Load, Sharing, Routing and Congestion Control in Distributed Systems as a Stackelberg Game

BY

A.A. Economides and John Silvester

Technical Report CENG 90-16

Electrical Engineering - Systems Department

University of Southern California

Los Angeles, CA. 90089-0781

3/5/91

Priority Load Sharing: An Approach Using Stackelberg Games

by

A. A. Economides and John A. Silvester

Technical Report CENG 90-16

Electrical Engineering Systems

University of Southern California

Los Angeles, CA. 90089-0781

PRIORITY LOAD SHARING: AN APPROACH USING STACKELBERG GAMES

by

Anastasios A. Economides *and* John A. Silvester

Computer Engineering Division
Electrical Engineering - Systems Department
University of Southern California, SAL 300
Los Angeles, CA 90089-0781

Abstract

In this paper we consider the load sharing problem in a multiprocessor, where different classes of jobs have different priorities and each priority class optimizes its own objective function. First, we formulate the problem as a leader-follower Stackelberg game, where the high priority jobs constitute the leader and the low priority jobs constitute the follower. Then we focus on a special case of two preemptive resume priority classes that share a two-processor system. In this case, we find the Stackelberg equilibrium solution that minimizes the average job delay of each class. An interesting result is that if both classes have equal mean service requirements, when both processors are used, then the low priority load sharing decisions are independent of the arrival rates. Also, for equal mean service requirements and constant total arrival rate, the overall average delay of jobs from both classes is constant, i.e. does not depend on the mix of high and low priority jobs. Finally, we comment on our approach for multiobjective optimization of distributed systems with multi-priority classes.

1 INTRODUCTION

The usual approach to distributed system design and control is the optimization of a single function [6, 3]. If multiple objectives are desired, then the usual approach is to combine the objectives as seen by the system administrator [2] into a single function. Thus, it is assumed that all customers in the system are treated similarly and they cooperate for the socially optimum, such as optimizing the average customer performance. However, in a real distributed environment there is a diversity of customer classes, each with possibly different objectives and different service and accounting requirements. In [4], we have taken a game theoretic approach for performance optimization of competing classes in a distributed computing system. In that paper, we have formulated and solved the routing problem among competing classes of jobs as

a Nash game.

It is quite common to require differentiated service among different classes by assigning different priorities to different classes, for example interactive jobs have higher priority than batch jobs. A high priority class may acquire most of the resources that it needs, while a low priority class should wait for the high priority class to complete service. Since the reason for having priorities is to give preferential treatment to the high priority jobs, it is *not* meaningful to define a *single* multi-objective function (ex. a convex combination of the objective functions of the different priority classes) for global optimization *across all the priority classes simultaneously.* However, we can still optimize the behavior of jobs within each priority class. Therefore a different approach should be taken for performance optimization of multipriority systems. In this paper, we formulate and optimize the performance of different priority classes as a Stackelberg game.

For simplicity of presentation, we consider two priority classes of customers which select between two servers. Jobs from the high priority class and jobs from the low priority class arrive to a two-processor system requiring execution. The problem of deciding to which processor each job will be assigned is the load sharing problem [6, 3] (Fig. 1).

In section 2, we define the notation for a simple two-processor system that is shared among two priority classes. In section 3, we formulate the priority load sharing problem as a Stackelberg non-cooperative game [1] where the leader is the high priority class, while the follower is the low priority class. In section 4, we solve a preemptive resume priority load sharing problem for a two processor system, where each priority class wants to minimize the average delay of its jobs. In section 5, we discuss the Stackelberg equilibrium solution for this priority load sharing problem. Finally, in section 6, we conclude on this new approach for performance evaluation and optimization of multipriority distributed computing systems.

2 QUEUEING MODEL

In this section, we introduce a simple queueing model of two servers that are shared by customers of two priority classes (Fig. 1). The problem is to assign these customers to the two servers so as to minimize the average delay of each class. An application is load sharing for a multiprocessor system, where interactive jobs (high priority) and batch jobs (low priority) may use two processors for execution. Another application is routing, where voice packets (high priority) and data packets (low priority) may use two different links for transmission between source-destination.

Let the high priority class α jobs arrive to the system with rate λ^α (Poisson arrivals) and require service times with mean $1/\mu^\alpha$ (exponential). On the other hand, the low priority class β jobs arrive to the system with rate λ^β (Poisson arrivals) and require service times with mean $1/\mu^\beta$ (exponential). Jobs of both classes may be served at either of the two processors, which have service rates C_1 and C_2, respectively. Furthermore, for stability reasons it is assumed that the total arrival rate of service requirements is less than the total service rate :

$$\frac{\lambda^\alpha}{\mu^\alpha} + \frac{\lambda^\beta}{\mu^\beta} \leq C_1 + C_2$$

Class α assigns its jobs to server 1 with probability P_1^α and to server 2 with probability P_2^α, $(P_1^\alpha + P_2^\alpha = 1, \;\; P_1^\alpha, P_2^\alpha \geq 0)$, such that its cost function $J^\alpha(P_1^\alpha, P_2^\alpha, P_1^\beta, P_2^\beta)$ is minimized. Similarly, class β assigns its jobs to server 1 with probability P_1^β and to server 2 with probability P_2^β, $(P_1^\beta + P_2^\beta = 1, \;\; P_1^\beta, P_2^\beta \geq 0)$, such that its cost function $J^\beta(P_1^\alpha, P_2^\alpha, P_1^\beta, P_2^\beta)$ is minimized.

In the following sections, we formulate and solve the load sharing problem of two processors among two priority classes as a Stackelberg game.

3 STACKELBERG EQUILIBRIUM

In this section, we consider the load sharing problem, when two priority classes, with different objectives, share two processors. We formulate this priority multiobjective optimization problem as a non cooperative Stackelberg game [1] between the two priority classes.

Next, we give some definitions for a two-priority class (any kind of priorities) game similar to those in [1] for Stackelberg games:

Definition 1: In a two-priority class finite game, with the high priority class α as the leader and the low priority class β as the follower, the set $R^\beta(P_1^\alpha, P_2^\alpha)$, defined for the high priority strategy (P_1^α, P_2^α) that satisfies $P_1^\alpha + P_2^\alpha = 1, \quad P_1^\alpha, P_2^\alpha \geq 0$, by:

$$\begin{aligned} R^\beta(P_1^\alpha, P_2^\alpha) = \{ \quad & (P_1^\beta, P_2^\beta) \ \textit{such that} \ P_1^\beta + P_2^\beta = 1, \quad P_1^\beta, P_2^\beta \geq 0 : \\ & J^\beta(P_1^\alpha, P_2^\alpha, P_1^\beta, P_2^\beta) \leq J^\beta(P_1^\alpha, P_2^\alpha, \hat{P}_1^\beta, \hat{P}_2^\beta), \\ & \forall \ (\hat{P}_1^\beta, \hat{P}_2^\beta), \ \textit{such that} \ \hat{P}_1^\beta + \hat{P}_2^\beta = 1, \quad \hat{P}_1^\beta, \hat{P}_2^\beta \geq 0\} \end{aligned}$$

is the *optimal response* (rational reaction) set of the low priority class β to the strategy of the high priority class α.

What the above definition says is that the low priority class β finds the set of its controls (P_1^β, P_2^β), that minimize its cost function $J^\beta(P_1^\alpha, P_2^\alpha, P_1^\beta, P_2^\beta)$, for given strategy (P_1^α, P_2^α) of the high priority class α.

Definition 2: In a two-priority class finite game with the high priority class α as the leader, a strategy $(P_1^{\alpha*}, P_2^{\alpha*})$, such that $P_1^{\alpha*} + P_2^{\alpha*} = 1, \quad P_1^{\alpha*}, P_2^{\alpha*} \geq 0$, is called a *Stackelberg equilibrium* strategy for the leader if

$$\inf_{(P_1^\beta, P_2^\beta) \in R^\beta(P_1^{\alpha*}, P_2^{\alpha*})} J^\alpha(P_1^{\alpha*}, P_2^{\alpha*}, P_1^\beta, P_2^\beta) \leq \inf_{(P_1^\beta, P_2^\beta) \in R^\beta(P_1^\alpha, P_2^\alpha)} J^\alpha(P_1^\alpha, P_2^\alpha, P_1^\beta, P_2^\beta)$$

$$\forall \ (P_1^\alpha, P_2^\alpha) \ \textit{such that} \ P_1^\alpha + P_2^\alpha = 1, \quad P_1^\alpha, P_2^\alpha \geq 0$$

This means that the high priority class α finds the set of its optimal controls $(P_1^{\alpha*}, P_2^{\alpha*})$ that minimize its cost function $J^\alpha(P_1^\alpha, P_2^\alpha, P_1^\beta, P_2^\beta)$, given the optimal response set $R^\beta(P_1^{\alpha*}, P_2^{\alpha*})$ of the low priority class β to its strategy $(P_1^{\alpha*}, P_2^{\alpha*})$.

Definition 3: Let $(P_1^{\alpha*}, P_2^{\alpha*})$, such that $P_1^{\alpha*}+P_2^{\alpha*} = 1, \quad P_2^{\alpha*}, P_2^{\alpha*} \geq 0$, be a Stackelberg strategy for the leader α. Then any element $(P_1^{\beta*}, P_2^{\beta*}) \in R^{\beta}(P_1^{\alpha*}, P_2^{\alpha*})$ is an optimal strategy for the follower β that is in equilibrium with $(P_1^{\alpha*}, P_2^{\alpha*})$. The strategy $(P_1^{\alpha*}, P_2^{\alpha*}, P_1^{\beta*}, P_2^{\beta*})$ is a *Stackelberg solution* for the game with the high priority class α as the leader and the cost pair $J^{\alpha}(P_1^{\alpha*}, P_2^{\alpha*}, P_1^{\beta*}, P_2^{\beta*})$, $J^{\beta}(P_1^{\alpha*}, P_2^{\alpha*}, P_1^{\beta*}, P_2^{\beta*})$ is the corresponding *Stackelberg equilibrium outcome.*

So, after the high priority class α has found its Stackelberg equilibrium strategy $(P_1^{\alpha*}, P_2^{\alpha*})$, then the optimal response set of the low priority class β is given by $R^{\beta}(P_1^{\alpha*}, P_2^{\alpha*})$. Any element $(P_1^{\beta*}, P_2^{\beta*}) \in R^{\beta}(P_1^{\alpha*}, P_2^{\alpha*})$ is an optimal strategy for the low priority class β.

4 PREEMPTIVE RESUME PRIORITY LOAD SHARING

In this section, we give a simple example for two preemptive resume priority classes of jobs that share two processors. When a high priority job is assigned to a processor, if there is another high priority job there, then it is put in the queue. If there are only low priority jobs there, then the low priority job is preempted and the high priority one starts been executing immediately. When all the high priority jobs have finished receiving service, then the low priority job that was preempted resumes and continues receiving service [5, 2]. The high priority class α (leader) assigns its jobs to the two processors, such that the average delay of its jobs is minimized. On the other hand, the low priority class β (follower) assigns its jobs to the two processors, such that the average delay of its jobs is minimized, after the high priority class α has optimally assigned its jobs. Thus a Stackelberg equilibrium is achieved.

4.1 General Two Class Solution

The cost function that we use for the high preemptive resume priority class α is its average job delay [5]:

$$J^{\alpha}(P_1^{\alpha}, P_2^{\alpha}) = \frac{P_1^{\alpha} * \frac{1}{\mu^{\alpha} C_1}}{1 - \frac{\lambda^{\alpha} P_1^{\alpha}}{\mu_{\alpha} C_1}} + \frac{P_2^{\alpha} * \frac{1}{\mu^{\alpha} C_2}}{1 - \frac{\lambda^{\alpha} P_2^{\alpha}}{\mu_{\alpha} C_2}}$$

This is a strictly convex function over the convex space $P_1^{\alpha} + P_2^{\alpha} = 1, \ P_1^{\alpha}, \ P_2^{\alpha} \geq 0$.

Similarly, the cost function for the low preemptive resume priority class β is its average job delay [5]:

$$J^{\beta}(P_1^{\alpha}, P_2^{\alpha}, P_1^{\beta}, P_2^{\beta}) = \frac{P_1^{\beta} * [\frac{1}{\mu^{\beta} C_1} - \frac{\lambda^{\alpha} P_1^{\alpha}}{\mu^{\alpha} C_1 * \mu^{\beta} C_1} + \frac{\lambda^{\alpha} P_1^{\alpha}}{(\mu^{\alpha} C_1)^2}]}{(1 - \frac{\lambda^{\alpha} P_1^{\alpha}}{\mu^{\alpha} C_1}) * (1 - \frac{\lambda^{\alpha} P_1^{\alpha}}{\mu^{\alpha} C_1} - \frac{\lambda^{\beta} P_1^{\beta}}{\mu^{\beta} C_1})} +$$

$$+ \frac{P_2^{\beta} * [\frac{1}{\mu^{\beta} C_2} - \frac{\lambda^{\alpha} P_2^{\alpha}}{\mu^{\alpha} C_2 * \mu^{\beta} C_2} + \frac{\lambda^{\alpha} P_2^{\alpha}}{(\mu^{\alpha} C_2)^2}]}{(1 - \frac{\lambda^{\alpha} P_2^{\alpha}}{\mu^{\alpha} C_2}) * (1 - \frac{\lambda^{\alpha} P_2^{\alpha}}{\mu^{\alpha} C_2} - \frac{\lambda^{\beta} P_2^{\beta}}{\mu^{\beta} C_2})}$$

This is a strictly convex function over the convex space $P_1^{\beta} + P_2^{\beta} = 1, \ P_1^{\beta}, \ P_2^{\beta} \geq 0$. It is also a strictly convex function over the convex space $P_1^{\alpha} + P_2^{\alpha} = 1, \ P_1^{\alpha}, \ P_2^{\alpha} \geq 0$.

The overall average job delay is:

$$J(P_1^{\alpha}, P_2^{\alpha}, P_1, P_2^{\beta}) = \frac{\lambda^{\alpha}}{\lambda^{\alpha} + \lambda^{\beta}} * J^{\alpha}(P_1^{\alpha}, P_2^{\alpha}) + \frac{\lambda^{\beta}}{\lambda^{\alpha} + \lambda^{\beta}} * J^{\beta}(P_1^{\alpha}, P_2^{\alpha}, P_1^{\beta}, P_2^{\beta})$$

Theorem 1 : There exists a Stackelberg equilibrium.

Proof : This is a two player non zero-sum continuous kernel game on the square, for which the follower's cost functional $J^{\beta}(P_1^{\alpha}, P_2^{\alpha}, P_1^{\beta}, P_2^{\beta})$ is strictly convex with respect to the leader's strategy over the convex space $P_1^{\alpha} + P_2^{\alpha} = 1, \quad P_1^{\alpha}, \ P_2^{\alpha} \geq 0$. Therefore, it admits a Stackelberg equilibrium solution [1]. □

The high preemptive resume priority class α solves the following problem:

minimize

$$J^{\alpha}(P_1^{\alpha}, P_2^{\alpha}) = \frac{P_1^{\alpha}}{\mu^{\alpha} C_1 - \lambda^{\alpha} P_1^{\alpha}} + \frac{P_2^{\alpha}}{\mu^{\alpha} C_2 - \lambda^{\alpha} P_2^{\alpha}}$$

with respect to P_1^α, P_2^α

such that $P_1^\alpha + P_2^\alpha = 1, \quad P_1^\alpha, \ P_2^\alpha \geq 0.$

On the other hand, the low preemptive resume priority class β solves the following problem:

minimize

$$J^\beta(P_1^{\alpha*}, P_2^{\alpha*}, P_1^\beta, P_2^\beta) = \frac{P_1^\beta * [\frac{C_1}{\mu^\beta} - \frac{\lambda^\alpha P_1^{\alpha*}}{\mu^\alpha \mu^\beta} + \frac{\lambda^\alpha P_1^{\alpha*}}{(\mu^\alpha)^2}]}{(C_1 - \frac{\lambda^\alpha P_1^{\alpha*}}{\mu^\alpha}) * (C_1 - \frac{\lambda^\alpha P_1^{\alpha*}}{\mu^\alpha} - \frac{\lambda^\beta P_1^\beta}{\mu^\beta})} +$$

$$+ \frac{P_2^\beta * [\frac{C_2}{\mu^\beta} - \frac{\lambda^\alpha P_2^{\alpha*}}{\mu^\alpha \mu^\beta} + \frac{\lambda^\alpha P_2^{\alpha*}}{(\mu^\alpha)^2}]}{(C_2 - \frac{\lambda^\alpha P_2^{\alpha*}}{\mu^\alpha}) * (C_2 - \frac{\lambda^\alpha P_2^{\alpha*}}{\mu^\alpha} - \frac{\lambda^\beta P_2^\beta}{\mu^\beta})}$$

with respect to P_1^β, P_2^β

such that $P_1^\beta + P_2^\beta = 1, \quad P_1^\beta, \ P_2^\beta \geq 0.$

Let define the auxiliary variables

$$C_1^\alpha = C_1 - \frac{\lambda^\alpha P_1^{\alpha*}}{\mu^\alpha}$$

$$C_2^\alpha = C_2 - \frac{\lambda^\alpha P_2^{\alpha*}}{\mu^\alpha}$$

$$C_1^\beta = \frac{C_1}{\mu^\beta} - \frac{\lambda^\alpha P_1^{\alpha*}}{\mu^\alpha \mu^\beta} + \frac{\lambda^\alpha P_1^{\alpha*}}{(\mu^\alpha)^2}$$

$$C_2^\beta = \frac{C_2}{\mu^\beta} - \frac{\lambda^\alpha P_2^{\alpha*}}{\mu^\alpha \mu^\beta} + \frac{\lambda^\alpha P_2^{\alpha*}}{(\mu^\alpha)^2}$$

Then, the following policy allocates the arriving jobs to the two servers such that a Stackelberg equilibrium is achieved:

$If \qquad \dfrac{\lambda^\alpha}{\mu^\alpha} \leq C_1 + C_2,$

then

$$If \quad C_1 - \sqrt{C_1 C_2} \leq \frac{\lambda^\alpha}{\mu^\alpha} \quad and \quad C_2 - \sqrt{C_1 C_2} \leq \frac{\lambda^\alpha}{\mu^\alpha}$$

$$then \quad P_1^{\alpha *} = \frac{C_1}{\dfrac{\lambda^\alpha}{\mu^\alpha}} - \frac{C_1 + C_2 - \dfrac{\lambda^\alpha}{\mu^\alpha}}{\dfrac{\lambda^\alpha}{\mu^\alpha}} * \frac{\sqrt{C_1}}{\sqrt{C_1} + \sqrt{C_2}}$$

$$If \quad 0 \leq \frac{\lambda^\alpha}{\mu^\alpha} \leq C_1 - \sqrt{C_1 C_2}$$

$$then \quad P_1^{\alpha *} = 1$$

$$If \quad 0 \leq \frac{\lambda^\alpha}{\mu^\alpha} \leq C_2 - \sqrt{C_1 C_2}$$

$$then \quad P_1^{\alpha *} = 0$$

$$If \qquad \frac{\lambda^\beta}{\mu^\beta} \leq C_1^\alpha + C_2^\alpha$$

then

$$If \quad C_1^\alpha - C_2^\alpha * \sqrt{\frac{C_1^\beta}{C_2^\beta}} \leq \frac{\lambda^\beta}{\mu^\beta} \quad and \quad C_2^\alpha - C_1^\alpha * \sqrt{\frac{C_2^\beta}{C_1^\beta}} \leq \frac{\lambda^\beta}{\mu^\beta}$$

$$then \quad P_1^{\beta *} = \frac{\frac{C_1^\alpha}{\frac{\lambda^\beta}{\mu^\beta}} - \frac{C_1^\alpha + C_2^\alpha - \frac{\lambda^\beta}{\mu^\beta}}{\frac{\lambda^\beta}{\mu^\beta}}}{\sqrt{C_1^\beta} + \sqrt{C_2^\beta}} * \sqrt{C_1^\beta}$$

$$If \quad \frac{\lambda^\beta}{\mu^\beta} \leq C_1^\alpha - C_2^\alpha * \sqrt{\frac{C_1^\beta}{C_2^\beta}}$$

$$then \quad P_1^{\beta *} = 1$$

$$If \quad \frac{\lambda^\beta}{\mu^\beta} \leq C_2^\alpha - C_2^\alpha * \sqrt{\frac{C_2^\beta}{C_1^\beta}}$$

$$then \quad P_1^{\beta *} = 0$$

Of course, the Stackelberg equilibrium load sharing probabilities to the other server are $P_2^{\alpha *} = 1 - P_1^{\alpha *}$ and $P_2^{\beta *} = 1 - P_1^{\beta *}$.

Substituting these Stackelberg equilibrium probabilities into the average delay functions, we have the Stackelberg equilibrium outcome of the game (Appendix A).

4.2 Interesting Results

From the above Stackelberg equilibrium solution and outcome of the game, we have some interesting results:

Proposition 1: For a given system $C_1 \geq C_2$,

if $\mu^\alpha = \mu^\beta = \mu$, $\lambda^\alpha + \lambda^\beta \leq \mu(C_1 + C_2)$, and $\lambda^\alpha + \lambda^\beta = \lambda = constant$,

then $J(P_1^{\alpha*}, P_2^{\alpha*}, P_1^{\beta*}, P_2^{\beta*}) = constant$

Proof: see Appendix B.

The above proposition says that for equal mean service requirements for both priority classes and constant total arrival rate the overall average job delay is constant, i.e. it does not depend on the mix of high and low priority jobs.

In Fig. 2, we show the Stackelberg equilibrium average delay of the high priority class α, $J^\alpha(P_1^{\alpha*}, P_2^{\alpha*})$, of the low priority class β, $J^\beta(P_1^{\alpha*}, P_2^{\alpha*}, P_1^{\beta*}, P_2^{\beta*})$, and of the system $J(P_1^{\alpha*}, P_2^{\alpha*}, P_1^{\beta*}, P_2^{\beta*})$ versus different mixes of the high and low priority arrival rates $\frac{\lambda^\alpha}{\lambda^\beta}$, for fixed server capacities $C_1 = 2, C_2 = 1$, fixed total arrival rate $\lambda^\alpha + \lambda^\beta = 2.5$, and equal mean service requirement of the high and the low priority jobs $1/\mu^\alpha = 1/\mu^\beta = 1$. We note that the overall average job delay is constant and independent from the mix of the high and low priority jobs.

Proposition 2: For a given system with $C_1 \geq C_2$ and $\mu^\alpha = \mu^\beta = \mu$,

if $\mu(C_1 + C_2) \leq \lambda^\alpha + \lambda^\beta$ and $C_1 - \sqrt{C_1 C_2} \leq \lambda^\alpha/\mu$,

then $P_1^{\beta*} = \dfrac{\sqrt{C_1}}{\sqrt{C_1} + \sqrt{C_2}}$

Proof: When $\mu^\alpha = \mu^\beta = \mu$, for Case 1 Appendix A, we have $C_1^\beta = \dfrac{C_1}{\mu}$, $C_2^\beta = \dfrac{C_2}{\mu}$. The proof follows immediately. □

What the above proposition says is that for equal mean service requirements for both priority classes, when the high priority class α uses both servers, then the Stackelberg equilibrium decisions of the low priority class β are constant and independent of the arrival rates $(\lambda^\alpha, \lambda^\beta)$. This result is not intuitive, because we might expect that the load sharing decisions for the low priority class β should also depend on the arrival rates (as it is the case for the high priority class α). It is also very important, because even when the arrival rates vary over time, the load sharing policy for the low priority jobs remains the same (Fig. 3).

Proposition 3: For a given system $C_1 \geq C_2$:

if $\dfrac{\lambda^\alpha}{\mu^\alpha} + \dfrac{\lambda^\beta}{\mu^\beta} \to C_1 + C_2$, then $P_1^{\beta*} \to \dfrac{\sqrt{C_1}}{\sqrt{C_1} + \sqrt{C_2}}$.

Proof: see Appendix C.

The above proposition says that even for different service requirements for the two priority classes, when the total arriving service requirement $\frac{\lambda^\alpha}{\mu^\alpha} + \frac{\lambda^\beta}{\mu^\beta}$ approaches the total service capacity $C_1 + C_2$, the Stackelberg load sharing decisions for the low priority class β become constant and independent from the arrival rates.

In Fig. 3, we show the Stackelberg equilibrium load sharing probabilities of both the high priority class α, $P_1^{\alpha *}$, and of the low priority class β, $P_1^{\beta *}$, versus the system load $\frac{\lambda^\alpha/\mu^\alpha + \lambda^\beta/\mu^\beta}{C_1 + C_2}$, for fixed server capacities $C_1 = 2, C_2 = 1$, equal arrival rates $\lambda^\alpha = \lambda^\beta$ and different ratio of the mean service requirement of the high and the low priority jobs $1/\mu^\alpha = 2/\mu^\beta = 1$, $1/\mu^\alpha = 1/\mu^\beta = 1$, $1/\mu^\beta = 2/\mu^\alpha = 1$.

For equal mean service requirements of the high and low priority jobs, we note that when the high priority class α uses both processors ($0 < P_1^{\alpha *} < 1$), then the load sharing decisions $P_1^{\beta *}$ for the low priority class β are constant and independent of the arrival rates $\lambda^\alpha, \lambda^\beta$. For different mean service requirements of the high and low priority jobs, we note that when the system load $\frac{\lambda^\alpha/\mu^\alpha + \lambda^\beta/\mu^\beta}{C_1 + C_2}$ approaches 1, then the load sharing decisions $P_1^{\beta *}$ for the low priority class β approach the same constant value as for the case of equal mean service requirements.

In this section, we have explicitly solved a two processor load sharing problem, when two preemptive resume priority classes of jobs share the two processors. For equal mean service requirements of jobs from both classes, when the high priority class uses both servers, then the Stackelberg equilibrium decisions of the low priority jobs do not depend on the arrival rates of the jobs. That means that even if the arrival rates change during operation, our load sharing algorithm will still perform "optimally" for the low priority class. Also, for constant total arrival rate, even if we change the mix of high and low priority classes, then the overall average job delay remains constant.

5 NUMERICAL RESULTS & DISCUSSION

In this section, we discuss some other results that can be derived from the Stackelberg solution of the two processor load sharing problem among jobs from two preemptive resume priority classes.

5.1 Constant λ^α

Consider a two processor system $C_1 \leq C_2$ with fixed arrival rate of interactive (high priority) jobs $\lambda^\alpha = constant$. If this multiprocessor is also to be used by batch (low priority) jobs and we want to secure an upper bound on the average delay of batch jobs $J^\beta \leq J_0^\beta$, then we should restrict the arrival rate of the batch jobs up to an upper limit. For example, if $\mu^\alpha = \mu^\beta = \mu$, Case 1, then

$$\frac{(\sqrt{C_1}+\sqrt{C_2})^2}{\mu * (C_1 + C_2 - \frac{\lambda^\alpha}{\mu}) * (C_1 + C_2 - \frac{\lambda^\alpha}{\mu} - \frac{\lambda^\beta}{\mu})} \leq J_0^\beta \quad \Rightarrow$$

$$\lambda^\beta \leq \mu * (C_1 + C_2) - \lambda^\alpha - \frac{(\sqrt{C_1}+\sqrt{C_2})^2}{J_0^\beta * (C_1 + C_2 - \frac{\lambda^\alpha}{\mu})}$$

In Fig. 4, we show the Stackelberg equilibrium average delay of both the higher priority class α, $J^\alpha(P_1^{\alpha*}, P_2^{\alpha*})$, of the lower priority class β, $J^\beta(P_1^{\alpha*}, P_2^{\alpha*}, P_1^*, P_2^{\beta*})$, and of the system $J(P_1^{\alpha*}, P_2^{\alpha*}, P_1^*, P_2^{\beta*})$ versus the low priority class β arrival rate, λ^β, for fixed server capacities $C_1 = 2, C_2 = 1$, fixed mean service requirements $1/\mu^\alpha = 1/\mu^\beta = 1$ and fixed high priority class α arrival rate $\lambda^\alpha = 1.0$. So, for example, if the average delay of batch jobs J^β should be less than 10, then the arrival rate of batch jobs should be $\lambda^\beta < 1.71$.

5.2 Constant λ^β

Next, consider a two processor system $C_1 \leq C_2$ with fixed arrival rate of batch jobs (low priority) $\lambda^\beta = constant$. If this multiprocessor is also to be used by interactive

jobs (high priority) and we want to secure an upper bound on the average delay of batch jobs $J^\beta \leq J_0^\beta$, then we should restrict the arrival rate of the interactive jobs up to an upper limit. For example, if $\mu^\alpha = \mu^\beta = \mu$, Case 1, then

$$\frac{(\sqrt{C_1}+\sqrt{C_2})^2}{\mu * (C_1 + C_2 - \frac{\lambda^\alpha}{\mu}) * (C_1 + C_2 - \frac{\lambda^\alpha}{\mu} - \frac{\lambda^\beta}{\mu})} \leq J_0^\beta \quad \Rightarrow$$

$$\lambda^\alpha \leq \mu(C_1 + C_2) - \frac{\lambda^\beta}{2} - \sqrt{(\frac{\lambda^\beta}{2})^2 + \frac{\mu(\sqrt{C_1}+\sqrt{C_2})^2}{J_0^\beta}}$$

In Fig. 5, we show the Stackelberg equilibrium average delay of both the higher priority class α, $J^\alpha(P_1^{\alpha*}, P_2^{\alpha*})$, of the lower priority class β, $J^\beta(P_1^{\alpha*}, P_2^{\alpha*}, P_1^*, P_2^{\beta*})$, and of the system $J(P_1^{\alpha*}, P_2^{\alpha*}, P_1^*, P_2^{\beta*})$ versus the high priority class α arrival rate, λ^α, for fixed server capacities $C_1 = 2, C_2 = 1$, fixed mean service requirements $1/\mu^\alpha = 1/\mu^\beta = 1.$ and fixed low priority class β arrival rate $\lambda^\beta = 1.0$.

So, for example, if the average delay of batch jobs J^β should be less than 10, then the arrival rate of interactive jobs should be $\lambda^\beta < 1.59$.

5.3 Constant λ

Thirdly, consider a two processor system $C_1 \leq C_2$ with fixed total arrival rate of both interactive and batch jobs $\lambda^\alpha + \lambda^\beta = \lambda = constant$. We want to determine what mix of interactive and batch jobs will secure an upper bound on the average delay of interactive jobs $J^\alpha \leq J_0^\alpha$ as well as on batch jobs $J^\beta \leq J_0^\beta$. Let $k = \frac{\lambda^\alpha}{\lambda^\beta}$ be the mix of interactive over batch jobs. Then we can write the arrival rate of interactive jobs as $\lambda^\alpha = \frac{k}{k+1}\lambda$ and the arrival rate of batch jobs as $\lambda^\beta = \frac{1}{k+1}\lambda$. For example, if $\mu^\alpha = \mu^\beta = 1$, Case 2.2, then

$$\frac{1}{\mu C_1 - \frac{k}{k+1}\lambda} \leq J_0^\alpha \qquad \Rightarrow \; k \leq \frac{J_0^\alpha * \mu C_1}{1 - J_0^\alpha * (\mu C_1 - \lambda)}$$

$$\frac{\mu C_1}{(\mu C_1 - \frac{k}{k+1}\lambda) * (\mu C_1 - \lambda)} \leq J_0^\beta \quad \Rightarrow \; k \leq \frac{\mu C_1 * [J_0^\beta * (\mu C_1 - \lambda) - 1]}{\mu C_1 - J_0^\beta * (\mu C_1 - \lambda)^2}$$

In Fig. 2, we show the Stackelberg equilibrium average delay of both the high priority class α, $J^\alpha(P_1^{\alpha*}, P_2^{\alpha*})$, of the low priority class β, $J^\beta(P_1^{\alpha*}, P_2^{\alpha*}, P_1^*, P_2^{\beta*})$, and of the system $J(P_1^{\alpha*}, P_2^{\alpha*}, P_1^*, P_2^{\beta*})$ versus different mix of the high and low priority arrival rates $k = \frac{\lambda^\alpha}{\lambda^\beta}$, for fixed server capacities $C_1 = 2, C_2 = 1$, fixed total arrival rate $\lambda = \lambda^\alpha + \lambda^\beta = 2.5$, and equal mean service requirement of the high and the low priority jobs $1/\mu^\alpha = 1/\mu^\beta = 1$.

So, for example, if the average delay of interactive jobs should be less than 2 and the average delay of batch jobs should be less than 10, then mix of interactive and batch jobs should be $\frac{\lambda^\alpha}{\lambda^\beta} < 2.8$.

5.4 Different Server Rate Ratios

Finally, in Fig. 6, we show the Stackelberg equilibrium probability to processor 1, of both the high and low priority classes versus the system load $\frac{\lambda^\alpha/\mu^\alpha + \lambda^\beta/\mu^\beta}{C_1 + C_2}$ for equal arrival rates $\lambda^\alpha = \lambda^\beta$, equal mean sevice requirements $1/\mu^\alpha = 1/\mu^\beta = 1$ and different ration of the service rates of the two processors $\frac{C_1}{C_2} = 5, 4, 3, 2, 1, 1/2, 1/3, 1/4, 1/5$.

When the service rate of server 1 is substantially larger than the service rate of server 2, $(C_1 = 5 * C_2)$, then server 1 is used exclusively for almost all arrival rates. When the service rates are $C_1 = 4 * C_2$, then for low and medium load, server 1 is exclusively used, but for heavy system load, server 2 also is used. When the service rates are $C_1 = 3 * C_2$, then the slow server starts been used for lower system load. When the service rates are $C_1 = 2 * C_2$, then the slow server starts been used for even lower system load. When the service rates are equal $C_1 = C_2$, then both servers are used equally $(P_1^{\alpha*} = P_2^{\alpha*} = P_1^{\beta*} = P_2^{\beta*} = 0.5)$. Now, when server 2 is faster, a similar

scenario happens, i.e. the faster server 2 is, the more it is exclusively used.

6 CONCLUSIONS

In this paper, we formulated and solved a *priority load sharing* problem. Real distributed systems assign different priorities to different classes of jobs, in order to give preferential treatment to some classes of jobs. Therefore, it is not meaningful to optimize a single function over all different priority classes simultaneously. In this paper, we have introduced an alternative methodology for dealing with multipriority optimization problems. We formulated a two-priority class load sharing problem as a Stackelberg game with leader the high priority class and follower the low priority class. Furthermore, we gave the explicit solution when two preemptive resume priority classes want to minimize their average job delay. We found that for equal mean service requirements of jobs from both classes, when both processors are used, then the decisions of the low priority class do not depend on the arrival rates, i.e. even if the arrival rates vary, the same routing probabilities can be used for the low priority jobs. Also, for equal mean service requirements of jobs from both classes, when the total arrival rate of jobs is constant but the mix of high and low priority jobs varies, then the overall average job delay remains constant.

Straightforward extensions are to consider multiple priority (> 2) classes, as well as more than two servers. More difficult but more interesting currently is the problem of *dynamic* policies for a multi-priority system. This is currently under investigation.

APPENDIX A

Substituting the Stackelberg equilibrium probabilities into the average delay functions, we have the Stackelberg equilibrium outcome of the game. Let a two processor system $C_1 \geq C_2$. Then we consider several cases:

Case 1: If $C_1 - \sqrt{C_1 C_2} \leq \frac{\lambda^\alpha}{\mu^\alpha}$, then the Stackelberg equilibrium load sharing decisions for the high priority class are given by:

$$P_1^{\alpha *} = \frac{C_1}{\frac{\lambda^\alpha}{\mu^\alpha}} - \frac{C_1 + C_2 - \frac{\lambda^\alpha}{\mu^\alpha}}{\frac{\lambda^\alpha}{\mu^\alpha}} * \frac{\sqrt{C_1}}{\sqrt{C_1} + \sqrt{C_2}}$$

$$P_2^{\alpha *} = \frac{C_2}{\frac{\lambda^\alpha}{\mu^\alpha}} - \frac{C_1 + C_2 - \frac{\lambda^\alpha}{\mu^\alpha}}{\frac{\lambda^\alpha}{\mu^\alpha}} * \frac{\sqrt{C_2}}{\sqrt{C_1} + \sqrt{C_2}}$$

and the average delay of the high priority jobs is:

$$J^\alpha(P_1^{\alpha *}, P_2^{\alpha *}) = \frac{(\sqrt{C_1} + \sqrt{C_2})^2}{\lambda^\alpha * (C_1 + C_2 - \frac{\lambda^\alpha}{\mu^\alpha})} - \frac{2}{\lambda^\alpha}$$

The auxiliary variables become

$$C_1^\alpha = (C_1 + C_2 - \frac{\lambda^\alpha}{\mu^\alpha}) * \frac{\sqrt{C_1}}{\sqrt{C_1} + \sqrt{C_2}}$$

$$C_2^\alpha = (C_1 + C_2 - \frac{\lambda^\alpha}{\mu^\alpha}) * \frac{\sqrt{C_2}}{\sqrt{C_2} + \sqrt{C_2}}$$

$$C_1^\beta = \frac{C_1}{\mu^\alpha} + (C_1 + C_2 - \lambda^\alpha/\mu^\alpha) * (\frac{1}{\mu^\beta} - \frac{1}{\mu^\alpha}) * \frac{\sqrt{C_1}}{\sqrt{C_1} + \sqrt{C_2}}$$

$$C_2^\beta = \frac{C_2}{\mu^\alpha} + (C_1 + C_2 - \lambda^\alpha/\mu^\alpha) * (\frac{1}{\mu^\beta} - \frac{1}{\mu^\alpha}) * \frac{\sqrt{C_2}}{\sqrt{C_1} + \sqrt{C_2}}$$

Then the load sharing decisions for the low priority class β are:

$$P_1^{\beta*} = \frac{\mathcal{C}_1^\alpha}{\frac{\lambda^\beta}{\mu^\beta}} - \frac{C_1 + C_2 - \frac{\lambda^\alpha}{\mu^\alpha} - \frac{\lambda^\beta}{\mu^\beta}}{\frac{\lambda^\beta}{\mu^\beta}} * \frac{\sqrt{\mathcal{C}_1^\beta}}{\sqrt{\mathcal{C}_1^\beta} + \sqrt{\mathcal{C}_2^\beta}}$$

$$P_2^{\beta*} = \frac{\mathcal{C}_2^\alpha}{\frac{\lambda^\beta}{\mu^\beta}} - \frac{\mathcal{C}_1^\alpha + \mathcal{C}_2^\alpha - \frac{\lambda^\alpha}{\mu^\alpha} - \frac{\lambda^\beta}{\mu^\beta}}{\frac{\lambda^\beta}{\mu^\beta}} * \frac{\sqrt{\mathcal{C}_2^\beta}}{\sqrt{\mathcal{C}_1^\beta} + \sqrt{\mathcal{C}_2^\beta}}$$

and the Stackelberg equilibrium average delay of the low priority jobs is:

$$J^\beta(P_1^{\alpha*}, P_2^{\alpha*}, P_1^{\beta*}, P_2^{\beta*}) = \frac{(\sqrt{\mathcal{C}_1^\beta} + \sqrt{\mathcal{C}_2^\beta})^2}{\frac{\lambda^\beta}{\mu^\beta} * (C_1 + C_2 - \frac{\lambda^\alpha}{\mu} - \frac{\lambda^\beta}{\mu^\beta})} - \frac{(\sqrt{C_1} + \sqrt{C_2}) * (\mathcal{C}_1^\beta \sqrt{C_2} + \mathcal{C}_2^\beta \sqrt{C_1})}{\frac{\lambda^\beta}{\mu^\beta} * \sqrt{C_1 C_2} * (C_1 + C_2 - \frac{\lambda^\alpha}{\mu^\alpha})}$$

Case 2: If $C_1 - \sqrt{C_1 C_2} \geq \frac{\lambda^\alpha}{\mu^\alpha}$,

then the Stackelberg equilibrium load sharing decisions for the high priority class α are given by: $P_1^{\alpha*} = 1, \quad P_2^{\alpha*} = 0$

and the average delay of the high priority jobs is:

$$J^\alpha(P_1^{\alpha*}, P_2^{\alpha*}) = \frac{1}{\mu^\alpha C_1 - \lambda^\alpha}$$

The auxiliary variables become

$$\mathcal{C}_1^\alpha = C_1 - \frac{\lambda^\alpha}{\mu^\alpha}$$

$$\mathcal{C}_2^\alpha = C_2$$

$$\mathcal{C}_1^\beta = \frac{C_1}{\mu^\beta} - \frac{\lambda^\alpha}{\mu^\alpha \mu^\beta} + \frac{\lambda^\alpha}{(\mu_a)^2}$$

$$\mathcal{C}_2^\beta = \frac{C_2}{\mu^\beta}$$

Then we consider two cases:

Case 2.1: If $C_1^\alpha - C_2^\alpha * \sqrt{\frac{C_1^\beta}{C_2^\beta}} \leq \frac{\lambda^\beta}{\mu}$ *and* $C_2^\alpha - C_1^\alpha * \sqrt{\frac{C_2^\beta}{C_1^\beta}} \leq \frac{\lambda^\beta}{\mu}$
then the Stackelberg equilibrium load sharing decisions for the low priority class β are given by:

$$P_1^{\beta*} = \frac{C_1 - \frac{\lambda^\alpha}{\mu^\alpha}}{\frac{\lambda^\beta}{\mu^\beta}} - \frac{C_1 + C_2 - \frac{\lambda^\alpha}{\mu^\alpha} - \frac{\lambda^\beta}{\mu^\beta}}{\frac{\lambda^\beta}{\mu^\beta}} * \frac{\sqrt{C_1^\beta}}{\sqrt{C_1^\beta} + \sqrt{C_2^\beta}}$$

$$P_2^{\beta*} = \frac{C_2}{\frac{\lambda^\beta}{\mu^\beta}} - \frac{C_1 + C_2 - \frac{\lambda^\alpha}{\mu^\alpha} - \frac{\lambda^\beta}{\mu^\beta}}{\frac{\lambda^\beta}{\mu^\beta}} * \frac{\sqrt{C_2^\beta}}{\sqrt{C_1^\beta} + \sqrt{C_2^\beta}}$$

$$J^\beta(P_1^{\alpha*}, P_2^{\alpha*}, P_1^\beta, P_2^\beta) = \frac{(\sqrt{C_1^\beta} + \sqrt{C^\beta}_2)^2}{\frac{\lambda^\beta}{\mu^\beta} * (C_1 + C_2 - \frac{\lambda^\alpha}{\mu^\alpha} - \frac{\lambda^\beta}{\mu^\beta})} -$$

$$- \frac{C_1^\beta C_2 + C_2^\beta (C_1 - \frac{\lambda^\alpha}{\mu^\alpha})}{\frac{\lambda^\beta}{\mu^\beta} * (C_1 - \frac{\lambda^\alpha}{\mu^\alpha}) * C_2}$$

Case 2.2: If $C_1^\alpha - C_2^\alpha * \sqrt{\frac{C_1^\beta}{C_2^\beta}} \geq \frac{\lambda^\beta}{\mu}$,
then the Stackelberg equilibrium load sharing decisions for the low priority class β are given by: $P_1^{\beta*} = 1 \quad P_2^{\beta*} = 0$,

and the Stackelberg equilibrium average delay of the low priority jobs is:

$$J^\beta(P_1^{\alpha*}, P_2^{\alpha*}, P_1^{\beta*}, P_2^{\beta*}) = \frac{C_1^\beta}{(C_1 - \frac{\lambda^\alpha}{\mu^\alpha}) * (C_1 - \frac{\lambda^\alpha}{\mu^\alpha} - \frac{\lambda^\beta}{\mu^\beta})}$$

APPENDIX B

Proposition 1: For a given system $C_1 \geq C_2$,
if $\mu^\alpha = \mu^\beta = \mu$, $\lambda^\alpha + \lambda^\beta \leq \mu(C_1 + C_2)$, and $\lambda^\alpha + \lambda^\beta = \lambda = constant$,

then $J(P_1^{\alpha*}, P_2^{\alpha*}, P_1^{\beta*}, P_2^{\beta*}) = constant$

Proof:

Case 1: If $C_1 - \sqrt{C_1 C_2} \leq \frac{\lambda^\alpha}{\mu}$, then the average delay of the high priority jobs is:

$$J^\alpha(P_1^{\alpha*}, P_2^{\alpha*}) = \frac{(\sqrt{C_1} + \sqrt{C_2})^2}{\lambda^\alpha * (C_1 + C_2 - \frac{\lambda^\alpha}{\mu})} - \frac{2}{\lambda^\alpha}$$

and the average delay of the low priority jobs is:

$$J^\beta(P_1^{\alpha*}, P_2^{\alpha*}, P_1^{\beta*}, P_2^{\beta*}) = \frac{(\sqrt{C_1} + \sqrt{C_2})^2}{\mu * (C_1 + C_2 - \frac{\lambda^\alpha}{\mu}) * (C_1 + C_2 - \frac{\lambda^\alpha}{\mu} - \frac{\lambda^\beta}{\mu})}$$

Finally, the overall average job delay is:

$$J(P_1^{\alpha*}, P_2^{\alpha*}, P_1^{\beta*}, P_2^{\beta*}) = \frac{(\sqrt{C_1} + \sqrt{C_2})^2}{\mu * \lambda * (C_1 + C_2 - \lambda)} - \frac{2}{\lambda}$$

Case 2.1: If $\frac{\lambda^\alpha}{\mu} \leq C_1 - \sqrt{C_1 C_2} \leq \frac{\lambda^\alpha}{\mu} + \frac{\lambda^\beta}{\mu}$ *and* $C_2 - (C_1 - \frac{\lambda^\alpha}{\mu}) * \sqrt{\frac{C_1}{C_2}} \leq \frac{\lambda^\beta}{\mu}$, then the average delay of the high priority jobs is:

$$J^\alpha(P_1^{\alpha*}, P_2^{\alpha*}) = \frac{1}{\mu C_1 - \lambda^\alpha}$$

and the average delay of the low priority jobs is:

$$J^\beta(P_1^{\alpha*}, P_2^{\alpha*}, P_1^{\beta*}, P_2^{\beta*}) = \frac{(\sqrt{C_1} + \sqrt{C_2})^2}{\lambda^\beta * (C_1 + C_2 - \frac{\lambda}{\mu})} - \frac{2C_1 - \frac{\lambda^\alpha}{\mu}}{\lambda^\beta * (C_1 - \frac{\lambda^\alpha}{\mu})}$$

Finally, the overall average job delay is:

$$J(P_1^{\alpha*}, P_2^{\alpha*}, P_1^{\beta*}, P_2^{\beta*}) = \frac{(\sqrt{C_1} + \sqrt{C_2})^2}{(\lambda^\alpha + \lambda^\beta) * (C_1 + C_2 - \frac{\lambda^\alpha}{\mu} - \frac{\lambda^\beta}{\mu})} - \frac{2}{\lambda^\alpha + \lambda^\beta}$$

Case 2.2: If $C_1 - \sqrt{C_1 C_2} \geq \frac{\lambda^\alpha}{\mu} + \frac{\lambda^\beta}{\mu}$,
and the Stackelberg equilibrium average delay of the low priority jobs is:

$$J^\beta(P_1^{\alpha*}, P_2^{\alpha*}, P_1^{\beta*}, P_2^{\beta*}) = \frac{C_1}{\mu * (C_1 - \frac{\lambda^\alpha}{\mu}) * (C_1 - \frac{\lambda^\alpha}{\mu} - \frac{\lambda^\beta}{\mu})}$$

Finally, the overall average job delay is:

$$J(P_1^{\alpha*}, P_2^{\alpha*}, P_1^{\beta*}, P_2^{\beta*}) = \frac{1}{\mu C_1 - \lambda}$$

Therefore for constant λ, the $J(P_1^{\alpha*}, P_2^{\alpha*}, P_1^{\beta*}, P_2^{\beta*})$ is also constant. □

APPENDIX C

Proposition 3: For a given system $C_1 \geq C_2$,
if $\frac{\lambda^\alpha}{\mu^\alpha} + \frac{\lambda^\beta}{\mu^\beta} \rightarrow C_1 + C_2$,
then $P_1^{\beta*} \rightarrow \frac{\sqrt{C_1}}{\sqrt{C_1} + \sqrt{C_2}}$.
Case 1: If $C_1 - \sqrt{C_1 C_2} \leq \frac{\lambda^\alpha}{\mu^\alpha}$ *and* $C_2 - \sqrt{C_1 C_2} \leq \frac{\lambda^\alpha}{\mu^\alpha}$,

When the arriving service requirement $\frac{\lambda^\alpha}{\mu^\alpha} + \frac{\lambda^\beta}{\mu^\beta}$ approaches the total service capacity $C_1 + C_2$, then the low priority class β uses both servers. The Stackelberg equilibrium load sharing decisions for the low priority class β approach

$$P_1^{\beta*} \rightarrow \frac{\sqrt{C_1}}{\sqrt{C_1} + \sqrt{C_2}}$$

$$P_2^{\beta*} \rightarrow \frac{\sqrt{C_2}}{\sqrt{C_1} + \sqrt{C_2}}$$

Case 2: If $C_1 - \sqrt{C_1 C_2} \geq \frac{\lambda^\alpha}{\mu}$,
Then we consider two cases:
Case 2.1: If $C_1^\alpha - C_2^\alpha * \sqrt{\frac{C_1^\beta}{C_2^\beta}} \leq \frac{\lambda^\beta}{\mu}$ *and* $C_2^\alpha - C_1^\alpha * \sqrt{\frac{C_2^\beta}{C_1^\beta}} \leq \frac{\lambda^\beta}{\mu}$,

then the Stackelberg equilibrium load sharing decisions for the low priority class βapproach

$$P_1^{\beta*} \rightarrow \frac{\sqrt{C_1}}{\sqrt{C_1}+\sqrt{C_2}}$$

$$P_2^{\beta*} \rightarrow \frac{\sqrt{C_2}}{\sqrt{C_1}+\sqrt{C_2}}$$

□

References

[1] T. Basar and G. J. Olsder. *Dynamic Noncooperative Game theory.* Academic Press, 1982.

[2] D.P. Bertsekas and R. Gallager. *Data Networks.* Prentice Hall, 1987.

[3] E.D.Lazowska D.L.Eager and J.Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. on Software Eng., Vol SE-12, No 5*, pp. 662-675, May 1986.

[4] A.A. Economides and J.A. Silvester. Routing of cooperating and competing multiple classes: Team optimization and nash equilibrium. *Electrical Engineering - Systems Dept.*, University of Southern California, 1989.

[5] L. Kleinrock. *Queueing Systems, Vol. 2: Applications.* J. Wiley & Sons, 1976.

[6] Y.-T.Wang and R.T.J.Morris. Load sharing in distributed systems. *IEEE Trans. on Computers, Vol C-34, No 3*, pp. 204-217, March 1985.

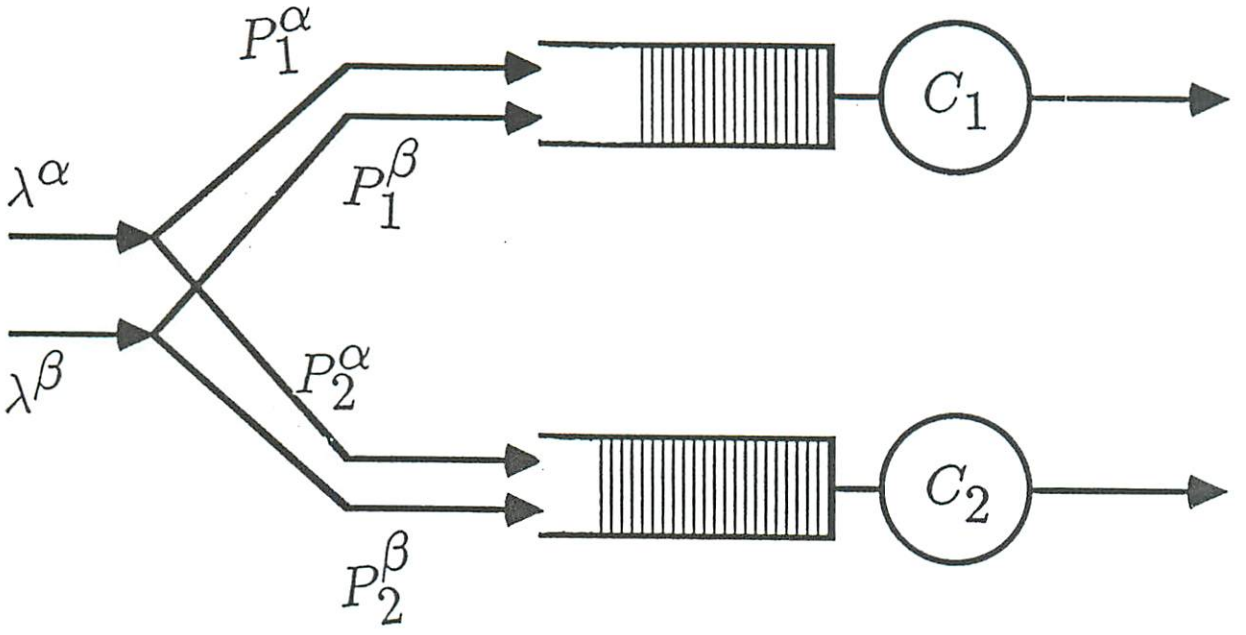


Fig. 1 A Two Priority Load Sharing Problem

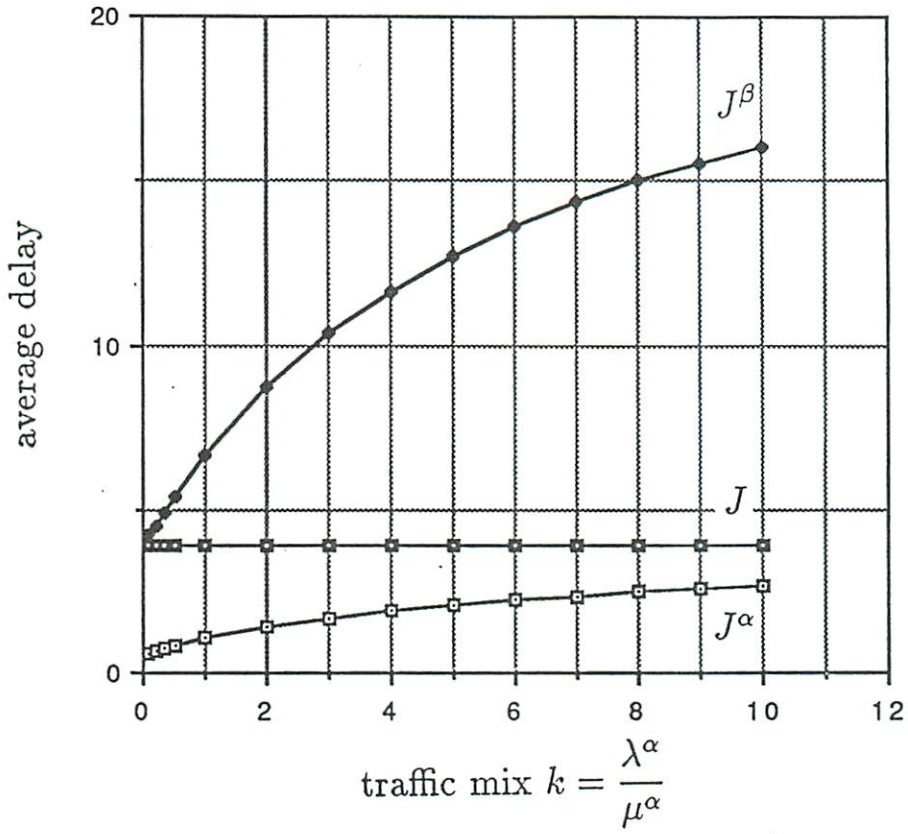


Fig. 2 Stackelberg equilibrium average delay J^α, J^β and J versus different mix of the high and low priority arrival rates $\frac{\lambda^\alpha}{\lambda^\beta}$, for $C_1 = 2, C_2 = 1$, $\lambda^\alpha + \lambda^\beta = 2.5$, and $\mu^\alpha = \mu^\beta = 1$.

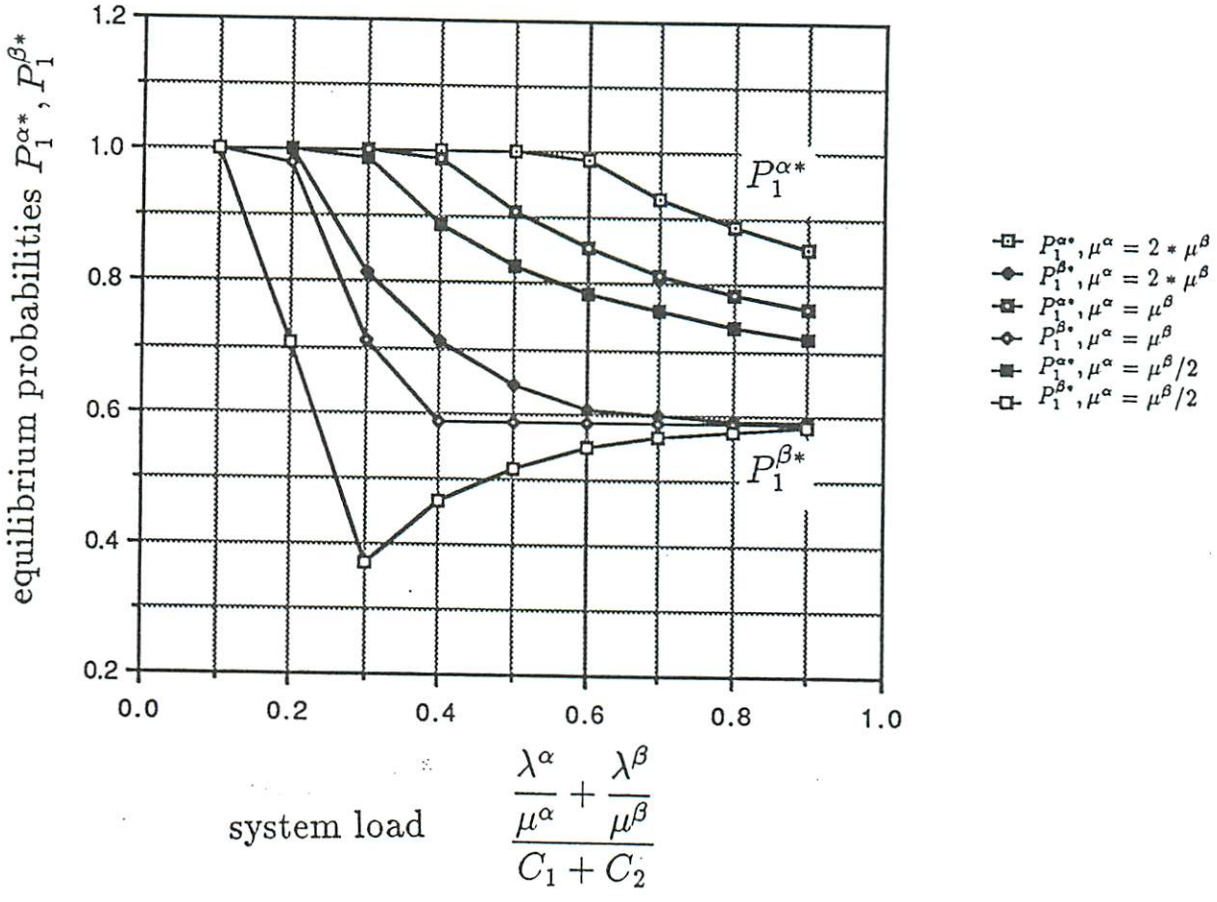


Fig. 3 Stackelberg equilibrium probabilities $P_1^{\alpha*}$ and $P_1^{\beta*}$ versus $\dfrac{\lambda^\alpha/\mu^\alpha + \lambda^\beta/\mu^\beta}{C_1+C_2}$, for $C_1 = 2, C_2 = 1$, $\lambda^\alpha = \lambda^\beta$ and $1/\mu^\alpha = 2/\mu^\beta = 1$, $1/\mu^\alpha = 1/\mu^\beta = 1$, $1/\mu^\beta = 2/\mu^\alpha = 1$.

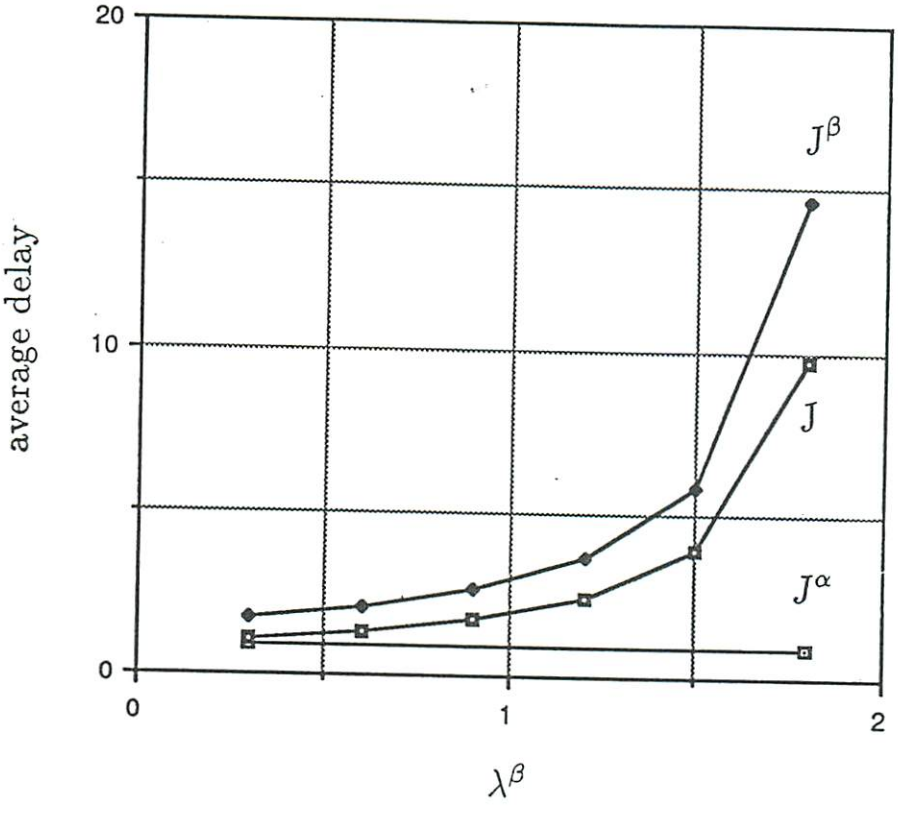


Fig. 4 Stackelberg equilibrium average delay J^α, J^β and J versus λ^β, for $C_1 = 2, C_2 = 1$, $1/\mu^\alpha = 1/\mu^\beta = 1$ and $\lambda^\alpha = 1.0$.

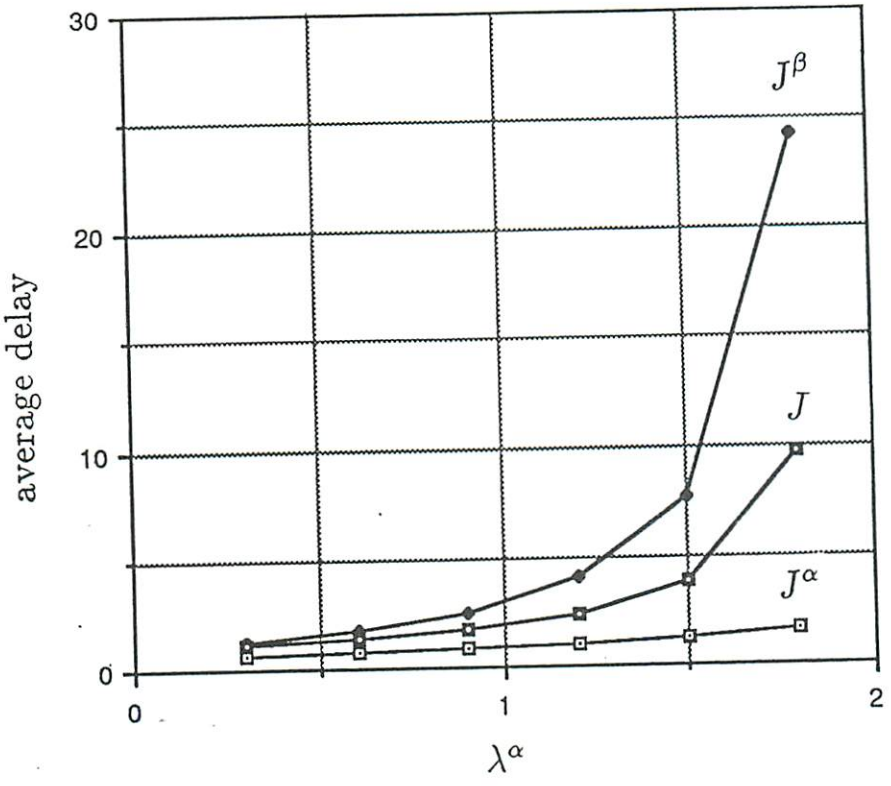
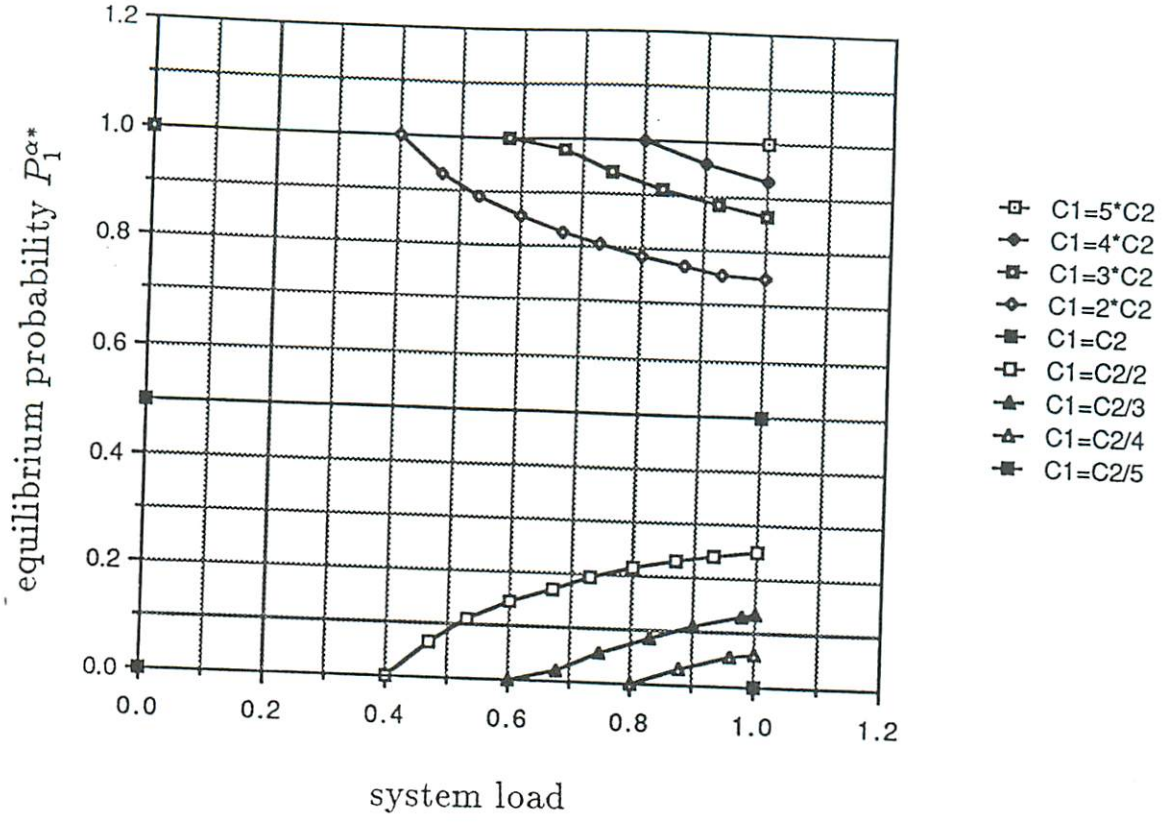


Fig. 5 Stackelberg equilibrium average delay J^α, J^β, and J versus λ^α, for $C_1 = 2, C_2 = 1$, $1/\mu^\alpha = 1/\mu^\beta = 1$ and $\lambda^\beta = 1.0$.



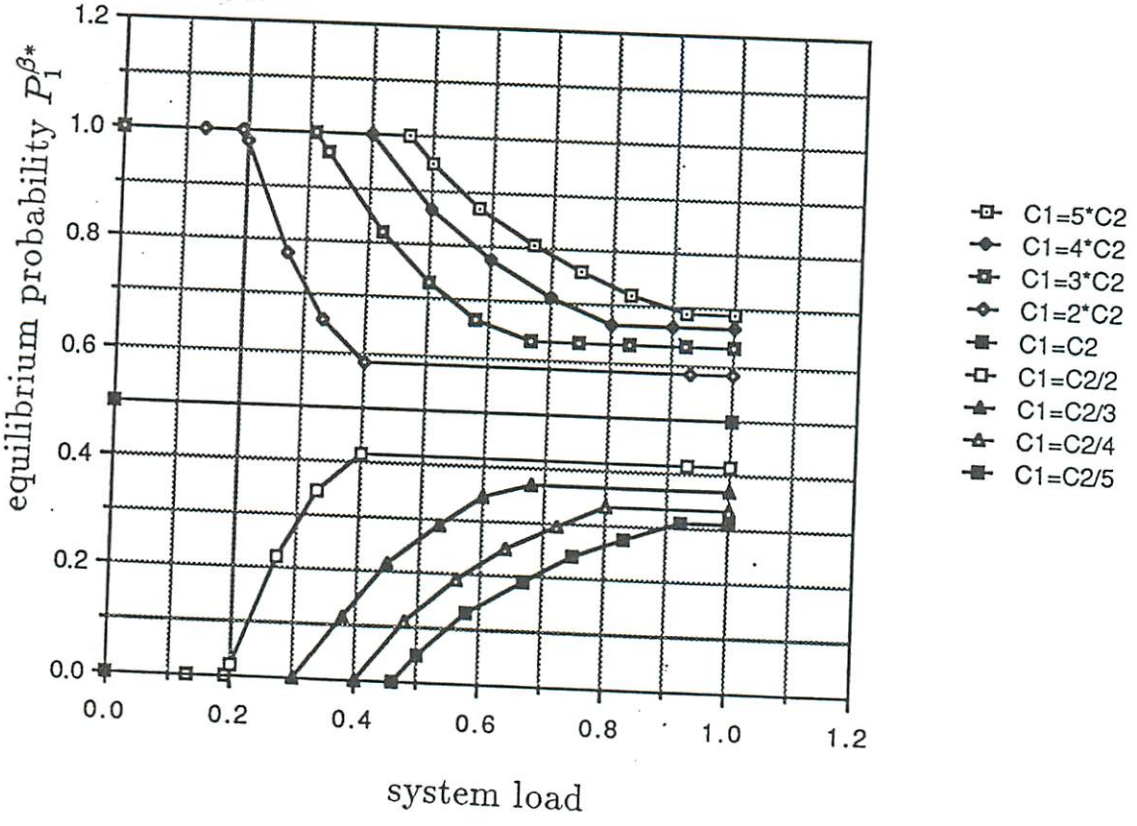


Fig. 6 Stackelberg equilibrium probabilities $P_1^{\alpha*}$ and $P_1^{\beta*}$ versus $\dfrac{\lambda^\alpha/\mu^\alpha + \lambda^\beta/\mu^\beta}{C_1 + C_2}$, for $\lambda^\alpha = \lambda^\beta$, $1/\mu^\alpha = 1/\mu^\beta = 1$ and $\dfrac{C_1}{C_2} = 5, 4, 3, 2, 1, 1/2, 1/3, 1/4, 1/5$.

ASYNCHRONOUS ITERATIVE ALGORITHMS FOR PROBLEMS WITH DISCRETE DATA

by

Aydın Üresin

Technical Report: 90-15

A Dissertation Presented to the

FACULTY OF THE GRADUATE SCHOOL

UNIVERSITY OF SOUTHERN CALIFORNIA

In Partial Fulfillment of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

(Electrical Engineering)

June 1990

Dedication

To my parents Dr. Necati Üresin and Gönül Üresin.

Acknowledgements

I wish to thank a number of individuals who helped me successfully complete my Ph.D. studies. My advisor, Dr. Michel Dubois, certainly played the major role in all my accomplishments throughout my studies at USC. His technical and personal support was invaluable; without this support I would not have been able to achieve my goal. I do not have enough room here to sufficiently express my appreciation for his help.

Dr. Kai Hwang and Dr. Dean Jacobs made contributions in this thesis work by serving on my committee. I am also grateful to Dr. Hwang for giving me the opportunity to join the OMP project, in the late stage of my studies. It was a wonderful experience to work with him and the other excellent people on the project.

Dr. George Papavassilopoulos and Dr. François Robert (from France) were trusty sources of advice, information, and encouragement. They too deserve my gratitude for showing their appreciation in my research work. In the early stages of my studies at USC, Dr. James Ellison recognized my talent and encouraged me. I am very fortunate to know him. Dr. Dan I. Moldovan, who taught the graduate course in parallel processing played a role in my interest in this area. I appreciate Dr. Cauligi S. Raghavendra, Dr. Jean-Luc Gaudiot and Dr. Ramakant Nevatia for serving on my qualifying exam committee.

Finally, I would like to mention my friends Jin Chin Wang, Krishnan Ramamurthy, Koray Öner, Christoph Scheurich and our group secretary Lucille Stivers, all of whom made my life enjoyable during my studies. Unfortunately, I am unable to list all of my friends in the department. I wish them success.

Contents

Dedication ii

Acknowledgements iii

List Of Tables vi

List Of Figures vii

Abstract ix

1 INTRODUCTION 1
- 1.1 Iterative Algorithms . . . 1
- 1.2 MIMD Architectures . . . 1
- 1.3 Task Allocation . . . 2
- 1.4 Synchronous Iterative Algorithms . . . 4
- 1.5 Asynchronous Iterative Algorithms . . . 5
- 1.6 Previous Work . . . 7
- 1.7 Contributions of the Thesis . . . 8

2 MODELS OF ASYNCHRONOUS COMPUTATIONS 9
- 2.1 Mathematical Background . . . 9
- 2.2 Asynchronous Iterations . . . 10
  - 2.2.1 Formal Definition . . . 10
  - 2.2.2 Interpretations of t . . . 11
  - 2.2.3 Redundancy . . . 12
- 2.3 Totally Asynchronous Iterations . . . 12
  - 2.3.1 Definition . . . 12
  - 2.3.2 Interpretations of Total Asynchronism . . . 14
  - 2.3.3 Main Convergence Result . . . 16
  - 2.3.4 Necessary Condition . . . 19
  - 2.3.5 Other Sufficient Conditions . . . 22
    - 2.3.5.1 Monotonic Operators . . . 22
    - 2.3.5.2 Componentwise Contraction . . . 25
    - 2.3.5.3 A Condition Based on Dependencies . . . 26
    - 2.3.5.4 Numerical Problems . . . 27
- 2.4 Asynchronous Iterations With Bounded Delay . . . 27
  - 2.4.1 Definition . . . 27
  - 2.4.2 Convergence . . . 29
  - 2.4.3 Relationship With Total Asynchronism . . . 31
  - 2.4.4 A Neural Network Example . . . 32
  - 2.4.5 Enforcing the Model on a Shared Memory Architecture . . . 39
- 2.5 Non-Redundant Asynchronous Computations . . . 43
  - 2.5.1 Definition . . . 43
  - 2.5.2 Convergence . . . 43

2.6 Partial Asynchronism . . . 46
2.6.1 Definition . . . 46
2.6.2 Summary of Previous Work . . . 47
2.6.2.1 Maximum Contraction . . . 47
2.6.2.2 Gradient–Like Optimization Algorithms . . . 49
2.7 Dynamic Iteration Operators . . . 50
2.8 Other Models of Asynchronism . . . 52
2.8.1 Ordered Schedules . . . 52
2.8.2 Serial Computations . . . 53
2.8.3 Stochastic Models . . . 53

3 APPLICATIONS 54
3.1 Constraint Satisfaction Problems . . . 54
3.1.1 Overview . . . 54
3.1.2 Correctness . . . 55
3.1.3 An Implementation . . . 56
3.2 Transitive Closure . . . 57
3.2.1 Functional Dependencies in Relational Database Design . . . 57
3.2.2 Production Systems . . . 57
3.3 Logic Programming . . . 58
3.3.1 Logic Programming as Constraint Satisfaction Problem . . . 58
3.3.2 Logic Programming With Uncertainties . . . 58
3.4 An All Pairs Shortest Path Algorithm . . . 58
3.4.1 Synchronous Parallel Floyd's Algorithm . . . 58
3.4.2 Simple Asynchronous Floyd's Algorithm . . . 59
3.4.3 Partitioning . . . 61
3.4.4 Early Updating of $ROUND$ . . . 62
3.4.5 Experimental Results . . . 64
3.5 Other Shortest Path Algorithms . . . 67

4 CONVERGENCE RATE 68
4.1 Introduction . . . 68
4.2 Summary of Related Research . . . 68
4.3 Shared Memory Multiprocessors . . . 70
4.3.1 General Model . . . 70
4.3.2 Strongly Coupled Computations . . . 72
4.3.2.1 Description of the Model . . . 72
4.3.2.2 Probabilistic Analysis . . . 76
4.3.3 Partially Coupled Iteration Operators . . . 81
4.3.3.1 General Considerations . . . 81
4.3.3.2 Self Coupled Iteration Operators . . . 83
4.3.3.3 Colorable Iteration Operators . . . 83
4.3.4 Heterogeneous Tasks . . . 86
4.3.4.1 Global and Local Computations . . . 86
4.3.4.2 Static Allocation . . . 88
4.4 Summary of Results . . . 90

5 CONCLUSION 92

Reference List 93

List Of Tables

2.1 A numerical example 39

3.1 Initial allocation of data 56
3.2 Implementation of simple Floyd's Algorithm 65
3.3 Implementation Floyd's Algorithm with early updating 66

List Of Figures

1.1 Multiprocessor architectures 3

2.1 A redundant computation 13
2.2 Asynchronous distortion 17
2.3 A monotone operator 24
2.4 An asynchronously contracting operator which is not monotone 24
2.5 A componentwise contracting operator which is not monotone 26
2.6 Demonstration of Definition 2.4 29
2.7 Demonstration of Lemma 2.9 30
2.8 The function $\phi(\cdot)$ 33
2.9 Update directions in different regions 34
2.10 Totally asynchronous convergence 35
2.11 Sequences y^k and z^k 36
2.12 Demonstration of the implementation 41
2.13 The algorithm to implement Model 2.3 44
2.14 An operator convergent under non-redundancy but not under total asynchronism 46

3.1 Synchronized parallel Floyd's algorithm 59
3.2 Simple asynchronous Floyd's algorithm 60
3.3 Asynchronous Floyd's algorithm for partitioned data 62
3.4 Partitioning A 63
3.5 Early updating of $ROUND$ 64

4.1 An asynchronous iteration 71
4.2 Iteration vs. pseudo–cycle 74
4.3 Comparison of estimated and simulation results for execution time (Uniform Distribution) 77
4.4 Comparison of estimated and simulation results for slowdown (Uniform Distribution) 77
4.5 Comparison of estimated and simulation results for execution time (Normal Distribution) 78
4.6 Comparison of estimated and simulation results for slowdown (Normal Distribution) 78
4.7 Slowdown vs. Q for small fluctuations 79
4.8 Slowdown vs. Q for large fluctuations 79
4.9 Effect of scheduling policy 80
4.10 An example of partial coupling 82
4.11 Estimated execution time for strong and self coupling 84
4.12 Estimated slowdown factor for strong and self coupling 84
4.13 Red-black ordering 85
4.14 The k-th pseudo–cycle of $\{\varphi''(k)\}$ 87
4.15 The computation is not skewed 89
4.16 The computation is skewed 89

Abstract

A significant number of algorithms in numerical and symbolic computing are iterative. It is not unusual that the computations in each iteration of an iterative algorithm can be decomposed, leading to a parallel execution. Such an algorithm can be implemented synchronously or asynchronously in a multiprocessor system. In a synchronous parallel algorithm (synchronous iteration), processors have to synchronize at the end of each iteration. This may sometimes cause a significant performance degradation. Asynchronous iterative algorithms (asynchronous iterations), on the other hand, do not need this type of synchronization and they allow each processor to run at its own pace. Many asynchronous iterative algorithms for important numerical problems have previously been shown to converge to desired solutions.

The first part of the thesis is devoted to general convergence conditions under different asynchronous computational models. The emphasis is on problems with discrete data, although the results have implications in numerical computing. The general convergence conditions are applied to some non-numerical problems including constraint satisfaction, transitive closure, logic programming and some shortest path problems. It is seen that for most of these applications, the convergence is guaranteed under the most general model which allows redundancy and multiple and inconsistent copies of the same data. Redundancy means that the same component may be computed by multiple processors at the same time. In order to guarantee the convergence of redundant computations, the corresponding iteration operator must have a unique fixed point in the considered data domain. On the contrary, convergence under a non-redundant computational model does not imply the uniqueness of a fixed point. When there are multiple fixed points, different non-redundant executions of the same algorithm may converge to different fixed points. If there is a unique fixed point, on the other hand, the existence of multiple inconsistent copies of the same data in the system does not affect the convergence. In the last part of the thesis, a model is given to estimate the performance of asynchronous iterative algorithms. Using this model, the performance of asynchronous iterative algorithms is compared to the performance of their synchronized counterparts.

Chapter 1

INTRODUCTION

1.1 Iterative Algorithms

A significant number of algorithms in the areas of both numerical and non-numerical computing fall into the class of *iterative algorithms*. These algorithms are characterized by an iteration operator F successively applied to some data x, starting with the initial value of the data $x(0)$, and are represented by the following recursive relationship:

$$x(k+1) = F(x(k)), \quad k = 0, 1, 2, \ldots \tag{1.1}$$

where $x(k)$ is the value of x at the k-th iteration.

Some examples of numerical analysis problems that are solved by iterative algorithms are linear systems of equations, partial and ordinary differential equations and optimization [7]. Similarly, many graph problems such as network flow, shortest path, and transitive closure can be cast into the above formulation. Iterative algorithms and paradigms are also quite common in artificial intelligence and computer vision, including constraint satisfaction algorithms, execution of production systems and logic programs.

Typically, the iterated data x and the iteration operator F have multiple components, and this allows us to write (1.1) as

$$\begin{aligned} x_1(k+1) &= F_1(x(k)) \\ x_2(k+1) &= F_2(x(k)) \\ &\vdots \\ x_n(k+1) &= F_n(x(k)) \end{aligned}$$

for all $k = 0, 1, 2, \ldots$ It immediately follows that in each iteration, F_i's can be computed in parallel.

1.2 MIMD Architectures

In the parallel processing systems considered in this thesis, no assumption can be made about either the speed of each individual processor in the system, or the time it takes to compute each F_i even if the processing speed is predictable. This is the characteristic of MIMD (multiple–instruction multiple–data) systems, data-flow computers and distributed systems. Our emphasis is on MIMD systems, although the results are applicable to the other models. An MIMD architecture is composed of several independent processors, each capable of executing a separate program [22], [45]. The processors are interconnected in a way which permits programs

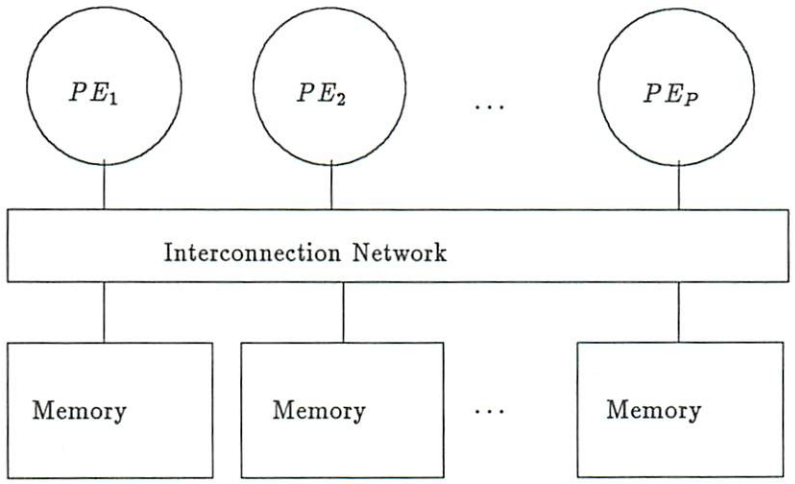
to exchange data and synchronize activities. We will interchangably use the term *multiprocessor* to refer to an MIMD architecture, although this terminology may not be universally accepted.

There are two major classes of MIMD systems, namely, *shared memory multiprocessors* and *message passing systems.* In shared memory multiprocessors (Figure 1.1(a)), there are globally shared memory modules which can be accessed by all the processors through the interconnection network. The interconnection network can be as simple as a bus, or as complex as a crossbar network. The interactions occur only through the shared memory system. In message passing multiprocessors (Figure 1.1(b)), each processor has its own local memory which cannot be accessed by other processors. The system supports communication through point–to–point exchange of information.

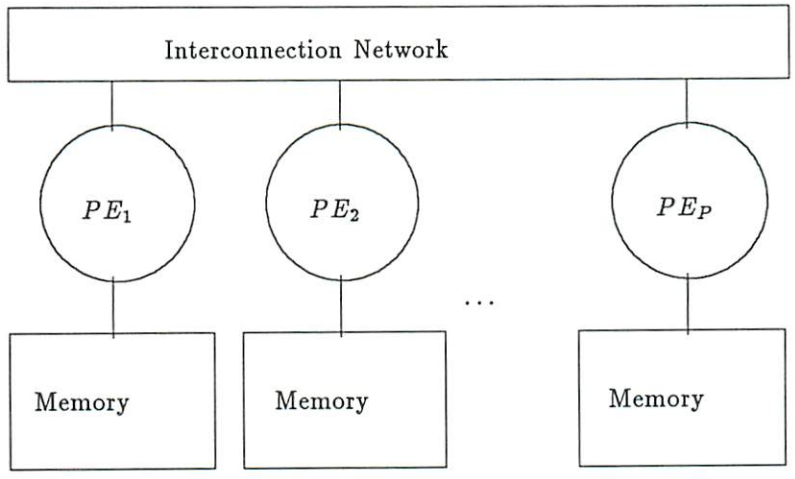
These two classes of multiprocessors represent the extreme cases. Practical designs may lie at the extremes, or anywhere in between.

1.3 Task Allocation

For our purposes, the computation of an F_i in (1.1) can be referred as a *task*, and the assignment of tasks to available time slots of processors is called the *task allocation.* We will assume that the execution of a task is exclusively performed by a processor without interruption. If (1.1) is implemented on an MIMD system in which there are as many processors available as components, then the computation of each F_i can be assigned to a different processor. Assuming that the processors in the system are identical and ignoring the communication between the processors, the task allocation scheme in this case is unique. However, it is not uncommon that there are fewer processors available than components. In this latter case, the allocation of F_i's to processors is not unique and *static allocation* or *dynamic allocation* may be adopted. In the static allocation, the assignment of F_i's to processors is made a priori before the start of the computations and remains fixed throughout. In the dynamic case, the decision as to which component to execute next is made dynamically at runtime. Typically, there is a global queue of tasks and when a processor becomes available, the current state of the task queue determines the task it executes next. Different policies may be chosen to organize the task queue, such as FIFO (First–In–First–Out), R-R (Round–Robin), etc. Conceptually, the allocation scheme based on a unique global queue implies that the selection of the next task is *centralized.* On the contrary, in a scheme based on static allocation, the processors do not have to consult a global structure to select the next task: this decision is *decentralized.* It is also possible to design an allocation scheme which is a combination of the two extreme cases. For example, we can maintain multiple queues in the system; each queue schedules only a predetermined group of components and is accessed only by a predetermined group of processors. The advantage of the static allocation is that it suffers no overhead of maintaining and runtime accessing a global structure for scheduling purposes. However, in the case when it is not possible to predict the total workload (i.e., the total amount of computation to execute F), and not possible to distribute the load equally among the processors, the static allocation results in a poor load balance. Dynamic allocation improves load balance, but at the expense of the overhead caused by global queue accesses.



(a)



(b)

(a) Shared memory multiprocessor architecture
(b) Message passing multiprocessor architecture
(PE_i : Processor i)

Figure 1.1: Multiprocessor architectures

1.4 Synchronous Iterative Algorithms

If we desire to strictly enforce the sequence of events described by (1.1), regardless of the task allocation strategy we implement, we must ensure that all the computations for the k-th iteration are completed before the computations for the $(k+1)$-st iteration can start. This requires a *barrier synchronization* after each iteration. A barrier synchronization is a logical point in the control flow of an algorithm at which all the processors must arrive before any of them are allowed to proceed further. The iterative algorithms for which the state changes are exactly described by (1.1) are called *synchronous iterative algorithms* or *synchronous iterations*, and they use barrier primitives extensively. They suffer performance degradation caused by synchronization barriers. Synchronization barriers impose two kinds of performance penalties. The first one is due to the load imbalance, i.e., due to the fluctuations in the time taken by the processors to complete their work section before arriving at the barrier. The second kind of penalty results from the execution of the barrier.

The overhead introduced by the load imbalance was formulated by an approximation by Kruskal and Weiss [24]. They estimated the expected time T to complete the execution of Q independent tasks allocated by batches of size K on P processors. It was assumed that running times of tasks were independent identically distributed (i.i.d.) random variables with mean μ and variance σ^2. They require some additional restrictions. The precise description of these restrictions is beyond the scope of this thesis, but exponential and Gaussian distributions and large values of P will satisfy the requirements. The result is as follows.

$$T \approx \frac{Q}{P}\mu + \sigma\sqrt{2\,K \cdot \log P}.$$

It should be clear that the second term corresponds to the penalty of load imbalance.

The second form of performance penalty is related to the implementation of the synchronization barrier. In shared memory multiprocessors, a common implementation involves two stages [2], [18]. In the first stage, the processors check in at the barrier by simply incrementing a global counter which is initially reset. The last processor that checks in records this event, typically by setting a shared flag which is initially reset. Each processor, after checking in, polls the flag until it is set. The last processor will see the counter value as P (the number of processors), and will set the flag. Regarding the execution time of the barrier, the worst case occurs when the processors arrive at the barrier simultaneously. Usually, the counter can be accessed by no more than one processor at a time. Assume the worst case, and suppose that incrementing the counter takes t_c units of time. Then, the barrier execution takes at least $P\,t_c$ units. In general, the time complexity of this type of barrier is roughly $O(P)$.

The above described linear barrier is clearly damaging to the system performance for large numbers of processors. In a faster scheme, there are more than one barrier counters organized as a tree structure [2], [18]. The processors are partitioned and each partition is assigned to a leaf of the tree. The maximum value of the counter on a leaf is the size of the corresponding partition, and the maximum value of an inner node counter is equal to the number of its children. Each node also has a flag associated with it and it is set when its counter reaches the maximum value. The leaf counters count the number of processors in each partition that have checked in, and the other counters count the number of set child flags. The barrier execution is complete when the root flag is set. If the tree is balanced, it has $O(\log P)$ levels. Since we can consider the whole process as a propagation from the leaves to the root, the complexity of this type of barriers is roughly

$O(\log P)$. Although for a large P, logarithmic barriers should be preferred over linear barriers, for smaller numbers of processors it may not be worthwhile to do so.

In message passing systems, the barrier at the end of each iteration k involves communicating all the components of $x(k)$ to all the processors. The time complexity of this operation (multinode broadcast) depends on the interconnection architecture, and it is given in [7] for several architectures. In most cases it is $O(P)$.

So far we have assumed that the computation of each F_i in (1.1) depends on all the components of x. Sometimes there is only local dependency and it is not necessary to demand that all the computations in the k-th iteration are completed before the computations in the next iteration can start. It follows that if the computation of an F_i depends only on a subset of all the components, we do not need a barrier, and we can design synchronization schemes in which the computation of F_i waits only for the components it depends on. Evidently, this type of relaxed synchronization reduces the overhead caused by both load unbalance and execution of synchronization. The disadvantage is that programming such a scheme may become complicated.

1.5 Asynchronous Iterative Algorithms

An *asynchronous iterative algorithm* (also called *asynchronous iteration*) can be defined as an algorithm which is obtained by simply removing the synchronization points between the iterations in a synchronous iterative algorithm. Each processor participating in an asynchronous algorithm computes at full speed and efficiency. It uses component values as it needs them, even if they are not the most recent ones, and it releases new values as it produces them [25]. In its most general form, an asynchronous iteration allows any type of task scheduling: static, dynamic, centralized or decentralized. Since in a decentralized scheme the processors do not have information about the status of each other, it is possible that the same component is computed by more than one processors at the same time. As a result of this redundancy, the computation of a component does not necessarily use the most recent value of even the same component.

Let us make it clear that asynchronous iterations are not always free of all forms of synchronization. In shared memory systems, the need for synchronization arises in order to protect the integrity of data elements, i.e., to provide *atomic* access to data elements. For example, each component of x in (1.1) should be accessed atomically. This means that while a processor is updating a component x_i, no other processor should be allowed to read x_i, possibly having an intermediate and illegitimate value. In practical systems, atomic access to a data component is usually guaranteed by hardware, if the size of the component is the same as the size of an addressable physical memory location. However, components of x may be more complex data, such as high precision floating point numbers, records, lists. They may also be smaller pieces of addressable memory locations, such as bits and characters. In these cases, components of x should be accessed in *critical sections*. A critical section with label L is a form of synchronization, and can be defined as a code section of a parallel program such that only one critical section with label L can be executed at any given time. A common implementation of critical sections is to use the LOCK and UNLOCK operations [22]. These operations are atomic and have an argument of data type *lock*. A lock is defined to have two states: open and locked. LOCK(L) waits until the lock L is open and it locks L when it is open. UNLOCK(L) simply changes the state of L to open. Therefore, a critical section with label L is equivalent to the following code.

LOCK(L)
execute critical section
UNLOCK(L)

Critical sections are also required to access task queues when centralized allocation schemes are employed.

Naturally, the major concern regarding an iterative algorithm is whether it converges to a desired result. It is clear that an asynchronous version of a synchronous algorithm does not always converge. Fortunately, however, for significant number of problems, asynchronous iterative implementations are guaranteed to converge to desired results. Some of the problems that can be solved by asynchronous iterations are linear systems of equations, optimization problems, partial differential equations, ordinary differential equations, shortest path problems, network flow problems, routing in data networks, transitive closure, parallel prefix problem, filtering for constraint satisfaction problems, neural network simulation [7], [52].

Asynchronous iterations were originally motivated by their potential to improve performance by removing synchronization, which is a major source of performance degradation in MIMD multiprocessors. However, they have several other interesting properties that make them useful. The advantages of asynchronous iterations can be summarized as follows.

1. They increase processor efficiency, because they do not suffer performance penalties due to load imbalance and to synchronization. This usually makes them faster than their synchronous counterparts.

2. The processors work on more recent component values as compared to their synchronized counterparts since component values are released as soon as they are computed. This is another reason why they converge faster. For exactly the same reason, Gauss-Seidel iterations are superior to their Jacobi versions on a uniprocessor system.

3. In cache based shared memory machines, processors work on their local cache copies of shared data, but if there are multiple copies of the same data element in the system, they must be kept consistent. This major source of overhead in such systems is called *cache coherence*. Asynchronous iterations do not require to enforce cache coherence. Processors can simply work on their possibly incoherent copies. The only condition for convergence is that updates eventually (in a finite amount of time) propagate to all processors.

4. In synchronous iterations implemented on message passing systems, communication between the processors should be synchronous. In other words, a processor which sends a message is blocked until it acknowledges the reception of the message. Since asynchronous iterations do not need this restriction, they can overlap communication with computation. Therefore, maximum throughput is attainable by keeping both processors and communication channels busy.

5. For a synchronized execution of an iterative algorithm in a shared memory system, processors are blocked during a shared memory access. Since asynchronous iterations do not require processors to wait on memory accesses, they can achieve both maximum processing speed and maximum memory throughput at the same time.

6. The task allocation scheme of an asynchronous iteration can be such that each component can be computed by more than one processor (not necessarily at the same time). In this case, if a processor

fails, no component is discarded forever from future updates. Such asynchronous iterations are therefore fault-tolerant.

7. In some distributed and real-time applications, the coefficients of a problem may change with time as new information is gathered by the system. An iteration designed as an asynchronous iteration may be "robust" enough to converge even if a change occurs in the problem. In this case, the iteration may not have to restart, but can simply continue.

Even with these advantages, asynchronous iterations are not always superior to their synchronized counterparts. Some of their disadvantages are listed below.

1. Although as a result of the recent developments in asynchronous iterations there is a comprehensive theory which provides us techniques for convergence proofs, the techniques may not always be easily applicable to particular problems. Needless to say, an asynchronous implementation of a problem cannot always be proven to be convergent.

2. Even though an asynchronous iteration can be proven to be convergent, in the case where there are multiple fixed points, the asynchronous implementation may show nondeterministic behavior. In this case, the asynchronous iteration may converge to one or the other, depending on the timing of events.

3. Usually asynchronous iterations converge faster than their synchronized counterparts. However, it is not possible to generalize this result. Under some worst-case conditions, asynchronous iterations may be up to two times slower.

4. Detecting the convergence of an asynchronous iteration and terminating the algorithm accordingly may sometimes be difficult. For example, consider a synchronous algorithm which is guaranteed to converge in some fixed number of steps, say M. The asynchronous implementation may not converge after each task is executed M times.

1.6 Previous Work

In this section, a summary of the previous related work by other researchers is given. In the remaining chapters of the thesis, more detail will be provided when needed.

Asynchronous iterations were first introduced by Chazan and Miranker [12]. They were then called *chaotic iterations*. The assumptions of their computational model were twofold: (i) all the components are updated infinitely often; (ii) there exists an upper bound such that the input components for the updates cannot be more outdated than the bound. They showed that asynchronous iterations corresponding to a linear system of equations with non-negative coefficients converge to the desired solution if and only if the spectral radius of the underlying matrix is less than one. In [25], Kung discussed issues related to implementations of asynchronous iterations in multiprocessors. Robert, Charnay and Musy [41] considered *serial–parallel iterations* for solving non-linear problems. Serial–parallel scheme of computations is a restricted model and it does not allow outdated input values. [28], [38] and [31] also studies asynchronous iterations corresponding to non-linear operators. Baudet proved a convergence condition for non-linear problems, under the most general asynchronous computational model, where no bound exists for the recency of the input components.

The general computational model described by Baudet was later called *totally asynchronous* in [7]. In [44], [29] and [32], asynchronous iterations for some applications in numerical computing were studied.

Bertsekas obtained a general convergence result that does not assume real data domains [4]. In [30] and [4] the convergence for monotone operators were considered. [46], [49], [50] and [51], contain several applications for a model more restricted than partial asynchronism but weaker than serial–parallel computations. This model is called *partial asynchronism.* The partially asynchronous computational model was also considered in [27].

Robert derived a convergence condition in terms of dependency information, for serial–parallel computations, and when the data domain is binary [39] , [40]. Tsitsiklis considered a similar computational model and established general necessary and sufficient conditions for convergence [48].

In a recent publication, Bertsekas and Tsitsiklis studied some results related to convergence rate and termination of asynchronous iterations [6]. The book by these authors contains most of the previous work in the area of asynchronous iterations [7].

1.7 Contributions of the Thesis

Most of the earlier work has concentrated on numerical problems and the mathematical tools used to derive convergence results have been borrowed from the numerical analysis discipline. The initial purpose of this thesis research was to generalize the previous results to allow asynchronous algorithms in symbolic computing. Consequently, a general convergence condition for asynchronous iterations has been established in Chapter 2. This convergence result has later been seen to be similar to the one given in [4]. However, our result is conceptually simpler and the computational model of [4] is more restricted which assumed that the processors are assigned to fixed and disjoint sets of components.

Another contribution of the thesis is the general convergence condition for *asynchronous iterations with bounded delay.* This model differs from partial asynchronism by allowing redundant computations. Chapter 2 also provides a sharper classification of asynchronous computational models than those given elsewhere. The implications of the redundancy and the boundedness of delays are clarified.

The thesis also makes a contribution in the area of applications by establishing the convergence of constraint satisfaction and transitive closure problems. A significant number of problems in artificial intelligence can be reduced to the constraint satisfaction problem and there are even languages based on this problem. The transitive closure also has implications in database optimization and in production systems. The simple asynchronous version of the Floyd's Algorithm was also given in a different context by [11]. However, the extensions of the simple algorithm are new. The actual implementations of these asynchronous algorithms in a multiprocessor have also been done for the first time.

Finally, in Chapter 4, a performance model is introduced, useful to estimate the performance of asynchronous iterations. The differences between this model and the one given in [7] and [6] are that the computation times are not necessarily equal and that the overhead due to the implementation of synchronization primitives is ignored. The results of this thesis, along with [7] and [6] clearly show that the performance of asynchronous iterations is predictable, contrary to the previous belief. Asynchronous executions may not always be favorable. The thesis identifies such worst case conditions where asynchronous iterations can be up to two times slower than their synchronized counterparts.

Chapter 2

MODELS OF ASYNCHRONOUS COMPUTATIONS

2.1 Mathematical Background

Throughout the thesis, $X = X_1 \times X_2 \times \cdots \times X_n$ denotes the set from which the shared data x in (1.1) take values and the iteration operator $F : X \mapsto X$ in (1.1) can be decomposed as

$$F(x) = (F_1(x), F_2(x), \ldots, F_n(x)), \quad \forall x \in X,$$

where $F_i : X \mapsto X_i$, for all $i = 1, 2, \ldots, n$. X is said to be *finite* if it contains a finite number of elements. Furthermore, $\mathcal{N}$, $\mathcal{N}^+$ and $\Re$ denote the sets of non-negative integers, positive integers and real numbers, respectively.

We say that a sequence $\{z(k)\}$ elements of which take values from a set X *converges* to $z^* \in X$ and write $\lim_{k\to\infty} z(k) = z^*$ in either or both of the following cases.

- There exists an integer m such that $z(k) = z^*$ for all $k \geq m$.
- $X \subseteq \Re^n$ and for all $\epsilon > 0$, there exists an integer m such that $|z_i(k) - z_i^*| < \epsilon$ for all $k \geq m$ and $i = 1, 2, \ldots, n$. [1]

A sequence $\{Z(k)\}$ of subsets of X is said to *converge* to $z^* \in X$ ($\lim_{k\to\infty} Z(k) = z^*$) iff all sequences $\{z(k)\}$ such that $z(k) \in Z(k)$, for all k, converge to z^*. An element z^* of X is said to be a *limit point* of a sequence $\{z(k)\}$ if there exists a subsequence of $\{z(k)\}$ that converges to z^*

A *fixed point* of F is defined to be an element of X satisfying $x^* = F(x^*)$. The set of fixed points of F will be denoted as X^*. Also, a function $F : X \mapsto X$ is said to be *continuous* on a set X if for all converging sequences $\{z(k)\}$ of elements of X

$$\lim_{k\to\infty} F(z(k)) = F(\lim_{k\to\infty} z(k)).$$

From this definition, it is clear that when X is finite, every function F under which X is closed is continuous.

Some of our results will be given in terms of an ordering relation. A relation $\preceq$ in a set X is called a *partial order on* X iff, for every $a, b, c \in X$

- (*Reflexivity*) $a \preceq a$,

[1] The reader should be cautioned here that this is not the most general definition of convergence, although it will serve our purpose. In particular, it may not be adequate when X is an infinite and non-numerical domain. In general, a topology on X should be given beforehand and convergence is defined in terms of the topology [34]. This general approach is beyond our scope, but can be chosen in case of need. The results of the thesis would still be valid under the general convergence definition.

- (*Antisymmetry*) $a \preceq b, b \preceq a \Rightarrow a = b$,
- (*Transitivity*) $a \preceq b, b \preceq c \Rightarrow a \preceq c$. □

The ordered pair $(X, \preceq)$ is called a *partially ordered set* or a *poset*. An element $a \in X$ is a *minimal element of* X if no element $b \in X$ exists such that $a \neq b$ and $b \preceq a$. Similarly, we can also define a *maximal element of* X. We write $a \prec b$ if $a \preceq b$, but $a \neq b$, for all $a, b \in X$.

2.2 Asynchronous Iterations

2.2.1 Formal Definition

The following definition formulates the general model of asynchronous iterative algorithms. It was first introduced by Chazan and Miranker, and was then called *chaotic iterations* [12]. We define it in terms of a *schedule* ($\mathcal{S}$), along with an iteration operator and initial data. Informally, a schedule is a timing of accesses to global data. For example, in a shared memory multiprocessor these accesses include both the fetching and the storing of iterate components. In message passing systems the schedule describes the timing of component updates in local memories and communication of these updates to remote processors.

Definition 2.1 (Asynchronous Iteration) *An asynchronous iteration* with respect to the schedule $\mathcal{S} = (\{\alpha(t)\}, \{\tau(t)\})$ corresponding to F and starting with $x(0) \in X$ is a sequence $\{x(t)\}$ such that

$$x_i(t) = \begin{cases} x_i(t-1) & \text{if} \quad i \notin \alpha(t) \\ F_i(x_1(\tau_1^i(t)), x_2(\tau_2^i(t)), \ldots, x_n(\tau_n^i(t))) & \text{if} \quad i \in \alpha(t) \end{cases}$$

for all $t = 1, 2, \ldots$ and $i = 1, 2, \ldots, n$. It is denoted by $(F, x(0), \mathcal{S})$, where $\{\alpha(t)\}$ is a sequence of subsets of $\{1, 2, \ldots, n\}$ and $\{\tau(t)\}$ is a sequence of elements of $\mathcal{N}^{n \times n}$ satisfying $0 \leq \tau_j^i(t) < t$ for all $t = 1, 2, \ldots$ and $i, j = 1, 2, \ldots, n$. □

$\alpha(t)$ is the *update set at* t and we say that the i-th component is *updated at* t if $i \in \alpha(t)$. We assume that all of the components are updated at 0, i.e., $\alpha(0) = \{1, 2, \ldots, n\}$. The vector $u^i(t)$ with components $u_j^i(t) = x_j(\tau_j^i(t))$ is called the *input of the* i*-th component at* t and $F_i(u^i(t))$ is said to be *generated at* t *for the* i*-th component*. The definition implies that the computations start at $t = 0$.

In this formulation, synchronous iterations in fact belong to the class of asynchronous iteration. As an example, the schedule given below corresponds to the synchronous point–Jacobi iteration [3], where the inputs for the computations in each iteration are generated in the previous iteration.

$$\alpha(t) = \{1, 2, \ldots, n\}$$

$$\tau_j^i(t) = t - 1,$$

for all $t = 1, 2, \ldots$ and $i, j = 1, 2, \ldots, n$. Similarly, the schedule corresponding to the Gauss-Seidel iteration, where the components are updated sequentially and where each update uses the most recent iterate values, is shown in the following:

$$\alpha(t) = \{1 + [(t-1) \bmod n]\},$$

$$\tau_j^i(t) = t - 1,$$

for all $t = 1, 2, \ldots$ and $i, j = 1, 2, \ldots, n$.

2.2.2 Interpretations of t

It should be clear that for a given execution of an asynchronous algorithm, the schedule $\mathcal{S}$ is not uniquely defined. A virtually unlimited number of physical interpretations of t as well as x are possible.

A simple interpretation of t would be the index of each update instance of the shared memory. Ordering of t conforms with the real time ordering, i.e., if the updates with index t and $t+1$ occur at real times $r(t)$ and $r(t+1)$, then $r(t) < r(t+1)$ and there are no other updates between $r(t)$ and $r(t+1)$, for all t. With this interpretation, $x(t)$ is the value of shared data right after the t-th update. If the i-th component is updated at this instance, then this update is the result of the computation of F_i using the value of the j-th component right after the update with index $\tau_j^i(t)$, i.e., it is fetched sometime no earlier than $r(\tau_j^i(t))$ and before $r(\tau_j^i(t)+1)$.

Consider the parallel synchronized execution of the Jacobi iteration for three components and three processors. Suppose that the computation time of the first component is 10 real time units. Similarly, the computation times of the second and the third components are 20 and 30 real time units, respectively. With the interpretation described as above, the update instances of the first, the second and the third components in the first iteration are $t = 1, 2$, and 3, respectively. Then, at $t = 4$ the first component is updated, etc.

Although this interpretation of t is perfectly legitimate, it does not show the parallelism inherent in the computations. It describes the sequence of updates as if the same computations were implemented on a uniprocessor system. We wish to define the times of updates in a way that exhibits the parallelism in the computation. For example, in the above Jacobi iteration example, the exact order of updates in real time is not relevant to the convergence of the algorithm. We could also consider that, in each iteration, all the components are updated all at once.

In general, let us consider a sequence $\{\psi(t)\}$ of real time instances and the corresponding sequence $\{I(t)\} \equiv \{(\psi(t), \psi(t+1)]\}$[2] of the time intervals such that

- $I(0) \equiv (\psi_0, 0]$, where $\psi_0 < 0$ is an arbitrary time instance before the computation starts,[3]
- for all t, $I(t)$ covers at least one update instance of some component and at most one update instance of each component,
- for all $t = 1, 2, \ldots$ all the inputs of the updates in $I(t)$ are generated no later than in $I(t-1)$.

Given such $\{I(t)\}$, $x(t)$ can be interpreted as the value of the shared data right at the end of $I(t)$, i.e., at $\psi(t+1)$. If the i-th component is updated in $I(t)$, $\tau_j^i(t)$ can be interpreted as the index of the interval at which the j-th component of the input vector of this update is generated. Since there is an infinite number of sequences of the above form, for the same computation, an infinite number of interpretations of $\{x(t)\}$ is possible. If $I(t)$'s contain single updates, then this interpretation is the same as the one described previously. In order to show maximum parallelism, we will define $I(t)$ such that it contains the maximum number of updates that satisfy the above two conditions. With this definition, $I(t)$ is unique for a given computation.

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

[3] It is assumed that the computations start at real time zero.

2.2.3 Redundancy

In the schedules above, as well as in most typical cases, the computation of a component x_i uses its most recent value as an input component. In other words, $\tau_i^i(t) = t - 1$, for all $i = 1, 2, \ldots, n$ and $t = 1, 2, \ldots$ In general, this condition holds when there are no redundancies in the computations. Redundancy means that the same component may be computed by multiple processors at the same time. Non-redundancy can be identified by the following cases.

1. Static allocation is adopted and the component sets assigned to processors are not overlapping, i.e., each component is assigned to a unique processor.

2. Dynamic allocation is adopted and the task scheduling scheme is centralized. For example, the system may keep a unique copy of the program code for each F_i so that x_i cannot be computed by more than one processor. Or, the global queue governing the allocation of the tasks may be unique and does not allow the scheduling of a task which is presently executing.

Furthermore, these are the only cases that guarantee $\tau_i^i(t) = t - 1$, for all $i = 1, 2, \ldots, n$ and $t = 1, 2, \ldots$, because of the simple reason that in the computational systems we are assuming in this thesis, no assumption can be made about execution times of tasks. In the following, an example of a redundant computation is given.

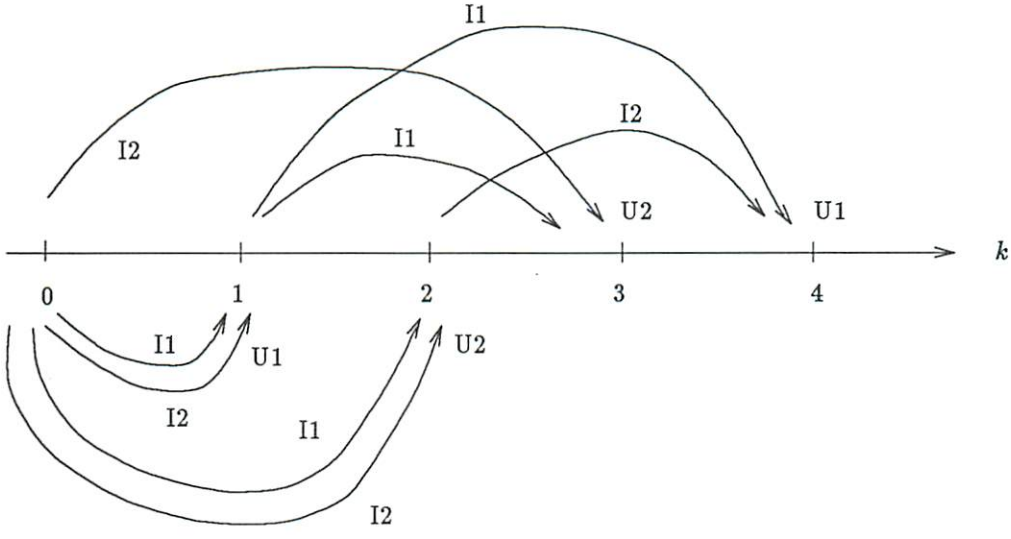
Example 2.1 Assume two components in the shared data computed by two processors in a shared memory system. Each processor computes the components in order using whichever value it fetches from the shared store. Figure 2.1 shows such a schedule. For example, the arrows pointing to $t = 4$ can be interpreted as follows. The processor assigned to component 1 fetches the first component of the shared data at 1 and the second one at 2. Using these values it computes the second component and updates the shared data at 4. The other arrows can be interpreted similarly. The computation of the second component in this example is redundant in the following way. The update of the second component at 3 uses the initial value of that component without being aware of the fact that the most recent value of the second component is generated at 2. □

2.3 Totally Asynchronous Iterations

2.3.1 Definition

Obviously, Definition 2.1 would not be of any value if we did not have any tools to predict the convergence of an algorithm described by it. It should be clear that without any restriction, it is not possible to guarantee convergence, for all possible schedules. At the very least, we should require that all the components should be updated sufficiently many times, i.e., no component drops out of the computations forever (*non-starvation of updates*). Furthermore, computations should not use an outdated component value as their input, forever (*non-starvation of inputs*), although they are allowed to do so for a while. These are very weak conditions which impose practically no restrictions on the system. Fortunately however, for many problems, they are the only requirements for convergence to the desired solutions. They were first introduced by Baudet [3], and the computational models that satisfy these conditions were called *totally asynchronous* by Bertsekas and Tsitsiklis [7]. It is formally defined as follows.

t	1	2	3	4	$\cdots$
$\alpha(t)$	$\{1\}$	$\{2\}$	$\{2\}$	$\{1\}$	$\cdots$
$\tau_1^1(t)$	0	–	–	1	$\cdots$
$\tau_2^1(t)$	0	–	–	2	$\cdots$
$\tau_1^2(t)$	–	0	1	–	$\cdots$
$\tau_2^2(t)$	–	0	0	–	$\cdots$



Ui: Update of the i-th component
Ii: The input component i

Figure 2.1: A redundant computation

Model 2.1 (Total Asynchronism)

- *(Non-starvation of Updates) Each $i = 1, 2, \ldots, n$ occurs in an infinite number of $\alpha(t)$'s, i.e., each component is updated infinitely often.*
- *(Non-starvation of Inputs) Given any $t_1 = 0, 1, \ldots$, there exists $t_2 > t_1$ such that $\tau_j^i(t) \geq t_1$, for all i, j and $t \geq t_2$, i.e., after a coordinate value is generated, this value can be used as inputs of only a finite number of updates.* □

It is useful to state at this point the following property for convergent totally asynchronous iterations.

Lemma 2.1 *Let F be a continuous function on $X = X_1 \times X_2 \times \cdots \times X_n$. Then every point of convergence of all totally asynchronous iterations $(F, x(0), \mathcal{S})$ with $x(0) \in X$, is also a fixed point of F in X .*

Proof. Consider the sequence $\{u^i(k)\}$ of input vectors for the i-th component. From non-starvation of inputs, $\{u^i(t)\}$ converges to the same point x^* as $\{x(t)\}$. From the definition of continuity

$$x_i^* = \lim_{t \to \infty} u_i^i(t) = \lim_{t \to \infty} x_i(t) = \lim_{t \to \infty} F_i(u^i(t)) = F_i(x^*).$$

Since this holds for all $i = 1, 2, \ldots, n$, the point of convergence is a fixed point. □

In Section 2.3.3, we will prove a general condition on the iteration operator F, under which all totally asynchronous iterations converge to the fixed point of F. It will further be proven that this condition is also necessary for convergence if the domain X of the shared data contains finite number of elements. Before that, we will discuss how to interpret totally asynchronous iterations implemented in some shared memory architectures.

2.3.2 Interpretations of Total Asynchronism

In Section 2.2.2 we have already shown that the interpretation of the *time* of $x(t)$ is not unique. Another degree of freedom we may exploit is in the interpretation of the *place* of $x(t)$. In this section, we explain this in three example shared memory systems. The last two examples show that totally asynchronous iterations allow multiple inconsistent copies of x in the system.

1. Consider a shared memory multiprocessor as shown in Figure 1.1(a), where the computations are controlled by a static allocation scheme. Each component is assigned to be computed by fixed processors throughout the execution of an asynchronous iteration. The requirement of total asynchronism on this system is trivial: No processor is allowed to drop out of the execution forever (non-starvation of updates), and the computation time of each component is finite (non-starvation of inputs). Because of the static allocation, this property satisfies both conditions of Model 2.1. Convergence is not affected when some of the processors occasionally drop out of the computations. However, common sense suggests that this increases the execution time. Although this is generally not true, for important class of iteration operators, results have been established in the same direction as this intuition, as we will discuss in Chapter 4.

2. Definition 2.1 by no means implies that x should be associated with a particular physical device, such as shared memory. Suppose that in the multiprocessor system shown in Figure 1.1, in addition to the shared memory, each processor has a local storage which can only be accessed by itself. The processors

may generally be allocated dynamically to components. Assume, with no loss of generality, that each local storage is large enough to hold the whole x and that the initial value of x is loaded into each local storage before the computation starts. The processors work (read and write) with their local copies of x, but once in a while, at some unspecified times, some of the local updates (not necessarily all) propagate to the shared memory, and also once in a while, the processors load the most recent value of the shared memory to the local storages. An immediate suggestion for $x(t)$ may be to interpret it as the value of the shared data (in the global memory) right at t. However, this conflicts with Definition 2.1. Since the processors are working on the local copies, the input vectors may never be seen in the global memory. Even if they do, data transfer to memory may take so long that the input vectors appear in the memory before they are used in the local updates which violates $\tau_j^i(t) < t$. Now suppose that $x_i(t)$ is interpreted such that it is the value of the most recent local update of x_i no later than t (in any local storage). Then, if x_i is updated at t, $\tau_j^i(t)$ can be interpreted as the index of the interval in which the j-th component of the input data for this update is generated. Notice that it may have been generated in the local storage, or it may have been generated in another processor and may have been propagated to the local storage through the global memory. The implications of total asynchronism for this system are as follows:

- Each component should once in a while be updated.
- Local updates should once in a while be propagated to the global memory in a finite period of time.
- Local copies should once in a while be replaced with recent component values of x in the global memory in a finite period of time.

3. We now consider a simpler architecture on which asynchronous iterations execute with the same efficiency. It can easily be noticed that, for our point of view, the sole function of the shared memory in the above system is to provide communication between the processors. Therefore, this part of the system can be replaced with an interconnection network which assumes the same responsibility. In other words, the last two conditions of total asynchronism for the above system can be combined so that we have the following conditions for total asynchronism:

   - Each component should be updated occasionally.
   - Newer updates of each component of x should occasionally be communicated to each processor.

   This leads to the architecture where processors maintain their own copies of the iterated data x in their local memories which interact through an interconnection network. Processors read their local copies. A write operation updates its local copy, and also notifies the interconnection network about this transaction. The interconnection network is responsible for broadcasting this update. The originating processor does not have to wait for the completion of this broadcast. As long as the updates are eventually propagated, the convergence is maintained. It should be noted that the ordering of the broadcasts does not affect the convergence. In other words, the interconnection network does not have to transmit writes in the same order as it receives them.

2.3.3 Main Convergence Result

We first give a different but equivalent formulation of Model 2.1 which is stated by the following lemma. This will be convenient in our later proofs.

Lemma 2.2 *A schedule $\mathcal{S} = (\{\alpha(t)\}, \{\tau(t)\})$ is totally asynchronous if and only if there exists a sequence of integers $\{\varphi(k)\}$ satisfying the following conditions.*

- $\varphi(0) = t_0$ *and* $\varphi(1) = 0$, *where* $t_0 < 0$ *is some arbitrary integer.*
- *Each component* $i = 1, 2, \ldots, n$ *is updated at some* t *such that*

$$\varphi(k) < t \leq \varphi(k+1), \qquad \forall k = 0, 1, 2, \ldots$$

- $t > \varphi(k) \Rightarrow \tau_j^i(t) \geq \varphi_j(k-1) > \varphi(k-1), \quad \forall k = 1, 2, \ldots, \ \forall i = 1, 2, \ldots, n$ *where* $\varphi_j(k-1)$ *is defined as the first instance later than* $\varphi(k-1)$ *at which the* j*-th component is updated.*

Proof. The proof is by mathematical induction. The proof of basis clause (the case for $k = 1$) is trivial. Suppose that there exists $\varphi(0), \varphi(1), \ldots, \varphi(l-1)$ satisfying the conditions. From non–starvation of updates, there exists a t' such that all the components are updated in the interval $(\varphi(l-1), t']$. On the other hand, from non–starvation of inputs, given $\varphi_s(l-1)$ there exists a t'_s such that $\tau_s^i(t) \geq \varphi_s(l-1)$, for all $i, j = 1, 2, \ldots, n$ and $t \geq t'_s$. If $\varphi(l)$ is chosen to be the maximum of all t' and t'_s's, it satisfies the above conditions.

It is also easy to see that the schedule is totally asynchronous, given the conditions of the lemma. □

Following the terminology in [40], we call the set $\Phi(k) = \{j | \varphi(k) < j \leq \varphi(k+1)\}$ the k-th *pseudo–cycle* of $\mathcal{S}$, and the sequence $\{\Phi(k)\}$ is said to be the *pseudo–cycle sequence* associated with $\mathcal{S}$. There are an infinite number of $\{\varphi(k)\}$'s and $\{\Phi(k)\}$'s satisfying the above conditions, given a schedule. In the rest of the thesis $\{\varphi(k)\}$ and $\{\Phi(k)\}$ (when there is no ambiguity) will represent an arbitrary one of these, although we will not explicitly mention it every time.

Because of the above lemma, a totally asynchronous schedule is also called a *pseudo–periodic schedule.* As we shall show in Theorem 2.1, at the end of a pseudo–cycle $\Phi(k)$ (from $\varphi(k)$ to $\varphi(k+1)$), it is guaranteed that there is an "improvement" made towards the solution. Therefore, associating $\Phi(k)$'s to actual periods of time suggests an obvious way of estimating the total computation time.

Now we define a class of operators that will later be shown to converge to the unique solution for all totally asynchronous schedules. In the next section, it is also proven that this is the largest such class for finite data domains.

Definition 2.2 An operator F is an *asynchronously contracting operator* (ACO) on a subset $X(0)$ of X iff there exists a sequence of sets $\{X(k)\}$ that satisfies the following conditions.

- (*Box Condition*) For all $k = 0, 1, 2, \ldots$, $X(k)$ is a Cartesian product of n sets; i.e.,

$$X(k) = X_1(k) \times X_2(k) \times \cdots \times X_n(k).$$

- For all $k = 0, 1, 2, \ldots$, $X(k+1) \subseteq X(k)$, and furthermore, $\{X(k)\}$ converges to a fixed point of F in $X(0)$;
- $x \in X(k) \Rightarrow F(x) \in X(k+1)$, for all $k = 0, 1, 2, \ldots$ □

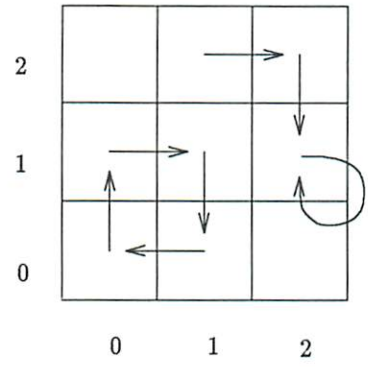


Figure 2.2: Asynchronous distortion

As will be shown in Theorem 2.1, $X(k)$ is the set in which the iterates stay after the k-th pseudo–cycle and therefore the above conditions constitute sufficient conditions for convergence of totally asynchronous iterations. The following lemma further states that the point of convergence, therefore the fixed point is unique.

Lemma 2.3 *F has a unique fixed point in a set $X(0)$ if it is asynchronously contracting on $X(0)$.*

Proof. Existence of a fixed point is guaranteed by definition of ACO (second condition). Uniqueness can be shown by contradiction. Suppose that there exists two distinct fixed points $x^*, y^* \in X(0)$. Then, from the third condition, $x^*, y^* \in X(k)$, for all k. This violates the second condition. □

The last two conditions in the above definition are intuitive. In fact, any definition of contraction imposes conditions of a similar nature. On the contrary, the box condition might seem counter–intuitive; yet, it is the requirement of an ACO that is due to the asynchronous nature of the computations which shows as *asynchronous distortions* in asynchronous algorithms. When asynchronous distortion occurs, an iterate value in $X(k)$ is generated, which is not possible to reach by successive applications of the iteration operator starting from the initial data. It will be shown later that there exists a schedule such that each point of $X(k)$ is visited in the k-th pseudo-cycle, when $X(0)$ contains a finite number of elements. In other words, all points in $X(k)$ are reachable by asynchronous distortion. Since we want convergence for *all* schedules, we need to guarantee the invariance of $X(k)$ under asynchronous distortion. Therefore, the box condition is *necessary* for convergence for finite $X(0)$ when no restriction is enforced on the schedule. Whether it is necessary for domains that contain an infinite number of elements, however, is still an open question. The reason for the uncertainty is that the proof of the necessary condition for the finite case is based on the construction of an asynchronous iteration such that all of the points in $X(k)$ are visited in the k-th pseudo-cycle of this asynchronous iteration. If $X(k)$ contained an infinite number of elements, then the constructed schedule would not be pseudo–periodic.

Example 2.2 Figure 2.2 displays an iteration operator in which the arc from each element in the domain points to the value we obtain when we apply the operator F to that element. Consider the iteration that starts with $x(0) = (1, 2)$ and that converges synchronously as well as for many asynchronous schedules to $y = (2, 1)$, as long as the iterates do not leave the path from $x(0)$ to y. However, some schedules may cause distortion; i.e., the iterates may be "deflected" to the cycle in the lower left part of the domain. For instance, let there be two processors: P1 and P2. P1 first updates x_1 and then x_2. Right after this, P2 updates x_2,

using the initial value of x_1 and the most recent value of x_2. At this point, the iterate value is $z = (1, 0)$, and starting with z, the algorithm cycles even if the remaining part of the schedule is synchronous. □

In the example above, we observe that the cause of asynchronous distortion is the redundant computations of x_2. If we disallow this so that each update of a component uses the most recent value of that component, then the iterates always converge to y. Therefore, the box condition is a necessary condition only when the schedule is not restricted.

We now formally state and prove that if F is asynchronously contracting, then the corresponding asynchronous iteration converges to the fixed point of F, for all totally asynchronous schedules.

Theorem 2.1 (Sufficient Condition) *If F is an ACO on a set $X(0)$, then any asynchronous iteration $\{x(t)\} = (F, x(0), \mathcal{S})$ with $x(0) \in X(0)$ converges to the unique fixed point x^* of F in $X(0)$, for all totally asynchronous schedules $\mathcal{S}$.*

Proof. We will simply show by induction that starting from $\varphi_i(k)$, the i-th component of the iterates indefinitely enters $X_i(k)$ for all $i = 1, 2, \ldots, n$; i.e.,

$$t \geq \varphi_i(k) \;\Rightarrow\; x_i(t) \in X_i(k), \qquad \forall i = 1, 2, \ldots, n,\ k = 0, 1, 2, \ldots \tag{2.1}$$

Suppose this is true for $k = l$ (induction hypothesis) and consider the update at $\varphi_{i'}(l+1)$. Since the i-th component of the input vector for this update is generated no earlier than $\varphi_i(l)$, the input vector is in $X(l)$, by the hypothesis. In other words, for $t = \varphi_{i'}(l+1)$

$$u^{i'}(t) \;=\; (x_1(\tau_1^{i'}(t),\ x_2(\tau_2^{i'}(t)), \ldots) \;\in\; X(l).$$

Note that this would not always be true if $X(l)$ were not a Cartesian product. As a consequence, we have

$$x_{i'}(j) \;=\; F_{i'}(u^{i'}(t)) \;\in\; X_{i'}(l+1)$$

for $t = \varphi_{i'}(l+1)$, and by induction it can be generalized for $t \geq \varphi_{i'}(l+1)$. Since this is valid for all $i' = 1, 2, \ldots, n$, (2.1) holds for $k = l+1$, and by induction for all $k = 0, 1, 2, \ldots$ Here, we are omitting the proof of the basis (that all the iterates are in $X(0)$), but the argument is the same.

(2.1) implies that after all the components are updated in the k-th pseudo-cycle the iterates indefinitely fall into $X(k)$, and therefore the iteration converges to the same point as the sequence $\{X(k)\}$. □

Asynchronous contraction is an extremely simple concept which generalizes and simplifies the convergence conditions given by other researchers. In the first paper on asynchronous algorithms [12], linear iterations of the form $F(x) = A\,x + b$, where A is an $n \times n$ matrix were considered. It was proven that for convergence of asynchronous iterations, it is necessary and sufficient that the spectral radius of $|A|$, $(\rho(|A|))$ is strictly less than 1. In Baudet, this result was generalized to cover non-linear iteration operators.

The convergence condition which is closest to asynchronous contraction was given by Bertsekas [4]. In fact, it can be shown to be equivalent to asynchronous contraction. However, unlike asynchronous contraction, it was expressed in a complicated and not so natural way. Interpretation of this result requires some manipulation of F and x which is not straightforward. As a result, in the formulation of [4], the box condition and the inclusion of $X(k+1)$ in $X(k)$ was not obvious. These were made clear later in [53], [52] and [7].

2.3.4 Necessary Condition

The following definitions and the subsequent lemmas establish the background required for the proof of Theorem 2.2, which states that asynchronous contraction is not only sufficient but also necessary for convergence under total asynchronism, for finite data domains.

Definition 2.3 Let $\{x(t)\} = (F, x(0), \mathcal{S})$ be an asynchronous iteration and $\{\Phi(k)\}$ be a pseudo–cycle sequence associated with it. The *outcome* of $\{x(t)\}$ in the k-th pseudo–cycle, $P(k) = P_1(k) \times P_2(k) \times \cdots \times P_n(k)$, is the set such that

$$P_i(k) = \{x_i(t) | i \in \alpha(t) \text{ and } t \in \Phi(k)\}, \qquad \forall i = 1, 2, \ldots, n, \quad k = 0, 1, 2, \ldots \qquad \square$$

In the following discussions, without explicitly mentioning it we assume that $P(k)$, $P'(k)$, $P''(k)$, ... correspond to $\{x(t)\}$, $\{x'(t)\}$, $\{x''(t)\}$, ... respectively.

Lemma 2.4 is based on the concept of *merging* asynchronous iterations. Merging $\{x'(t)\}$ and $\{x''(t)\}$, each starting with the same initial vector $x(0)$, means the construction of another asynchronous iteration $\{x(t)\}$ in the following way. In the k-th pseudo–cycle of $\{x(t)\}$, first the outputs of the k-th pseudo–cycle of $\{x'(t)\}$ are obtained in the same order, then this procedure is repeated for the k-th pseudo–cycle of $\{x''(t)\}$, as shown in Example 2.3.

Example 2.3

$$\begin{aligned}
\{x'(t)\} &= \underbrace{(1,1)}_{\Phi'(0)} \underbrace{(2,1)\,(2,2)}_{\Phi'(1)} \underbrace{(3,2)\,(3,3)}_{\Phi'(2)} \cdots \\
\{x''(t)\} &= \underbrace{(1,1)}_{\Phi''(0)} \underbrace{(1,5)\,(5,5)}_{\Phi''(1)} \underbrace{(6,5)\,(6,6)}_{\Phi''(2)} \cdots \\
\{x(t)\} &= \underbrace{(1,1)}_{\Phi(0)} \underbrace{(2,1)\,(2,2)\,(2,5)\,(5,5)}_{\Phi(1)} \underbrace{(3,5)\,(3,3)\,(6,3)\,(6,6)}_{\Phi(2)} \cdots
\end{aligned}$$

Each of these sequences starts with the same initial vector $x(0) = (1, 1)$, and for $k > 0$, $\Phi(k)$ has two phases. For example, in $\Phi(1)$ first x_1 then x_2 are updated, each time yielding the value 2, because this phase corresponds to $\Phi'(1)$. In the second phase of $\{\Phi(1)\}$, x_2 and x_1 are updated respectively, each time yielding 5, as in $\Phi''(1)$. $\square$

Lemma 2.4 *Let $\{x'(t)\} = (F, x(0), \mathcal{S}')$ and $\{x''(t)\} = (F, x(0), \mathcal{S}'')$ be totally asynchronous iterations w.r.t. $\mathcal{S}'$ and $\mathcal{S}''$, respectively. There exists an asynchronous iteration $\{x(t)\} = (F, x(0), \mathcal{S})$ such that $P(k) = P'(k) \cup P''(k)$ for all $k = 0, 1, 2, \ldots$*

Proof. The desired $\{x(t)\}$ is constructed by merging $\{x'(t)\}$ and $\{x''(t)\}$. The formal definition of the schedule for $\{x(t)\}$ is given by the series of formulas below.

$$\varphi(k) = \begin{cases} -1 & \text{if } k = 0 \\ 0 & \text{if } k = 1 \\ \varphi'(k) + \varphi''(k) & \text{otherwise} \end{cases}$$

Literally, the length of the k-th pseudo–cycle of $\{x(t)\}$ is the sum of the lengths of the k-th pseudo–cycles of $\{x'(t)\}$ and $\{x''(t)\}$, for $k > 0$.

$$\delta'(t) = \begin{cases} 0 & \text{if } t = 0 \\ t + \varphi''(k) & \text{if } t > 0 \text{ and } t \in \Phi'(k) \end{cases}$$

$$\delta''(t) = \begin{cases} 0 & \text{if } t = 0 \\ t + \varphi'(k+1) & \text{if } t > 0 \text{ and } t \in \Phi''(k) \end{cases}$$

δ' (δ'') simply maps the indices of the sequence $\{x'(t)\}$ ($\{x''(t)\}$) to the corresponding indices of $\{x(t)\}$. The inverse mapping is defined by

$$\delta^{-1}(t) = \begin{cases} 0 & \text{if } t = 0 \\ t - \varphi''(k) & \text{if } t > 0 \text{ and } 0 < t - \varphi(k) \leq |\Phi'(k)| \\ t - \varphi'(k+1) & \text{if } |\Phi'(k)| < t - \varphi(k) \leq |\Phi'(k)| + |\Phi''(k)| \end{cases}$$

δ^{-1} maps the indices of $\{x(t)\}$ to the corresponding indices of $\{x'(t)\}$ or $\{x''(t)\}$. In this definition, the second (third) line corresponds to the elements of the first (second) phase of $\Phi(k)$. If t is in the first (second) phase of a pseudo–cycle, the value of $\alpha(t)$ is the same as the value of α' (α'') for the corresponding element of $\{x'(t)\}$ ($\{x''(t)\}$); i.e.

$$\alpha(t) = \begin{cases} \alpha'(\delta^{-1}(t))) & 0 < t - \varphi(k) \leq |\Phi'(k)| \\ \alpha''(\delta^{-1}(t))) & \text{if } |\Phi'(k)| < t - \varphi(k) \leq |\Phi'(k)| + |\Phi''(k)| \end{cases}$$

τ can be defined in a similar manner, as follows.

$$\tau_j^i(t) = \begin{cases} \delta'(\tau'^i_j(\delta^{-1}(t))) & 0 < t - \varphi(k) \leq |\Phi'(k)| \\ \delta''(\tau''^i_j(\delta^{-1}(t))) & \text{if } |\Phi'(k)| < t - \varphi(k) \leq |\Phi'(k)| + |\Phi''(k)| \end{cases}$$

It is obvious that the resulting sequence indeed satisfies the assumptions of asynchronous iteration and the requirement of the lemma. □

Lemma 2.5 *Let* $\{x^{(1)}(t)\} = (F, x(0), \mathcal{S}^{(1)})$, $\{x^{(2)}(t)\} = (F, x(0), \mathcal{S}^{(2)}), \cdots$ and $\{x^{(m)}(t)\} = (F, x(0), \mathcal{S}^{(m)})$ *be asynchronous iterations corresponding to* F *and starting with* $x(0)$. *Then, there exists* $\{x(t)\} = (F, x(0), \mathcal{S})$, *such that*

$$P(k) = P^{(1)}(k) \cup P^{(2)}(k) \cup \cdots \cup P^{(m)}(k), \qquad \forall k = 0, 1, 2, \ldots$$

Proof. A straightforward generalization of the previous lemma. □

Lemma 2.5 states that the pseudo–cycles of different asynchronous iterations on the same F and $x(0)$ can be merged to form a new asynchronous iteration.

Lemma 2.6 *Let* $\{x'(t)\} = (F, x(0), \mathcal{S}')$ *be an asynchronous iteration. If* $a_{i_1} \in P'_{i_1}(k)$ *and* $a_{i_2} \in P'_{i_2}(k)$, *then there exists an asynchronous iteration* $\{x(t)\} = (F, x(0), \mathcal{S})$ *such that for some* $p \in \Phi(k)$, $x_{i_1}(p) = a_{i_1}$, and $x_{i_2}(p) = a_{i_2}$ *for all* $k > 0$, *and* $i_1 \neq i_2$.

Proof. Let a_{i_1} and a_{i_2} be the results of the updates at t_1 and $t_2 \in \Phi'(k)$, respectively; i.e.,

$$i_1 \in \alpha'(t_1) \ , \ x'_{i_1}(t_1) = a_{i_1},$$

and

$$i_2 \in \alpha'(t_2) \ , \ x'_{i_2}(t_2) = a_{i_2} \ .$$

Now, construct the first k pseudo–cycles of $\{x(t)\}$ by inserting an iterate at the end of the k-th pseudo–cycle of $\{x'(t)\}$ such that a_{i_1} and a_{i_2} are the outputs of this update. More formally, define

- $\alpha(t) = \alpha'(t)$, $\tau(k) = \tau'(k)$; therefore, $x(t) = x'(t)$, for $t \leq \varphi'(k+1)$;
- $\alpha(p) = \{i_1, i_2\}$, $\tau^{i_1}(p) = \tau'^{i_1}(j_1)$, $\tau^{i_2}(p) = \tau'^{i_2}(j_2)$;
- $\varphi(l) = \varphi'(l)$ for $l \leq k$;
- $\varphi(k+1) = \varphi'(k+1) + 1$,

where $p = \varphi'(k+1) + 1$. Also, for $t > p+1$, $\alpha(t)$ and $\tau(t)$ are chosen freely to satisfy the conditions of total asynchronism. Furthermore, it can easily be seen that the first k pseudo–cycles of $\{x(t)\}$ satisfy these conditions. □

The generalization of the above proof gives the following lemma.

Lemma 2.7 *Let $\{x'(t)\} = (F, x(0), S')$ be a totally asynchronous iteration. There exists $\{x(t)\} = (F, x(0), S)$ such that for any n-dimensional vector $a \in P'(k)$, there exists a t in the k-th cycle of $\{x(t)\}$ such that $x(t) = a$.*
□

This lemma simply states that given a vector a, each component of which is the output of an update in the k-th pseudo–cycle of a particular asynchronous iteration, we can construct another asynchronous iteration such that the value of the global data is a, at some time instance in the k-th pseudo–cycle.

Now, we can prove Theorem 2.2, which says that the sufficient condition of convergence given in Theorem 2.1 (i.e., F being asynchronously contracting) is also necessary for convergence for finite data domains.

Theorem 2.2 (Necessary Condition) *Let F be an operator defined in a domain X that contains a finite number of elements. If all totally asynchronous iterations $\{x(t)\} = (F, x(0), S)$, starting with $x(0) \in X$ and corresponding to F, converge to the same fixed point x^*, then F is asynchronously contracting on a set $X(0) \subseteq X$, where $x(0) \in X(0)$.*

Proof. Define $X(k) = X_1(k) \times X_2(k) \times \cdots \times X_n(k)$ such that

$$X_i(k) = \{a_i | a_i \in P_i(k) \text{ for some } S\}, \qquad \forall i = 1, 2, \ldots, n, \ \forall k = 0, 1, 2, \ldots$$

We shall show by mathematical induction that there exists an asynchronous iteration $\{x'(t)\} = (F, x(0), S')$ such that

$$P'(k) = X(k), \qquad \forall k = 0, 1, 2, \ldots$$

We will show only the induction part, since the argument for $k = 0$ is very similar. Assume that the above statement is true for $k < l$. Since the domain of F is finite, the iterates of all asynchronous iterations can take a finite number of values; therefore, $X(l)$ contains a finite number of elements. Consequently, there are $m - 1$ schedules that satisfy

$$X(l) = P^{(1)}(L) \cup P^{(2)}(l) \cup \cdots \cup P^{(m-1)}(l),$$

and from the hypothesis, there exists $S^{(m)}$, such that,

$$P^{(m)}(k) = X(k), \qquad k < l.$$

For all $j = 1, 2, \ldots, m$ and for all $k = 0, 1, 2, \ldots$, $P^{(j)}(k) \subseteq X(k)$. From Lemma 2.5 there exists an asynchronous iteration, $\{x'(t)\} = (F, x(0), S')$ such that

$$P'(k) = X(k), \qquad \forall k \leq l,$$

and by induction, $P'(k) = X(k)$ for all $k = 0, 1, 2, \ldots$ Then, from Lemma 2.7 there exists an asynchronous iteration $\{x(t)\} = (F, x(0), \mathcal{S})$ such that

$$X(k) \subseteq \{x(j) | j \in \Phi(k)\}, \quad \forall k = 0, 1, 2, \ldots \tag{2.2}$$

We can now show that the sequence $\{X(k)\}$ satisfies the conditions of Definition 2.2 :

- Box condition is satisfied by definition.
- The definition of pseudo–cycle permits us to coalesce two consecutive pseudo–cycles into one. In other words, given a pseudo–cycle sequence, $\{\Phi(k)\}$ corresponding to $\{x(t)\}$, there exists $\{\overline{\Phi}(k)\}$ such that $\overline{\Phi}(k) = \Phi(k) \cup \Phi(k+1)$, which is also a pseudo–cycle sequence. Therefore,

  $$X(k+1) \subseteq X(k) \cup X(k+1) \subseteq \overline{P}(k) \subseteq X(k).$$

  From (2.2) $\{X(k)\}$ converges to x^* and the second condition of Definition 2.2 is satisfied.
- There exists $\{x'(t)\} = (F, x(0), \mathcal{S}')$ such that $P'(k) = X(k)$ for $k = 0, 1, 2, \ldots$ Consequently, for all $x \in P'(k)$ and $i = 1, 2, \ldots, n$, there exists $\{x''(t)\} = (F, x(0), \mathcal{S}'')$ such that
  - $P''(k) = P'(k)$, and
  - $x_i(t_i) = F_i(x)$ for some $t_i \in \Phi''(k+1)$.

Therefore, $F_i(x) \in P_i''(k+1)$ for all $i = 1, 2, \ldots, n$, and hence $F(x) \in X(k+1)$, which completes the proof. □

Finally, for the sake of completeness, we give the following theorem, which is the restatement of Theorem 2.1 and Theorem 2.2 combined.

Theorem 2.3 *Let F be an operator defined in a domain X that contains a finite number of elements. For all totally asynchronous schedules, any asynchronous iteration $(F, x(0), \mathcal{S})$ corresponding to F and starting with any initial guess $x(0)$ converges to a fixed point of F if and only if F is asynchronously contracting in a set $X(0)$ that contains $x(0)$.* □

2.3.5 Other Sufficient Conditions

Although conceptually very simple, asynchronous contraction may be difficult to verify in practical problems. In this section, we will give simpler conditions on the iteration operator F each of which will be shown to be a special case of asynchronous contraction. This means that in each case, every totally asynchronous iteration corresponding to F and starting with the initial vector $x(0) \in X$ is guaranteed to converge to the unique fixed point x^*.

2.3.5.1 Monotonic Operators

Monotonic operators were first shown to converge under asynchronism, by Bertsekas [4] , [5]. In the following proposition, $\{y(k)\}$ refers to the sequence such that

- $y(0) = x(0)$,

- $y(k) = F(y(k-1)), \quad \forall k \in \mathcal{N}^+$.

In other words, $\{y(k)\}$ is the synchronous iteration corresponding to F and starting with $x(0)$. Furthermore, $\preceq_i$ denotes a partial order on the elements of X_i, for all $i = 1, 2, \ldots, n$. We write $a \preceq b$ if $a_i \preceq_i b_i$, for all $a, b \in X$ and $i = 1, 2, \ldots, n$.

Condition 2.1

- *F is continuous in X.*
- $F(x(0)) \preceq x(0)$.
- *$\{y(k)\}$ converges to an element of X.*
- *F is monotone in X; i.e., for all $a, b \in X$,* $a \preceq b \Rightarrow F(a) \preceq F(b)$.

Proof. From the first condition, X is closed under F, and therefore the elements of $\{y(k)\}$ take values from the set X. The continuity also implies that $\{y(k)\}$ converges to a fixed point x^* of F. From the second condition and from the monotonicity

$$y(0) \succeq y(1) \succeq \ldots y(k) \succeq y(k+1) \succeq \ldots \succeq x^*.$$

Define $\{X(k)\}$ such that

$$X(k) = \{a | x^* \preceq a \preceq y(k) \text{ and } a \in X\}.$$

Obviously, $X(k)$ satisfies the first two conditions of Definition 2.2 (asynchronous contraction). On the other hand, from monotonicity (last condition), for all $a \in X(k)$, $x^* \preceq F(a)$, and

$$F(a) \preceq F(y(k)) = y(k+1).$$

Also, since $F(a) \in X$ (from closure property), $F(a) \in X(k+1)$. Hence, all the conditions of Definition 2.2 are satisfied. □

When X is a finite set, Condition 2.1 can be simplified as follows.

Condition 2.2

- *X is a finite set.*
- *X is closed under F; i.e., for all $a \in X$,* $F(a) \in X$.
- *F is non-expansive in X; i.e., for all $a \in X$,* $F(a) \preceq a$.
- *F is monotone in X; i.e., for all $a, b \in X$,* $a \preceq b \Rightarrow F(a) \preceq F(b)$.

Proof. For finite domains the closure property implies continuity. Therefore, we only need to show the third condition of Condition 2.1. Since F is non-expansive and monotone,

$$y(0) \succeq y(1) \succeq \ldots y(k) \succeq y(k+1) \succeq \ldots$$

From the antisymmetry property of $\preceq$, no two elements of the sequence $\{y(k)\}$ are the same, except when they are equal to the fixed point. Consequently, $\{y(k)\}$ converges and Condition 2.1 applies. □

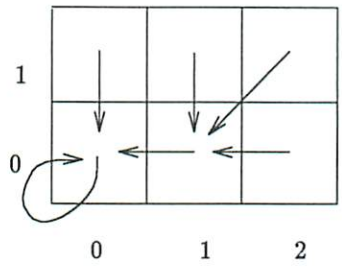


Figure 2.3: A monotone operator

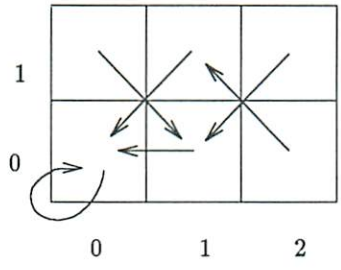


Figure 2.4: An asynchronously contracting operator which is not monotone

Example 2.4 The operator given by Figure 2.3 satisfies the conditions of the above propositions when $\preceq$ is selected as the regular ordering relation defined on numbers. The notation of the figure is the same as the one given in Example 2.2. □

The question may arise of whether the conditions of Proposition 2.1 and 2.2 are necessary for convergence. The following counter–example shows that they are not.

Example 2.5 F is an operator defined on the elements of $X = \{0,1,2\} \times \{0,1\}$, as shown in Figure 2.4. F is asynchronously contracting on X, because if we take

$$X(1) = \{0,1\} \times \{0,1\}, \qquad X(2) = \{0,1\} \times \{0\}, \qquad X(3) = X(4) = \cdots = \{(0,0)\},$$

then for all k,

$$x \in X(k) \Rightarrow F(x) \in X(k+1).$$

Suppose that F also satisfies the conditions of the above propositions. Since F is non-expansive,

$$F(0,1) \preceq (0,1) \Rightarrow 1 \preceq_1 0,$$

and

$$F(1,0) \preceq (1,0) \Rightarrow 0 \preceq_1 1,$$

which is a contradiction, and therefore, F is *not* non-expansive but still asynchronously contracting. □

We can redefine $\preceq$ such that Example 2.5 shows a monotone operator with respect to the new $\preceq$. We can state this in the following way.

Condition 2.3 *F is a contracting operator and $\{X(k)\}$ is the corresponding nested sequence of domains as defined in Definition 2.2. Furthermore, the following restriction is imposed on F.*

$$x \in R(k) \;\Rightarrow\; F(x) \in R(k+1), \quad \forall k = 0,1,2,\ldots,$$

where $R(k)$ is defined as

$$R(k) = X(k) - X(k+1), \qquad \forall\, k = 0, 1, 2, \ldots$$

□

The above restriction simply states that the application of the operator to some data in $R(k)$ moves the data to $X(k+1)$, but not any further, i.e., to $X(k+2)$. It is obvious that the restriction does not contradict with the definition of asynchronous contraction. Therefore, it constitutes a sufficient condition for convergence. Furthermore, it is implied by Condition 2.1. If we define $\preceq$ such that we say $x \preceq y$ if $k_x \leq k_y$, where k_x and k_y are the integers satisfying $x \in R(k_x)$ and $y \in R(k_y)$, for all $x, y \in X$. Then, Condition 2.1 holds, but $\preceq$ is not a partial order any longer since it violates the second condition (antisymmetry) of the definition of partial order. In Chapter 4, Condition 2.3 will be assumed for the purpose of comparing the speed of an asynchronous iteration to the speed of it synchronous version.

2.3.5.2 Componentwise Contraction

In the following case, each application of the iteration operator makes each component smaller with respect to a partial order.

Condition 2.4

- *X is a finite set.*
- *X is closed under F; i.e., for all $a \in X$, $F(a) \in X$.*
- *There exists a fixed point $x^* \in X$ such that for all $a \in X$ and $i = 1, 2, \ldots, n$, $F_i(a) \prec_i a_i$ if $a_i \neq x_i^*$, and $F_i(a) = x_i^*$ otherwise.*

Proof. x^* is the unique fixed point in X. This can easily be seen by contradiction; i.e., by assuming the existence of another fixed point $y^* \neq x^*$. Then, the last condition is not satisfied for $a = y^*$. Now, define $\{X(k)\}$ such that

$$X_i(k) = X_i(k-1) - R_i(k-1), \qquad \forall k > 0,\ \forall\, i,$$

where $R_i(k)$ is the set of maximal elements of $X_i(k)$, except x_i^*; i.e.,

$$R_i(k) = \{a_i | (a_i \in X_i(k)) \text{ and } (\forall b_i \in X_i(k),\ \ a_i \preceq_i b_i \Rightarrow a_i = b_i)\} - \{x_i^*\}.$$

It is obvious that every $X_i(k)$ has a maximal element, because of the fact that $X_i(k)$ is finite. Therefore, $R_i(k)$'s are non-empty, except when $X_i(k) = \{x_i^*\}$. Thus, $X_i(k+1) \subset X_i(k)$ except when $X_i(k) = \{x_i^*\}$, from which the first two conditions of Definition 2.2 are satisfied.

To prove the last part, we first note that $x^* \in X$, and since for all $i = 1, 2, \ldots, n$, $x_i^* \notin R_i(k)$, then $x^* \in X(k)$, for all $k = 0, 1, 2, \ldots$ Now, for a given $a \in X(k)$, there are two possibilities, for all k.

- If $a = x^*$, then $F(a) = x^* \in X(k+1)$;
- if $a \neq x^*$, then $F_i(a) \prec_i a_i$ or $a_i = x_i^*$, and therefore, $F_i(a) \notin R_i(k)$, which implies that $F_i(a) \in X_i(k+1)$, for all $i = 1, 2, \ldots, n$.

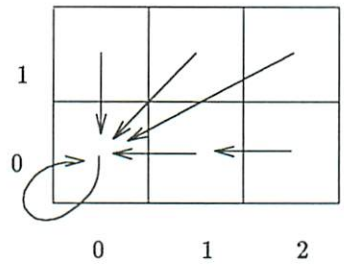


Figure 2.5: A componentwise contracting operator which is not monotone

In both cases, the conditions of asynchronous contraction are satisfied. □

Like Condition 2.1, Condition 2.4 is not necessary for asynchronous contraction. Example 2.5 can be used to support this. The major difference between Condition 2.2 and Condition 2.4 is that the latter one does not impose monotonicity.

Example 2.6 Figure 2.5 describes an operator satisfying Condition 2.4 but not Condition 2.2. In this example, the only ordering relation that would satisfy the second part of Condition 2.2 is the one defined for numbers in the usual sense, but for this ordering relation, monotonicity does not hold: although $(2,0) \preceq (2,1)$, $F(2,0) \succ F(2,1)$. □

2.3.5.3 A Condition Based on Dependencies

The following condition is expressed in terms of the dependency matrix $D(F)$ of the iteration operator, where D_{ij} is 0 if F_i does not depend on x_j and it is 1 otherwise. It is a generalization of the result given in [39], [40] and proven for a special class of asynchronous iterations in which each update of the shared data always uses the most recent value of the data as input.

Condition 2.5 *The transitive closure of the dependency matrix $D(F)$ is null.*

Sketch of Proof. Let,

$$D(F) = \begin{bmatrix} 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 & * \\ 0 & 0 & 0 & \cdots & 0 & * & * \\ & & & \cdots & & & \\ 0 & * & * & \cdots & * & * & * \end{bmatrix},$$

without loss of generality. Otherwise, we can obtain the above matrix by permuting on the rows and the columns [39]. For any x, $F_1(x)$ is constant, say x_1^* (from the first row). Given, $x_1 = x_1^*$, $F_2(x)$ is constant, say x_2^* (from the second row). Similarly,

$$x_1 = x_1^*, \ldots, x_i = x_i^* \;\Rightarrow\; F_{i+1}(x) = x_{i+1}^*.$$

Define,

$$\begin{aligned} X(0) &= X_1(0) \times X_2(0) \times \cdots X_n(0) \\ X(1) &= \{x_1^*\} \times X_2(0) \times \cdots \times X_n(0) \\ X(2) &= \{x_1^*\} \times \{x_2^*\} \times X_3(0) \times \cdots \times X_n(0) \\ &\vdots \end{aligned}$$

Then, F is asynchronously contracting due to Definition 2.2. □

This condition is also not sufficient for asynchronous contraction, e.g., it is not satisfied for the operator given in Example 2.5, because each component of F depend on each component of x.

2.3.5.4 Numerical Problems

The result of this section is due to Baudet [3]. The following notations are adopted. If z is a vector, $|z|$ denotes the vector with components $|z_i|$. Also, for any $x, y \in \Re^n$, $x \leq y$ implies $x_i \leq y_i$, for all i. An operator $F : \Re^n \mapsto \Re^n$ is said to be a *P-contracting operator* on a subset $X = X_1 \times X_2 \times \cdots \times X_n$ of $\Re^n$ if there exists a nonnegative $n \times n$ matrix A such that $\rho(A) < 1$ (the spectral radius of A) and,

$$|F(x) - F(y)| \leq A\,|x - y|, \quad \forall\, x, y \in X$$

[36].

Condition 2.6 *$F : \Re^n \mapsto \Re^n$ is P-contracting on a subset $X = X_1 \times X_2 \times \cdots \times X_n$ of $\Re^n$, where X is closed under F.*

Sketch of Proof. A complete proof can be found in [3]. Here, our goal is to relate it to the concept of asynchronous contraction. The starting point of the proof in [3] is a lemma that can be stated as follows. Let A be a nonnegative square matrix. Then $\rho(A) < 1$ if and only if there exists a positive scalar ω and a positive vector v such that

$$A\,v \leq \omega\,v \quad \text{and} \quad \omega < 1.$$

Without loss of generality, let us assume that the solution vector x^* is null. Otherwise, the origin of the coordinate system can be translated to the solution vector. Then, there exists a nonnegative $n \times n$ matrix A, a positive scalar $\omega < 1$ and a positive vector v such that

- $|F(x)| \leq B\,|x|$, for all $x \in X$, and
- $B\,v \leq \omega\,v$,

We can choose a scalar c such that $c\,v$ is larger than the absolute value of $x(0)$, and $X(k)$'s defined as follows satisfy the definition of asynchronous contraction (Definition 2.2).

$$X(k) = \{z \mid |z| \leq c\,\omega^k\,v\}.$$

□

Obviously, the result of the original paper on asynchronous iterations is a straightforward consequence of the above condition [12]. It states that for a linear operator given by $F(x) = A\,x + b$, $\rho(|A|) < 1$ is a sufficient condition for convergence. In [12], it was further proven that this condition is also necessary.

2.4 Asynchronous Iterations With Bounded Delay

2.4.1 Definition

For many computational systems, the conditions of total asynchronism (non-starvation of updates and inputs) are too weak. In such systems, there is an upper bound on the computational and communication delays. In

fact, in the original definition of asynchronous iterations (then called chaotic iterations) [12], this assumption was made. However, the result derived did not utilize the full power of boundedness. A computational model called *partial asynchronism* in [7], [46], [51], [50] did utilize boundedness, but also assumed non-redundancy. In Section 2.6.2, we will summarize the results of these publications. In this section, we first formally define the computational model which assume boundedness, but not non-redundancy. We call this *asynchronism with bounded delay* and derive a general convergence condition which is weaker than asynchronous contraction.

Model 2.2 (Asynchronism with Bounded Delay) *There exists a positive integer B such that*

- *(Bounded Non-starvation of Updates) For every $i = 1, 2, \ldots, n$ and for every $t \geq 0$ at least one element t' of the set $\{t, t+1, \ldots, t+B-1\}$ satisfies $i \in \alpha(t')$, i.e., in a time window of size B, all components are updated at least once.*
- *(Bounded Non-starvation of Inputs) $t - B \leq \tau_j^i(t) < t$, for all i, j and t such that $i \in \alpha(t)$, i.e., an update uses values of input components generated at most B time units ago.* □

In the proof of our main result, the starting point will be a property implied by Model 2.2. For convenience, we associate this property with a separate model (Model 2.3). In addition, when B is not a priori predictable in a realistic computational environment, it may be difficult to enforce Model 2.2. As will be discussed in Section 2.4.5, Model 2.3 is easier to deal with, in this respect. Let us first define the sequence $\{\varphi(k)\}$ of update instances as

$$\varphi(k) = \begin{cases} -1 & k = 0 \\ (k-1)\,C & k = 1, 2, \ldots \end{cases}$$

where C is a positive integer. Also let

$$\Phi(k) = \{t \mid \varphi(k) < t \leq \varphi(k+1)\}.$$

This is nothing else but a special pseudo–cycle sequence defined in the previous section, where $\Phi(k)$'s have constant size equal to C, for $k > 0$.

Model 2.3

- *For every i and k there exists a $t \in \Phi(k)$ such that $i \in \alpha(t)$, i.e., in a pseudo-cycle, all components are updated at least once.*
- *For every i, j, t' and $k > 0$, if $t' \in \Phi(k)$ and $i \in \alpha(t')$, there exists a $t'' > \varphi(k-1)$, that satisfies $u_j^i(t') = x_j(t'')$ and $j \in \alpha(t'')$, i.e., all updates use values that were generated no earlier than the previous pseudo-cycle.* □

The following lemma justifies the introduction of Model 2.3.

Lemma 2.8 *Model 2.2 implies Model 2.3 for $C = 2\,B - 1$.*

Proof. The first part is obvious from the first condition of the asynchronous model with bounded delay. Here we prove the second part. From the second condition of Model 2.2, for all $t' \in \Phi(k)$,

$$\tau_j^i(t') > \varphi(k) - B = \varphi(k-1) + B - 1$$

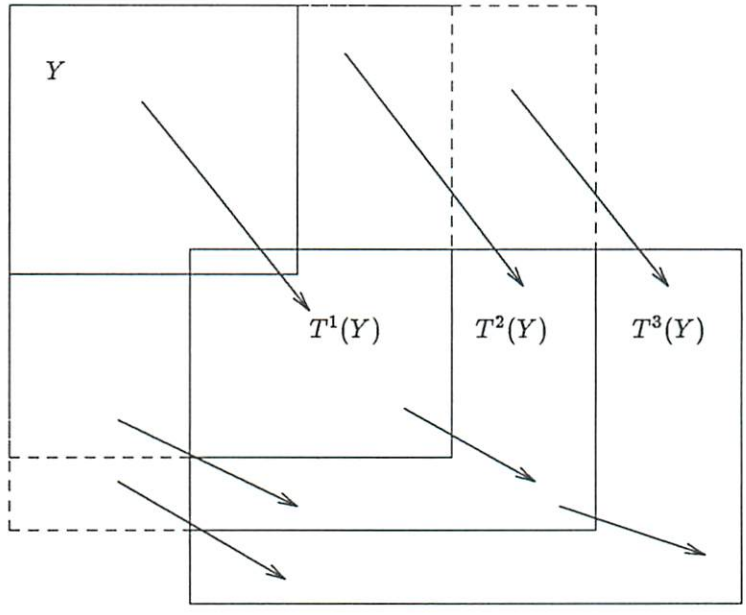


Figure 2.6: Demonstration of Definition 2.4

Let

$$u_j^i(t') = x_j(t''),$$

where t'' is the last update instance of the j-th component no later than $\tau_j^i(t')$ (i.e., $j \in \alpha(t'')$). From the first condition of Model 2.2, $t'' \geq \tau_j^i(t') - B + 1 > \varphi(k-1)$, and therefore $t'' > \varphi(k-1)$, for $C = 2B - 1$, which completes the proof. □

2.4.2 Convergence

Our general convergence result is based on the following definition of *d-closure* ($T^d(Y)$) of a set $Y = Y_1 \times Y_2 \times \cdots \times Y_n$. It simply defines the largest possible set of values generated in a time span of d units, given the input values Y.

Definition 2.4 (d-closure) *d-closure* $T^d(Y)$ of a set $Y = Y_1 \times Y_2 \times \cdots \times Y_n$ is recursively defined as:

$$T^d(Y) = \begin{cases} \prod_{i=1}^n \{F_i(x) \mid x \in Y\} & d = 1 \\ T^1(\prod_{i=1}^n \{T_i^{d-1}(Y) \cup Y_i\}) & \text{otherwise} \end{cases}$$

where $Y_i \subseteq X_i$, for all i. □

Figure 2.6 displays an example for a 2-dimensional real space. In the figure, each arrow originates from an element x and points to the value $F(x)$.

Although $T^d(Y)$ is conceptually not complicated, it may sometimes be cumbersome to estimate it directly from Definition 2.4. For this reason, the following lemma provides us with a simple set inclusion property regarding $T^d(Y)$, for an important class of functions, namely, monotonic real functions.

Lemma 2.9 *Let $F : \Re^n \mapsto \Re^n$ be a real function. Assume that it is monotonic in the box $U \subseteq \Re^n$ of the form*

$$U = \{x \mid \underline{x} \leq x \leq \overline{x}\},$$

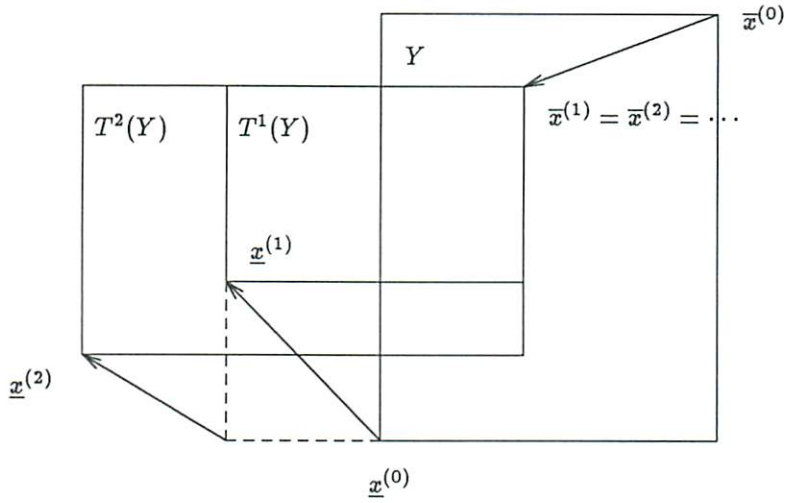


Figure 2.7: Demonstration of Lemma 2.9

for some $\underline{x}, \overline{x} \in U$. *In other words,* $F(x) \leq F(y)$ *for all* $x \leq y$ *such that* $x, y \in U$. *Here the inequality symbol* $\leq$ *denotes componentwise inequality. Suppose that* $Y = T^0(Y), T^1(Y), T^2(Y), \ldots \subseteq U$ *and let* $T^0(Y)$ *be of the form*

$$T^0(Y) = \{x \mid \underline{x}^{(0)} \leq x \leq \overline{x}^{(0)}\}.$$

Then,

$$T^d(Y) \subseteq \{x \mid \underline{x}^{(d)} \leq x \leq \overline{x}^{(d)}\}\ , \qquad d = 0, 1, 2, \ldots$$

for

$$\underline{x}^{(d)} = F(\underline{z}) \quad \textit{and} \quad \overline{x}^{(d)} = F(\overline{z})\ , \qquad d = 1, 2, \ldots$$

where

$$\underline{z}_i = \min\{\underline{x}_i^{(d-1)}, \underline{x}_i^{(0)}\} \quad \textit{and} \quad \overline{z}_i = \max\{\overline{x}_i^{(d-1)}, \overline{x}_i^{(0)}\}, \qquad i = 1, 2, \ldots, n. \qquad \square$$

This property results directly from Definition 2.4. Figure 2.7 demonstrates an example. Obviously, the above observation is also true when $\leq$ is replaced by a strict inequality $<$.

It is also worthwhile to point out the following observation.

Lemma 2.10 $T^{d-1}(Y) \subseteq T^d(Y)$ *for all* $Y \subseteq X$ *and* $d = 2, 3, \ldots$ $\square$

This can easily be shown by mathematical induction.

The main result of bounded delay model is stated by the following theorem. Convergence under bounded delay model does not require the uniqueness of a fixed point in the considered data domain. This was not the case for total asynchronism.

Theorem 2.4 *Under Model 2.3, any asynchronous iteration starting with* $x(0) \in X$ *converges to a fixed point* x^* *of* F *in* X *if the sequence* $\{Y(k)\}$ *converges to* x^* *where* $Y(0) = \{x(0)\}$ *and* $Y(k) = T^C(Y(k-1))$.

Proof. We will show by induction that at the l'-th time instance of $\Phi(k')$, i.e., at $t = \varphi(k') + l'$

$$x_i(t) \in T_i^{l'}(Y(k'-1)) \quad \text{for all} \quad k' = 1, 2, \ldots \quad \text{and} \quad i \in \alpha(t), \tag{2.3}$$

where $l' \leq C$. To avoid the redundancy of the arguments we omit the proof of this for $k' = 1$, which is the basis clause. Suppose this holds for all $k' = 1, 2, \ldots, k-1$. Then at $t = \varphi(k) + 1$

$$u_j^i(t) \in T_j^1(Y(k-2)) \cup T_j^2(Y(k-2)) \cup \cdots \cup T_j^C(Y(k-2)) = T_j^C(Y(k-2)) = Y_j(k-1),$$

for all $j = 1, 2, \ldots, n$ and $i \in \alpha(t)$. Then

$$x_i(t) = F_i(u^i(t)) \in T_i^1(Y(k-1)) \ , \quad \forall\, i \in \alpha(t)$$

which proves (2.3) for $k' = k$ and $l' = 1$. Suppose that it holds for $k' = k$ and $l' = 1, 2, \ldots, l-1$. To prove it for $l' = l$ we just have to notice that at $t = \varphi(k) + l$

$$u_j^i(t) \in Y_j(k-1) \cup T_j^1(Y(k-1)) \cup T_j^2(Y(k-1)) \cup \cdots \cup T_j^{l-1}(Y(k-1)) = Y_j(k-1) \cup T_j^{l-1}(Y(k-1))$$

for all $j = 1, 2, \ldots, n$ and $i \in \alpha(t)$. Thus,

$$x_i(t) = F_i(u^i(t)) \in T_i^l(Y(k-1)) \ , \quad \forall\, i \in \alpha(t)$$

and

$$x_i(t) \in T_i^C(Y(k-1)) \ , \quad \forall\, i \in \alpha(t).$$

Therefore, the elements of $x(t)$ are drawn from the elements of $\{Y(k)\}$ with nondecreasing indices. Since $\{Y(k)\}$ converges to x^*, $\{x(t)\}$ converges to x^*. □

2.4.3 Relationship With Total Asynchronism

Let us first introduce the notation $Y_C(k)$, in order to express the dependency of $Y(k)$ on the parameter C. From Theorem 2.4, it is clear that $Y_C(k)$ is the set of all possible values that can be generated in the k-th pseudo–cycle of a totally asynchronous iteration, when the pseudo–cycle is defined as it is given right before Model 2.2. If there is no upper–bound, then this set is equivalent to $X(k)$ in Definition 2.2, for $k > 0$. In other words, $Y_\infty(k) \equiv X(k)$, for all $k > 0$ and convergence of $\{Y_\infty(k)\}$ implies convergence of all totally asynchronous iterations. If $\{Y_\infty(k)\}$ does not converge, then there are two possibilities:

- $\{Y_C(k)\}$ converges for all possible values of C.
- There is a threshold C_{th} (possibly zero) such that $\{Y_C(k)\}$ converges for $C < C_{th}$ and it diverges otherwise.

Since $Y_C(k)$'s grow as C increases, there are no other possibilities.

Notice that the first case does not necessarily imply convergence under total asynchronism. However, if there exists an integer M such that $\{Y_M(k)\} \equiv \{Y_\infty(k)\}$, then all totally asynchronous iterations converge. If the domain X of shared data is finite, then this condition holds and the second case above does indeed imply totally asynchronous convergence. For further reference, let us restate this fact as a lemma.

Lemma 2.11 *If the domain X of shared data is finite, then convergence under bounded delay model, for all schedules, implies convergence under total asynchronism.* □

2.4.4 A Neural Network Example

Neural networks are computational paradigms, which are attracting an increasing number of researchers in recent years due to their promising capabilities of solving difficult artificial intelligence problems [26]. A neural network consists of simple processing elements, called neurons, interconnected in a particular way. Each neuron has a number of inputs and an output. Weights are associated with inputs, and the output is obtained by passing the weighted sum of the inputs through a non-linearity such as $\phi(\cdot)$ which is shown in Figure 2.8.

We will analyze a simple neural network consisting of two neurons, which can be represented by the following system of equations

$$\begin{aligned} x_1 &= F_1(x_1, x_2) = \phi(w_1\, x_1 + w_2\, x_2) \\ x_2 &= F_2(x_1, x_2) = \phi(w_2\, x_1 + w_1\, x_2) \end{aligned}$$

where $0 < w_2 < w_1 < 1$ and $w_1 + w_2 = 1$. $\phi(\cdot)$ is a continuous function with range $[-1, +1]$. It is monotonically non-decreasing, and in the domain $[-1, +1]$ it is strictly increasing. Also, it is symmetrical with respect to the origin (i.e., $\phi(z) = -\phi(-z),\ \forall z$) (Figure 2.8). For $z \in (0, \infty)$ we impose the following restrictions on ϕ:

- $y = z$ intersects $y = \phi(z)$ at $z = a < 1$.
- $y = \frac{1}{w_1} z$ intersects $y = \phi(z)$ at $z = b < a$.
- $y = \frac{1}{w_1 - w_2} z$ does not intersect $y = \phi(z)$.

Let $L1$ be the loci of points (x_1', x_2') satisfying

$$x_1' = F_1(x_1', x_2') = \phi(w_1\, x_1' + w_2\, x_2').$$

We can rewrite this as

$$x_2' = \frac{1}{w_2} [\phi^{-1}(x_1') - w_1\, x_1'].$$

setting $z = w_1\, x_1' + w_2\, x_2'$ and substituting $x_1' = \phi(z)$ we obtain

$$x_2' = \frac{w_1}{w_2} [\frac{1}{w_1} z - \phi(z)].$$

From the second assumption, for $x_1' \in (0, \phi(b))$, x_2' is negative and for $x_1' \in (\phi(b), 1)$, it is positive (Figure 2.9).

Because of the property $w_1 + w_2 = 1$, and because of the fact that a is a fixed point of $\phi(\cdot)$, $\xi^+ = (a, a)$ is a fixed point of F, and therefore, $L1$ passes through this point. Also, we can show that for $x_1' \in (0, \phi(b))$ $L1$ does not intersect the diagonal $D1$, because the vertical distance from the diagonal to $L1$ is

$$\delta = \frac{w_1}{w_2} [\frac{1}{w_1} z - \phi(z)] - (-\phi(z))$$

and

$$\frac{w_2}{w_1 - w_2} \cdot \delta = \frac{z}{w_1 - w_2} - \phi(z).$$

From the third assumption, δ does not become zero for $x_1' > 0$.

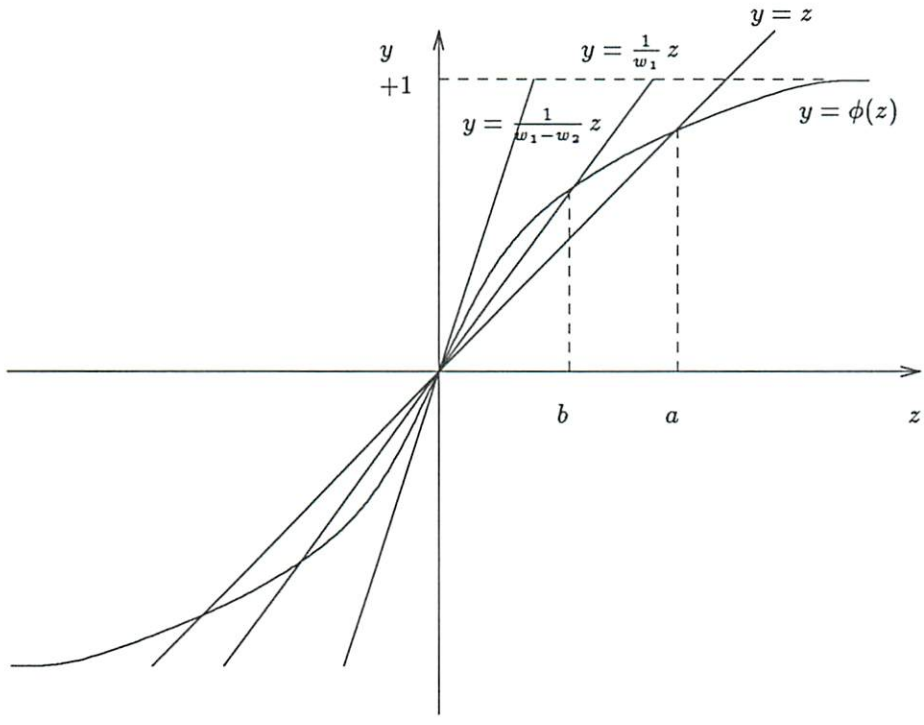


Figure 2.8: The function $\phi(\cdot)$

All this information makes it possible to obtain the part of $L1$ for positive x_1's (Figure 2.9). Because of the symmetry, it is also easy to obtain the remaining part of $L1$.

Since $\phi(\cdot)$ is increasing in $(-1, +1)$, for the points (x_1, x_2) above $L1$

$$x_1 < F_1(x_1, x_2)$$

and similarly below $L1$

$$x_1 > F_1(x_1, x_2).$$

$L2$ can similarly be defined as the loci of points (x_1'', x_2'') such that $x_2'' = F_2(x_1'', x_2'')$. From symmetry, $L2$ can be obtained as in Figure 2.9. In the figure, the arrows represent the directions of changes when F is applied to the points in different regions.

In what follows, we will analyze the convergence of asynchronous iterations corresponding to F, starting with initial points from different regions. From symmetry, it is sufficient to consider only quadrant I and the part of quadrant IV above the diagonal $D1$. The results for other regions can be obtained by symmetrical arguments.

Let us denote $\underline{R}$ as the set of points in the interior of the closed curve defined by $(0, 0)$, $L2$, ξ^+ and $L1$, and the points on the boundary except $(0, 0)$. Similarly, the interior points of the closed curve defined by ξ^+, $L1$, $(1, 1)$ and $L2$, and the boundary points constitute the set $\overline{R}$ (Figure 2.10).

Let $\{X(k)\}$ be the sequence defined by

$$X(k) = \{x \mid \underline{x}(k) \leq x \leq \overline{x}(k)\}, \qquad k = 0, 1, 2, \ldots \tag{2.4}$$

such that $\underline{x}(0) \in \underline{R}, \overline{x}(0) \in \overline{R}$ and

$$\underline{x}(k) = F(\underline{x}(k-1)), \quad \overline{x}(k) = F(\overline{x}(k-1)), \quad k = 1, 2, \ldots$$

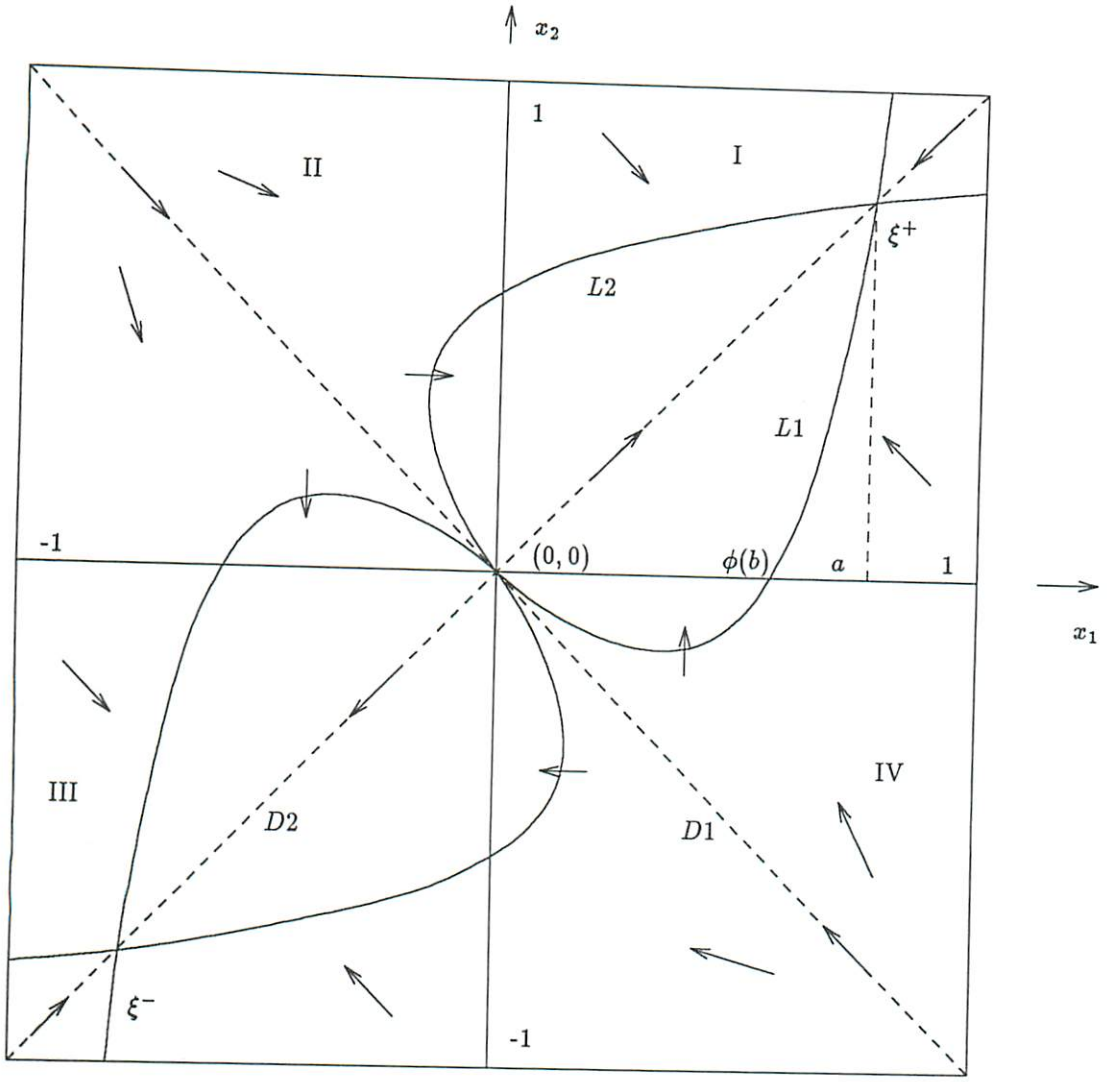


Figure 2.9: Update directions in different regions

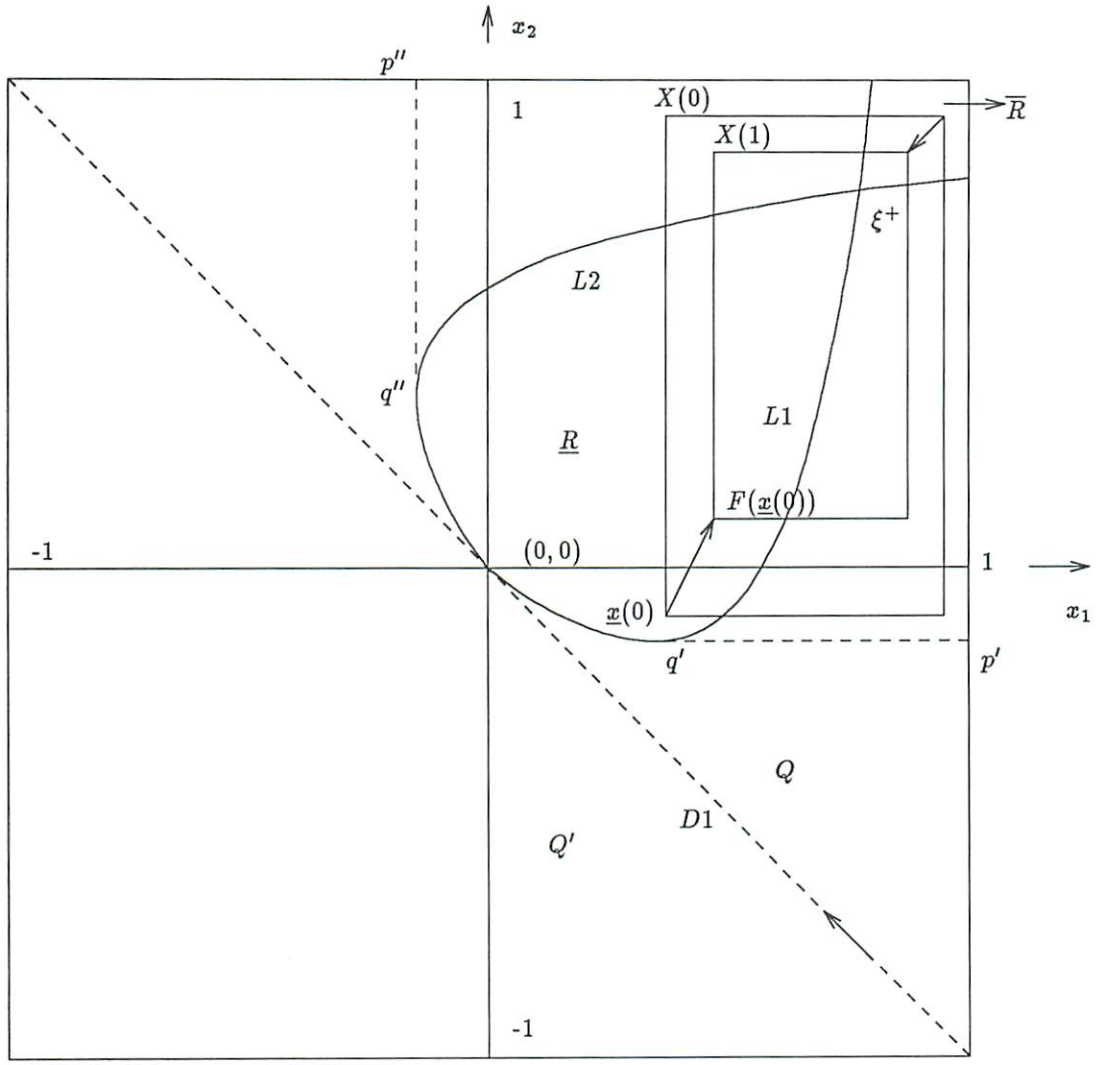


Figure 2.10: Totally asynchronous convergence

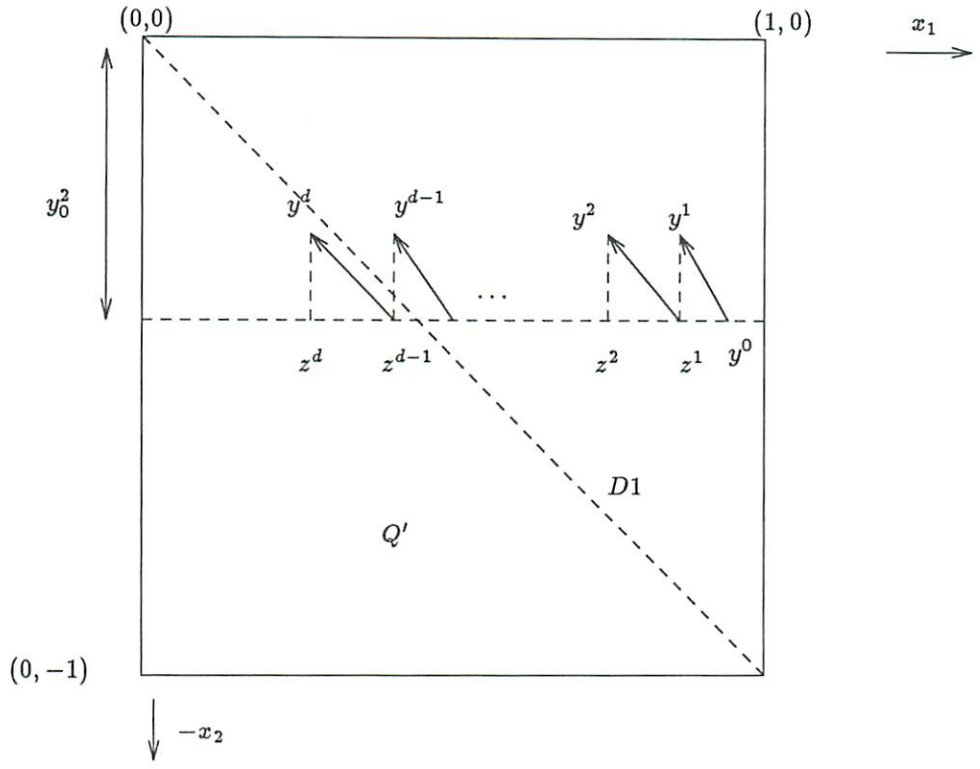


Figure 2.11: Sequences y^k and z^k

Since we have the monotonicity property, i.e.,

$$x \leq y \;\Rightarrow\; F(x) \leq F(y),$$

then

$$x \in X(k) \;\Rightarrow\; F(x) \in X(k+1).$$

On the other hand, for any point $\underline{x}$ in $\underline{R}$

$$\underline{x} < F(\underline{x}) \qquad \text{and} \qquad \lim_{k\to\infty} F^k(\underline{x}) = \xi^+ = (a, a),$$

where F^k denotes the function corresponding to k successive applications of F. Similarly, for any point $\overline{x}$ in $\overline{R}$

$$\overline{x} > F(\overline{x}) \qquad \text{and} \qquad \lim_{k\to\infty} F^k(\overline{x}) = \xi^+ = (a, a)$$

which proves the second part of asynchronous contraction. Subsequently, $\{X(k)\}$ satisfies Theorem 2.1. Now, let R be the set of points in the region bounded by p', q', $(0,0)$, q'', p'' and $(1,1)$, including the boundary except the point $(0,0)$. For any $x(0) \in R$, there exists $X(0)$ such that $\underline{x}(0) \in \underline{R}$ and $\overline{x}(0) \in \overline{R}$. Therefore, any totally asynchronous iteration converges to ξ^+, for such $x(0)$.

The only region left to consider is the one in quadrant IV and is surrounded by $(0,0)$, $L1$, q', p', $(1,-1)$ and $D1$. Q will denote the set of points in this region. Except for the part from p' to $(1,-1)$, the boundary is excluded from Q. Furthermore, the set of points to the left of $D1$, including $D1$, will be labelled as Q'.

Now consider the sequence $\{y^k\}$ of points starting with $y^0 \in Q$ such that

$$y^k = F(z^{k-1}), \qquad k = 1, 2, \ldots$$

where

$$z^k = (y_1^k, y_2^0), \qquad k = 0, 1, \ldots$$

(See Figure 2.11.)

The following observations about $\{y^k\}$ are crucial for our analysis.

Lemma 2.12 *For any positive integer C,*

- *if for some $d \leq C$, $y^d \in Q'$, then $y^k \in Q'$ for all $d \leq k \leq C$;*
- *otherwise, $y^C \in Q \cup R$. Furthermore, in this case, $z^k \in Q$ for all $k = 0, 1, 2, \ldots, C-1$.*

Proof. We will not give a formal proof. Instead, the reader is advised to follow the directions of updates in Figure 2.9. □

Lemma 2.13 *For all d satisfying $0 < d \leq C$, there exists an asynchronous schedule $\mathcal{S}$ with bounded delay B such that for some time instance t' of the asynchronous iteration $(F, y^0, \mathcal{S})$*

$$\begin{aligned} x(t') &= y^{d-1}, \\ x(t'+1) &= y^d, \end{aligned}$$

where $C = 2B - 1$.

Proof. For notational convenience we make use of negative time instances such that $x(t) = y^0$, $\forall\, t \leq 0$. We construct $\{x(t)\}$ such that x_1 is updated at each time instance and x_2 is updated only at $t = B, d-1, d$. For the update at each t the first component of the input is $x_1(t-1)$ and the second component of the input is $x_2(t-B)$. Therefore,

$$\begin{aligned} x_1(t) &= F_1(x_1(t-1), x_2(t-B)) = F_1(x_1(t-1), y_2^0), \\ x_2(t) &= \begin{cases} F_2(x_1(t-1), x_2(t-B)) = F_2(x_1(t-1), y_2^0) & t = B, d-1, d \\ x_2(t-1) & \text{otherwise} \end{cases} \end{aligned}$$

Notice that the update of x_2 at $t = B$ is only needed to satisfy the first requirement of asynchronism with bounded delay. Otherwise, it is not used as an input. It is clear that

$$\begin{aligned} x(d-1) &= y^{d-1} \\ x(d) &= y^d \end{aligned}$$

□

An immediate consequence of the above result is the following.

Lemma 2.14 *If y^C is in Q', then there exists an asynchronous schedule $\mathcal{S}$ with bounded delay $B > 1$ such that $(F, y^0, \mathcal{S})$ does not converge.*

Proof. From Lemma 2.12, there exists a $d \leq C$ such that y^{d-1} is in $Q \cup R$ and y^d is in Q'. Then, from the previous lemma, we can construct the first $t'+1$ elements of $\{x(t)\}$ such that

$$x(t') = y^{d-1} \quad \text{and} \quad x(t'+1) = y^d$$

and for $t > t'$ define

$$x(t) = F(x(t-2)).$$

Then the subsequence

$$x(t'), x(t'+2), x(t'+4), \ldots$$

converges to ξ^+ and the subsequence

$$x(t'+1), x(t'+3), x(t'+5), \ldots$$

converges to $(0,0)$ if y^d is on $D1$ and to ξ^- otherwise. This means that $\{x(t)\}$ does not converge. □

Let $g^k(\cdot)$ be the function that maps y^0 to y^k:

$$y^k = g^k(y^0),$$

and define the sequence $\{s(k)\}$ such that

$$s(k) = \begin{cases} \underline{x} & k = 0 \\ g^C(s(k-1)) & k > 0. \end{cases}$$

Also define

$$Y(0) = \{x \mid \underline{x} \leq x \leq \overline{x}\}$$

such that $\underline{x}$ is in Q and $\overline{x}$ is in $\overline{R}$, and $Y(k) = T^C(Y(k-1))$.

Lemma 2.15 *If for some $L > 0$, $s(L)$ is in $\underline{R}$ and $s(k)$'s are in Q for all $k < L$, then*

$$Y(k) \subseteq \{x \mid s(k) \leq x \leq F^k(\overline{x})\}, \quad \textit{for all } k \leq L$$

Proof. We omit the proof for $k = 1$. Let us assume that the hypothesis holds for $k - 1 < L$. Then, from Lemma 2.9 and from the second part of Lemma 2.12 it follows that

$$Y(k) = T^C(Y(k-1)) \subseteq \{x \mid g^C(s(k-1)) \leq x \leq F(F^{C-1}(\overline{x}))\}.$$

□

The final result of the section is as follows.

Proposition 2.1 *All asynchronous iterations with bounded delay $B > 1$ and starting with an initial vector $x(0)$ in Q converge to ξ^+ if and only if the sequence $\{s(k)\}$ converges to ξ^+.*

Proof. Let $s(L)$ be the first element of $\{s(k)\}$ which is not in Q. That means $s(L-1)$ is in Q and for $s(L)$ there are two cases:

- If $s(L) \in Q'$, then from Lemma 2.14 not all asynchronous iterations with delay B converge.
- If $s(L) \in \underline{R}$, then from Lemma 2.15 $Y(L)$ is in the form of (2.4) and from Theorem 2.1 all totally asynchronous iterations starting with an initial vector $Y(L)$ converges to ξ^+. From Lemma 2.9, there exists an asynchronous iteration which visits all the minimum (maximum) points of all the sets $Y(k)$. Therefore, $\{Y(k)\}$ converges to $\{\xi^+\}$. From Theorem 2.4 the claim follows. □

z	$\phi(z)$
0.0	0.00
(-)0.1	(-)0.12
(-)0.2	(-)0.24
(-)0.3	(-)0.35
(-)0.4	(-)0.45
(-)0.5	(-)0.55
(-)0.6	(-)0.64
(-)0.7	(-)0.72
(-)0.8	(-)0.8
(-)0.9	(-)0.87
(-)1.0	(-)0.94
(-)1.1	(-)1.0

k	$s(k)$	
	$B = 3$	$B = 4$
0	(0.65, -0.30)	(0.65, -0.30)
1	(0.46, -0.26)	(0.41, -0.27)
2	(0.37, -0.24)	(0.31, -0.25)
3	(0.32, -0.22)	(0.24, -0.24)
4	(0.28, -0.20)	(0.15, -0.24)
5	(0.26, -0.18)	(-0.01, -0.29)
6	(0.25, -0.17)	(-0.38, -0.49)
7	(0.24, -0.15)	(-0.70, -0.71)
8	(0.24, -0.14)	(-0.78, -0.78)
9	(0.25, -0.12)	(-0.80, -0.80)
10	(0.25, -0.10)	(-0.80, -0.80)
11	(0.26, -0.07)	(-0.80, -0.80)
12	(0.27, -0.05)	(-0.80, -0.80)
13	(0.28, -0.02)	(-0.80, -0.80)
14	(0.29, 0.01)	(-0.80, -0.80)
15	(0.31, 0.05)	(-0.80, -0.80)
...	...	...

Table 2.1: A numerical example

Consider the case where $\phi(\cdot)$ is a piecewise linear function that passes through the points given in Table 2.1, and

$$\begin{aligned} x_1 &= \phi(0.9\,x_1 + 0.1\,x_2) \\ x_2 &= \phi(0.1\,x_1 + 0.9\,x_2). \end{aligned}$$

It can be verified that $a = 0.8, b = 0.45$, and the assumptions stated at the outset of the section are satisfied. It is seen, for this example, that $\{s(k)\}$ converges to $\xi^+ = (0.8, 0.8)$ when $B = 3 (C = 5)$ and it does not converge to ξ^+ when $B = 4 (C = 7)$ (Table 2.1).

The significance of this result for this specific example is that convergence cannot be guaranteed for all possible initial points (x_1, x_2) under totally asynchronous conditions. For some initial values of the components, the algorithm must be restricted to an asynchronous iteration with delay $B < 4$. If an asynchronous iteration with delay $B < 4$ starts in R, and if it is observed that the iterate vector has been in the region R for at least B consecutive updates, then the computation could switch dynamically to a totally asynchronous scheme, converging to ξ^+.

2.4.5 Enforcing the Model on a Shared Memory Architecture

Although it may be safe to assume that Model 2.2 and Model 2.3 correspond to reasonable systems, it is not clear how to determine the value of the parameter B or C, for a realistic system, which we need in order to use our main result. Indeed, these parameters are affected by various random sources including not only hardware properties, such as processing time, memory access time, but also load balancing, and worst of all they may be data dependent. When we do not have a way of predicting B (or C), we must

have a mechanism to enforce its value. In this section, we discuss such a mechanism for a shared memory multiprocessor architecture.

We assume that the system supports critical sections, either in hardware or in software. We also assume that the shared memory access operations, including reading, writing, incrementing ($inc(\cdot)$) and decrementing ($dec(\cdot)$) shared variables are *atomic*. In other words, only one processor can execute one of these access operations on a given shared variable at a given time.

Consider the following scheme of computations on a shared memory multiprocessor defined as above, for the solution of the fixed point problem corresponding to F. The number of processors is equal to the number of components (n), and the execution on each processor consists of alternating phases of memory access and computation. Memory access phases are assumed to be instantaneous and will be called *checkpoints* here. In practical terms, these phases are executed in *critical sections*. At each checkpoint the component which is computed in the previous computation phase is updated in the shared memory, the component to be computed in the next computation phase is selected and the components required for the next computation phase are fetched from the shared memory. The status of the computations is also updated, as will be detailed later. In the compute phase, the component which is last selected is computed. Between two consecutive checkpoints on a processor, there may be other overhead, such as busy-waiting, but with no loss of generality we can include this overhead in the computation phase. It is enforced that there exists a sequence $\{\psi(t)\}$ of increasing time instances starting with the initial value $\psi(0)$ such that the following conditions hold.

- The time interval $(\psi(t), \psi(t+1)]$ which is denoted by $I(t)$, is the maximum time interval covering at most one checkpoint of each processor, for all t.
- At the checkpoint of processor p in $I(t)$, the component i for the next computation is selected by the following rule:

  $$i = (T + p) \bmod n,$$

  where $T = t \bmod (B-1)$ and p takes values from 1 to n.
- For $t = 0, B-1, 2(B-1), \ldots$, $I(t)$ covers one and only one checkpoint of each processor.

Figure 2.12 displays an example for $n = 4$ and $B = 4$. The horizontal lines correspond to the time axises of the processors. The checkpoints are marked with $\times$ symbols and each number refers to the component being updated at the current checkpoint and which was selected at the previous checkpoint according to the second condition. Notice that each interval bounded with double dotted lines covers one and only one checkpoint of each processor, as stated by the third condition. For the moment we can ignore the circles which will be explained later.

We now show that this scheme implements Model 2.3, with $C = 2B - 1$. A naive approach towards this goal might suggest mapping each interval $I(t)$ to the time instance t and considering the data sequence generated after this mapping as an asynchronous iteration as defined by Definition 2.1. This reasoning is not correct, because multiple updates of the same components, in an interval, is possible in our scheme, such as the updates of component 3 in $I(5)$ and $I(8)$, whereas Definition 2.1 does not allow more than one update of the same component, at the same time. Nevertheless, this can easily be corrected by simply extending the dimension of the data space of our original fixed point problem, so that different updates of the same component in the same interval can be considered as updates of different components. In other words, our implementation can be viewed as special case of the scheme where each processor keeps its own copy of x:

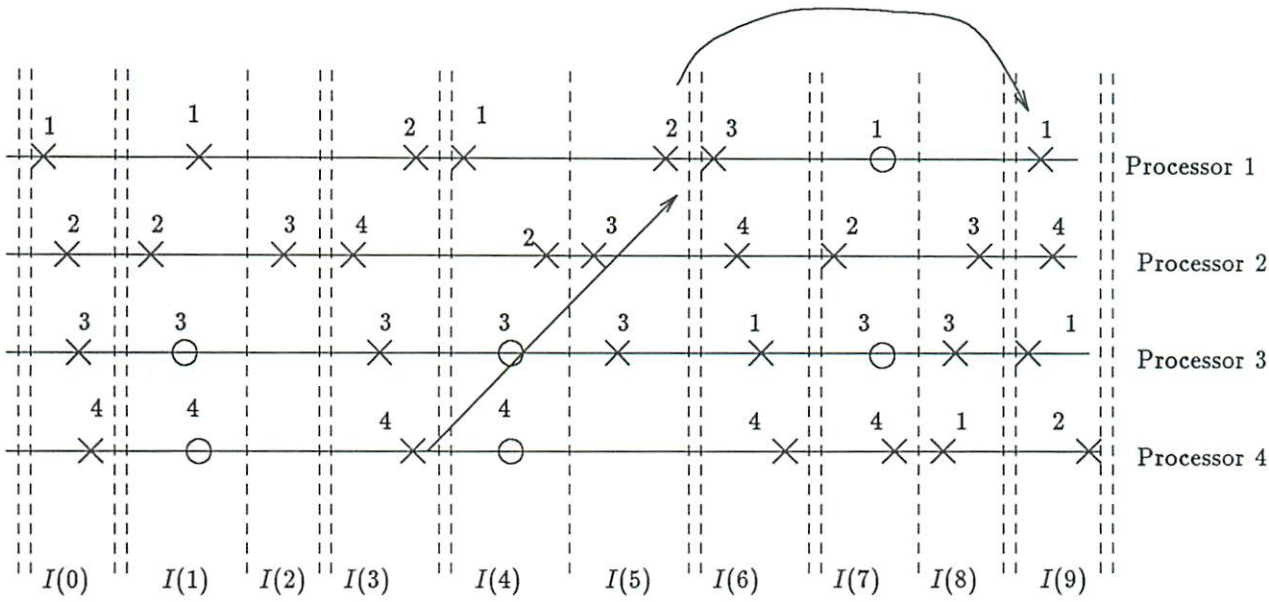


Figure 2.12: Demonstration of the implementation

each processor only updates its own copy, but it can fetch data from other processors. Each component of a copy can then be considered as a different component.

To be more precise, given our original fixed point problem corresponding to F defined in the data space X, consider the one corresponding to $\hat{F}$ defined in the data space $\hat{X} = X \times X \times \cdots \times X$, where

$$\hat{F}(\hat{x}) = (F(x^{(1)}), F(x^{(2)}), \ldots, F(x^{(n)})), \qquad \forall\, x^{(j)} \in X$$

for

$$\hat{x} = (x^{(1)}, x^{(2)}, \ldots, x^{(n)}).$$

Obviously, the solutions of the two problems are equivalent, in the sense that if ξ is a fixed point of the original problem, $\hat{\xi} = (\xi, \xi, \ldots, \xi)$ is the fixed point of the problem corresponding to $\hat{F}$, and vice versa.

By replacing n with $\hat{n} = n \cdot n$, F with $\hat{F}$, $x(0)$ with $\hat{x}(0) = (x(0), x(0), \ldots, x(0))$ and $\mathcal{S}$ with an $\hat{n}$-dimensional schedule $\hat{\mathcal{S}}$ in Definition 2.1, we can define an asynchronous iteration for the $\hat{n}$-dimensional problem corresponding to $\hat{F}$. We can now claim that our scheme corresponds to a schedule $\hat{\mathcal{S}}$ such that, for all t, the update values generated in $I(t)$ are exactly the same as the ones generated by $\{\hat{x}(t)\} = (\hat{F}, \hat{x}(0), \hat{\mathcal{S}})$ at the time instance t. We just need to see that only a section $x^{(p)}$ of the shared data $\hat{x} = (x^{(1)}, x^{(2)}, \ldots, x^{(n)})$ is exclusively updated by a processor p. On the other hand, convergence conditions of asynchronous iterations $\{x(t)\} = (F, x(0), \mathcal{S})$ and $\{\hat{x}(t)\} = (\hat{F}, \hat{x}(0), \hat{\mathcal{S}})$ are equivalent. The reason is as follows. If $\hat{Y}$ denotes the Cartesian product of Y's, i.e., $\hat{Y} = Y \times Y \times \cdots \times Y$, and $\hat{T}^d(\hat{Y})$ is defined by replacing the dimension of the problem by $\hat{n} = n \cdot n$ in Definition 2.4, then

$$\hat{T}^d(\hat{Y}) = T^d(Y) \times T^d(Y) \times \cdots \times T^d(Y).$$

This can easily be verified for $d = 1$ by noticing that the components of $\hat{T}^1(\hat{Y})$ with indices $i, i+n, i+2\,n, \ldots$ (the i-th components of different copies) are determined only from F_i and Y, as well as the i-th component of $T^1(Y)$, so they are all equivalent. By induction, we can verify this for all d, in a similar manner. Consequently, the condition of Theorem 2.4 for our original n-dimensional problem is equivalent to the corresponding condition for the $\hat{n}$-dimensional problem for $\hat{x}(0) = (x(0), x(0), \ldots x(0))$ and $\hat{\xi} = (\xi, \xi, \ldots, \xi)$

Having seen that the implemented scheme is a particular instance of the $\hat{n}$-dimensional problem corresponding to $\hat{F}$ and that the convergence properties of the $\hat{n}$-dimensional and the n-dimensional asynchronous iterations are the same, we now have to assert that our scheme satisfies the conditions of Model 2.3. Notice that if each computation initiated in $I(0), I(B-1), I(2(B-1)), \ldots$ completed in the very next interval, then each component of X would be updated in each of the intervals $I(1), I(B), I(2\,B-1), \ldots$. Obviously, this may not be the case. However, for each computation initiated in an interval $I(t)$, we can imagine the existence of a dummy processor that generates, in $I(t+1)$, the same value as the actual update for this computation. In Figure 2.12 dummy updates are denoted by circles. For example, the dummy updates of components 3 and 4 in $I(1)$ are duplications of their actual updates in $I(3)$. We can further continue duplicating the updates until we satisfy the requirement that all the components of the $\hat{n}$ dimensional vector $\hat{x}$ are updated in each of the intervals $I(1), I(B), I(2\,B-1), \ldots$. Therefore, with the aid of dummy updates, the first condition of Model 2.3 is satisfied.

The second condition of Model 2.3 can also be easily asserted. Consider an update in $I(t)$. Since the distance between two consecutive checkpoints on the same processor is at most $B-1$ intervals, then the update in $I(t)$ uses values from no earlier than $I(t-B+1)$, and these values are generated no earlier than $I(t-B+1-(B-1)) = I(t-2\,(B-1))$. For example, the update of component 1 in $I(9)$ uses the value of component 4 right before $I(6)$ and this value is generated in $I(3)$ as displayed by the arrows in Figure 2.12. This completes the discussion for correctness of our scheme.

In the remaining part of the section, implementation details of our scheme are covered. Here we have two main issues of concern:

- how to keep track of the indices of the intervals;
- how to enforce that each of the intervals $I(0), I(B-1), I(2\,(B-1)), \ldots$ covers a checkpoint of each processor.

For the first issue, suppose we have a shared variable t that contains the current interval index. It is clear that the value of t at a checkpoint should be larger than its value at the previous checkpoint on the same processor. If this is not true right before the checkpoint, then t should be incremented within the checkpoint. Let $t[p]$ be the local variable on processor p that contains the value of t at the time of the previous checkpoint. Then, at a checkpoint, t can be updated by the following assignment.

$$t \leftarrow \max\{t, t[p]+1\}.$$

Obviously, it is sufficient to keep track of t mod $(B-1)$ instead of t. In the algorithm we will develop, the shared variable T will serve this purpose.

The second issue we mentioned above corresponds to the requirement that after T becomes zero, all of the processors should execute their checkpoints once, before T can be incremented. In the situation where $T=0$ and all the processors have not executed their checkpoints since T last became zero, any checkpoint that attempts to increment T should be delayed. To detect this situation, we need a counter CNT to store the number of processors that have executed a checkpoint once after T last became zero. If a processor wants to increment T when CNT is not equal to n, it waits in a busy-waiting loop for CNT to become n. If we are not careful enough, this loop may cause a deadlock in the following way. While a processor p is waiting in this loop, other processors keep increasing CNT until it becomes n, but before processor p gets

this value, it is possible that another processor may change CNT again, causing processor p to get stuck in the busy-waiting loop. To prevent this situation, we use another counter W that contains the number of processors waiting for CNT to become n. When $CNT = n$ no processor is allowed to change CNT before $W = 0$.

The resulting algorithm to implement our scheme is displayed by the flowchart in Figure 2.13. It shows the execution on processor p. Critical section entry and exit points are denoted by circle blocks. Except for x, the shared variables are T, CNT and W. *Inc* and *dec* functions are assumed to be modulo n, i.e., $inc(n) = 1$.

There are two busy-waiting loops in the algorithm, "spinning" on variables CNT and W, which seem to cause performance degradation. However, these loops are executed only when the third condition of the definition of $I(t)$ is not satisfied by the natural timing of the computations. The better actual values of the delays match the implemented value of B, the less frequently the processors will loop on CNT. The possibility of looping on W is even less since W is introduced only to prevent deadlock, as mentioned above, and the expected occurrence of such deadlocks is negligible. Therefore, except at the critical section entry points, the processors are not blocked very often.

2.5 Non-Redundant Asynchronous Computations

2.5.1 Definition

In most computations, the computation of a component x_i always uses the most recent value of x_i as an input component. As we discussed in Section 2.2 this can be identified by the fact that the computation of a component by multiple processors, at the same time is not possible. We call this property *non-redundancy* and formulate it as follows.

Model 2.4 (Non-redundant Asynchronism) *Besides the requirements of Model 2.1 (total asynchronism), there holds:*

$$\tau_i^i(t) = t - 1, \qquad \forall i = 1, 2, \ldots, n, \quad t = 1, 2, \ldots$$

□

2.5.2 Convergence

Lemma 2.16 *A non-redundant asynchronous iteration $\{x(t)\} = (F, x(0), \mathcal{S})$ converges to a fixed point if every limit point of the input sequence $\{u^i(t)\}$ is a fixed point, for all $i = 1, 2, \ldots, n$, where F is continuous in a domain that contains $x(0)$.*

Proof. Let us fix i and define $\{u^i(t_l)\}$ as the subsequence of $\{u^i(t)\}$ obtained by eliminating the instances at which the i-th component is not updated. Now consider a limit point x^* of $\{u^i(t)\}$. There exists a subsequence $\{u^i(t_k)\}$ of $\{u^i(t_l)\}$ that converges to x^*. From continuity of F and since x^* is a fixed point

$$\lim_{t_k \to \infty} x_i(t_k - 1) = \lim_{t_k \to \infty} u_i^i(t_k) = F_i(\lim_{t_k \to \infty} u^i(t_k)) = \lim_{t_k \to \infty} (F_i(u^i(t_k))) = \lim_{t_k \to \infty} x_i(t_k),$$

which means that $\{x_i(t_k - 1)\}$ and $\{x_i(t_k)\}$ converges to the same point. On the other hand, we can partition $\{u^i(t_l)\}$ into subsequences such that each subsequence converges to a fixed point. We can repeat the above

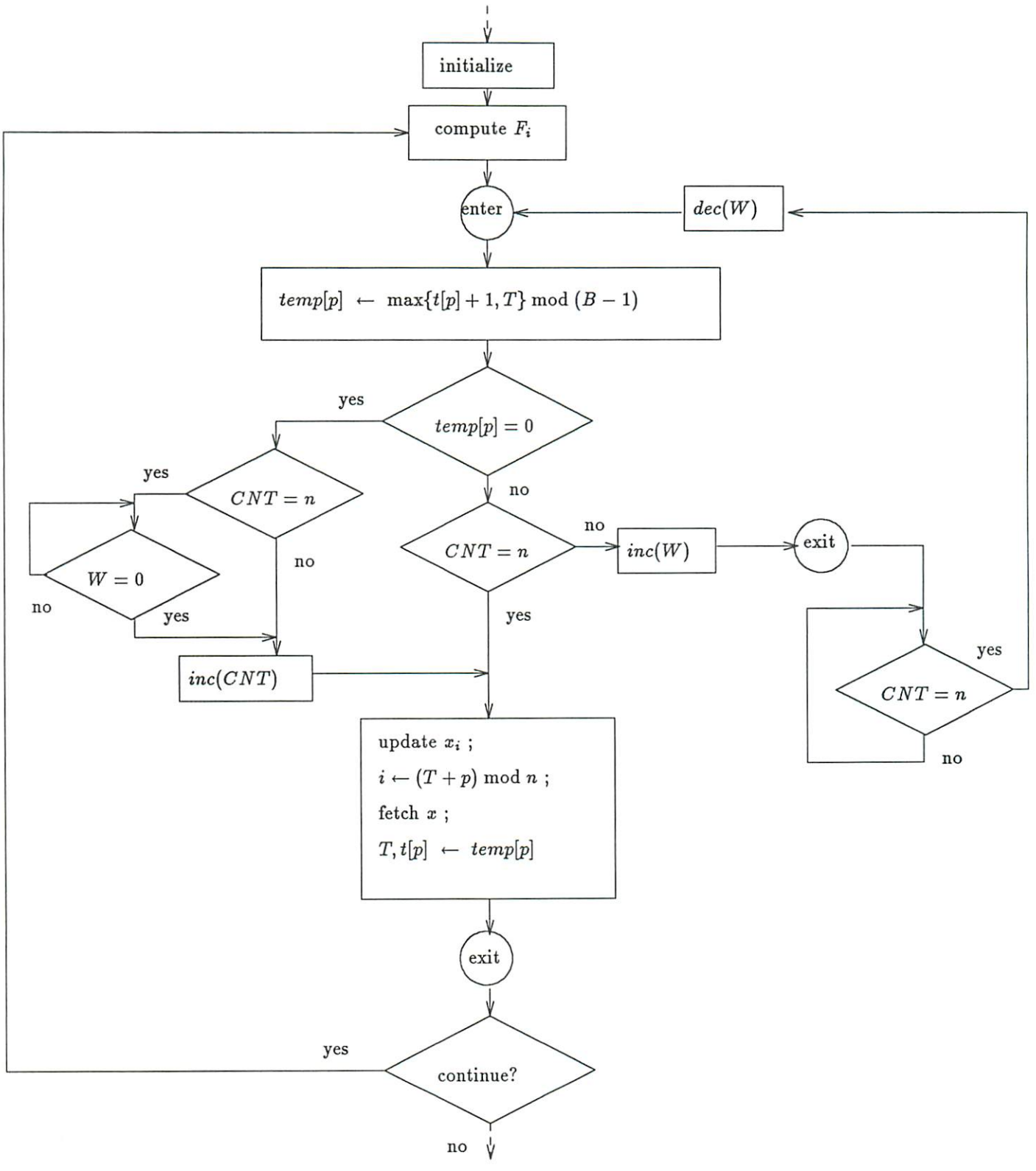


Figure 2.13: The algorithm to implement Model 2.3

arguments for each subsequence and as a result, we conclude that $\{x_i(t)\}$ converges. Therefore, $\{x(t)\}$ converges and from Lemma 2.1 it converges to a fixed point. □

The immediate consequence of the above lemma can be stated as follows.

Proposition 2.2 *Let $F : X \mapsto X$ be an operator, $\{X(k)\}$ be a sequence of subsets of X and $X^* = X_1^* \times X_2^* \times \cdots \times X_n^* \subseteq X$ be a set of fixed points of F. Then, all non-redundant asynchronous iterations with $x(0) \in X(0)$ converge to a fixed point in X^* if the following conditions hold.*

- (Box Condition) *For all $k = 0, 1, 2, \ldots, n$, $X(k)$ is the Cartesian product of n sets, i.e.,*

$$X(k) = X_1(k) \times X_2(k) \times \cdots \times X_n(k).$$

- *For all $k = 0, 1, 2, \ldots$, $X^* \subseteq X(k+1) \subseteq X(k)$, and furthermore, every limit point of all the sequences $\{z(k)\}$ such that $z(k) \in X(k)$ is a member of X^*.*

- $x \in X(k) \Rightarrow F(x) \in X(k+1), \quad k = 0, 1, 2, \ldots$

Proof. Let $\{\varphi(k)\}$ be the cycle sequence. Then, from the proof of Theorem 2.1

$$u^i(t) \in X(k), \quad \forall t > \varphi(k+1), \quad \forall i = 1, 2, \ldots, n.$$

Therefore, for all i, every limit point of $\{u^i(t)\}$ converges and the previous lemma applies. □

Proposition 2.3 *Given posets $(X_i, \preceq_i)$, for all $i = 1, 2, \ldots, n$, let $(X, \preceq)$ be a poset such that $x \preceq y$ if $x_i \preceq y_i$, for all $i = 1, 2, \ldots, n$ and $x, y \in X$. Then, all non-redundant asynchronous iterations $\{x(t)\} = (F, x(0), \mathcal{S})$ with $x(0) \in X = X_1 \times X_2 \times \cdots \times X_n$ satisfy the property that $x(t) \preceq x(t+1)$, for all $t = 0, 1, 2, \ldots$, if*

- *X is closed under F, and*

- *for all $i = 1, 2, \ldots$,*

$$F_i(x) \preceq x_i, \quad \forall x \in X$$

Proof. For all $t = 1, 2, \ldots$ and $i = 1, 2, \ldots, n$, there are two cases possible:

- $i \in \alpha(t)$ (x_i is updated at t). In this case,

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

  where,

$$u_i^i(t) = x_i(\tau_i^i(t)) = x_i(t-1).$$

  From the second condition,

$$x_i(t) \preceq_i x_i(t-1).$$

- $i \notin \alpha(t)$ (x_i is not updated at t). By definition, $x_i(t) = x_i(t-1)$. □

It immediately follows that if the conditions of the above lemma are satisfied, then all non-redundant asynchronous iterations converge to a fixed point in the following cases.

- X is a finite set.

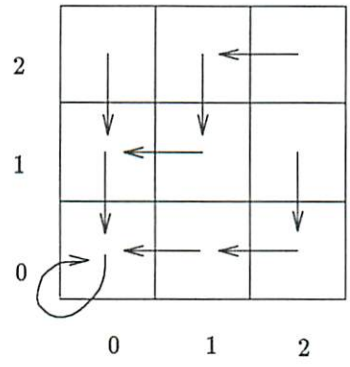


Figure 2.14: An operator convergent under non-redundancy but not under total asynchronism

- $F : \Re^n \mapsto \Re^n$ is a real and continuous function and $\preceq_i$ is defined as the order relation $\leq$ in the usual sense.

In both cases, a decreasing sequence implies convergence and from Lemma 2.1, convergence to a fixed point is guaranteed.

Example 2.7 The conditions of Proposition 2.3 do not imply asynchronous contraction. The operator given in Figure 2.14 supports this: it satisfies the conditions of the proposition, although it is not asynchronously contracting. □

2.6 Partial Asynchronism

2.6.1 Definition

Partial asynchronism can simply be defined as a model which satisfies the conditions of both non-redundancy and bounded delay. Results which are specific to this model were given in [27], [49], [50], [51], [46], [7] and the term partial was introduced by Bertsekas et.al. [7], [46]. Following the terminology of this thesis, we would probably call it *non-redundant asynchronism with bounded delay.* We nevertheless will prefer the original terminology. Partial Asynchronism is formally defined as follows.

Model 2.5 (Partial Asynchronism) *There exists a positive integer B such that:*

- *For every $i = 1, 2, \ldots, n$ and for every $t \geq 0$ at least one element t' of the set $\{t, t+1, \ldots, t+B-1\}$ satisfies $i \in \alpha(t')$.*
- *$t - B \leq \tau_j^i(t) < t$, for all i, j and t such that $i \in \alpha(t)$.*
- *$\tau_i^i(t) = t - 1$ for all i and t such that $i \in \alpha(t)$.* □

By definition, convergence under asynchronism with no redundancy implies convergence under partial asynchronism. This means that the results of the previous section can be used for partial asynchronism. In the following, we further prove that if the domain X of data is finite, then convergence under partial asynchronism for all B implies convergence under non-redundant asynchronism.

Lemma 2.17 *Let F be a closed function in a finite set X and let $x(0) \in X$. If all asynchronous iterations $(F, x(0), \mathcal{S})$ converge under partial asynchronism, then all asynchronous iterations $(F, x(0), \mathcal{S})$ also converge under non-redundancy.*

Proof. In this proof, given a schedule, $\{\varphi(k)\}$ will denote a particular pseudo–cycle sequence, for example the minimum pseudo–cycle sequence. Now, given $x(0)$ and F, let us fix a partially asynchronous schedule. Let $P(k)$ be the set of values generated in the k-th pseudo-cycle. Also let $v(k)$ be the last value of the data in the k-th pseudo–cycle, i.e., $v(k) = x(\varphi(k+1))$. We observe that there exists no two k' and k'' such that $P(k') \equiv P(k'')$ and $v(k') = v(k'')$, unless all the elements of $P(k')$ and $v(k')$ are fixed points. Otherwise, there would exist a partially asynchronous schedule, for which some points are repeated indefinitely, which are not fixed points. Since the domain of data is finite, it follows that there are finite choices of $(P(k), v(k))$ pairs. As a result, there exists an integer m such that all partially asynchronous iterations converge in at most m pseudo–cycles. This means that all non-redundant asynchronous iterations also converge in at most m pseudo–cycles. This can be seen by contradiction. Suppose that there exists a non-redundant asynchronous iteration that does not converge in m pseudo–cycles, then there would be a partially asynchronous schedule which does not converge in m pseudo–cycles. □

Given the results of asynchronous models with bounded delay and with non-redundancy, a straightforward approach to derive a convergence condition for partial asynchronism is to combine the two models as follows. Let the set Y be defined as follows:

$$Y = \bigcup_{k=0}^{\infty} \{\prod_{i=1}^{n} \{Y_i(k) \cup Y_i(k+1)\}\}.$$

From the proof of Theorem 2.4, it is clear that $u^i(t) \in Y$, for all $i = 1, 2, \ldots, n$ and $t = 0, 1, 2, \ldots$ Therefore, replacing X with Y in the previous section the results apply for partial asynchronism.

2.6.2 Summary of Previous Work

2.6.2.1 Maximum Contraction

This section is due to [46] and [7]. The results can also be used to obtain the results of [27].

In this section, $F : \Re^n \mapsto \Re^n$ is a real function. Let X^* be the set of fixed points of F and for each $x \in \Re^n$ let $||x|| = \max_{i=1,2,\ldots,n} |x_i|$ denote the maximum norm of x. For any $x \in \Re^n$, we denote by $\rho(x)$ the distance of x from X^*, defined by

$$\rho(x) = \inf_{y \in X^*} ||x - y||.$$

Finally, given any $x \in \Re^n$ and $x^* \in X^*$, we let $I(x; x^*)$ be the set of indices of coordinates of x that are farthest away from x^*, that is,

$$I(x; x^*) = \{i \mid |x_i - x_i^*| = ||x - x^*||\},$$

and we also denote

$$U(x; x^*) = \{y \in \Re^n \mid y_i = x_i \text{ for all } i \in I(x; x^*), \text{and} |y_i - x_i^*| < ||x - x^*|| \text{ for all } i \notin I(x; x^*)\}.$$

In other words, $U(x; x^*)$ is the set of all vectors y with $||y - x^*|| = ||x - x^*||$ that agree with x in the components that are farthest away from x^*.

The function F is assumed to satisfy the following assumption:

Assumption 2.1

(a) F is continuous.

(b) The set of fixed points X^* is convex and nonempty.

(c) $||F(x) - x^*|| \leq ||x - x^*||, \quad \forall x \in \Re^n, \forall x^* \in X^*$.

(d) For every $x \in \Re^n$ and $x^* \in X^*$ such that $||x - x^*|| = \rho(x) > 0$, there exists some $i \in I(x; x^*)$ such that $F_i(y) \neq y_i$, for all $y \in U(x; x^*)$.

(e) For any $i, x \in \Re^n$, and $x^* \in X^*$, if $F_i(x) \neq x_i$, then $|F_i(x) - x_i^*| < ||x - x^*||$. □

Proposition 2.4 *Let F be a function that satisfies Assumption 2.1. Then all partially asynchronous iterations $(F, x(0), \mathcal{S})$ converge to some element of X^*.*

Sketch of proof. For $t = 0, 1, \ldots$, define

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

$$d(z(t)) = \min_{x^* \in X^*} \{\max\{||x(t - B + 1) - x^*||, \ldots, ||x(t) - x^*||\}\}.$$

From Assumption 2.1(c), we first observe that

$$d(z(t + 1)) \leq d(z(t)), \quad \forall t \geq 0.$$

Let us now fix t. Let $d(z(t)) = \beta > 0$. The first goal is to show that there exists an integer m such that

$$d(z(t + m)) < d(z(t)). \tag{2.5}$$

For this purpose, we observe that if for some $t' \geq t$, and i, $|x_i(t') - x_i^*| < \beta$, then $|x_i(t'') - x_i^*| < \beta$, for all $t'' \geq t'$. This results from the last condition of partial asynchronism, and Assumption 2.1(e).

Consider $2\,n$ consecutive periods of time instances after t, such that the length of each period is B, i.e., $[t + 1, t + B], [t + B + 1, t + 2\,B], \ldots$ Assume that $B - 1$ units after the end of these periods, d value of z is still β, i.e.,

$$z(t) = z(t + 1) = \cdots = z(t + 2\,n\,B + B - 1) = \beta.$$

Otherwise, (2.5) is obtained. From assumptions (c) and (d), it can be concluded that for any two consecutive periods, there exists a component i and an update instance t_i of this component in these cycles, such that the distance from x_i to x_i^* ($|x_i - x_i^*|$) is β before t_i and it is less than β, after t_i. This means that after $2\,n$ periods, the distances from all the components are less than β. Then,

$$d(z(t + 2\,n\,B + B - 1)) < d(z(t)). \tag{2.6}$$

As a result, $\{d(z(t)\}$ converges to a limit d^*. If $d^* = 0$, $\{x(t)\}$ converges to a fixed point and the proof is complete. Assuming $d^* > 0$ will lead to a contradiction as follows. Since $\{z(t)\}$ is bounded, there exists a subsequence of it that converges to some z^* and since d is continuous $d(z^*) = d^*$. Now let $\Delta t = 2\,n\,B + B - 1$. We can express $z(t + \Delta t)$ as a function of $z(t)$:

$$z(t + \Delta t) = g(z(t); \theta(t)),$$

where $\theta(t)$ is some composition of F determined by the part of the schedule between t and $t+\Delta t$. Because of the finite delay assumption, there are only a finite number of $\theta(t)$'s and therefore there exists a subsequence $\{z'(t)\}$ of $\{z(t)\}$ such that $\theta(t)$'s are constant and equal to θ and $\{z'(t)\}$ converges to z^*. This means for another subsequence $\{z''(t)\}$

$$z''(t) = g(z'(t), \theta).$$

Since F is continuous, θ and g is a continuous function of $z'(t)$. Therefore, $\{z''(t)\}$ converges to $g(z^*, \theta)$. From (2.6) $d(g(z^*, \theta)) < d^*$, which is a contradiction. □

2.6.2.2 Gradient–Like Optimization Algorithms

The problem looked at in this section is the minimization of a cost function $G : \Re^n \mapsto \Re$. The iteration operator F for the solution of this problem is $F(x) = x - \gamma\, s(x)$, where $s(\cdot)$ is some function related to $G(\cdot)$. The synchronous iteration corresponding to F converges to the solution if γ is small enough [7]. The following result states that the convergence is preserved under partial asynchronism if γ is smaller than a threshold that depends on B. It was originally obtained by Tsitsiklis [49], [50], [51].

The following conditions are assumed.

Assumption 2.2

(a) There holds $G(x) \geq 0$, for every $x \in \Re^n$.

(b) (Lipschitz Continuity of ∇G) *The function G is continuously differentiable and there exists a constant K_1 such that*

$$||\nabla G(x) - \nabla G(y)|| \leq K_1\, ||x - y||, \quad \forall\, x, y, \in \Re^n.$$

(c) (Descent Property Along Each Coordinate) *For every i and $u \in \Re^n$, we have*

$$s_i(u) \nabla_i G(u) \leq 0.$$

(d) There exists positive constants K_1 and K_2 such that

$$K_2\, |\nabla_i G(u)| \leq |s_i(u)| \leq K_3\, |\nabla_i G(u)|, \quad \forall u \in \Re^n, \quad \forall\, i.$$

□

Proposition 2.5 *Under Assumption 2.2, a partially asynchronous iteration $(F, x(0), \mathcal{S})$ converge if $0 < \gamma < \gamma_0$, where*

$$\gamma_0 = \frac{1}{[(1 + B + n\, B)\, K_1\, K_3]}.$$

□

The complete proof requires the manipulation of the given assumptions and the descent lemma and is involved. We will not give a complete proof, but the key idea is to write the sequence generated by the partially asynchronous schedule as

$$x(t) = x(t-1) + \gamma\, r(t),$$

where

$$r_i(t) = s_i(u^i(t)).$$

The only difference between the synchronous algorithm is that in the synchronous case

$$r_i(t) = s_i(x(t-1))$$

and the difference between $x(t-1)$ and $u^i(t)$ is bounded, because

$$|u_j^i(t) - x_j(t)| = \gamma |\sum_{\tau=\tau_j^i(t)}^{t} r_j(\tau)| \leq \gamma \sum_{\tau=t-B}^{t} |r_j(\tau)|.$$

This suggests that the analysis follows the same steps as the one for the synchronous case. However, partial asynchronism introduces some additional error terms which depend on B and γ. It follows that given B, the error terms do not affect the convergence when γ is small enough.

2.7 Dynamic Iteration Operators

Up to this point we have considered only fixed point iterations for which the iteration operator remains fixed throughout the computations. There are cases, however, in which F changes dynamically as the computation progresses. For example, a common form of synchronous iterations looks like as follows.

```
for k := 1 to m do
    forall 1 ≤ i ≤ n do
        x_i := F_i(x, k)
```

The keyword **forall** indicates that x_i's are computed in parallel, for the current value of k. **For** has the same meaning as in the most declarative languages, and it implies that all the computations for k should be completed before the computations for $k+1$ can start. In other words, there is a barrier between each iteration.

In the above loop, we immediately observe that the iteration operator takes k as a parameter in the k-th iteration. This suggests an asynchronous implementation, for which the iteration operator takes k as a parameter in the k-th pseudo–cycle. For such asynchronous implementations the straightforward generalization of the convergence condition given by Definition 2.2 (asynchronous contraction) can be given as follows.

Condition 2.7 *There exists a sequence of sets* $\{X(k)\}$ *that satisfies the following conditions.*

- *(*Box Condition*) For all* $k = 0, 1, 2, \ldots$, $X(k)$ *is a Cartesian product of* n *sets; i.e.,*

$$X(k) = X_1(k) \times X_2(k) \times \cdots \times X_n(k).$$

- *For all* $k = 0, 1, 2, \ldots$, $X(k+1) \subseteq X(k)$, *and furthermore,* $\{X(k)\}$ *converges to a fixed point of* F ;
- $x \in X(k) \Rightarrow F(x,k) \in X(k+1)$, *for all* $k = 0, 1, 2, \ldots$ □

Here we omit the proof that the condition above is sufficient for convergence of totally asynchronous iterations, in which, for all k, the iteration operator takes k as a parameter, in the k-th pseudo-cycle, because this is almost identical to the proof of Theorem 2.1.

How to implement the asynchronous version of the above synchronous loop still needs some consideration. The main issue in this respect is how to keep track of the pseudo–cycles. Simply maintaining a single, global variable k that contains the current pseudo–cycle at all times is not sufficient, because by using only this information, it is not possible to update k, i.e., to determine when the next pseudo–cycle starts. However, if we maintain a variable P_CYCLE_i which contains the pseudo–cycle of the i-th component, for all i, then from this information it is possible to determine k and also the next pseudo–cycle of the i-th component. In other words, for all i, P_CYCLE_i is the variable such that $x_i \in X_i(P\_CYCLE_i)$, at all times. Therefore, at all times, $x \in X(\min_i\{P\_CYCLE_i\})$, i.e., $k = \min_i\{P\_CYCLE_i\}$. Furthermore, after each update of the i-th component, the new value of P_CYCLE_i can be updated by

$$P\_CYCLE_i := \min_j\{P\_CYCLE_j\} + 1.$$

These ideas amount to the following asynchronous loop.

```
task(i) ;
    begin {task}
        k := min_j{P_CYCLE_j} ;
        x_i := F_i(x, k) ;
        P_CYCLE_i := k + 1;
        if (P_CYCLE_i = m) then terminate else return;
    end {task}
```

In this notation, a task is an indivisible unit of execution that is held by the same processor from its start until the statement **return** or **terminate** is reached. There is a pool of tasks in the system: initially n of them, one for each component. There are also a number of processors, and each available processor selects a task from the pool and executes it until the **return** statement. At this point it returns the task to the pool and makes another task selection, or possibly continues with the same task. The execution of the **terminate** statement in $task(i)$ discards this task from the system for good, because x_i has reached the solution. It should also be mentioned that P_CYCLE_i's are initially assigned to 0. We also assume a computational system that supports atomic accesses (e.g. by hardware) to individual components of P_CYCLE and x. If not, accesses to P_CYCLE and x should be performed in critical sections.

A final remark that has to be made is that the asynchronous implementation described above is not always correct when the underlying schedule is totally asynchronous. As an example, suppose that at some instance of the computations two processors start executing the same task (say i) at exactly the same time. They execute the first two statements in parallel, generating the same update of x_i. Since both processors do the same work on x_i the next pseudo–cycle of the i-th component should be obtained by incrementing k once. However, execution of the third statement by two processors one after another, in our example, increments P_CYCLE_i more than once, which is incorrect. To avoid this type of race condition, the schedule should be non-redundant. This result seems to contradict the above generalization of Theorem 2.1 which states that the totally asynchronous implementation is correct. However, the general result does not consider implementation details and assumes an abstract mechanism that automatically makes the pseudo–cycle information available to the processors, as the computation proceeds, and since this abstraction is not available to the real–life implementation, it needs the additional restriction (non-redundancy) for correct operation.

2.8 Other Models of Asynchronism

2.8.1 Ordered Schedules

Another reasonable restriction we can impose is to assume that the information in the system is received in the order it is generated by the processors. In other words, after a processor uses, as input, the value of component x_i which is generated at some time instance t, it does not use values of x_i older than t. Another way to state this is that τ_j^i's have the monotonicity property as functions of t. In [6], this assumption was made for convergence rate comparison of asynchronous and synchronous iterations. However, we are not aware of any convergence condition which is specific to this model. The formal statement of the model is as follows.

Model 2.6 (Temporally Ordered Schedule) *For all $i, j = 1, 2, \ldots, n$ and $t = 0, 1, 2, \ldots$,*

$$\tau_j^i(t) \leq \tau_j^i(t+1).$$

□

We call this "temporal" ordering, because the ordering of τ_j^i's conform with the ordering of time. In [6], the following result was proven as an intermediate step to other results concerning the convergence rate.

Proposition 2.6 *Suppose that Condition 2.1 (monotonicity) holds and let $\{x(t)\} = (F, x(0), \mathcal{S})$ be an asynchronous iteration, where $\mathcal{S}$ is a horizontally ordered schedule. Then, there holds $x(t+1) \preceq x(t)$, for all t.*

Sketch of Proof. The proof is by induction. Here we omit the basis clause and assume that

$$x(t) \preceq x(t-1) \preceq \ldots \preceq x(1) \preceq x(0).$$

Let us fix a component i. There are three cases for i.

- It is not updated at $t+1$.
- It is updated at $t+1$ and its most recent update before $t+1$ occurs at some $t' > 0$.
- It is updated at $t+1$ and not updated previously.

It is trivial that the first case yields $x_i(t+1) \preceq x_i(t)$. In the second case, from the requirement of the schedule and from the induction hypothesis,

$$u^i(t+1) \preceq u^i(t').$$

From monotonicity,

$$F_i(u^i(t+1)) = x_i(t+1) \preceq F_i(u^i(t')) = x_i(t).$$

The third case can be handled in a similar manner. □

In other cases, the ordering conforms with j. In other words, for the computation of some F_i, the input components are received (or fetched) in some predetermined order. For example, some component x_r is always fetched before the other component x_s. We call this property *vertical ordering of the schedule* and formally state it as follows.

Model 2.7 (Spatially Ordered Schedule) *For all* $i = 1, 2, \ldots, n$, *there exists a permutation function* $\pi_i : \{1, 2, \ldots, n\} \mapsto \{1, 2, \ldots, n\}$ *such that*

$$\tau_r^i(t) \leq \tau_s^i(t)$$

if

$$\pi_i(r) \leq \pi_i(s),$$

for all $t = 0, 1, \ldots$ □

This is called "spatial" ordering, because the ordering of τ_j^i's conform with the ordering of the components. It may be an interesting research problem to derive convergence conditions specific to these models.

2.8.2 Serial Computations

Another mode of operation is *serial* which assumes negligible computation time. In other words, $\tau_j^i(t) = t-1$, for all i, j and t. This has been studied in [40], and in [48] necessary and sufficient conditions for convergence were derived.

2.8.3 Stochastic Models

All the computational models defined above are *deterministic*. This means that they only state some restrictions on the schedule, and any convergence result is expected to guarantee convergence *for every* restricted schedule. In stochastic models [49], [7], the schedule is seen as a random process. In this sense, the analysis of a stochastic model yields a probability of convergence. Convergence with probability 1 does not imply convergence for all possible schedules. For example, for all the asynchronous models we have defined, a synchronous iteration is a special case of the class of asynchronous iterations. Therefore, it is not possible that an asynchronous iteration converges deterministically and its synchronous version does not converge. In probabilistic models, however, it is possible that a synchronous iteration does not converge and its asynchronous version converges.

The disadvantage of stochastic models is that the analysis is difficult, but they may yield sharper convergence results. In this thesis, the emphasis is on deterministic models and we will not discuss stochastic models any further.

Chapter 3

APPLICATIONS

3.1 Constraint Satisfaction Problems

3.1.1 Overview

The *Constraint Satisfaction Problem* (CSP), also known as the *Consistent Labelling Problem* is a generic formulation of a large class of problems, mainly in artificial intelligence. Simply stated, an instance of CSP can be defined by three components: *objects*, *labels*, and *constraints*. The goal is to find an assignment of labels to the objects such that all the constraints are satisfied. Sometimes it is necessary to find *all* solutions instead of one. A tremendous amount of research work has been focused on this problem, motivated by the facts that it is an NP-complete problem and that it is frequently encountered in applications. A significant portion of a recent book on search problems in artificial intelligence is devoted to this topic [35], [17], [13], [33]: it contains an overview of the techniques developed in both past and current approaches. In another recent book [21], a new logic programming language based on constraint satisfaction paradigm is developed. Some real-life problems are expressed and solved using this new language and the approach is related to other recent projects in constraint logic programming, including CLP, Prolog III, and Trilogy.

Examples of the CSP are scene interpretation, graph isomorphism, graph coloring, theorem proving, belief maintenance, computer firmware development, event scheduling, floor-plan design, planning genetic experiments, specifications of software systems, and representation of physical systems [20], [35], [17], [13], [33], [21].

In the scene interpretation (scene labelling) problem, we are given a picture of a scene, such as an office, that has objects which need to be identified, or labelled. The label set is also given which may include "chair," "table," "desk," or "telephone." The relative positions of the objects in the picture and a priori world information define the constraints. For example, if object i is on top of object j in the picture, then object i cannot be labelled with "table" if object j is labelled with "telephone," because the world model dictates that a table cannot sit on top of a telephone.

In the graph isomorphism problem the goal is to find, if any, a one-to-one correspondence between the vertices of two graphs, such that the matched vertices have exactly the same connections. In this case, the objects are the vertices of one graph, and the labels are the vertices of the other one. The constraints are derived from the connections between the vertices. Other applications can be reduced to the CSP problem in a similar manner.

The common algorithm for solving the CSP is the Backtrack Search Algorithm. At each step of the algorithm, an object is selected and an assignment is made that satisfies the constraints, considering the previous assignments. If no assignment can be found, the algorithm backtracks. All the research work on the topic aims at reducing and possibly eliminating backtracking. Usually, before initiating an exhaustive search, the domains of the objects are *filtered*, i.e., the labels that do not satisfy local constraints are eliminated from the domains. The filtering process is also called *discrete relaxation* and can be described by an equation in the form of (1.1) and as shown below, it can be executed asynchronously. In many cases, the filtering process gives the solution without requiring any further search. In other cases, algorithms have been given which adaptively filter the domains in the course of the backtrack search.

3.1.2 Correctness

The CSP can formally be described as follows [43]. Let $A = \{a_1, \ldots, a_n\}$ be the set of objects to be labelled and $\Lambda = \{\lambda_1, \ldots, \lambda_m\}$ be the set of possible labels. Also, let Λ_i, be the set of labels compatible with a_i and Λ_{ij} be the set of label pairs compatible with a_i and a_j ; thus $(\lambda, \lambda') \in \Lambda_{ij}$ means that it is possible to label a_i with label λ and a_j with label λ'. By a *labelling* $L = (L_1, \ldots, L_n)$ of A , we mean an assignment of a set of labels $L_i \subseteq \Lambda_i$ to each $a_i \in A$. The labelling is *consistent* if for all i, j and for all $\lambda \in L_i$, there exists a $\lambda' \in L_j$ that is compatible with λ, i.e. $(\lambda, \lambda') \in \Lambda_{ij}$. We say that a labelling $L = \{L_1, \ldots, L_n\}$ *contains* another labelling L' if $L'_i \subseteq L_i$ for $i = 1, 2, \ldots, n$. The *greatest consistent labelling* L^∞ is a consistent labelling such that any other consistent labelling is contained in L^∞.

According to this model, the discrete relaxation procedure operates as follows. It starts with the initial labelling $L(0) = \{\Lambda_1, \ldots, \Lambda_n\}$. During each step, we eliminate from each L_i all labels λ, such that λ violates the condition of consistent labelling. We shall refer to the operation executed at each iteration as Δ. Therefore, for each iteration t, $L(t+1) = \Delta(L(t))$ and Δ_i is the operator that discards from L_i all labels λ such that for at least one a_j there is no label $\lambda' \in L_j$ which is compatible with λ. It has been shown in [43] that this iterative procedure, implemented synchronously, converges to L^∞, and the following proposition states that it can also be implemented without any synchronization.

Proposition 3.1 *Any totally asynchronous iteration* $(\Delta, L(0), \mathcal{S})$ *converges to* L^∞.

Proof. Define $X = X_1 \times \ldots \times X_n$ such that

$$X_i = \{L_i | L_i^\infty \subseteq L_i \subseteq L_i(0)\}, \qquad i = 1, 2, \ldots, n.$$

Since the set of all labels is finite, X is a finite set, and it is readily shown in [43] that X is closed under Δ, Δ is non-expansive and the first three parts of Condition 2.2 are satisfied, w.r.t. the ordering relation $\subseteq$.

Now, let $L, \bar{L} \in X$ be labelings such that $L \subseteq \bar{L}$ and λ be a label discarded from $\bar{L}_i$ by Δ_i when applied to $\bar{L}$. This implies that for at least one a_j, there is no label $\lambda' \in \bar{L}_j$ which is compatible with λ. Consequently, there is no label $\lambda' \in L_j$ which is compatible with λ, and therefore, if λ is in L_i, it is also discarded from L_i, by Δ_i, when applied to L. Thus, $\Delta(L) \subseteq \Delta(\bar{L})$; i.e., the last part of Condition 2.2 is satisfied, and the claim follows. □

PE	PRIVATE MEMORY	GLOBAL MEMORY
1	$L_1 = \{a{-}\uparrow F_{1a}, b{-}\uparrow F_{1b}, c{-}\uparrow F_{1c}\}$ $\Lambda_{12} = \{ab{-}\uparrow F_{2b}, cb{-}\uparrow F_{2b}\}$ $\Lambda_{13} = \{ac{-}\uparrow F_{3c}\}$	$(F_{1a}, F_{1b}, F_{1c}) = (1, 1, 1)$
2	$L_2 = \{a{-}\uparrow F_{2a}, b{-}\uparrow F_{2b}, c{-}\uparrow F_{2c}\}$ $\Lambda_{21} = \{ba{-}\uparrow F_{1a}, bc{-}\uparrow F_{1c}\}$ $\Lambda_{23} = \{bc{-}\uparrow F_{3c}, ba{-}\uparrow F_{3a}\}$	$(F_{2a}, F_{2b}, F_{2c}) = (1, 1, 1)$
3	$L_3 = \{a{-}\uparrow F_{3a}, b{-}\uparrow F_{3b}, c{-}\uparrow F_{3c}\}$ $\Lambda_{31} = \{ca{-}\uparrow F_{1a}\}$ $\Lambda_{32} = \{cb{-}\uparrow F_{2b}, ab{-}\uparrow F_{2b}\}$	$(F_{3a}, F_{3b}, F_{3c}) = (1, 1, 1)$

Table 3.1: Initial allocation of data

3.1.3 An Implementation

In the following, an asynchronous implementation of the discrete relaxation algorithm on a multiprocessor architecture with distributed global memory is described. This implementation is similar to the one given in [15] with the difference that communication is performed through shared memory instead of message passing and no synchronization is used. In this model architecture, there are N processing elements (PE's) executing asynchronously. Each PE has a CPU, a private memory and a part of the global memory. Private memory can only be accessed by the PE that owns it. The part of the global memory that resides in a PE can be directly accessed by that PE via the local bus. However, any other PE can also access it by using the global bus.

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

Example 3.1

$$\begin{aligned} A &= \{1, 2, 3\} \\ \Lambda_1 &= \Lambda_2 = \Lambda_3 = \Lambda = \{a, b, c\} \\ \Lambda_{12} &= \{ab, cb\} \quad \Lambda_{13} = \{ac\} \quad \Lambda_{23} = \{bc, ba\} \\ \Lambda_{ij} &= \Lambda_{ji} \end{aligned}$$

The initial allocation of this data is given by Table 3.1. In this table, the notation $F_{i\lambda}$ represents the flag associated with the label, λ, in Λ_i. $\uparrow F_{i\lambda}$ is the pointer that points to this flag and $x{-}\uparrow F_{i\lambda}$ means that this pointer is associated with x. □

Processor PE_i does the following computations. For each $\lambda \in L_i$ it scans through Λ_{ij}'s. If there is no pair (λ', λ'') in at least one Λ_{ij}, such that $\lambda' = \lambda$, then λ is discarded from L_i. For $(\lambda', \lambda'') \in \Lambda_{ij}$ such that $\lambda' = \lambda$, PE_i checks if $\lambda'' \in L_j$. If not, i.e. the flag $F_{j\lambda''}$ is reset, then (λ', λ'') is discarded from Λ_{ij}.

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

3.2 Transitive Closure

One of the fundamental building blocks of many algorithms is the following transitive closure problem [1]. Given a finite domain D, the inputs of the algorithm are a set $S \subseteq D$ and a family $\mathcal{F}$ of ordered pairs (P, Q) such that $P, Q \subseteq D$. Each (P, Q) can be interpreted such that the set P implies the set Q. The algorithm returns the closure of S under $\mathcal{F}$, i.e., the largest set implied by S. It has the following structure:

```
until convergence do
    for every (P, Q) in F such that P ⊆ S
        S ← S ∪ Q
```

This algorithm has the same structure as the one given given by (1.1). Each F_i can be viewed as a process that adds the i-th element of D to S. Since F_i's only add elements, but do not delete, $F(S) \supseteq S$, for all S. It is also monotonic, because if F_i adds an element to S for a certain value of S it is guaranteed that F_i also adds that element for a larger S. Therefore, Condition 2.2 applies and the algorithm can be implemented totally asynchronously. Some techniques are used for the sequential implementations to improve efficiency, such as discarding a dependency after it is used to insert elements to the closure. Similar techniques can easily be adopted in the asynchronous implementation.

3.2.1 Functional Dependencies in Relational Database Design

An important problem in relational database design is to find the set of attributes uniquely determined by another set of attributes S, given a set $\mathcal{F}$ of *functional dependencies*. A functional dependency (FD) is denoted by $P \rightarrow Q$ which means that the set P of attributes uniquely determines the set Q of attributes. It is clear that this problem is an instance of the transitive closure problem described above.

3.2.2 Production Systems

A Production System (PS) is composed of: (i) a database, (ii) rules, (iii) control mechanism [10]. We consider the model where each rule R_j is represented by a triple of sets: (C_j, A_j, D_j). Each time a rule completes execution, the control mechanism selects a rule R_j whose condition (C_j) matches the database. The execution of R_j consists of adding the elements in A_j to the database and deleting the elements in D_j from the database. In the special case where no deletion occurs, i.e., D_j's are empty sets, the execution of a production system is an instance of the transitive closure problem and it can be implemented totally asynchronously.

Now suppose that the union of all D_j's does not intersect the union of all C_j's and A_j's. Also in this case, the execution of the PS is similar to the above closure algorithm and can be performed totally asynchronously, because the deletion process is completely independent from the addition process. Unfortunately, this condition is generally not satisfied.

However, a PS is a non-deterministic system, in which the path from the initial condition to the solution is not specified, and the uniqueness, even the existence of the solution is not guaranteed. This suggests that we can add another dimension to the non-determinism inherent in a PS. Suppose that the semantics of a PS allows us to interpret a rule (C_j, A_j, D_j) as equivalent to $|A_j| + |D_j|$ different rules each of which has the same precondition (C_j) and adds or deletes a single element. This interpretation immediately relaxes the

requirement that the addition/deletion activity of a rule has to be atomic. Moreover, the elements of the precondition of a rule, which at any time are found to match the database, can be assumed to match the database forever. Another way to state this is that for any asynchronous parallel execution of a PS, there exists a uniprocessor execution which gives the same result. Suppose that an asynchronous execution adds the elements in the set A and deletes the ones in the set D. We can construct a uniprocessor execution such that the elements in A are added first using the single element rules and then the elements in D are deleted again using single element rules.

3.3 Logic Programming

3.3.1 Logic Programming as Constraint Satisfaction Problem

A logic program is a collection of clauses of the form

$$A \longleftarrow (B_1 \& \cdots \& B_n) \vee (C_1 \& \cdots \& C_m) \vee \cdots$$

where $\longleftarrow$ means "implied by," and $\&$, $\vee$ denote "and," "or" logic operations. [23]. The execution of a logic program is, conceptually, the repeated application of the clauses until convergence. Each activation of a clause effectively constrains the domain in which the left hand side (A) of the clause is true. In this sense, a logic program can be considered as a constraint satisfaction problem. The state of the program X can be represented by the current domain in which the left hand sides are true and the the activation of the i-th clause can be denoted by F_j. We then have a computation which has the same structure as (1.1). F_j's are monotonic, i.e., if X' is more constrained than X'' then $F_j(X')$ is at least as constrained as $F_j(X'')$. F_j's are also non-expansive and we can execute a logic program asynchronously.

3.3.2 Logic Programming With Uncertainties

In [54] an extension of logic programming in which the semantics of a logic program are generalized from true-false assignments to assignments of real values between 0 and 1, which can be viewed as representing some kind of certainty factor. For example, to represent the above clause in this extended form, we need the certainty factors $(k_B, k_C, \ldots)$ of the terms and using this information it can be mapped into the following equation.

$$a = \max\{k_B \cdot \min\{b_1, b_2, \ldots, b_n\}, k_C \cdot \min\{c_1, c_2, \ldots, c_m\}\}$$

where a, b_i, c_i are certainty factors of the logical facts A, B_i, C_i, respectively. Consequently, the execution of a logic program consisting of this type of equations can be considered as a fixed point iteration in the form of (1.1). Here, the iteration operator satisfies the conditions of Condition 2.1 and therefore this type of logic programs can be executed totally asynchronously.

3.4 An All Pairs Shortest Path Algorithm

3.4.1 Synchronous Parallel Floyd's Algorithm

Given a directed graph in which each arc has a nonnegative cost, the problem of all-pairs shortest path is to find for each ordered pair of vertices (i, j) the smallest length of any path from i to j, where the length of a

path is defined as the sum of the costs of the arcs on that path [1]. Floyd's Algorithm is a popular one for the solution of this problem and a parallel version of it is given in Figure 3.1 [14]. In this algorithm, n is the

```
for ROUND := 1 to n do
    forall 1 ≤ i, j ≤ n do
        A[i, j] := min{A[i, j], A[i, ROUND] + A[ROUND, j] }
```

Figure 3.1: Synchronized parallel Floyd's algorithm

number of nodes in the graph[1] and initially $A[i,j]$ is the cost of the arc from i to j for all $i \neq j$; if there is no arc from i to j, $A[i,j]$ is set to ∞. Each diagonal element, $A[i,i]$, is set to 0. The keyword **forall** indicates that the $A[i,j]$'s are computed in parallel. To distinguish the shared writable variables from the other ones we adopt the convention of using uppercase names for such variables ($A, ROUND$). The read/only shared variables (such as n) can be considered as local variables, because each processor can keep a copy of these in its local store and can efficiently access them. In the synchronized Floyd's algorithm, at the end of the k-th iteration (round), for all i,j : (i) $A[i,j]$ is the length of a path from i to j , (ii) $A[i,j]$ is not greater than the length of any path from i to j passing through nodes labelled k or less. Therefore, after the execution of the loop, matrix A contains the lengths of the minimum paths for each ordered pair.

In the synchronized parallel version of Floyd's algorithm the computations of all components of A must be synchronized after each iteration. Therefore, a barrier synchronization is needed after each iteration of the loop. Let P be the number of processors such that $P \leq n^2$. If we ignore the execution of the barrier, each round of the algorithm in Figure 3.1 takes $O(n^2 P^{-1})$ time. However, in [2] it has been shown that the barrier introduces significant overhead and a straightforward implementation of it, that uses a single semaphore, takes $O(P\, t'(P))$ time where $t'(P)$ is the time complexity for executing the semaphore. A faster scheme is also given in [2] which uses $O(P \log P)$ memory and takes $O(t'(P) \log P)$ time. Here, we assume the second implementation without further elaborating on its complexity. As a result, the synchronized parallel Floyd's algorithm takes $O(n\,(n^2 P^{-1} + t'(P) \log P))$ time. When P is $O(n^2)$ then the second term dominates and the complexity becomes $O(n\, t'(n^2) \log n)$.

3.4.2 Simple Asynchronous Floyd's Algorithm

The results of Section 2.7 can be used to map the synchronous Floyd's algorithm to its asynchronous version. In this section, we will not take this approach but prefer to work on the specifics of this problem.

We first observe that the single variable $ROUND$ in the above algorithm has the role of keeping track of the "progress" of the data A towards the solution. After all the components of A are updated once, one unit of progress has been made and the next "round" of computation is entered. In the first asynchronous version of the algorithm which is given in Figure 3.2, on the other hand, each processor only keeps track of the progress of the component it is assigned to. Therefore, instead of only one $ROUND$ number for the whole data, we have a $ROUND$ number for each individual component. The idea is as follows. Suppose that at a certain instance of the computations the component $A[i,j]$ is in the k-th round ($k = ROUND[i,j]$). After

[1] This notation is not consistent with the remaining of the thesis where n denotes the number of components. However, since this does not cause much confusion, only in this section, we follow the conventional notation for the number of nodes.

```
        task(i, j);
            begin {task}
S1:             k_old := ROUND[i, j] ;
S2:             k := k_old + 1 ;
S3:             ki := ROUND[i, k] ; kj := ROUND[k, j] ;
S4:             A[i, j] := min{A[i, j] , A[i, k] + A[k, j]} ;
S5:             if (k_old ≤ ki) and (k_old ≤ kj) then
                    begin
                        ROUND[i, j] := k ;
                        if (k = n) then terminate
                    end ;
                return
            end {task}
```

Figure 3.2: Simple asynchronous Floyd's algorithm

$A[i,j]$ is updated in this round, there is a unit progress in $A[i,j]$ towards the solution if its predecessors are at least in the same round. In this case, $ROUND[i,j]$ can be incremented. By predecessors, we mean the components on which $A[i,j]$ is dependent in the k-th round, i.e., $A[i,j]$, $A[i,k+1]$, $A[k+1,j]$.

The notation for the algorithm specification is the same as the one adopted in Section 2.7. Also, as in Section 2.7, a non-redundant scheduling policy is assumed.

The proof of correctness of the above described algorithm is based on the truth of the following property throughout the execution of the algorithm. Let $ROUND_t[i,j]$ and $A_t[i,j]$ be the values of $ROUND[i,j]$ and $A[i,j]$ at time t.

Property 1 *$A_t[i,j]$ is the length of a path from i to j which is not greater than the length of any path passing through the nodes labelled $ROUND_t[i,j]$ or less.*

Thus, $A_t[i,j]$ is the length of a minimum path when $ROUND_t[i,j] = n$. The correctness immediately follows from the following facts :

(1) At any time instance t during the execution of the algorithm Property 1 holds for all i, j,

(2) $ROUND[i,j]$'s are nondecreasing,

(3) For all i, j and t there exists a finite period of time Δt such that $ROUND_{t+\Delta t}[i,j]$ is greater than $ROUND_t[i,j]$.

We can show (1) by induction. It is obvious that initially Property 1 holds for all i, j (Recall that initially $ROUND[i,j] = 0$ for all i, j). After the initial state, the only events that would affect the truth of (1) are the executions of statements S4 and S5 in Figure 3.2. Let us assume that Property 1 holds just before the execution of S4. It will obviously be satisfied after the execution of S4, as well, because the new value of $A[i,j]$ will not be greater than the old value. Now, let the old value of A be A_o and consider the time instance just before the execution of S4. If $ki \geq k\_old$, then $A_o[i,k]$ is not greater than the length of any path passing through nodes labelled k_old or less. The reason is that by induction hypothesis this is true at the time when ki is fetched and even if $A[i,k]$ might have been changed since then, this change can only decrease $A[i,k]$ without affecting the truth of the above statement. Similarly, if $kj \geq k\_old$, then $A_o[k,j]$ is not greater than

the length of any path passing through nodes labelled k_old or less. Therefore, $A_o[i,k] + A_o[k,j]$ is smaller than or equal to the length of any path passing through k once and not passing through nodes with a label larger than k. Consequently, $\min\{A_o[i,j], A_o[i,k] + A_o[k,j]\}$ is the length of a path which is not greater than the length of any path passing through nodes labelled k or less, and Property 1 holds just after the execution of S5.

We should note here that we have implicitly assumed a property called *sequential consistency* of the computational systems. In other words, the memory system services the memory requests issued by a processor, in the same order as it is implied by the program. Many commercial and prototype systems such as Sequent/Symmetry [37] and IBM/RP3 [8] support sequential consistency, although not all systems enforce it.

(2) is obvious from statement S5 which is the only statement that updates $ROUND[i,j]$. (3) is not as obvious. Let l be the minimum of all the components of $ROUND$ at time t and i', j' be the coordinates of this minimum, i.e., $ROUND[i',j'] = l$. From the non-starvation condition, in a finite period of time $task(i',j')$ will execute its code. During this execution $k\_old = l$ and therefore the condition of S5 is satisfied and $ROUND[i',j']$ is incremented. This means that $\min\{ROUND[i,j]\}$ is increased in a finite period of time and all the components are increased in a finite period of time.

The time complexity of this asynchronous Floyd's algorithm generally depends on the task allocation scheme, which affects the number of executions of $task(x,y)$, although the number of useful executions of $task(x,y)$ in which $ROUND[x,y]$ is incremented is always n. However, in Chapter 4, we will show that under reasonable conditions, the ratio between the useful and the wasted task executions is upper bounded by a constant independent of n and P. Then, each task is executed $O(n)$ times and between two consecutive executions of a certain task, there are $O(n^2\,P^{-1})$ task executions by a single processor where P is the number of processors which is less than n^2. If accessing a shared variable takes $O(t(P))$ time, then a single execution of a task takes $O(t(P))$ time. Also assume an allocation scheme that does not take more than $O(t(P))$ to initiate the next task. Then the time complexity of the algorithm is $O(n^3\,P^{-1}\,t(P))$. In the case when we have sufficient number of processors such that P is $O(n^2)$, the complexity will be $O(n\,t(n^2))$, which is less than the synchronized case. Note that the time complexity of executing a semaphore ($O(t'(P))$) is at least as large as the time complexity of accessing a shared variable ($O(t(P))$).

The asynchronous algorithm explained in this section has also been independently published in [11]. In the next section, we give an extension to it.

3.4.3 Partitioning

The synchronized Floyd's algorithm and the asynchronous Floyd's algorithm in the previous section are the two extreme cases with respect to the size of the $ROUND$ data. As mentioned before, in the synchronized case only one $ROUND$ number is maintained for the whole matrix A and in the simple asynchronous case, there is a $ROUND$ number for each component. Using an additional memory for each component of A may be considered a waste, especially when there are fewer processors available than the number of components making partitioning necessary. Consequently, in the case of partitioning, we can utilize a $ROUND$ number for each partition, instead of each component, and the result is the algorithm in Figure 3.3.

Here, the rows and the columns of matrix A are partitioned and a column partition x together with a row partition y defines a submatrix of A, i.e., a rectangular domain of indices which is denoted by $s[x,y]$ (Figure

```
        task(x, y);
            begin {task}
S1:             k_old := ROUND[x, y] ;
S2:             k := k_old + 1 ;
S3:             zx := Col_Partition(k) ; zy := Row_Partition(k) ;
S4:             kx := ROUND[x, zy] ; ky := ROUND[zx, y] ;
S5:             for (i, j) ∈ s[x, y] do
                    A[i, j] := min{A[i, j], A[i, k] + A[k, j]} ;
S6:             if (k_old ≤ kx) and (k_old ≤ ky) then
                    begin
                        ROUND[x, y] := k ;
                        if (k = n) then terminate
                    end
S7:             return
            end {task} ;
```

Figure 3.3: Asynchronous Floyd's algorithm for partitioned data

3.4). $Task(x, y)$ corresponds to a single update of the whole domain $s[x, y]$ instead of a single component update. The column partition that contains the k-th column is denoted by $Col\_Partition(k)$ and similarly the row partition that contains the k-th row is denoted by $Row\_Partition(k)$. The predecence relationships which existed among the individual components in the previous algorithm are now among the partitions.

The relationship between A and $ROUND$ is stated by Property 2 below, which is analogous to Property 1.

Property 2 *$A_t[i, j]$ is the length of a path from i to j which is not greater than the length of any path passing through the nodes labelled $ROUND_t[x, y]$ or less, for all $(i, j) \in s[x, y]$.*

The argument to show that the elements of $ROUND$ eventually increase in a finite period of time till the solution, is the same as in the previous case; therefore, we will not repeat it here. The proof that Property 2 holds for all i, j and t is also similar to the proof of Property 1. We only have to notice that the only events that could affect the truth of Property 2 are the executions of S5 or S6. As a result, after $task(x, y)$ terminates the value of $ROUND[x, y]$ is n and from Property 2 $A[x, y]$ contains the solution.

As in the previous case, the time complexity of this version can be obtained easily as $O(n^3 p^{-1} t(p))$. Although the complexity is the same, the size of $ROUND$, in this version, is smaller. In addition, this version is faster due to the less frequent accesses to $ROUND$.

3.4.4 Early Updating of $ROUND$

Notice that in the previous algorithm for partitioned data, $ROUND[x, y]$ is fetched at the beginning of $task(x, y)$ and it is updated at the end. This may cause the execution to slow down unnecessarily. To see this, consider the following scenario. Consider the first executions of the tasks $task(x, y)$ and $task(x', y')$. $Task(x, y)$ updates its partition and then updates $ROUND[x, y](= 1)$. After this, it starts the second execution: it fetches $ROUND[x, y](= 1)$ and later $ROUND[x, zy](= 0)$ (at S4). Let $y' = zy$. If $task(x, y')$ is slower such that it updates $ROUND[x, zy]$ in the first execution after $task(x, y)$ fetches it in the second execution, then in the second execution of $task(x, y)$ $ROUND[x, y]$ will not be incremented, because

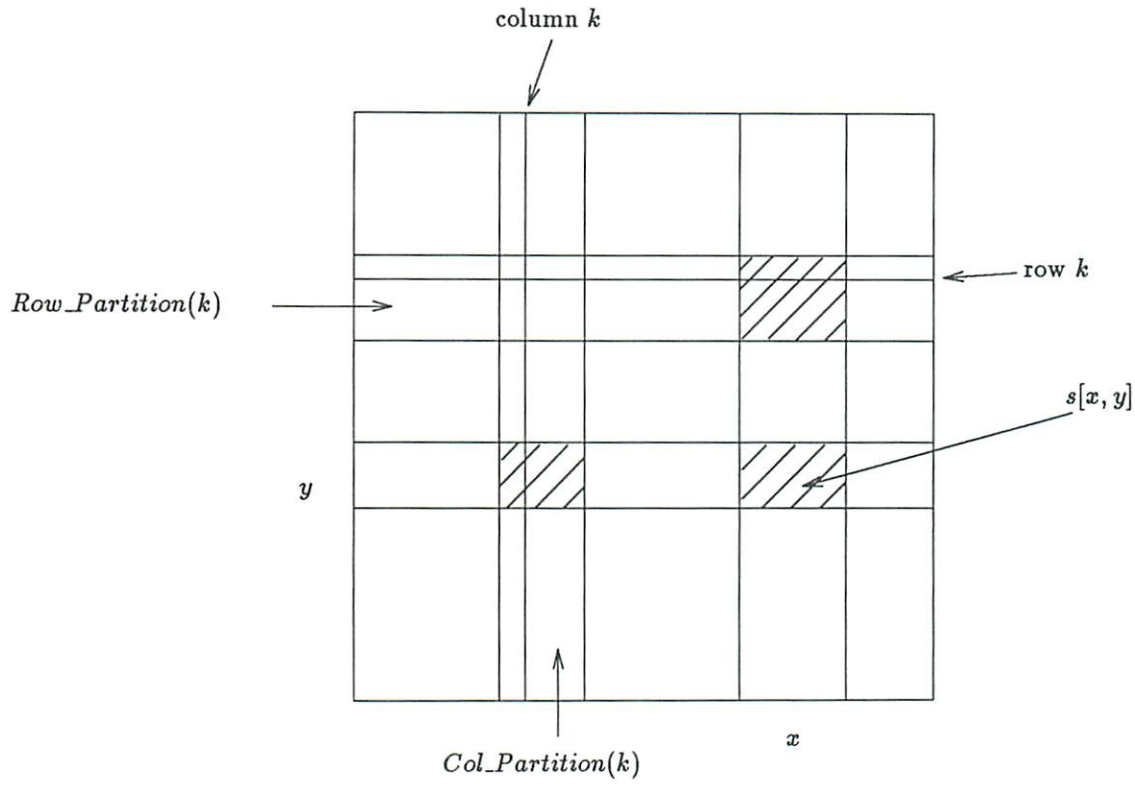


Figure 3.4: Partitioning A

$task(x, y)$ will still see $ROUND[x, zy] = 0$. In general, the worst case is that a task increments its $ROUND$ component in one of the two consecutive executions.

One of the ways to reduce the slowdown due to the above effect is to update $ROUND[x, y]$ as early as possible in $task(x, y)$. This is done in the algorithm in Figure 3.5.

The only difference between this version and the previous one is that $task(x, y)$ updates $ROUND[x, y]$ prematurely: $task(x, y)$ first updates the points in $s[x, y]$ either coordinate of which is $k\_next = ROUND_t[x, y]+2$, then $ROUND[x, y]$ is updated before the remaining components in $s[x, y]$ are processed.

Property 3 *$A_t[i, j]$ is the length of a path which is not greater than the length of any path passing through the nodes labelled $ROUND_t[x, y]$ or less, for all i, j such that*

- $(i, j) \in s[x, y]$, *and*
- $i = ROUND_t[x, y] + 1$ *or* $j = ROUND_t[x, y] + 1$.

Since the only different part of the algorithm is between S7 and S9, the only possible cause of incorrect behavior of the algorithm may be this part. Particularly, the only reason the algorithm would not be correct could be as follows. $task(x, y)$ might update $ROUND[x, y]$ and another $task(\overline{x}, \overline{y})$ might fetch this premature data before $task(x, y)$ finishes the execution of S8, and then it may use this premature data to produce incorrect data. On the other hand, $task(\overline{x}, \overline{y})$ can only fetch $ROUND[x, y]$ at S5. Therefore, after S5 of $task(\overline{x}, \overline{y})$, either $\overline{kx} = ROUND[x, y]$ or $\overline{ky} = ROUND[x, y]$. Without loss of generality, we assume the first case; from symmetry the argument for the second case is the same. $\overline{kx}$ is used only in S7 and

```
        task(x, y);
            begin {task}
S1:             k_old := ROUND[x, y] ;
S2:             k := k_old + 1 ;
S3:             k_next := k + 1 ;
S4:             zx := Partition_x(k) ; zy := Partition_y(k) ;
S5:             kx := ROUND[x, zy] ; ky := ROUND[zx, y] ;
S6:             for ((i, j) ∈ s[x, y] and (i = k_next or j = k_next)) do
                    A[i, j] := min{A[i, j], A[i, k] + A[k, j]} ;
S7:             if (k_old ≤ kx and k_old ≤ ky) then
                    ROUND[x, y] := k ;
S8:             for ((i, j) ∈ s[x, y] and i ≠ k_next and j ≠ k_next do
                    A[i, j] := min{A[i, j], A[i, k] + A[k, j]} ;
S9:             if ROUND[x, y] = n then terminate else return ;
            end {task}
```

Figure 3.5: Early updating of $ROUND$

since the difference between the premature and the old values of $ROUND[x, y]$ is 1, the correctness may be affected only when $\overline{k\_old} = \overline{kx}$, therefore $\overline{k} = ROUND[x, y] + 1$. On the other hand, the only elements of A that $task(\overline{x}, \overline{y})$ uses as inputs are the ones either coordinate of which is $\overline{k} = ROUND[x, y] + 1$. Property 3 states that Property 2 is satisfied for such elements. Therefore, the early update of $ROUND[x, y]$ does not make the algorithm incorrect.

Obviously, the complexity of this version of the asynchronous algorithm is the same as the previous one. However, since the processors make the $ROUND$ data available to other processors earlier than in the previous version, the components of $ROUND$ are incremented more frequently. Therefore, this algorithm executes faster.

3.4.5 Experimental Results

The algorithms presented above are deterministic in the sense that they are guaranteed to give the minimum path. This is enforced by coordinating the shared data components by means of the $ROUND$ data. In practice, we usually need a good solution close to the minimum path, instead of the ultimate minimum. In this case, there is no need for the $ROUND$ data, and we can implement asynchronous algorithms by removing the overhead caused by $ROUND$ from the algorithms presented in this paper. We have done this on a Sequent Symmetry shared memory multiprocessor with 30 processors [37].

Table 3.2 shows the execution times of the asynchronous and synchronous version of the simple Floyd's algorithm in Figure 3.1 for different numbers of processors. The asynchronous version is obtained by simply removing the barriers at the end of each iteration. The initial data are random and the number of nodes in the graph are chosen to be $n = 300$. The execution times are measured in milliseconds. The third column in the table shows the number of node pairs for which the asynchronous version did not give the minimum path. In other words, it is the number of node pairs that the asynchronous and the synchronous versions differ.

In Table 3.3, the execution times of the asynchronous and synchronous implementations of the algorithm in Figure 3.5 without $ROUND$ are displayed.

Processors	Asynchronous	Synchronous	Non-minimum node pairs
1	320,392	320,803	0
2	161,120	161,412	0
3	107,459	108,288	0
4	80,522	81,343	0
5	64,000	66,045	0
6	52,889	54,661	0
7	45,187	46,170	0
8	40,070	40,621	0
9	35,722	35,753	0
10	31,410	31,684	0
11	29,330	29,476	0
12	26,360	26,378	0
13	25,069	25,204	0
14	23,041	23,322	0
15	21,099	21,234	0
16	20,099	20,087	0
17	19,125	19,277	3
18	17,936	18,021	5
19	16,811	17,029	0
20	15,838	16,011	0
21	15,892	15,964	0
22	14,820	14,970	7
23	14,846	14,858	0
24	14,090	14,040	0
25	12,844	12,984	0
26	12,772	12,964	3
27	12,792	12,937	0
28	11,838	11,994	3
29	11,899	12,051	4

Table 3.2: Implementation of simple Floyd's Algorithm

Processors	Asynchronous	Synchronous	Non-minimum node pairs
1	320,599	320,888	0
2	160,608	160,903	0
3	106,924	107,471	0
4	79,947	80,624	0
5	63,683	64,561	0
6	53,044	53,579	0
7	45,324	45,794	0
8	40,037	40,323	0
9	35,767	35,871	0
10	31,597	31,809	0
11	29,440	29,486	0
12	26,428	26,472	0
13	25,231	25,248	0
14	23,224	23,267	0
15	21,163	21,296	0
16	20,151	20,290	0
17	19,242	19,401	0
18	18,019	18,196	0
19	17,015	17,154	0
20	15,969	16,123	0
21	15,885	16,021	0
22	14,933	15,116	0
23	14,843	14,972	0
24	14,065	14,076	0
25	12,984	13,114	0
26	12,949	13,050	0
27	12,876	12,966	0
28	11,852	12,072	0
29	11,974	12,184	0

Table 3.3: Implementation Floyd's Algorithm with early updating

3.5 Other Shortest Path Algorithms

Another shortest path problem is to find the cost of the shortest path from a given node of the graph which is called single-source shortest path problem. In [4] and [7], the totally asynchronous version of the Bellman-Ford algorithm for this problem is shown to be correct. This was accomplished by observing that the iteration operator is monotonic.

A similar iterative algorithm is given below for the all pairs shortest path problem.

$$A_{i,j}(k+1) = \min_p \{A_{i,p}(k) + A_{p,j}(k)\}, \qquad k = 0, 1, 2, \ldots$$

In matrix form the above can be written as

$$A(k+1) = \Gamma(A(k)), \qquad k = 0, 1, 2, \ldots$$

which has the same format as (1.1).

Let us define $X(0)$ as a Cartesian product of $X_{i,j}(0)$'s where $X_{i,j}(0)$ is the union of $\{\infty\}$ and the set of the costs of all paths from i to j. $A \in X(0)$ implies that $A_{i,p}$ and $A_{p,j}$ are the costs of some paths from i to p and p to j, respectively. Consequently, $A_{i,p} + A_{p,j}$, and therefore, $\Gamma_{i,j}(A)$ is the cost of some path from i to j. As a result, $\Gamma(A) \in X(0)$, which satisfies the closure property of Condition 2.1. The other conditions can also be satisfied easily, which means that the iterative algorithm given above can be implemented totally asynchronously.

Chapter 4

CONVERGENCE RATE

4.1 Introduction

Efficiency is 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.

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 [2] 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 [16] and [42].

Our main interest in this chapter is the comparison of asynchronous and synchronous executions of the same problem under similar conditions. In Section 4.2, we review the results of other researchers, which favor asynchronous executions over synchronous ones. In Section 4.3, we give a computational model for which asynchronous iterations are not necessarily superior and can be up to 2 times slower than their synchronous counterparts. Although no architecture is specified in either section, the results of Section 4.2 seem to be more useful for message passing and distributed systems, whereas the results of Section 4.3 are more suitable for shared memory systems. The underlying assumptions of Section 4.2 are that the computation of a component takes unit time and that the system allows the overlapping of computations and communications. As a result, a synchronous algorithm suffers idle times between iterations, due to the communication of the recent data, in contrast to its asynchronous version. In Section 4.3, the penalty associated with the execution of synchronization is ignored. The overhead of the communication and the memory accesses is also ignored, or equivalently, it is assumed that both the asynchronous and the synchronous executions suffer the same overhead. In any case, we make some assumptions on the iteration operator, since no result can be obtained for operators that show arbitrarily irregular behavior.

4.2 Summary of Related Research

In this section, we summarize the results originally given by Bertsekas and Tsitsiklis [6], [7]. The main result is stated by Proposition 4.1 and assumes the monotonicity of F (Condition 2.1). It is also assumed that the

asynchonous schedule is temporally ordered (Model 2.6). Roughly speaking, the following effects are shown to make an asynchronous iteration faster:

- If we increase the parallelism in an asynchronous iteration by allowing more component updates, while maintaining the updates in the original version, the resulting iteration is at least as fast as the original version.
- Consider two asynchronous iterations with the same update instances. The difference between the two is that the second one utilizes more recent information in corresponding updates. Then, the second one is at least as fast as the first one.

The formal statement of the above is as follows.

Proposition 4.1 *Let F be a monotonic operator on a set X satisfying Condition 2.1. Also, $\{x(t)\} = (F, x(0), \mathcal{S})$ and $\{\hat{x}(t)\} = (\hat{F}, x(0), \hat{\mathcal{S}})$ are asynchronous iterations with temporally ordered schedules corresponding to F and starting with $x(0)$. Furthermore, $\alpha(t) \subseteq \hat{\alpha}(t)$ for all t and $\tau_j^i(t) \leq \hat{\tau}_j^i(t)$, for all i, j and t such that $i \in \alpha(t)$. Then, $x^* \preceq \hat{x}(t) \preceq x(t)$, where x^* is the unique fixed point of F in X.*

Sketch of Proof. The proof is by induction. We already have $\hat{x}(0) \preceq x(0)$ (Basis clause). Assume that $\hat{x}(t') \preceq x(t')$, for all $t' < t$. There are three cases for the i-th component:

- $i \notin \hat{\alpha}(t)$. Then, $i \notin \alpha(t)$ and therefore $\hat{x}_i(t) \preceq x_i(t)$.
- $i \in \hat{\alpha}(t)$ and $i \notin \alpha(t)$. Then, $\hat{x}_i(t) \preceq \hat{x}_i(t-1)$ (from Proposition 2.6) and therefore $\hat{x}_i(t) \preceq x_i(t)$.
- $i \in \hat{\alpha}(t)$ and $i \in \alpha(t)$. We have $\hat{x}_j(\hat{\tau}_j^i(t)) \preceq x_j(\tau_j^i(t))$ and therefore, from monotonicity, $\hat{x}_i(t) \preceq x_i(t)$. □

The following conclusions can be drawn, for monotonic iteration operators, from the above result:

1. In a uniprocessor environment, the Gauss-Seidel version of an iterative algorithm executes at least as fast as its Jacobi version. This is a well known classical result and is a consequence of Proposition 4.1 for the following reason. In both executions, the same components are updated at the same time instances ($\alpha(t) = \hat{\alpha}(t)$, for all t), but in the Gauss-Seidel execution, the updates are based on more recent information.
2. In a parallel processing environment with virtually unlimited number of processors, the Jacobi version of an iterative algorithm executes at least as fast as its Gauss-Seidel version, in contrast to the classical result above. This was proven also in [47] and can be reached from Proposition 4.1. We just need to notice that the only difference between the schedules of the two versions is that in the Jacobi version all the components are updated at each time instance while in the Gauss-Seidel version, a subset of components are updated (The sets $\alpha(t)$'s are larger in the Jacobi version). Notice, however, that the efficiency of computations in the Jacobi case may be lower since it uses more processors and the reduction of the execution time may not be proportional to the number of processors used.
3. In many systems, processors work only on their local copies of data and at certain times they exchange updated data with each other, or with some global unit (e.g. shared memory). Computation times

(including accesses to local data) can clearly be distinguished from communication times, and a processor that initiates an exchange of data does not, in general, have to wait for the completion of this transaction. This is true in distributed systems, message passing multiprocessors and sometimes shared memory systems. Furthermore, assume that the computation of a component takes unit time and that we have an unlimited number of processors. Under these conditions, the asynchronous version of an algorithm executes at least as fast as its synchronous one, because the asynchronous version updates all the components in a unit of time. In the synchronous version, after each computation, processors wait idle for the results of the previous computation before they can proceed. Therefore, the sets $\alpha(t)$'s are larger in the asynchronous case. On the other hand, since the communication delays are identical in both cases, corresponding updates use information with the same recency.

4. The above result is still true when a processor in the asynchronous version does not initiate a transfer of data at each time instance, but only when the synchronous version initiates a transfer. The reason is that the above proposition applies when $\tau_j^i(t) \leq \hat{\tau}_j^i(t)$ for all $i \in \alpha(t)$, and not necessarily for all i.

In [7] and [6], similar results have been obtained, which state the superiority of asynchronous iterations, for contracting operators, i.e., for operators F satisfying the following property:

$$\max_i \frac{1}{w_i} ||F_i(x) - x_i^*||_i \leq \gamma \max_i \frac{1}{w_i} ||x_i - x_i^*||_i, \quad \forall x.$$

It is not clear how the results of this section can be applied, when the computation times are not constant. The following section is devoted mainly to this issue.

4.3 Shared Memory Multiprocessors

4.3.1 General Model

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

We consider a multiprocessor system with P processors. The components of the global data x are 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. An asynchronous iterative algorithm (asynchronous iteration) corresponds to an allocation of each task to infinitely many time slots of available processors. Figure 4.1 displays an example. We further make the following assumption.

Assumption 4.1

- *The length of each task interval is finite, and in a finite period of time all the components are updated (total asynchronism).*

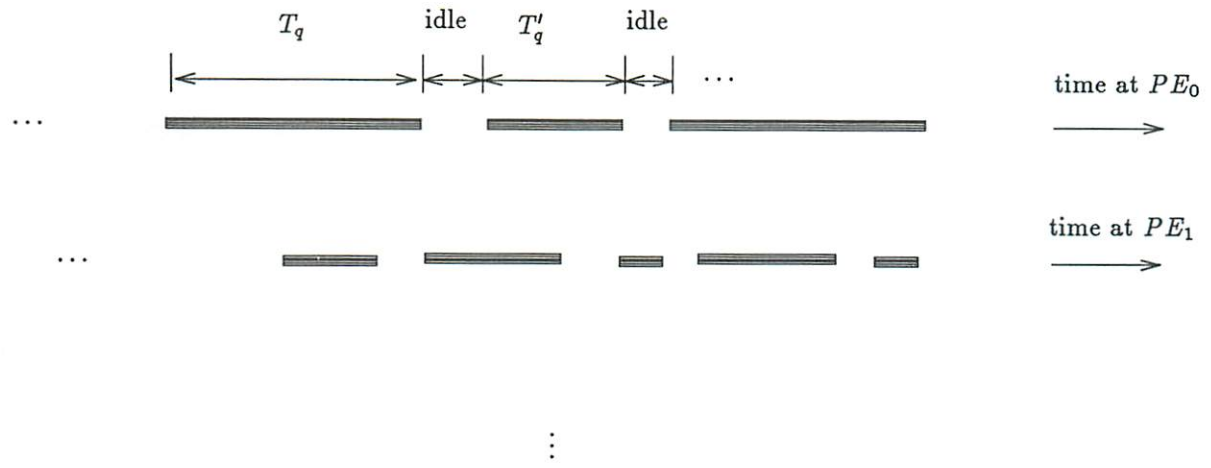


Figure 4.1: An asynchronous iteration

- *The processors never stay idle except at the synchronization points between the iterations. The overhead due to other type of synchronizations such as critical sections are ignored.*
- *The task intervals of the same task never overlap in time (non-redundancy).*

□

Although no particular architecture is specified, this model is more useful for shared memory systems. The major restriction is the non-redundancy condition. Although the results are derived under this condition, they still are good approximations when a small amount of redundancy is allowed.

Let us now further restrict F in the following manner.

Assumption 4.2 *Let $R(k)$ be defined as*

$$R(k) = X(k) - X(k+1),$$

for all k. Then,

$$x \in R(k) \Rightarrow F(x) \in R(k+1),$$

for all k. □

This is the same as Condition 2.3. It simply states that the application of the operator to some data in $R(k)$ moves the data to $X(k+1)$, but not any further, i.e., to $X(k+2)$. Let us define the *age* $A(x)$ of data x as the largest integer that satisfies $x \in X(A(x))$. Then, another way to state this assumption is

$$A(F(x)) = A(x) + 1.$$

The main problem addressed in this chapter is to compare the convergence rate of an asynchronous iteration $\mathcal{A}$ corresponding to F and starting with some $x(0) \in R(0)$ to that of its synchronous version starting with the same initial vector. The synchronous version can be represented by

$$y(k) = F(y(k-1)),$$

for all k, where $y(k)$ is the value of the data at the end of the k-th iteration. Given the above assumption, at the end of the k-th iteration, the iterates enter $R(k)$, i.e.,

$$y(k) \in R(k), \tag{4.1}$$

for all k.

The execution time of an asynchronous and synchronous iteration is the time it takes for the iterates to indefinitely enter a certain domain $X(M)$. From the definition of pseudo–cycle sequence, this period is no more than M pseudo–cycles in the asynchronous case and from (4.1) it is exactly M iterations in the synchronous case. The result is that the ratio between asynchronous and synchronous execution times is upper-bounded by the ratio between average pseudo–cycle time and average iteration time. This is the key result for our later analysis. Notice that if we did not make the above assumption on F this would not always be true, because, otherwise, the synchronous version might "luckily" hit the solution in an arbitrarily small number of iterations.

Let $\{\varphi_{min}(k)\}$ the minimum possible pseudo–cycle sequence corresponding to $\mathcal{A}$. In other words, for all pseudo–cycle sequences and for all k, $\varphi_{min}(k) \leq \varphi(k)$. Also let ϕ_{min} and I denote the average pseudo–cycle time of $\{\varphi_{min}(k)\}$ and the average iteration time of the synchronous version, respectively. The averages are taken over all the pseudo–cycles of one execution. Also we call the the ratio between the asynchronous and synchronous versions *the slowdown factor* and we denote this by S. Then,

$$S \leq \frac{M \cdot \phi_{min}}{M \cdot I}. \tag{4.2}$$

Unfortunately, ϕ_{min} is difficulty to estimate using the assumptions made, so far. We can however identify some other pseudo–cycle sequences which are not always minimum. Such a sequence $\{\varphi'(k)\}$ is defined as follows.

Definition 4.1 $\{\varphi'(k)\}$ is an increasing sequence of the time instances starting with $\varphi'(1) = 0$ 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. □

It should be emphasized that $\{\varphi'(k)\}$ is only a pessimistic estimate of the pseudo–cycle sequence. For a given iteration operator F, the worst we can expect is that in each pseudo–cycle, each task needs the outcome of all the tasks executed in the previous pseudo–cycle, in order to make any progress towards the solution. This is the situation when there exists the strongest possible "coupling" between the tasks. As we show in the next section, for the class of iteration operators with strong coupling, $\{\varphi'(k)\}$ is the exact estimate of the minimum pseudo–cycle sequence if the input data is fetched right at the beginning, and the computed values are released only at the end of each task execution.

4.3.2 Strongly Coupled Computations

4.3.2.1 Description of the Model

We define *strong coupling* by the following two assumptions. The first one restricts the computational model and the second one describes the condition on the iteration operator F.

Assumption 4.3 *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 the ones that are available right at the starting instance of TI_q.* □

For the following, the *age* $A_i(x_i)$ of component x_i is defined as the largest integer such that $x_i \in X_i(A_i(x_i))$. It can be seen that $A(x) = \min_j\{A_j(x_j)\}$.

Assumption 4.4 *For all x and i the iteration operator F satisfies*

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

□

Roughly speaking, a component update depends on all the components of its input, in order to make a progress towards the solution. This corresponds to the worst case, as far as the execution time of the asynchronous iteration is concerned. Usually, the coupling among the components is only partial.

Let ϕ' be the average pseudo–cycle of $\{\varphi'(k)\}$ which is defined by Definition 4.1. The following proposition shows that under the above assumptions, ϕ' is the exact estimate of ϕ_{min}.

Proposition 4.2 $t < \varphi'(k+1) \Rightarrow x(t) \notin X(k)$, *for all* $k > 0$.

Proof. We can prove this by induction. Since there exists a component x_i which is not updated before $\varphi'(2)$, the age of x_i, therefore, the age of x is 0, prior to $\varphi'(2)$. This proves the proposition for $k = 1$. Suppose that it holds for $k = 1, 2, \ldots, l$. Consider the first task interval TI_q on a processor PE_p covered by the l-th pseudo–cycle of $\{\varphi'(k)\}$. Because of Theorem 2.1, right after TI_q, each component i updated in TI_q indefinitely enter $X_i(l+1)$. However, the only other instance within the l-th pseudo–cycle, such a component might be updated is before TI_q (because of non-redundancy). The input for this update is generated earlier than $\varphi'(l)$, therefore it is not in $X(l)$, therefore the updated component x_i at this instance is not in $X_i(l+1)$. This means that x does not enter $X(l+1)$ before all the processors cover at least one task interval in the l-th pseudo–cycle. This proves the proposition for $k = l + 1$ and the claim follows. □

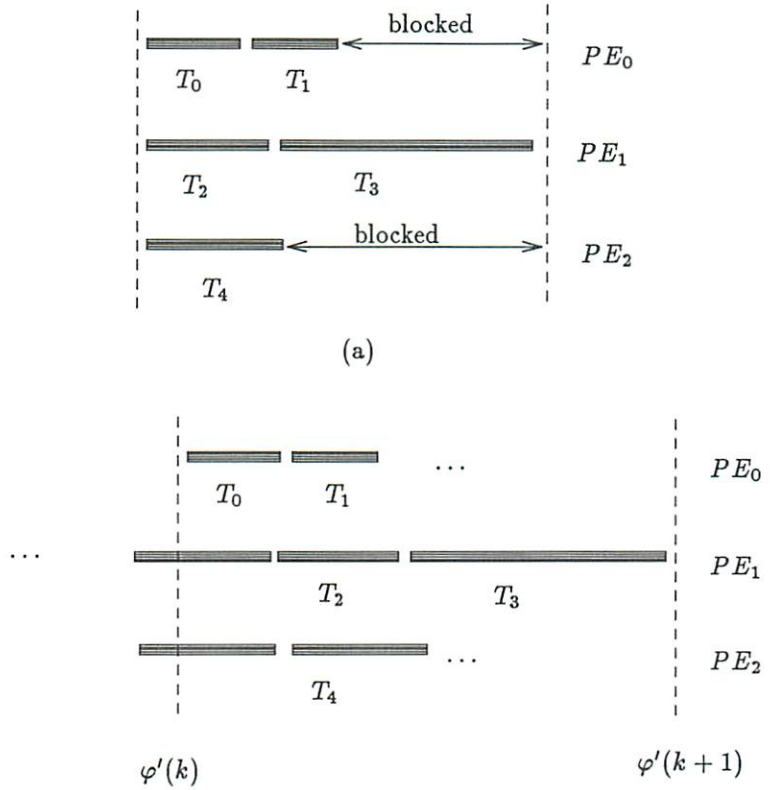
From Theorem 2.1, after $\varphi'(k)$ the iterates indefinitely enter $X(k)$ and the above proposition shows that this is the earliest such instance. Consequently, an asynchronous iteration takes exactly M pseudo–cycles of $\{\varphi'(k)\}$.

As a result, under the assumptions of this section, the slowdown factor S can be written as

$$S = \frac{\phi_{min}}{I} = \frac{\phi'}{I}.$$

Notice that ϕ' and I are nothing else but execution times of asynchronous and synchronous versions, scaled down by a constant factor M. Therefore, we can consider ϕ' and I as *normalized execution times.*

Figure 4.3(a) shows an iteration of a synchronous algorithm for 5 tasks and 3 processors; a pseudo–cycle of the corresponding asynchronous algorithm is demonstrated by Figure 4.3(b). We immediately observe that in the synchronous case all the processors restart right after the previous iteration, while in the asynchronous case, only one processor restarts right after the previous pseudo–cycle (PE_0 in the figure). From Proposition 4.2, the tasks that start before the k-th pseudo–cycle and that complete within the k-th pseudo–cycle do not increment the ages of their components. Therefore, they can be considered as "wasted." Consequently we identify three parts of the total processor time ($\phi(k) \cdot P$) spent in the k-th pseudo–cycle of an asynchronous iteration.



(a) Synchronous iteration
(b) Asynchronous iteration

Figure 4.2: Iteration vs. pseudo–cycle

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 PE_p. Then, $\phi_1(k) \cdot P$ is defined as the sum of all $d_p(k)$'s (In Figure 4.3(b) $d_1(k) + d_2(k)$). Notice that there are up to $(P-1)$ nonzero $d_p(k)$'s.

2. $(\phi_2(k) \cdot P)$. This is the total "useful" work in the pseudo–cycle which covers the first full execution of each task (In Figure 4.3(b) the total processor time covered by T_0 through T_4).

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 from PE_p completes its useful work until the completion of the pseudo–cycle.

In the next sections, $\phi_1, \phi_2, \phi_3, d_p$, and e_p will denote the average quantities where the averages are taken over k's. Our analysis will be based on estimating these parts. For the estimation of the second part, we make the following assumption on the scheduling of tasks to processors which ensures that this part covers exactly Q tasks.

Assumption 4.5 *The scheduling policy satisfies the following requirement. When a processor becomes available at any time instance τ in a pseudo–cycle k, it always selects a task T_q, for the next execution, which has not yet started in the k-th pseudo–cycle, unless all the tasks have been started.* □

This is some kind of a *fairness* condition. Let us discuss this assumption in some more detail. Let τ denote any time instance in the k-th pseudo–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 pseudo–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 pseudo–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 pseudo–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 beginning of the k-th pseudo–cycle until τ is less than $(Q-L)$. This period is upper bounded by the upper bound t_{max} on the task interval lengths, since some tasks working at τ have been started before the k-th pseudo–cycle. Let t_{min} be the lower bound on the task interval lengths. Then, the upper bound on the task initiations in the same period, on $(P-L)$ processors is $\lceil \frac{t_{max}}{t_{min}} (P-L) \rceil$. We need this number to be less than $(Q-L)$. This is satisfied when

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

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

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

Although this condition is sufficient to satisfy the fairness assumption, it is not necessary. The question may now arise as to what happens when the condition above and the assumption are not satisfied. In this case, $\phi_2(k) \cdot P$ may cover more than Q tasks. We believe that this will have little effect on the accuracy of the estimations.

4.3.2.2 Probabilistic Analysis

In this section, we use the Kruskal–Weiss formula [24] for an estimate of ϕ', I and of S. This formula assumes that the task intervals are independent and identically distributed (i.i.d.) random variables with mean μ and variance σ^2. All the tasks are drawn from the same distribution function with increasing failure rate (IFR). A distribution function $G(z)$ 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 $z_0 > 0$ we have

$$\frac{1 - G(z + z_0)}{1 - G(z)} \quad \text{is monotone decreasing in } z.$$

When G has a density g then this is equivalent to

$$\frac{g(z)}{1 - G(z)} \quad \text{is monotone increasing in } z.$$

Intuitively, an IFR random variable shows aging. IFR distributions are quite common, and include the following distributions: Exponential, Gamma with $\frac{\mu}{\sigma} \geq 1$, Weibull, Truncated Normal (i.e., Normal constrained to be positive), Uniform on the interval $(0, A)$ for any $A > 0$, Constant $= A$, for any A. With the above restrictions, the expected time to complete one iteration of a synchronous algorithm can be approximated by

$$E(I) \approx \frac{Q}{P}\mu + \sigma\sqrt{2 \log P}. \tag{4.3}$$

In [24], this formula is derived assuming normal distribution, but it is also claimed that it is a good approximation quite generally.

In order to use this formula to estimate the pseudo–cycle time, we need to make the following assumption.

Assumption 4.6 *The task interval lengths are i.i.d. random variables drawn from the same IFR distribution function with mean μ and variance σ^2.* □

In the following formulae, the coefficient of variation $(\frac{\sigma}{\mu})$ will be denoted by c_v. As in the previous section, the above model guarantees that a pseudo–cycle is the time it takes to complete Q tasks. Let $d_p(k)$ be defined as above, i.e., the time from the beginning of a pseudo–cycle k until the first processor initiation on PE_p. IFR condition assures that d_p is stochastically bounded by the time it takes to complete a whole task [24]. Since there are up to $(P-1)$ $d_p(k)$'s for each k, the upper bound on the overhead at the beginning of a pseudo–cycle can be represented by $(P-1)$ task executions. Therefore, a pseudo–cycle takes no more than $(Q + P - 1)$ task executions. Using the Kruskal–Weiss formula

$$E(\phi') \leq \frac{P-1}{P}\mu + \frac{Q}{P}\mu + \sigma\sqrt{2 \log P}. \tag{4.4}$$

The first, second and the third terms correspond to ϕ_1, ϕ_2 and ϕ_3, respectively. The first term represents the overhead due to the use of outdated information in component updates. The second term is the useful work, and the third term is the result of fluctuations. Combining (4.3) and (4.4) we obtain

$$E(S) \leq 1 + \frac{P-1}{Q + c_v\, P\sqrt{2 \log P}} \leq 1 + \frac{P-1}{Q}. \tag{4.5}$$

The conclusion is that the ultimate upper bound of $1 + \frac{(P-1)}{Q}$ is reached for small σ. On the other hand, the slowdown factor decreases as σ increases. For very large fluctuations, the slowdown factor is close to 1.

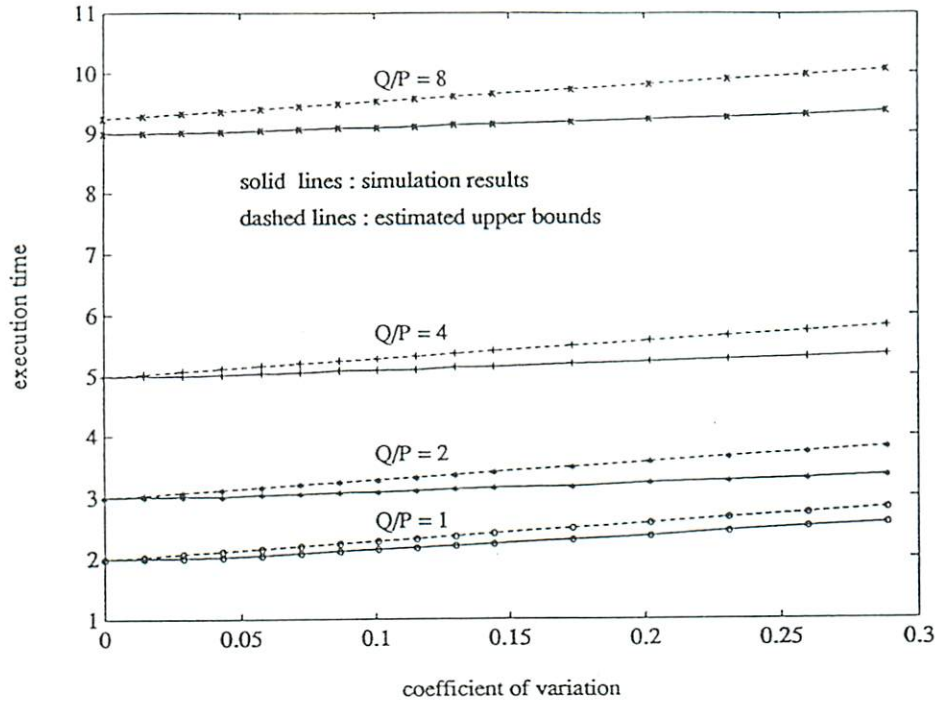


Figure 4.4: Comparison of estimated and simulation results for execution time (Uniform Distribution)

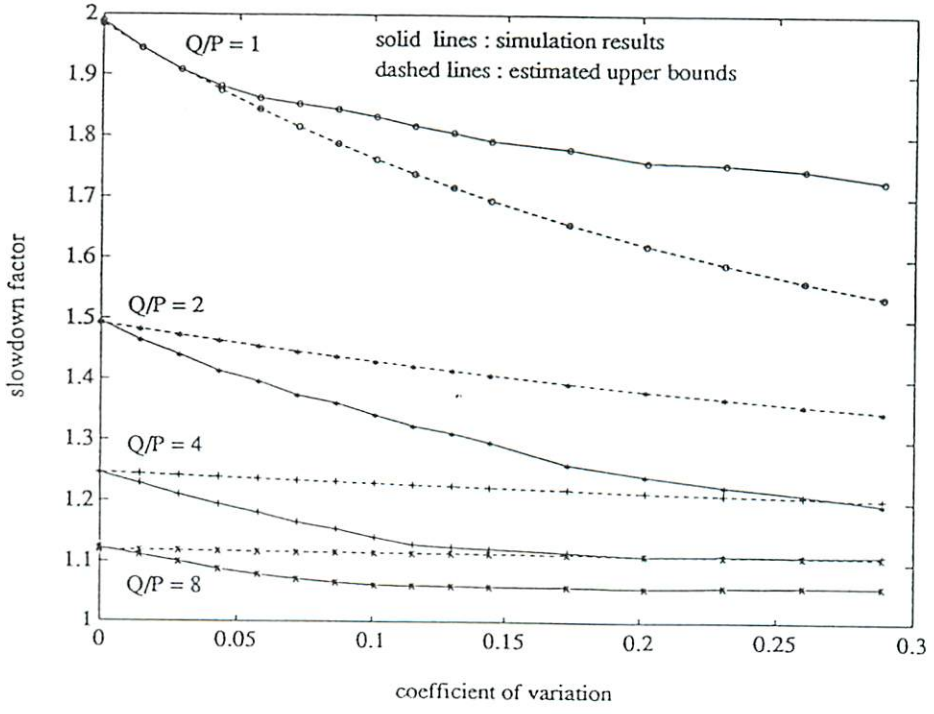


Figure 4.5: Comparison of estimated and simulation results for slowdown (Uniform Distribution)

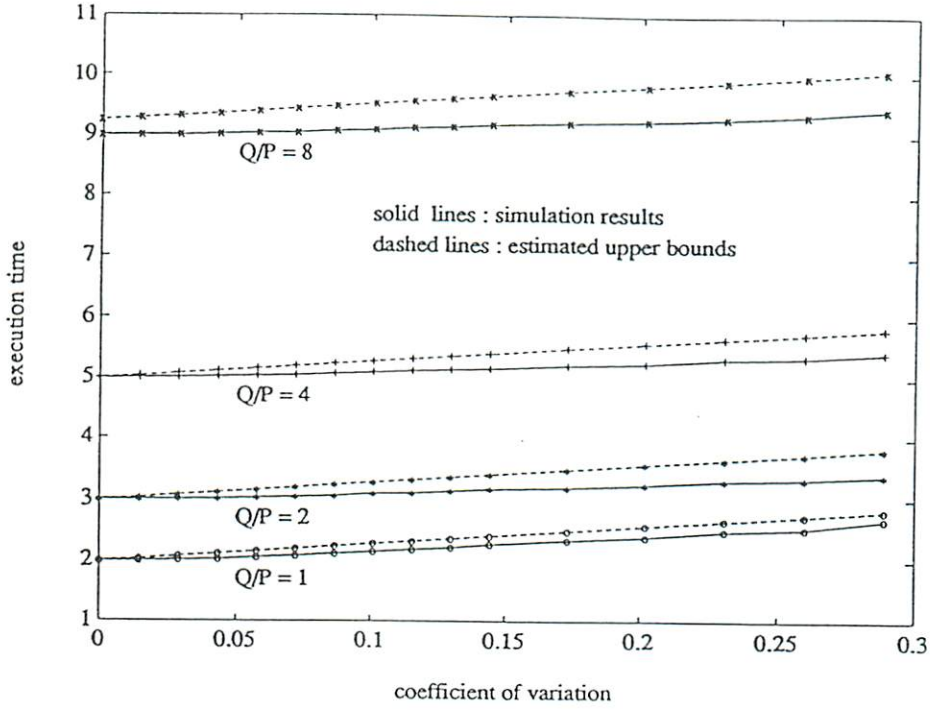


Figure 4.6: Comparison of estimated and simulation results for execution time (Normal Distribution)

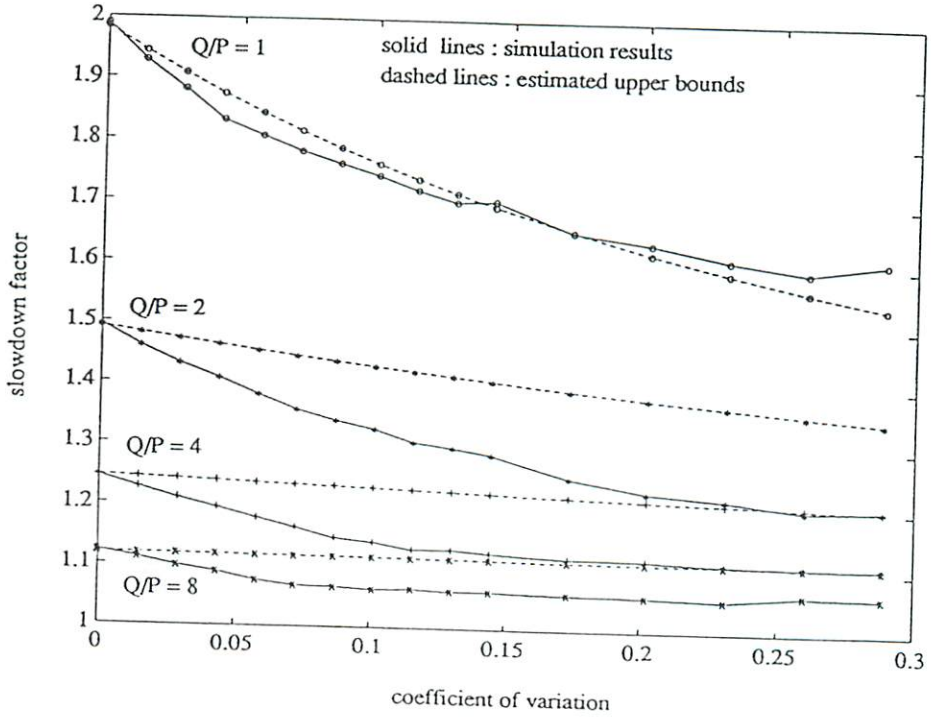


Figure 4.7: Comparison of estimated and simulation results for slowdown (Normal Distribution)

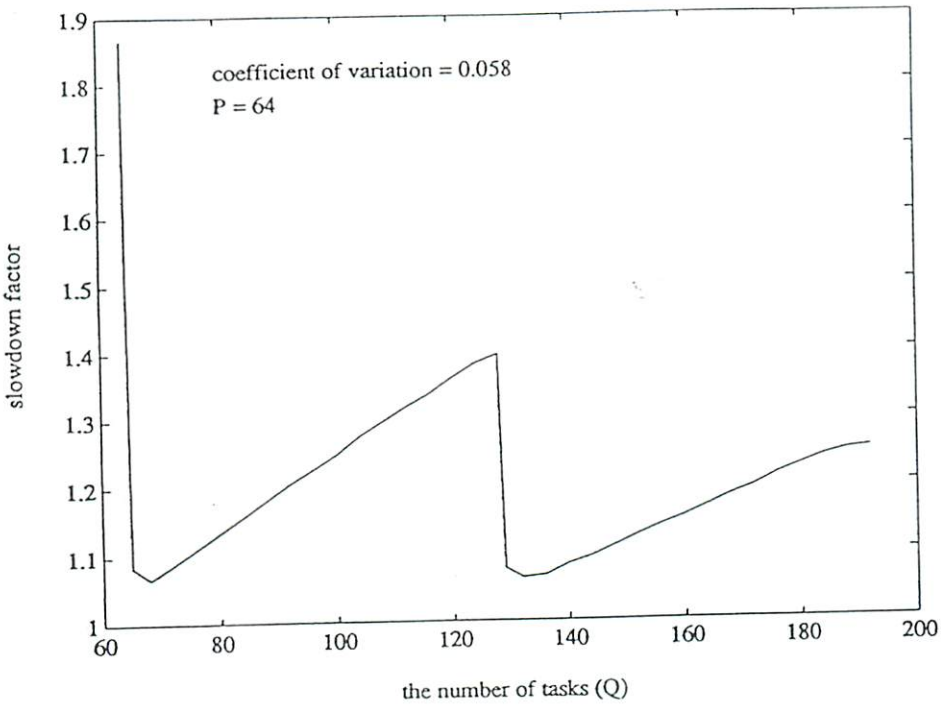


Figure 4.8: Slowdown vs. Q for small fluctuations

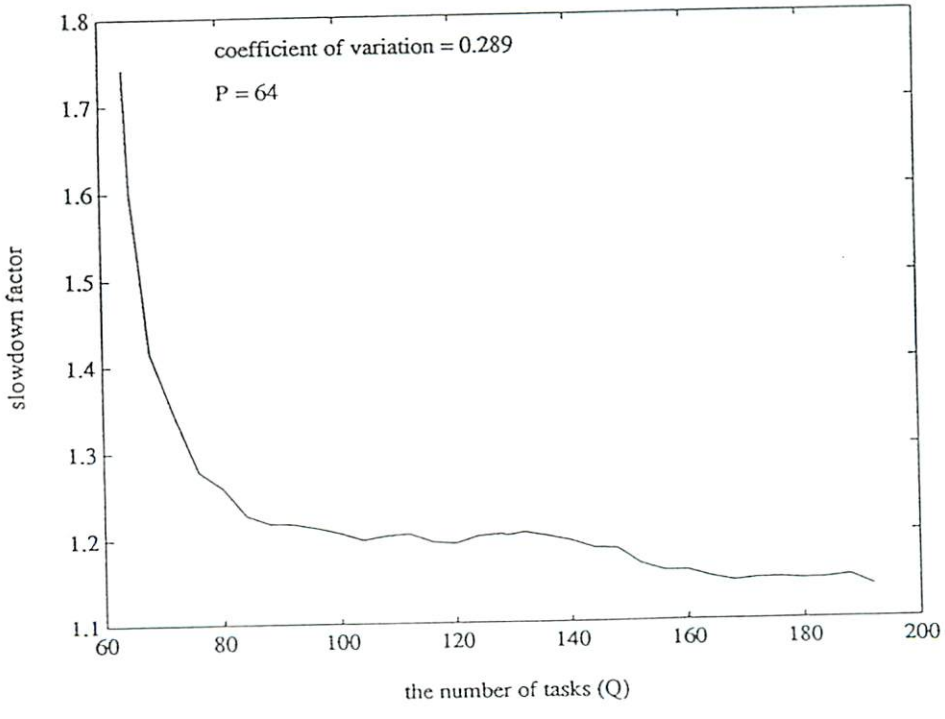


Figure 4.9: Slowdown vs. Q for large fluctuations

We have run some simulations to verify (4.4) and (4.5). Comparative results are displayed in Figures 4.4–4.10. In all of these figures $P = 64$. Also in the figures, the average task execution time is chosen as $\mu = 1$. $E(\phi')$ and $E(I)$ are labeled as the execution time, since they are the normalized total execution times, scaled down by the total number of iterations. In Figures 4.4–4.7 the results are for $Q = 64, 128, 256, 512$. In Figure 4.4, the estimated upper bound on the execution time is compared to the execution time obtained by simulation, for the uniform distribution. It is observed that the simulation results are very close to the right hand side of (4.4). Figure 4.5 displays the slowdown factor, for the same set of data. The simulation results verify the conjecture that the slowdown is upper–bounded by $1 + \frac{P}{Q}$. Figures 4.6 and 4.7 are identical to Figures 4.4 and 4.5, with the only difference that the former show the simulation results for the normal distribution. It is clear that the results for normal and uniform distributions are very close.

Figures 4.8 and 4.9 show what happens when $\frac{Q}{P}$ is not an integer. Integer values of $\frac{Q}{P}$ correspond to good load balancing in which case the advantage of the synchronous algorithm is maximized. This explains the peaks at integer values of $\frac{Q}{P}$. In Figure 4.9, the peaks are diminished, because the task intervals are changing more wildly.

In all the simulations above, the scheduling policy is round–robin, restricted to be non-redundant. In other words, the tasks are selected from a circular queue and if the task selected is currently executing, then the next task is selected. In Figure 4.10 simulation results are displayed for four different scheduling

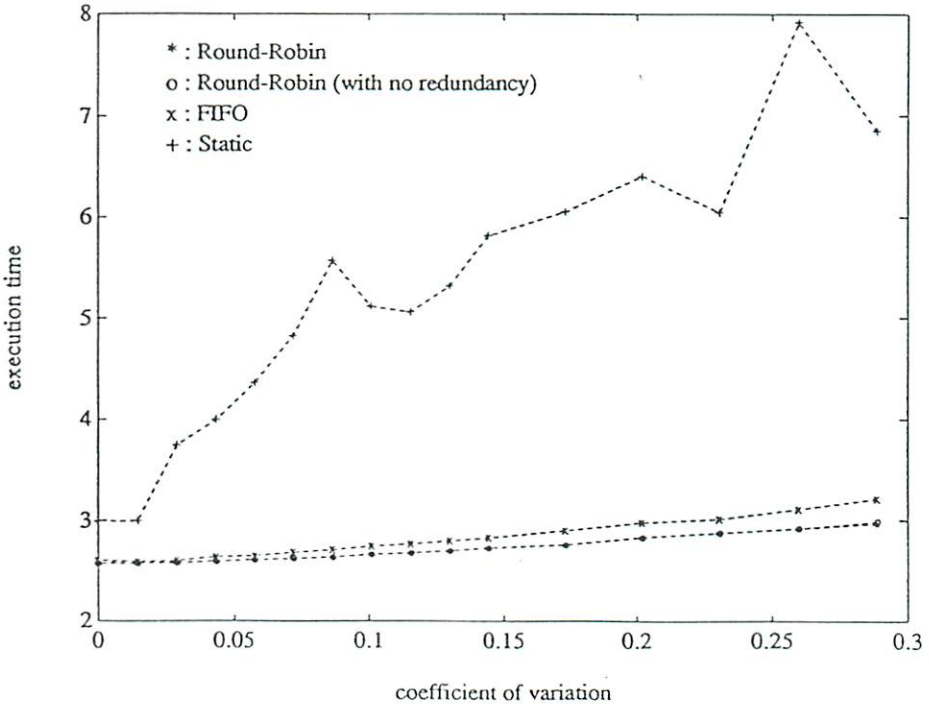


Figure 4.10: Effect of scheduling policy

policies: non-redundant round–robin, round–robin, FIFO and a decentralized scheme. In round–robin, the next task in the circular queue is selected whether or not it is currently executing. We observe no difference between the execution times of the two versions of the round–robin scheme, although we should note that the simulation results may not be reliable because of the redundancy, i.e., because $\{\varphi'(k)\}$ given by Definition 4.1 is the exact estimation of the minimum pseudo–cycle sequence only under non-redundancy. In the FIFO policy, there is a global queue of tasks. When a processor is available, it selects the task from the head

of the queue and it removes this task from the queue. When the task completes its execution it is added to the tail of the queue. It is clear that this policy is guaranteed to be non-redundant. The last policy does not use a global queue. Instead, the next task to be executed is selected locally in a static manner as follows: Processor PE_p starts by executing task T_p. It then executes $T_{(p+P) \bmod Q}, T_{(p+2P) \bmod Q}$, etc. Among all these policies, the performance of the decentralized scheduling is unacceptable, because of the significant amount of redundancy. It seems that the simple round–robin is a reasonable choice, in the case where the dependency information is not available. Obviously, when there is a known set of dependencies among the tasks, the task scheduling should observe these dependencies.

4.3.3 Partially Coupled Iteration Operators

4.3.3.1 General Considerations

The upper bounds obtained so far 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 pseudo–cycle time, where the execution of a task depends critically on the outcome of *all* the tasks in the previous pseudo–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 $S_q(k)$ be the set of tasks that the computation of T_q depends on in the k-th pseudo–cycle, in order to increment its age. Taking this coupling into account, we can slightly modify the definition of the pseudo–cycle sequence, to obtain smaller upper bounds.

Definition 4.2 For all q, $\{\varphi^{(q)}(k)\}$ is defined as a nondecreasing sequence of time instances, starting with $\varphi^{(q)}(1) = 0$, such that the interval $(\beta^{(q)}(k), \varphi^{(q)}(k+1)]$ covers at least one global computation phase of the task T_q, where

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

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 2.1.

If the time distance between $\beta^{(q)}(k)$ and the starting instance of the first execution of T_q after $\beta^{(q)}(k)$ is denoted by $r_q(k)$ and $g_q(k)$ is the duration of this execution, then

$$\varphi^{(q)}(k+1) = \max_{j \in S_q(k)} \{\varphi^{(j)}(k)\} + r^{(q)}(k) + g^{(q)}(k) \tag{4.6}$$

and the total execution time will be

$$T \leq \max_q \{\varphi^{(q)}(M)\}.$$

Example 4.1 Consider the computation in Figure 4.11. 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 τ. Notice that the second task interval of the first component (x_0) does not contribute to the pseudo–cycle from $\varphi'(2)$ to $\varphi'(3)$. This is because the definition of $\varphi'(k)$ assumes the worst case in which F_0 cannot advance the data without the 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.

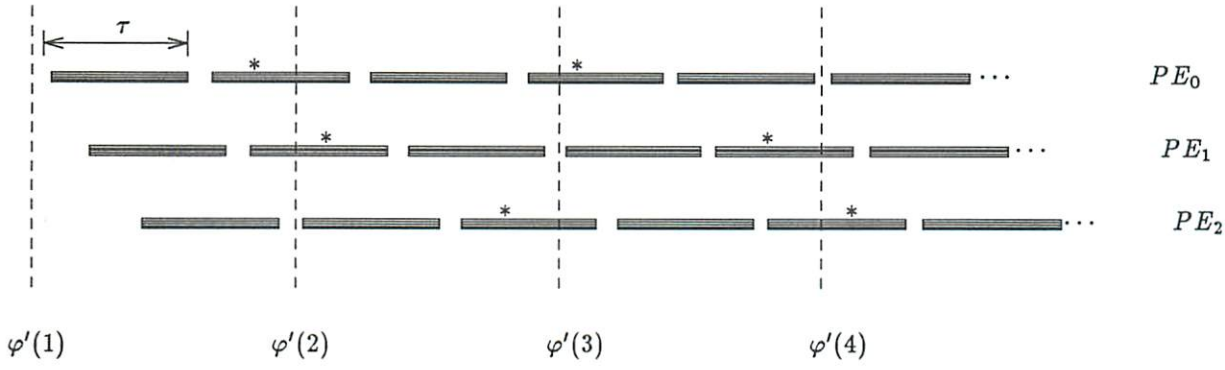


Figure 4.11: An example of partial coupling

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

$$\begin{aligned} x_0 &= F_0(x_0) \\ x_1 &= F_1(x_0, x_1) \\ x_2 &= F_2(x_0, x_1, x_2) \end{aligned}$$

Since F_0 only depends 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, each computation of x_0 makes a progress towards the solution. Similarly, each computation of x_1 and x_2 also makes a progress. Consequently, the average minimum pseudo–cycle time is upper bounded by τ. □

The discussion above suggests that the sequence specified in Definition 4.2 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 $S_q(k)$'s are replaced, in (4.6) 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 $\overline{\varphi}^{(q)}(k) \geq \varphi^{(q)}(k)$, $\varphi'(k)$ cannot be less than $\varphi^{(q)}(k)$. Furthermore, from (4.6), $\varphi^{(q)}(k)$ gets smaller as $S_q(k)$'s get smaller. It can be concluded that when the coupling is small asynchronous implementations perform better than their synchronous versions.

By taking into account the weaker coupling, we can tighten the upper–bound given by (4.4) for Assumption 4.6. It is clear that both of the overhead terms in (4.6) (i.e., the first and the third) decrease with decreasing coupling among components. Let $k_1 < 1$ and $k_3 < 1$ be the positive factors by which the first and the third terms are scaled down. Then,

$$E(\phi') \leq k_1 \frac{P-1}{P} \mu + \frac{Q}{P} \mu + k_3 \sigma \sqrt{2 \log P}. \tag{4.7}$$

k_1 and k_3 are affected by not only the coupling but also by other factors such as scheduling policy, fluctuations, etc. From (4.3) and (4.7) the following upper–bound on the slowdown factor can be obtained.

$$E(S) \leq 1 + \frac{k_1 (P-1) - (1-k_3) c_v P \sqrt{2 \log P}}{Q + c_v P \sqrt{2 \log P}}. \tag{4.8}$$

Therefore, an asynchronous execution is faster than its synchronous version if

$$k_1 < (1-k_3) c_v \sqrt{2 \log P}.$$

4.3.3.2 Self Coupled Iteration Operators

An optimistic approach for the estimation of the execution time of an asynchronous iteration is to assume that each task execution makes a progress. This approach was taken in [9] and [19]. It is equivalent to assuming that each task is coupled only 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 $Q \cdot M$ tasks. Using the Kruskal–Weiss formula

$$M \cdot E(\phi) \approx \frac{M\,Q}{P}\,\mu + \sigma\,\sqrt{2\,\log P}. \tag{4.9}$$

Therefore, the expectation of the slowdown can be written as

$$E(S) \approx 1 - \frac{M-1}{M} \cdot \frac{c_v\,P\sqrt{2\,\log P}}{Q + c_v\,P\sqrt{2\,\log P}} \geq 1 - \frac{c_v\,P\sqrt{2\,\log P}}{Q + P\,c_v\sqrt{2\,\log P}}, \tag{4.10}$$

which is less than 1.

Figure 4.12 displays equations (4.4) and (4.9) for $P = 64$ and $Q = 64, 128, 256, 512$. Similarly, Figure 4.13 compares equations (4.5) and (4.10). In each case, $\mu = 1$.

4.3.3.3 Colorable Iteration Operators

Definition 4.2 does not even rule out the possibility of pseudo–cycles of length zero. We explain this on a popular scheme, called red-black ordering, for the solution to the problem given by Example 4.2.

Example 4.2 In red-black ordering, each component has the color red or black, such that the computation of red (black) components depend only on black (red) components. Consider the computational scheme as shown in Figure 4.14. 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 τ. $\varphi^{(r)}(k)(\varphi^{(b)}(k))$ is equal to $\varphi^{(q)}(k)$, given in Definition 4.2, 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 depend only 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 τ, which means the average pseudo–cycle time is t. According to the definition of $\{\varphi'(k)\}$, however, a pseudo–cycle should cover all the tasks and in this example has a length of $2\,\tau$. □

We can also generalize the example as follows. Let $\mathcal{S} = \{\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_{R-1}\}$ be a partitioning of the set of all the tasks $\mathcal{T}$ such that the tasks in $\mathcal{T}_i$ are only coupled with the tasks in $\mathcal{T}_{(i-1) \bmod R}$, for all $0 \leq i \leq R-1$. Iteration operators for which the components can be partitioned in this way are called *colorable* and for each $\mathcal{T}_C$, the tasks in $\mathcal{T}_C$ are said to have color C. Then it is sufficient that the k-th pseudo–cycle only covers the tasks that has color ($k \mod R$). For such defined pseudo–cycle sequence, in k pseudo–cycles, all the tasks that has color ($k \mod R$) make at least k units of progress.

This result can be used to make the upper–bounds given equations (4.7) and (4.8) even smaller. We just need to notice that the above partitioning of the set of tasks into R subsets has the effect of reducing the number of tasks a pseudo–cycle has to cover. That is, an average pseudo–cycle has to cover $\frac{Q}{R}$ tasks instead of Q. However, the total number of iterations may increase since all the components do not reach the

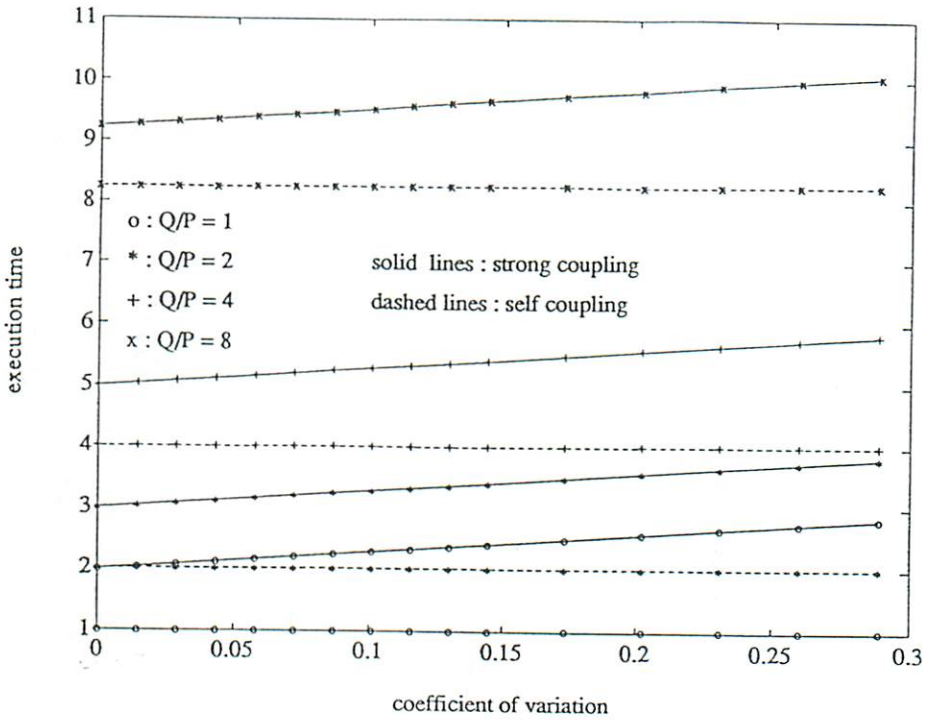


Figure 4.12: Estimated execution time for strong and self coupling

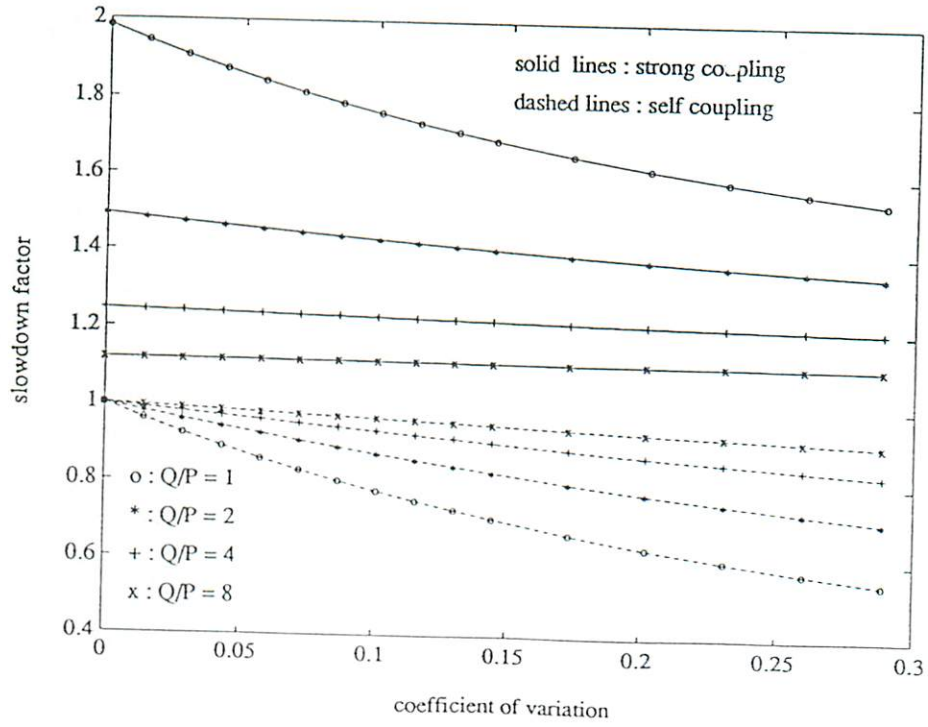


Figure 4.13: Estimated slowdown factor for strong and self coupling

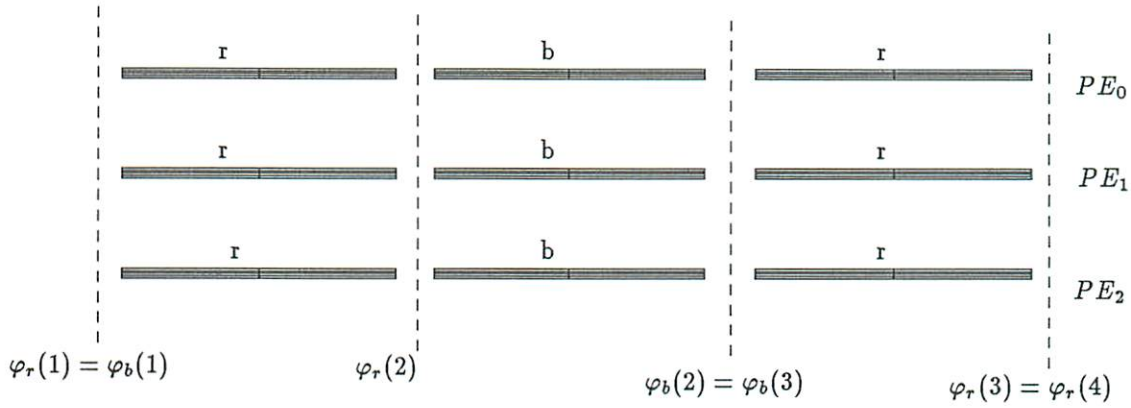


Figure 4.14: Red-black ordering

solution in M iterations (Only the ones updated in the M-th pseudo–cycle do so). Ignoring this overhead, the second term in (4.7) is scaled down by a factor of $k_2 \approx \frac{1}{R}$ in the following way.

$$E(\phi) \leq k_1 \frac{P-1}{P} \mu + k_2 \frac{Q}{P} \mu + k_3 \sigma \sqrt{2 \log P}.$$

Then,

$$E(S) \leq 1 + \frac{k_1 (P-1) - (1-k_2) Q - (1-k_3) c_v P \sqrt{2 \log P}}{Q + c_v P \sqrt{2 \log P}}.$$

$E(S)$ is less than 1 if

$$k_1 < (1-k_2) \frac{Q}{P} + (1-k_3) c_v \sqrt{2 \log P}.$$

In the case where $S_q(k)$'s are constant in the whole data domain and are detectable by simple dependency analysis, it is not necessary to use global barrier synchronization. Furthermore, if $S_q(k)$'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 waits only for the tasks in the set $S_q(k)$ to be completed to start its next task execution. If the tasks can be preempted such that a task T_q starts right after all the tasks in $S_q(k)$ are executed (even though it is busy) then for this synchronization scheme all $r_q(k)$'s in (4.6) are zero. Hence, unless the synchronization cost is too high, this form of synchronization is justified. A more detailed discussion of this idea can be found in [19].

There may still be cases where $S_q(k)$'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 [20]. In such a situation, $S_q(k)$'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.

We can conclude that the elimination of synchronization points improves the performance of asynchronous iterations significantly when $S_q(k)$'s are small. Unfortunately, we do not yet know how to measure and estimate $S_q(k)$'s and therefore parameters k_1, k_2, k_3, exactly. 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.

4.3.4 Heterogeneous Tasks

The results we presented so far assume homogeneous tasks. In this section, we relax this restriction. On the other hand, we assume that the task interval lengths of each task are constant.

Assumption 4.7 *For each q, the task interval length of T_q is constant and equal to t_q. With no loss of generality we assume*

$$t_{max} = t_0 \geq t_1 \geq \cdots \geq t_{Q-1} = t_{min}.$$

□

t_D and t_{av} are defined as follows.

$$(P-1)\,t_D = \sum_{q=0}^{P-2} t_q,$$

$$Q\,t_{av} = \sum_{q=0}^{Q-1} t_q.$$

Then, $\phi_1(k), \phi_2(k), \phi_3(k)$ defined in Section 4.3.2.1 can be estimated as follows.

$$\begin{aligned} \phi_1(k) &\leq (P-1)\,t_D \\ \phi_2(k) &= Q\,t_{av} \\ \phi_3(k) &\leq (P-1)\,t_{max} \end{aligned}$$

Thus,

$$\phi' \leq \frac{Q}{P}\,t_{av} + \frac{P-1}{P}\,(t_D + t_{max}).$$

On the other hand,

$$I \cdot P \geq Q\,t_{av}.$$

Therefore,

$$S \leq 1 + \frac{P-1}{Q}\,\frac{t_D + t_{max}}{t_{av}}.$$

4.3.4.1 Global and Local Computations

The definition of $\{\varphi'(k)\}$ assumes that the computed values in each task are released at the end of the task. This implies that the task intervals which fall on the pseudo-cycle boundaries are wasted. In many cases, however, it is possible to rearrange the computations so that the critical components to 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 wasted. We therefore make the following assumption.

Assumption 4.8 *A task interval has two phases:* the global computation phase *and* the local computation phase. *In the second phase, a task does not interact with other tasks, i.e., it does not access a component shared with other tasks. All the interactions occur in the global computation phase. The global and the local computation phases of task T_q have constant lengths which are denoted by g_q and l_q respectively.* □

Example 4.3 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 computes and updates the boundary points of its corresponding region first (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. Therefore, the following defined sequence $\{\varphi''(k)\}$ has the property that all the components reach the solution no later that $\varphi''(M)$.

Definition 4.3 $\{\varphi''(k)\}$ 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.

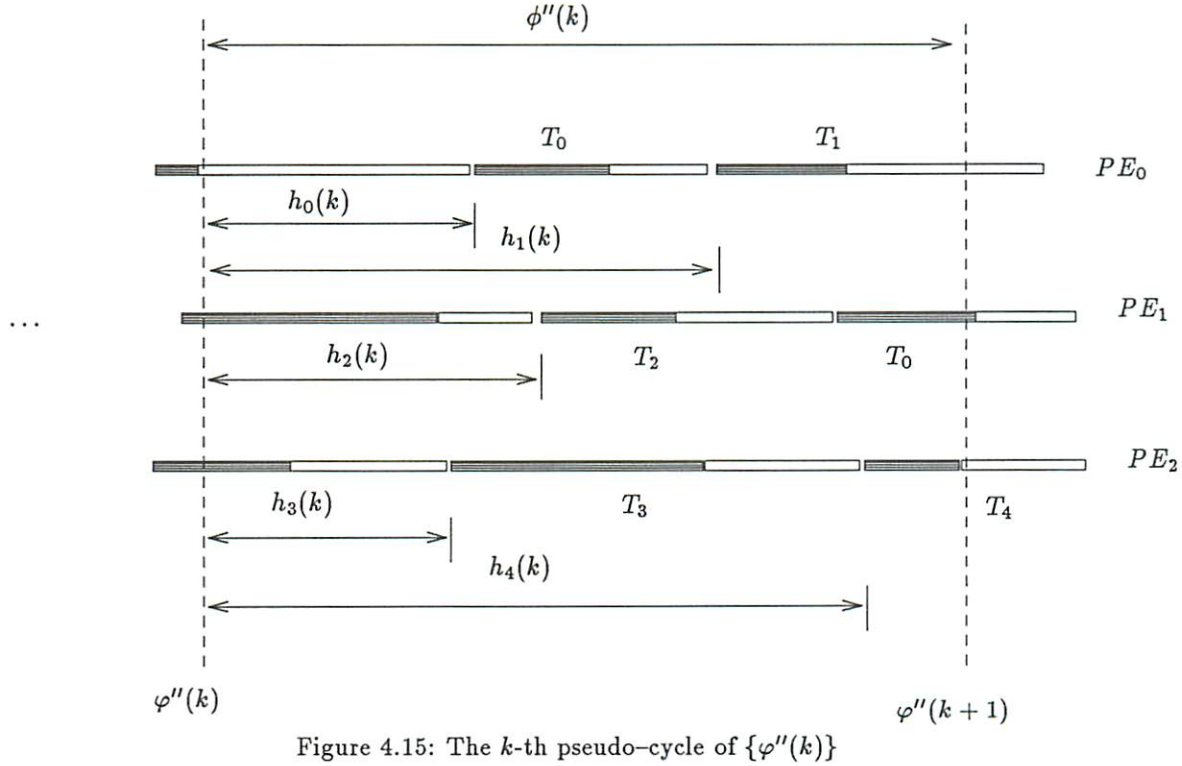


Figure 4.15: The k-th pseudo–cycle of $\{\varphi''(k)\}$

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. (See Figure 4.15 which shows a case for 5 tasks and 3 processors). Then,

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

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

$$\phi''(k) = \max_q \{h_q(k) + g_q\}. \tag{4.11}$$

An important case from which we can obtain meaningful results is the static allocation, where each processor is assigned to a fixed set of tasks. In this case, it is reasonable to assume $P = Q$, because otherwise we can combine all the tasks assigned to a processor, into one. Conversely, for the case $P = Q$, we can assume static allocation. 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 be completed at exactly 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.

4.3.4.2 Static Allocation

The assumption of this section is as follows,

Assumption 4.9 *There are as many processors as tasks* $(P = Q)$. □

An important result on the upper bound for the execution time can be stated for Assumption 4.9. Since $h_q(k)$ varies between 0 and $t_q = g_q + l_q$, from (4.11)

$$\phi''(k) \leq \max_q \{2\, g_q + l_q\}. \tag{4.12}$$

Note that the minimum pseudo–cycle time of the synchronous version is t_{max}. Then,

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

where ϕ'' is the average pseudo–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 4.3 *Under Assumption 4.9, any asynchronous iteration is at most 2 times slower than its synchronous counterpart.* □

From (4.12), the slowdown factor is 1 when for all q, $g_q \ll l_q$. Another related observation is not a consequence of (4.12) 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 pseudo–cycle. These results are summarized below:

Proposition 4.4 *Under Assumption 4.9, 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* $q \neq 0$. □

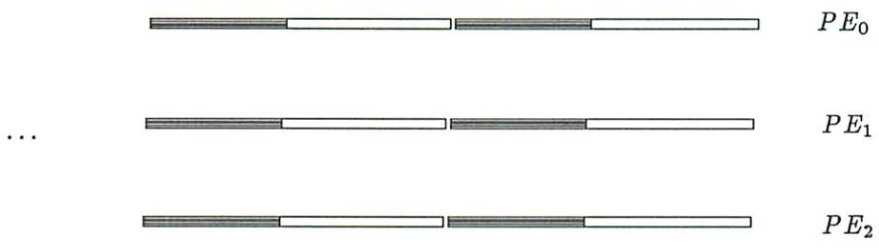


Figure 4.16: The computation is not skewed

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 4.16). 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 transient period, task intervals will be skewed (Figure 4.17). Suppose we neglect the transient period and let the skew distance $h \leq t$ be defined as the time

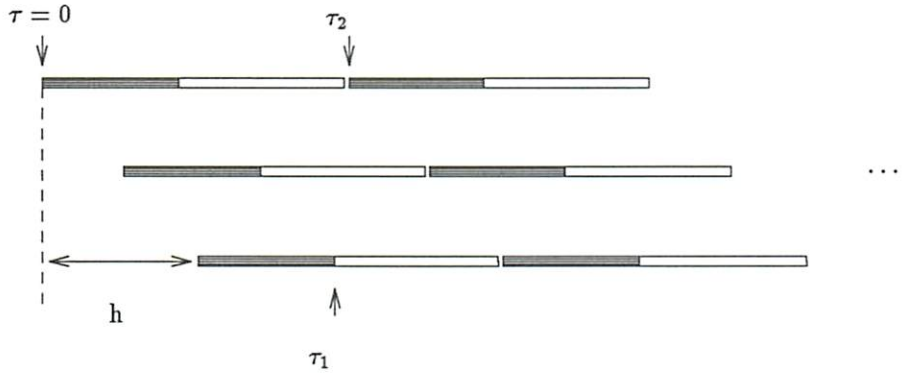


Figure 4.17: The computation is skewed

difference between the initial starting points of the tasks that start the earliest and the latest (Figure 4.17). 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 4.5 *Under Assumption 4.9, 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 Assumption 4.9, but the timing of events can be deviated from Assumption 4.9 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 pseudo–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 4.6 *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 Assumption 4.9 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 4.4 is also sensitive to small fluctuations in the timing.

4.4 Summary of Results

In what follows we will summarize the results of this chapter. All of these results are intuitive, and the contribution of this chapter is to establish their correctness, through solid reasoning or through simulations. It should also be emphasized that we have made some assumptions on the iteration operator F, since it is not possible to obtain results on the convergence rate of either asynchronous or synchronous executions, for arbitrary iteration operators. Furthermore, along the way, we have specified some restrictions on the computational models, but the assumptions do not seem to be stringent in most reasonable cases.

1. The time complexity of an asynchronous iterative algorithm cannot be worse than the complexity of its synchronous version. Under worst case conditions, i.e., when the components depend strongly on each other, and also when the overhead of synchronization can be neglected, the asynchronous algorithm can be up to $1 + \frac{P}{Q}$ times slower than its synchronous version, which means that the ultimate upper bound for the slowdown factor is 2. Obviously, when the synchronization overhead is high, asynchronous implementations are favorable.

2. Given the number of processors P and the total workload per iteration, an asynchronous iteration can be made faster by increasing the number of tasks (Q), i.e., by breaking down the total workload into more tasks.

3. An asynchronous iteration executes faster when it delays the fetching of input data for component updates and by releasing the updated data earlier. Consequently, 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 will only be used locally.

4. An asynchronous iteration converges faster by increasing parallelism, i.e., by allowing more updates while maintaining the original updates.

5. If the dependency among the tasks is weak and a priori known, then the scheduling strategy that controls the allocation of tasks to processors should observe this dependency. Otherwise, any scheduling policy that gives equal chance to tasks should be adopted. The simple round–robin scheme seems to be a good choice.

6. Again, when the dependency 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.

Chapter 5

CONCLUSION

In recent years, we have witnessed significant advances in parallel processing technology. Today, multiprocessor systems are very popular and support parallel processing. Given this trend towards more parallelism, it is not unrealistic to anticipate systems, in the near future, with large numbers of processors, for example a few hundreds, may be thousands. On the other hand, research in parallel algorithms and programming, in the past, concentrated either on "simulating" known sequential algorithms on parallel systems, or on designing new parallel algorithms, exploiting the parallel nature of the applications, but with a synchronous computational model in mind. Consequently, multiprocessor implementations of such algorithms involve extensive synchronizations. While the value of such research cannot be denied, it is also important to understand and exploit the additional dimension of parallelism introduced by multiprocessors: asynchronism. This is even more critical for multiprocessing systems with large numbers of processors, since it is more difficult to support synchronization in such systems.

In this thesis, we have explored an aspect of asynchronous computing which does not appear in sequential or synchronous parallel computing, that is, the correctness of a computation even when the sequence of interactions among the participating processors shows a chaotic behavior. For this purpose, we have defined models of asynchronous computations, and have derived convergence results for these models; we have also studied some issues related to performance. The idea of asynchronous iterative algorithms with chaotic processor interactions is not new. Previous work emphasized numerical computing; the primary goal of this research was to generate more general results applicable also to symbolic computing. We have concentrated on underlying fundamental issues rather than on application driven approaches.

One of the major criticisms of asynchronous algorithms is that they require mathematical knowledge too sophisticated for average algorithm designers. This thesis demonstrates that significant results can be obtained by relying on very little mathematical knowledge and that the fundamental concepts are in fact very simple. Another criticism has been that the performance of asynchronous algorithms is unpredictable. We have presented many results on performance; it appears that their performance is no more unpredictable than their synchronized counterparts.

In the future, we may expect many applications of asynchronous algorithms in various areas, especially in artificial intelligence. This would be facilitated if asynchronism was integrated in the semantics of languages. Several parallel languages have been designed by integrating parallelism in existing sequential languages. A similar approach should be taken for asynchronism. It seems natural for example to extend production system and logic programming languages to allow asynchronous computations which are not derived from

uniprocessor computations. Another possible area of future research is the design of architectures for asynchronous computations. In order to achieve maximum performance from asynchronous algorithms, systems must be designed with fully asynchronous data accesses and transfers. Overlapping of data transfer and computation will become more critical as the gap between the speed of interprocessor communication and the speed of processors increases.

Reference List

[1] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

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

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

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

[5] Bertsekas, D. P. Distributed dynamic programming. *IEEE Trans. on Automatic Control 27* (1982), 610–616.

[6] Bertsekas, D. P., and Tsitsiklis, J. N. Convergence rate and termination of asynchronous iterative algorithms. In *Proceedings, International Conference on Supercomputing* (Crete, Greece, June 1989).

[7] Bertsekas, D. P., and Tsitsiklis, J. N. *Parallel and Distributed Computation.* Prentice Hall, 1989.

[8] Brantley, W. C., McAuliffe, K. P., and Weiss, J. RP3 Processor-Memory element. In *ICPP* (1985).

[9] Brochard, L., Prost, J., and Fauire, F. Synchronization and load unbalance effects of parallel iterative algorithms. In *ICPP* (1989), pp. 153–160.

[10] Brownston, L., et al. *Programming Expert Systems in OPS5.* Addison-Wesley, Reading, Massachusetts, 1985.

[11] Chandy, K. M., and Misra, J. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

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

[13] Dechter, R., and Pearl, J. Network-based heuristics for constraint satisfaction problems. In *Search in Artificial Intelligence*, L. Kanal and V. Kumar, Eds., Springer-Verlag, New York, 1988.

[14] Deo, N., Pang, C., and Lord, R. Two parallel algorithms for shortest path problems. In *IEEE International Conference on Parallel Processing* (1980), pp. 244–253.

[15] Dixit, V., and Moldovan, D. I. *Discrete Relaxation on SNAP.* Tech. Rep., Dept. of EE-Systems, Univ. of Southern California, Los Angeles, 1984.

[16] 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.

[17] Freuder, E. C. Backtrack-free and backtrack-bounded search. In *Search in Artificial Intelligence*, L. Kanal and V. Kumar, Eds., Springer-Verlag, New York, 1988.

[18] GOODMAN, J. R., VERNON, M. K., AND WOEST, P. J. *Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors.* Tech. Rep. TR-814, Univ. of Wisconsin at Madison, 1989.

[19] GREENBAUM, A. Synchronization costs on multiprocessors. *Parallel Computing 10* (1989), 3–14.

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

[21] HENTENRYCK, P. V. *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.

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

[23] KOWALSKI, R. *Logic for Problem Solving.* North-Holland, Amsterdam, 1979.

[24] KRUSKAL, C. P., AND WEISS, A. Allocating independent subtasks on parallel processors. In *ICPP* (1984), pp. 236–240.

[25] 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.

[26] LIPPMANN, R. P. Introduction to computing with neural nets. *IEEE ASSP Magazine* (April 1987), 4–22.

[27] LUBACHEVSKY, B. D., AND MITRA, D. A chaotic asynchronous algorithm for computing the fixed point of a nonnegative matrix of unit spectral radius. *J. ACM 33*, 1 (January 1986), 130–150.

[28] MIELLOU, J. Algorithmes de relaxation à retards. *Revue d'Automatique, Informatique et Recherche Opérationnelle 9*, R-1 (April 1975), 55–82.

[29] MIELLOU, J. Asynchronous iterations and order intervals. In *Parallel Algorithms & Architectures*, M. C. et al., Ed., North-Holland, 1986.

[30] MIELLOU, J. Itérations chaotiques à retards: études de la convergence dans le cas d'espaces partiellement ordonnés. *CRAS*, 278 (1974), 957–960.

[31] MIELLOU, J., AND SPITERI, M. Un critère de convergence pour des méthodes générales de point fixe. *Math. Modelling & Numer. Analy. 19* (1985), 645–69.

[32] MITRA, D. Asynchronous relaxations for the numerical solution of differential equations by parallel processors. *SIAM J. Sci. Stat. Comput. 8* (1987), 43–58.

[33] MONTANARI, U., AND ROSSI, F. Fundamental properties of networks of constraints: a new formulation. In *Search in Artificial Intelligence*, L. Kanal and V. Kumar, Eds., Springer-Verlag, New York, 1988.

[34] MUNKRES, J. R. *Topology: A First Course.* Prentice Hall, Englewood Cliffs, NJ, 1975.

[35] NADEL, B. A. Tree search and arc consistency in constraint satisfaction algorithms. In *Search in Artificial Intelligence*, L. Kanal and V. Kumar, Eds., Springer-Verlag, New York, 1988.

[36] ORTEGA, J. M., AND RHEINBOLDT, W. C. *Iterative Solution of Nonlinear Equations in Several Variables.* Academic Press, 1970.

[37] OSTERHAUG, A. *Guide to Parallel Programming on Sequent Computer Systems.* Sequent Computer Systems, Inc., 1987.

[38] ROBERT, F. Contraction en norme vectorielle: convergence d'itérations chaotiques pour des équations non linéaires de point fixe à plusieurs variables. *Lin. Algeb. & Appl. 13* (1976), 19–36.

[39] ROBERT, F. *Discrete Iterations.* Springer-Verlag, Berlin, 1986.

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

[41] ROBERT, F., CHARNAY, M., AND MUSY, F. Iterations chaotiques serie-paralleles pour des équations non-lineaires de point fixe. *Aplikace Mat. 20* (1975), 1–38.

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

[43] ROSENFELD, A., HUMMEL, R. A., AND ZUCKER, S. W. Scene labeling by relaxation operations. *IEEE Trans. Systems, Man, and Cybernetics 6*, 6 (June 1976), 420–433.

[44] SPITERI, P. Parallel asynchronous algorithms for solving boundary value problems. In *Parallel Algorithms & Architectures*, M. C. et al., Ed., North-Holland, 1986.

[45] STONE, H. S. *High-performance computer architecture.* Addison-Wesley, 1987.

[46] TSENG, P., BERTSEKAS, D. P., AND TSITSIKLIS, J. N. Partially asynchronous, parallel algorithms for network flow and other problems. *SIAM J. Cont. and Opt. 28*, 3 (May 1990), 678–710.

[47] TSITSIKLIS, J. N. A comparison of Jacobi and Gauss-Seidel parallel iterations. *Applied Mathematics Letters.* To appear.

[48] TSITSIKLIS, J. N. On the stability of asynchronous iterative processes. *Mathematical Systems Theory 20* (1987), 137–153.

[49] TSITSIKLIS, J. N. *Problems in Decentralized Decision Making and Computation.* PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, 1984.

[50] TSITSIKLIS, J. N., AND BERTSEKAS, D. P. Distributed asynchronous optimal routing in data networks. *IEEE Trans. on Automatic Control 31* (1986), 325–332.

[51] TSITSIKLIS, J. N., BERTSEKAS, D. P., AND ATHANS, M. Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *IEEE Trans. on Automatic Control 31* (1986), 803–812.

[52] ÜRESIN, A., AND DUBOIS, M. *Parallel asynchronous algorithms for discrete data.* Tech. Rep. CENG 89-15, Computer Engineering Division, Department of Electrical Engineering-Systems, University of Southern California, 1989. Also to appear in *J. ACM.*

[53] ÜRESIN, A., AND DUBOIS, M. Sufficient conditions for the convergence of asynchronous iterations. *Parallel Computing 10* (1989), 83–92.

[54] VANEMDEN, M. H. Quantitative deduction and its fixpoint theory. *Journal of Logic Programming 3*, 1 (April 1986), 37–53.

4.3.2.2 Probabilistic Analysis

In this section, we use the Kruskal–Weiss formula [24] for an estimate of ϕ', I and of S. This formula assumes that the task intervals are independent and identically distributed (i.i.d.) random variables with mean μ and variance σ^2. All the tasks are drawn from the same distribution function with increasing failure rate (IFR). A distribution function $G(z)$ 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 $z_0 > 0$ we have

$$\frac{1 - G(z + z_0)}{1 - G(z)} \quad \text{is monotone decreasing in } z.$$

When G has a density g then this is equivalent to

$$\frac{g(z)}{1 - G(z)} \quad \text{is monotone increasing in } z.$$

Intuitively, an IFR random variable shows aging. IFR distributions are quite common, and include the following distributions: Exponential, Gamma with $\frac{\mu}{\sigma} \geq 1$, Weibull, Truncated Normal (i.e., Normal constrained to be positive), Uniform on the interval $(0, A)$ for any $A > 0$, Constant $= A$, for any A. With the above restrictions, the expected time to complete one iteration of a synchronous algorithm can be approximated by

$$E(I) \approx \frac{Q}{P}\mu + \sigma\sqrt{2 \log P}. \tag{4.3}$$

In [24], this formula is derived assuming normal distribution, but it is also claimed that it is a good approximation quite generally.

In order to use this formula to estimate the pseudo–cycle time, we need to make the following assumption.

Assumption 4.6 *The task interval lengths are i.i.d. random variables drawn from the same IFR distribution function with mean μ and variance σ^2.* □

In the following formulae, the coefficient of variation $(\frac{\sigma}{\mu})$ will be denoted by c_v. As in the previous section, the above model guarantees that a pseudo–cycle is the time it takes to complete Q tasks. Let $d_p(k)$ be defined as above, i.e., the time from the beginning of a pseudo–cycle k until the first processor initiation on PE_p. IFR condition assures that d_p is stochastically bounded by the time it takes to complete a whole task [24]. Since there are up to $(P-1)$ $d_p(k)$'s for each k, the upper bound on the overhead at the beginning of a pseudo–cycle can be represented by $(P-1)$ task executions. Therefore, a pseudo–cycle takes no more than $(Q + P - 1)$ task executions. Using the Kruskal–Weiss formula

$$E(\phi') \leq \frac{P-1}{P}\mu + \frac{Q}{P}\mu + \sigma\sqrt{2 \log P}. \tag{4.4}$$

The first, second and the third terms correspond to ϕ_1, ϕ_2 and ϕ_3, respectively. The first term represents the overhead due to the use of outdated information in component updates. The second term is the useful work, and the third term is the result of fluctuations. Combining (4.3) and (4.4) we obtain

$$E(S) \leq 1 + \frac{P-1}{Q + c_v\, P\sqrt{2 \log P}} \leq 1 + \frac{P-1}{Q}. \tag{4.5}$$

The conclusion is that the ultimate upper bound of $1 + \frac{(P-1)}{Q}$ is reached for small σ. On the other hand, the slowdown factor decreases as σ increases. For very large fluctuations, the slowdown factor is close to 1.

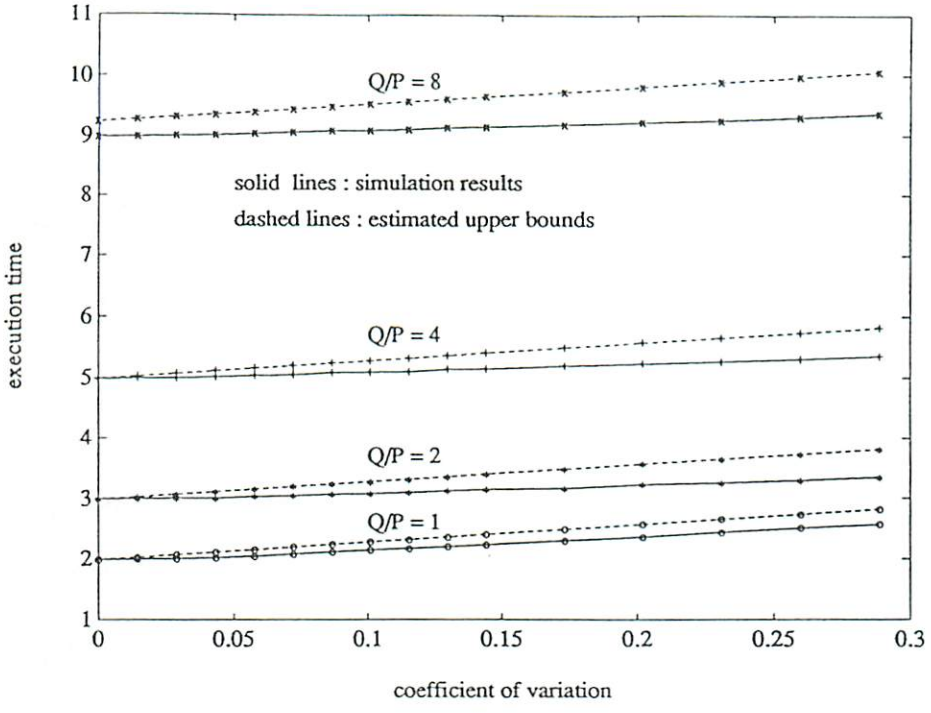


Figure 4.4: Comparison of estimated and simulation results for execution time (Uniform Distribution)

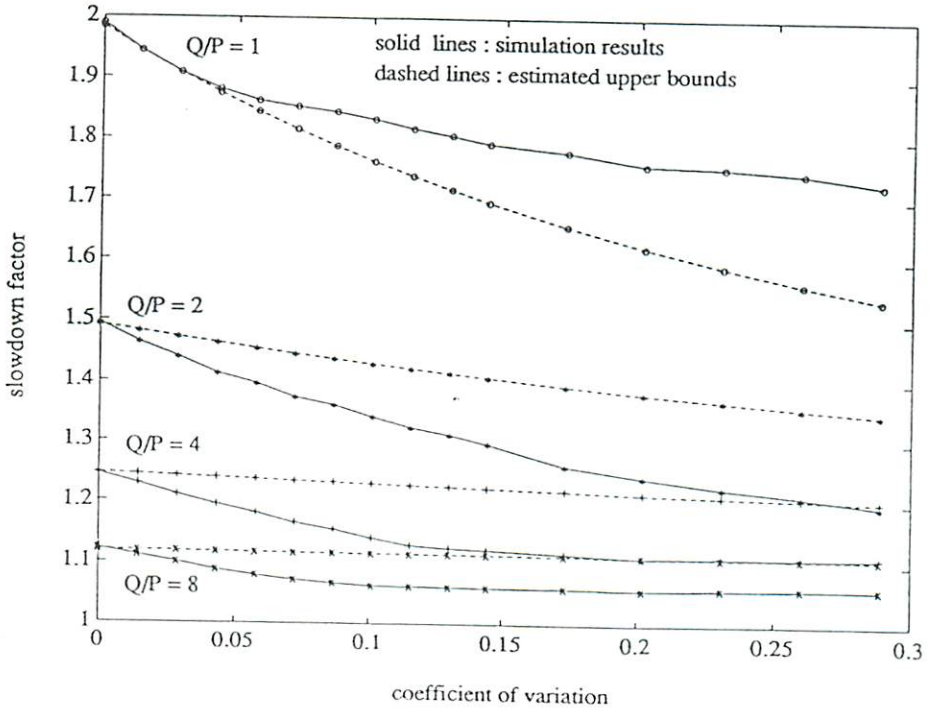


Figure 4.5: Comparison of estimated and simulation results for slowdown (Uniform Distribution)

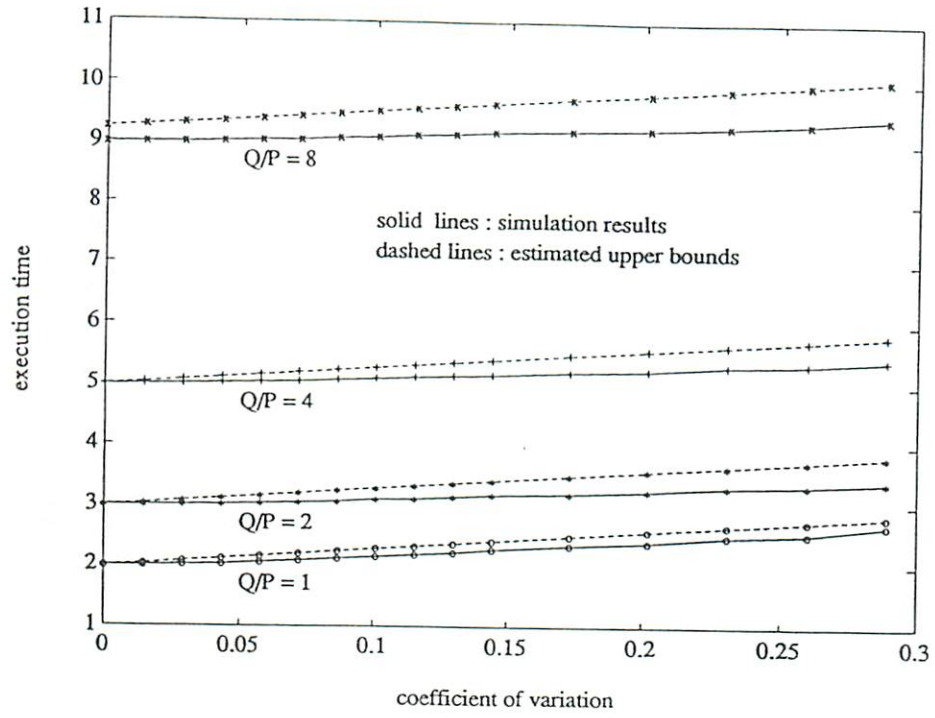


Figure 4.6: Comparison of estimated and simulation results for execution time (Normal Distribution)

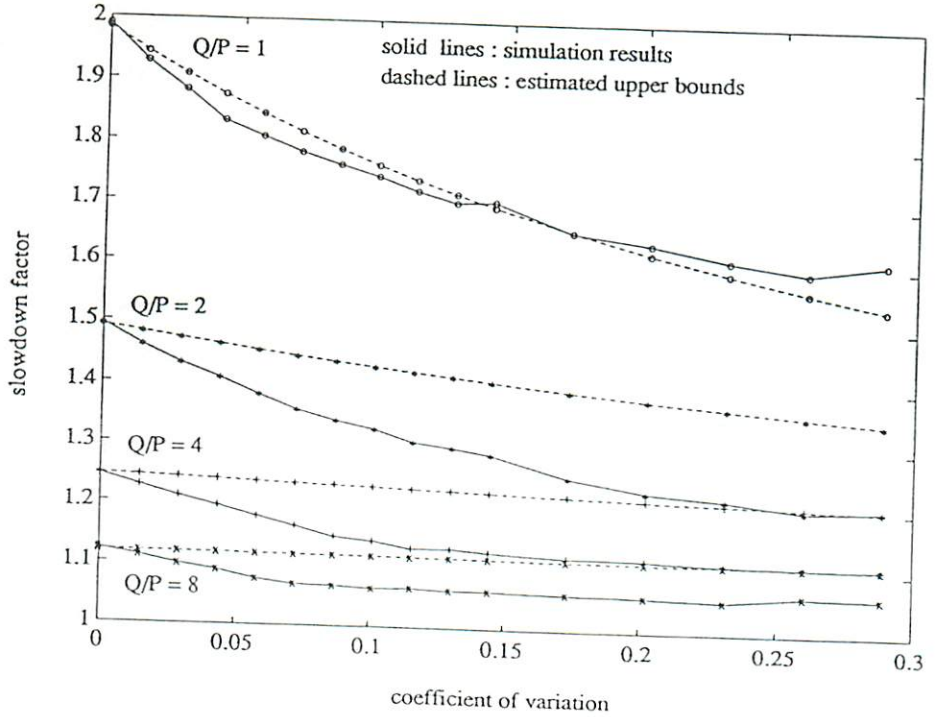


Figure 4.7: Comparison of estimated and simulation results for slowdown (Normal Distribution)

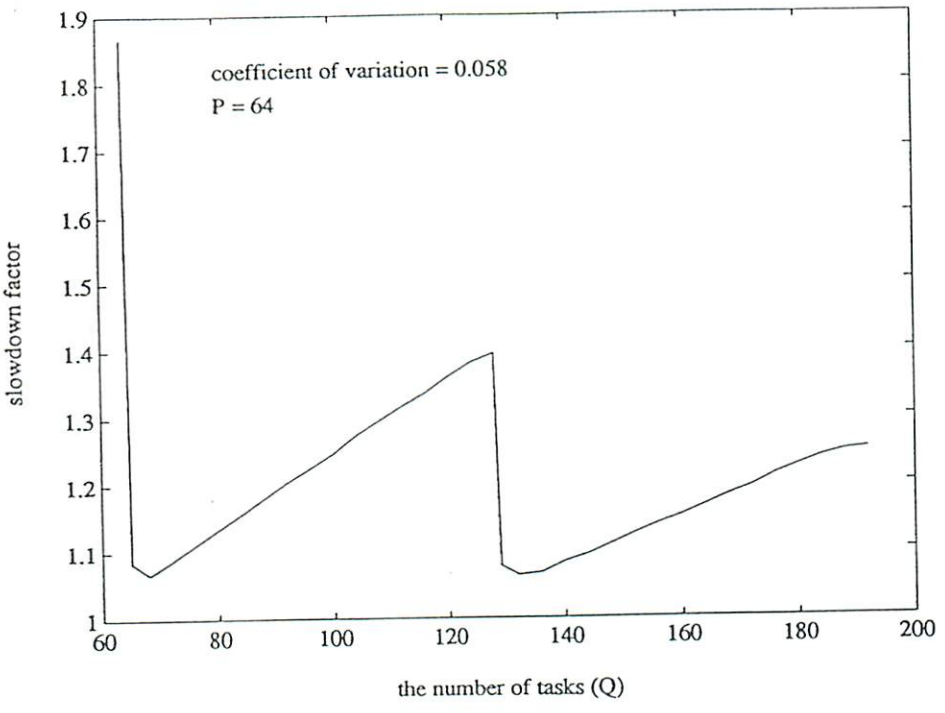


Figure 4.8: Slowdown vs. Q for small fluctuations

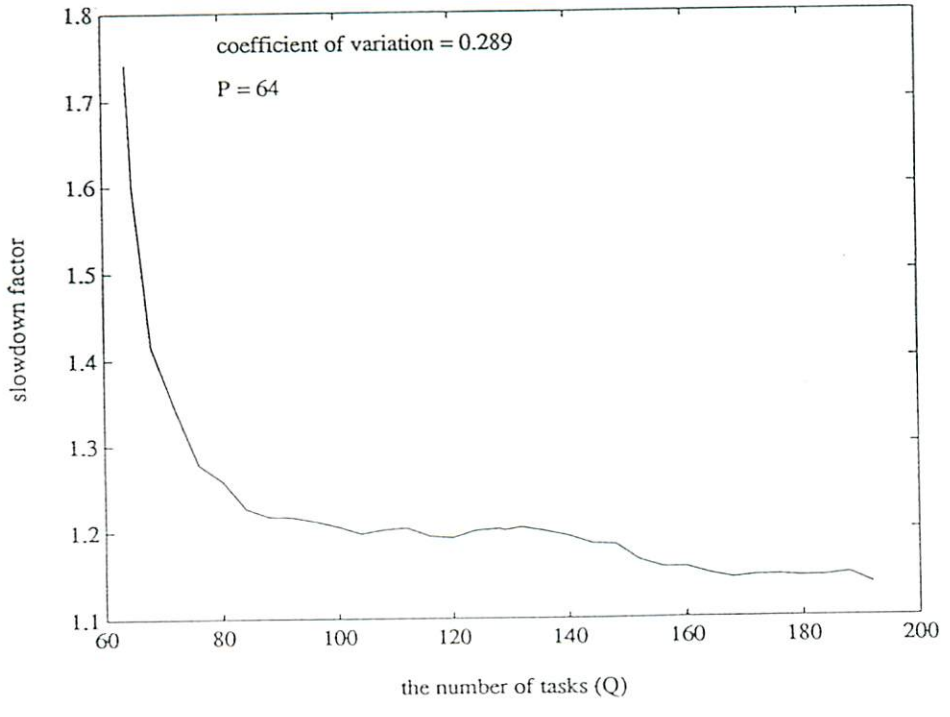


Figure 4.9: Slowdown vs. Q for large fluctuations

We have run some simulations to verify (4.4) and (4.5). Comparative results are displayed in Figures 4.4–4.10. In all of these figures $P = 64$. Also in the figures, the average task execution time is chosen as $\mu = 1$. $E(\phi')$ and $E(I)$ are labeled as the execution time, since they are the normalized total execution times, scaled down by the total number of iterations. In Figures 4.4–4.7 the results are for $Q = 64, 128, 256, 512$. In Figure 4.4, the estimated upper bound on the execution time is compared to the execution time obtained by simulation, for the uniform distribution. It is observed that the simulation results are very close to the right hand side of (4.4). Figure 4.5 displays the slowdown factor, for the same set of data. The simulation results verify the conjecture that the slowdown is upper–bounded by $1 + \frac{P}{Q}$. Figures 4.6 and 4.7 are identical to Figures 4.4 and 4.5, with the only difference that the former show the simulation results for the normal distribution. It is clear that the results for normal and uniform distributions are very close.

Figures 4.8 and 4.9 show what happens when $\frac{Q}{P}$ is not an integer. Integer values of $\frac{Q}{P}$ correspond to good load balancing in which case the advantage of the synchronous algorithm is maximized. This explains the peaks at integer values of $\frac{Q}{P}$. In Figure 4.9, the peaks are diminished, because the task intervals are changing more wildly.

In all the simulations above, the scheduling policy is round–robin, restricted to be non-redundant. In other words, the tasks are selected from a circular queue and if the task selected is currently executing, then the next task is selected. In Figure 4.10 simulation results are displayed for four different scheduling

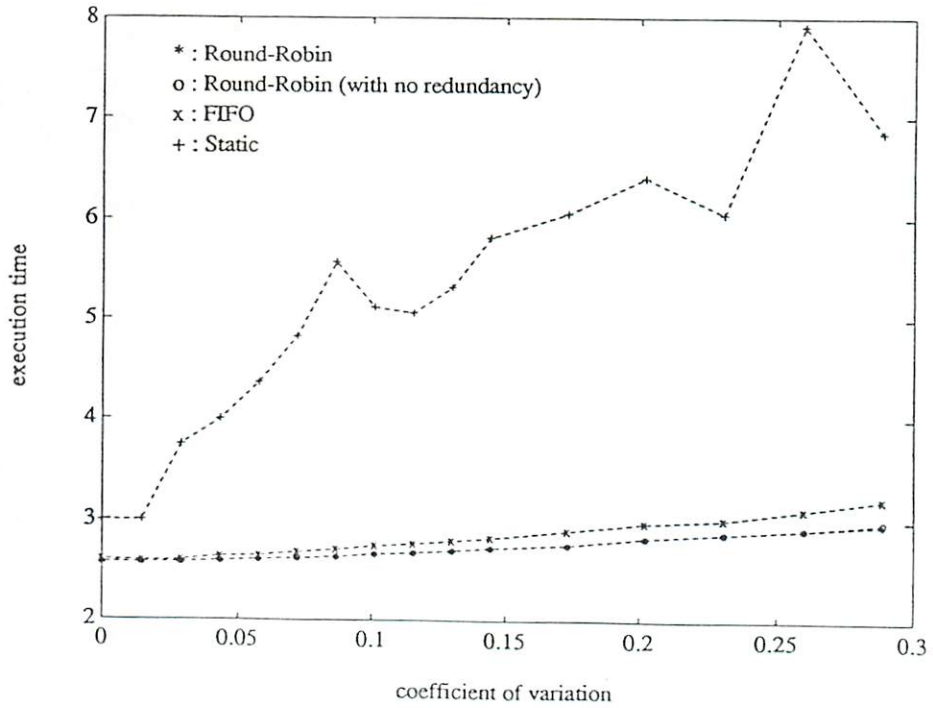


Figure 4.10: Effect of scheduling policy

policies: non-redundant round–robin, round–robin, FIFO and a decentralized scheme. In round–robin, the next task in the circular queue is selected whether or not it is currently executing. We observe no difference between the execution times of the two versions of the round–robin scheme, although we should note that the simulation results may not be reliable because of the redundancy, i.e., because $\{\varphi'(k)\}$ given by Definition 4.1 is the exact estimation of the minimum pseudo–cycle sequence only under non-redundancy. In the FIFO policy, there is a global queue of tasks. When a processor is available, it selects the task from the head