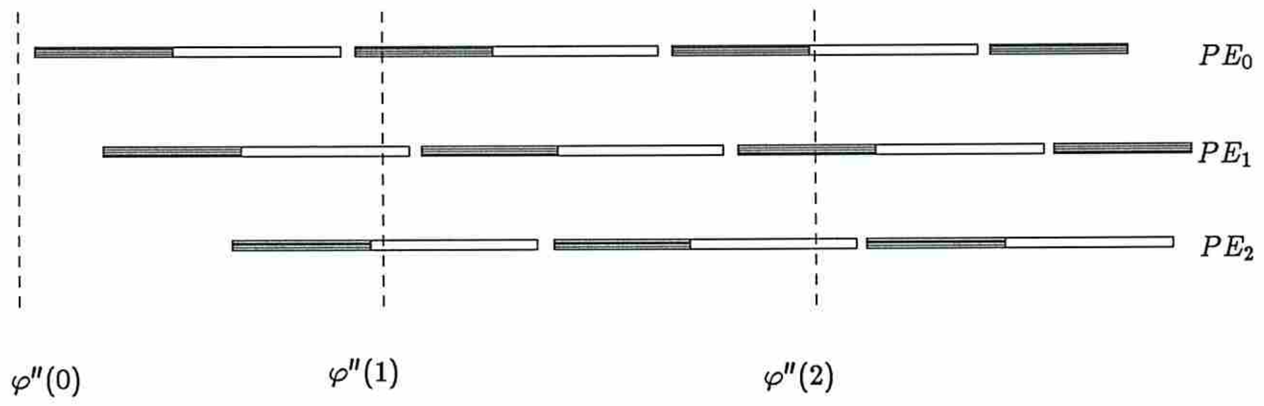

Figure 11: An example for Definition 6

If the time distance between $\tau_q(k)$ and the starting instance of the first global phase of T_q after $\tau_q(k)$ is denoted by $r_q(k)$ and $g_q(k)$ is the length of this phase, then

$$\varphi_q(k+1) = \max_{j \in S_q}\{\varphi_j(k)\} + r_q(k) + g_q(k) \tag{4}$$

and the total execution time will be

$$T \leq \max_q\{\varphi_q(M)\}.$$

Example 2 Consider the computation in Figure 11. There are three components of the data and each component x_i is assigned statically to PE_i. Each task interval length is constant and equal to t. Notice that the second global computation interval of the first component (x_0) does not contribute to the cycle from $\varphi''(1)$ to $\varphi''(2)$. This is because the definition of $\varphi''(k)$ does not take into account the dependency between the components and considers the worst case in which F_0 cannot advance the data without the recent value of x_2. In this way, one out of three global computation intervals of x_0 is wasted and the average cycle time is $1.5\,t$.

Now, suppose the dependency between the components is restricted in the following manner.

$$\begin{aligned} x_0 &= F_0(x_0, x_1) \\ x_1 &= F_1(x_0, x_1, x_2) \end{aligned}$$

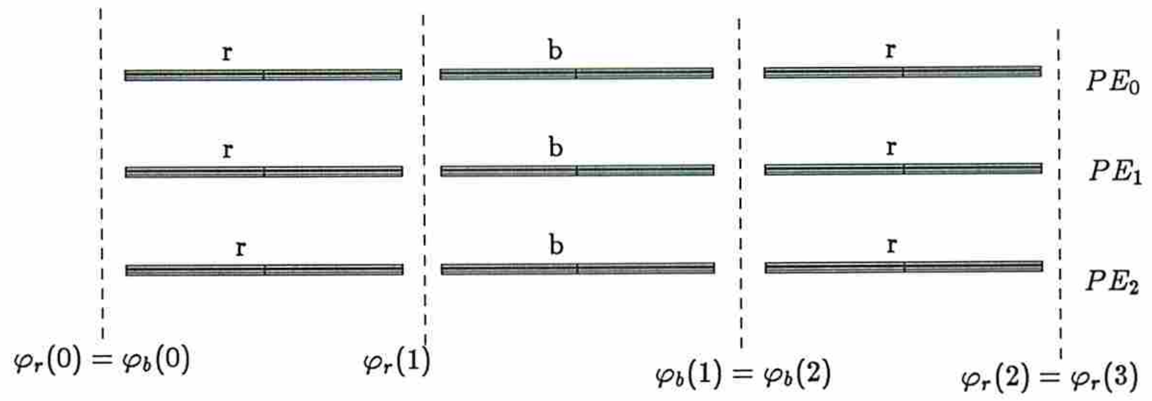
r
b
r
PE_0
r
b
r
PE_1
r
b
r
PE_2
$\varphi_r(0) = \varphi_b(0)$
$\varphi_r(1)$
$\varphi_b(1) = \varphi_b(2)$
$\varphi_r(2) = \varphi_r(3)$

Figure 12: red-black ordering

$$x_2 = F_2(x_0, x_1, x_2)$$

Since F_0 does not depend on x_2, the computation of x_0 in a cycle may overlap with the computation of x_2 in the previous cycle. As a result, each computation of x_0 makes a progress towards the solution. Consequently, the average cycle time of $\{\phi_{min}(k)\}$ is upper bounded by t. □

Definition 6 does not even rule out the possibility of cycles of length zero. We explain this on a popular scheme, called red-black ordering, for the solution to the problem given by Example 1.

Example 3 In red-black ordering, each component has a color, red or black, such that the computation of red (black) components only depend on black (red) components. Consider the computational scheme as shown in Figure 12. There are 3 processors and 6 tasks. 3 of the tasks correspond to red (black) components and are labelled by r (b) in the figure. All the task intervals are constant and equal to t. $\varphi_r(k)(\varphi_b(k))$ is equal to $\varphi_q(k)$, given in Definition 6, for red (black) tasks. Notice that, for example, at the end of the first computation phase of black tasks, the black components have made at least 1 unit of progress, since the initial time. On the other hand, the black components only depend on the red ones and the red components have completed their first computations. Therefore, the black components have made also at least 2 units of progress. Effectively, 1 unit of progress is made in a time period of t, which means the average cycle time is t. According to the definition of $\{\varphi''(k)\}$, however, a cycle should cover all the tasks and in this example has a length of $2\,t$. $\square$

We can also generalize the example as follows. Let $\mathcal{S} = \{\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_{S-1}\}$ be a partitioning of the set of all the tasks $\mathcal{T}$ such that the tasks in $\mathcal{T}_i$ only depends on the tasks in $\mathcal{T}_{(i-1) \bmod S}$, for all $0 \le i \le S-1$. Then it is sufficient that the k-th cycle only covers the tasks in $\mathcal{T}_{k \bmod S}$. For such defined cycle sequence, in k cycles, all the tasks in $\mathcal{T}_{k \bmod S}$ make at least k units of progress.

The above discussion suggests that the sequence specified in Definition 6 sometimes yields a much smaller bound on the execution time than $\{\varphi''(k)\}$. To see this in general, define $\overline{\varphi}_q(k)$ as the value of $\varphi_q(k)$ when S_q is replaced, in (4) by the set $\mathcal{T}$ of all the tasks. Clearly, $\varphi''(k)$ can be written as

$$\varphi''(k) = \max_q \overline{\varphi}_q(k)$$

Since $\overline{\varphi}_q(k) \ge \varphi_q(k)$, $\varphi''(k)$ cannot be less than $\varphi_q(k)$. Furthermore, from (4), $\varphi_q(k)$ gets smaller as S_q gets smaller. It can be concluded that when the dependency is small and the upper bound of the cycle time of $\{\varphi''(k)\}$ is equal to the iteration time of the synchronous version with global barrier synchronization some gain in execution speed may be expected by removing the barriers.

Nevertheless, in the case where S_q's are constant in the whole data domain and are detectable by simple dependency analysis, it is not necessary to use global barrier synchronization. If, furthermore, S_q's are small, it makes little sense to compare an asynchronous iteration with its version with global barrier synchronization. Instead, a more restricted form of synchronization can be adopted, in which a task T_q only waits for the tasks in the set S_q to be completed to start its next global computation phase. If the tasks can be preempted such that a task T_q starts right after all the tasks in S_q are executed (even though it is busy) then for this synchronization scheme all $r_q(k)$'s in (4) are zero. Hence, unless the synchronization cost is too high, this form of synchronization is justified.

There may still be cases where S_q's are small, but they are not constant and dynamically changing throughout the computation. For example, in the consistent labeling problem the dependency between the components is data dependent [7]. In such a situation, S_q's are not known a priori, therefore it is not possible to use the restricted form of synchronization. Barrier synchronization is the only possible synchronizer and since the dependency is small, removing it will increase the performance.

Even Definition 6 leads to a pessimistic estimation of the upper bound to the total execution time. This bound corresponds to the case where T_q relies critically on all the components in S_q, in order to make any progress towards the solution. However, a dependency from a task $T_{q'}$ to T_q does not necessarily imply that T_q cannot make a progress towards the solution without the contribution of $T_{q'}$. Therefore, we can replace S_q, in Definition 6, by a smaller set $S_q^{\prime} \subseteq S_q$ that only contains all the components x_i, such that the values of x_i's are close enough to the solution to make T_q advance. As it is obvious from its definition, $S_q^{\prime}$ dynamically changes throughout the computation, depending on the value of the data and it is a measure of the coupling between the tasks.

We can conclude that the elimination of synchronization points improves the performance of asynchronous iterations significantly when $S_q^{\prime}$'s are much smaller than S_q's. Unfortunately, we do not yet know how to measure and estimate $S_q^{\prime}$'s and therefore we are not able to derive smaller upper bounds on the execution times of asynchronous iterations. This information would allow us not only to better analyze performance but also to design better algorithms that would optimally schedule the tasks dynamically during the computation by making use of this information.

6 Conclusion

We have introduced a model useful to estimate the speed of asynchronous iterations. For this model, we have studied the effect of the timing of shared memory accesses in asynchronous iterations. A poorly designed asynchronous algorithm can be slower than its synchronous counterpart. However, in all the cases we have analyzed the slowdown factor is upper bounded by a constant. Partitioning is essential to the performance of asynchronous iterations. A good partitioning should consider the coupling between the components and should increase local computations compared to global ones.

References

[1] AXELROD, T. S. Effects of synchronization barriers on multiprocessor performance. *Parallel Computing 3* (1986), 129–140.

[2] BAUDET, G. M. Asynchronous iterative methods for multiprocessors. *J. ACM 25*, 2 (April 1978), 226–244.

[3] BERTSEKAS, D. P. Distributed asynchronous computation of fixed points. *Mathematical Programming 27* (1983), 107–120.

[4] BERTSEKAS, D. P., AND TSITSIKLIS, J. N. *Parallel and Distributed Computation*. Prentice Hall, 1989.

[5] CHAZAN, D., AND MIRANKER, W. Chaotic relaxation. *Linear Algebra and Its Applications 2* (1969), 199–222.

[6] DUBOIS, M., AND BRIGGS, F. A. Performance of synchronized iterative processes in multiprocessor systems. *IEEE Transaction on Software Engineering 8*, 4 (July 1982), 419–431.

[7] HARALICK, R. M., AND SHAPIRO, L. G. The consistent labeling problem: Part I. *IEEE Trans. on PAMI 1*, 2 (April 1979), 173–184.

[8] HWANG, K., AND BRIGGS, F. A. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1984.

[9] KUNG, H. T. Synchronized and asynchronous parallel algorithms for multiprocessors. In *Algorithms and Complexity : New Directions and Recent Results*, J. F. Traub, Ed., Academic Press, New-York, 1976.

[10] ROBERT, F. *Iterations Discrètes Asynchrones.* Tech. Rep. R.R. 671-M, Informatique et Mathemetiques Appliquées de Grenoble, September 1987.

[11] ROBINSON, J. Some analysis techniques for asynchronous multiprocessor algorithms. *IEEE Transaction on Software Engineering 5*, 1 (January 1979), 24–31.

[12] ÜRESİN, A., AND DUBOIS, M. *Parallel asynchronous algorithms for discrete data.* Tech. Rep. CRI-88-05, Computer Research Institute, Department of Electrical Engineering-Systems, University of Southern California, 1988.