SELF-SYNCHRONIZING CONCURRENT COMPUTING SYSTEMS

by

Vijay Krishna Madisetti

Memorandum No. UCB/ERL M89/122

16 October 1989

SELF-SYNCHRONIZING CONCURRENT COMPUTING SYSTEMS

by

Vijay Krishna Madisetti

Memorandum No. UCB/ERL M89/122

16 October 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

SELF-SYNCHRONIZING CONCURRENT COMPUTING SYSTEMS

by

Vijay Krishna Madisetti

Memorandum No. UCB/ERL M89/122

16 October 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

SELF-SYNCHRONIZING CONCURRENT COMPUTING SYSTEMS

Vijay Krishna Madisetti

Department of Electrical Engineering and Computer Sciences

University of California

Berkeley, CA 94720

ABSTRACT

While the past few years have witnessed an unprecedented advance in the status of parallel computing hardware, software has not caught up with this pace of development. Our effort has been focused on the development of efficient algorithms and software for high-speed parallel scientific computing in an effort to meet this demand.

This thesis presents the theory and design of a new distributed computing system, the *Self-Synchronizing Concurrent Computing System (SESYCCS)*, for efficient solution of a large class of compute-bound scientific problems. This thesis establishes that *separating* synchronization from computation has a number of merits, especially in boosting the efficiency of implementation and reducing memory requirements.

In this thesis, we propose two new models for distributed computation; Static Computation Graphs (SCGs) and Dynamic Computation Graphs (DCGs), and a robust theory is developed for understanding their behavior. A new algorithm is proposed for efficient self-synchronization for SCGs that optimizes computing resource allocation.

We present new algorithms for self-synchronization for DCGs and derive concrete quantitative results for the efficiency of their implementation. We study in some detail the tradeoff between finite memory and speed of computation.

Application of the algorithms to simulation of discrete-event systems is described and a new algorithm, Wolf, is proposed and analyzed that promises a high processor utilization along with a significant speedup in the computation.

Committee chair

David G. Messerschmitt

David Messerschmitt

ACKNOWLEDGMENTS

I have been fortunate to have been able to meet with and work with a very large number of outstanding individuals at Berkeley. Their stimulating company and support has made my stay here the happiest and probably the most productive five years of my life.

I owe much to David Messerschmitt for giving me the benefit of his clear style of thinking and experience. I thank him for his guidance and generous advice at all times. I have learnt much from him.

I am indebted to Jean Walrand for teaching me most of what I learnt at Berkeley. I am very grateful to him for giving me his time unhesitatingly, and for many pleasurable discussions on life and philosophy.

I also take this opportunity to thank my parents, Anant and Madhavi, for their love, guidance, patience and encouragement without which I could not have taken so many steps in the right direction. My uncle and aunt, Somnath and Lakshmi, have been a major source of inspiration during my formative years, I thank them for taking so much interest and showing so much love. I thank my brother Avanindra for his unquestioning love and support through our childhood together and now through our advent into adulthood. I thank Sandhya for bringing him so much happiness. Years have fallen into the right places.

I thank Sherry and Judd Smith for accepting me into their family and for allowing me to share their happiness. I also thank Kim, Heather, Jenny and Alan for their affection.

I take this opportunity to thank my friend Chaitali for her love and encouragement through the years. I thank my friend Partha for his company and his advice on matters practical. My heartfelt gratitude goes to my present and past friends at Stebbins Hall, especially Margaret, Maria, Adam, Chris, Anupam, Jaideep, Rosa, Rene, John, Alison, Cristina, Lori, Ingrid, Deborah, Kristy, Herman and Gani and the rest, for their company. I cherish many happy memories at Stebbins.

At Cory I thank my friends, Teresa, Randy, Srinivas, Denise, Keshab, Pranav, Sonia, Venkat, Ashutosh, Vasant, Meena, Andy, Takis, Joe, Shimone, and Chedsada for being there when it mattered. I am also grateful to messer550 and eal550 (Horngdar, Johnson, Valerie, Biswa, Dev, Ilo, Tom, Wen-lung, Gil, John, Jeff, Shuvra, Edwin, Alan, Maureen, Holly, Cindy, Paul, and Janet) for their company and friendship. I thank Edward Lee and Graham Brand for being a ready source of guidance and advice in times of need and necessity (especially on market-investment strategies). I am indebted to many others at the department, who are my friends and will always be. For them that is friends need no thanky, but I am grateful to them (especially Pearl, Beatrice, Maureen, Leah, Carol, Beth, Gen Thiebaut, Kathryn, Pat, Chris, Gwyn and others with whom I had the good fortune of coming in contact with).

I thank NSF, Shell Development Co. (Bill Moorhead), NCUBE Corp. (Tom Bauer), and COPPE/UFRJ Brazil, for their generous assistance and financial support for this research.

Last but not the least, I thank Professor Bob Brodersen and Professor H. Frank Morrison for serving on my thesis committee and for their criticism which greatly improved my work.

CONTENTS

Chapter 1

Introduction

This thesis is dedicated to algorithms and distributed computing architectures for high speed scientific computing. It describes how a set of interconnected high performance VLSI processors, or a *multicomputer*, can be used to solve a wide range of computationally intensive engineering problems. The results presented can be extended to a distributed network of workstations. Most of the theory presented in this thesis is new, for interest in these machines as an alternative to traditional supercomputers has been rekindled only recently, spurred by the rapid progress in VLSI technology and the commercial viability of such machines. We have studied the programming and implementation of these computing machines and have developed a body of theory for their efficient utilization. We owe much to the rich heritage in parallel algorithms and technology developed in the 70's as reflected in the seminal work of Arvind, Chandy, Dijkstra, Hoare, Karp, Knuth, Kung, Kuck, Tukey and Winograd, and in the late 70's to early 80's by Cray, Siegel, Seitz, Stone, and Valiant among others.

Multicomputers consist of a set of VLSI processors (usually 100-1000), each with its own private memory and a capability to communicate asynchronously with other independently executing processors in the system. There is no shared state among these processors. Concurrent computation can be best described by the *Multiple-Instruction-Multiple-Data (MIMD)* model of computation, where each

processor executes a different part of the computation. It is expected that the result of the distributed and asynchronous computation would be identical to that of an equivalent serial (or sequential) computation, though available much faster. Considering the fact that each decade (or less) has witnessed an order of magnitude enhancement in the computational power of uniprocessors, one could argue that multicomputers would be considered viable if and only if they can offer a speedup of at least two orders of magnitude. For certain applications, multicomputers have indeed provided such a speedup.

While our objectives are straightforward, the means to reach these ends are challenging. Asynchrony is at the heart of concurrent multicomputing and its effect on distributed computation has been poorly understood. Time is no longer a shared variable, and consequently each processor is provided with a local time clock. Communication between asynchronous processors would, therefore, require the existence of an efficient synchronization mechanism, not only to ensure correctness of computation but also to guarantee its forward progress. For instance, let the progress of computation with time in a processor i be measured by a local clock $C_i(t)$ that increases monotonically with the wall clock time t. At some later time t_1, processor i may need some data from a remote processor j that is at a local clock value of $C_j(t_1)$. For a number of reasons, this result from processor j may not be made immediately available to i as requested. Non-zero communication delay is one such reason. Alternatively, the local clock $C_j(t)$ on processor j may not have reached $C_i(t_1)$. Processor i may have to wait until such time t_2, when $C_j(t_2)$ equals $C_i(t_1)$ for some $t_2 > t_1$. These reasons bring to light another fundamental issue in multicomputing: *latency*. Latency (due to suboptimal scheduling of distributed computation) enforces idle times in the processors, thus penalizing efficiency. Therefore, the performance of a VLSI multicomputer can best be evaluated through a better understanding of the issues of *latency* and *synchronization*.

This thesis is dedicated to exploring these issues in considerable depth. We present a robust theory with relatively few assumptions about the behavior of compute-bound problems. The theory was tested, whenever feasible, through implementation on a multiple-processor system. The grindstone of implementation has taught us many a lesson and our experience with these machines has been valuable (and not rarely painstaking) to the extent that it served as a testbed for validation of the theory

developed. This "proof-of-concept" approach has given us the confidence required to come up with a general framework for evaluating the performance of these computing engines.

Chapter 2 provides a brief quantitative description of synchronization in parallel systems. Not much is known about synchronization in multicomputers, hence this chapter motivates the chapters that follow.

We first address the issue of latency in multicomputers. Latency can be eliminated by time-sharing the multicomputer in a *multi-threaded* computation, where a number of independent programs can execute concurrently on the same set of processors. We introduce the notion of a *time-shared* multicomputer, and propose new optimal algorithms for efficient resource allocation in such systems. Our emphasis is on the development of stable, efficient, real-time distributed algorithms (using feedback). The onus of improving the efficiency of the implementation is removed from the shoulders of the user, and moved to a scheduler on the multicomputer itself. The programmer, however, retains the responsibility of presenting the algorithm to the multicomputer in a form suitable for such a time-shared approach. As we describe in Chapter 3, this effort is minimal, relative to the rewards accrued.

We then address the important issue of synchronization for general asynchronous systems. We develop a theory for a class of synchronization algorithms. Synchronization issues are separated from the computational (data) aspects of distributed computation. Use of special synchronization hardware is proposed. In Chapter 4, we present new algorithms for synchronization and derive analytical results for their performance. The question of whether a distributed system should be resynchronized at regular intervals or allowed to proceed without resynchronization is posed, and some analytical results indicating when either approach is preferred are presented. This chapter provides the theory for the design and analysis of a new class of concurrent computing systems, *Self-Synchronizing Concurrent Computing Systems (SESYCCS), where the synchronization is provided by the computing system itself.*

A new class of algorithms, appropriately called the Randomized Algorithms (RA), are introduced for synchronizing asynchronous distributed computing systems. These algorithms, as described in Chapter 5, promise a further enhancement in the efficiency of SESYCCS, by learning from the past

behavior of communicating processors. These algorithms find immediate application in the distributed simulation of dynamical discrete-event systems.

Chapter 6 describes how our algorithms for distributed synchronization fit naturally in the distributed simulation of discrete-event systems. We describe how such computation can be efficiently performed out on a SESYCCS. Chapter 7 discusses present work in the area of synchronization in SESYCCS and some future directions.

Chapter 2

Synchronization in Distributed Systems

Synchronization is defined as a mechanism that ensures that the result of a distributed computation is correct, in that it provides the same result as that of an equivalent serial or sequential computation. Synchronization is, therefore, a fundamental issue in the design of multiple-processor systems. The twin responsibilities of allowing fair access to shared resources and ensuring correctness of the distributed computation are delegated to the synchronization algorithm. One of the primary aims of distributed and parallel computation has been to partition a compute-bound problem and assign the parts to independent processors, such that the computation terminates much faster as a consequence. The increase in the speed of computation, called *speedup*, depends both on the natural concurrency available in the application as well as on the task partitioning and synchronization strategies used. Thus, distributed and parallel computation can only be viable if the computational burden is shared evenly among processors, if the overhead in communications of data and control variables is small, and if the synchronization is efficient.

As the number of processors in the system could be as large as a few thousands, it is of vital importance that the algorithms used be distributed, and that their performance scale well. Time is a distributed variable, and correctness of the computation has to be enforced by each processor on the basis of local information alone. Algorithms which enforce global correctness based on local decisions are

therefore zealously sought. In this chapter, we will introduce the problem of synchronization in distributed systems. Later chapters will propose and analyze specific algorithms for synchronization in message-passing distributed-memory systems. Finally, we introduce a new class of systems called the **self-synchronizing concurrent computing systems (SESYCCS).**

2.1. Synchronization in shared-memory systems

In this section, we will review in brief the synchronization mechanisms available in shared-memory multiple-processor systems. While our aim is to develop the theory of synchronization in distributed-memory machines, we will examine some shared memory synchronization schemes to allow later comparisons of performance.

In a shared-memory multiple-processor system, the processes (residing on processors), access common memory locations to obtain data and tasks for parallel execution. This model of computation is fine-grained (parallelism at the instruction and loop level) in nature, and scheduling of tasks is often centralized. Best efficiency is achieved whenever a process with a task has a processor to run on.

The need for synchronization in shared-memory systems can arise when two or more processes wish to read or write the same data structure. The final result of the computation then depends on the times (and the order) of accesses by each competing process. As an example, consider the following statements to be executed concurrently by two processes, 1 and 2.

$$x = y * z$$

$$y = x + z$$

If x and y are shared between the processes, 1 and 2, the result of the parallel computation is unpredictable and depends on the order of access by the processes. Synchronization is of importance in resolving this issue of dependency in such a *critical* section. Likewise, when two processes *compete* for the control of a shared resource (an I/O device, for instance), they both cannot be assigned the use of device, and some synchronization protocol has to be enforced to serialize access.

2.1.1. Semaphores:

Semaphores are used to synchronize access in a shared-memory environment and represent a shared data structure into which each process can read or write. A *lock* is a very common type of semaphore, regulating access to a critical section.

Lock: A lock is a mechanism which ensures that only one process can access a shared data structure at one time. Processes compete to acquire locks on a shared variable. If a process finds a desired shared variable unlocked, it acquires the lock, and releases the lock after completion of its task in the critical section. Processes awaiting the release of a lock have the choice of either spinning in a *busy loop* (wasting cycles) or can switch context to some other task. Hardware locks, called *atomic locks*, are sometimes provided, where actions to ensure that the lock is unlocked before acquiring it, and relocking it, are performed as one *indivisible* action. Hardware locks are quick and efficient because they do not require operating system intervention in the synchronization.

Ordering Semaphores: If there are data dependencies between successive iterations of a loop, it is essential that the iterations be executed in certain order. Ordering semaphores allow this by assigning to each competing process an *iteration* number, I. A process is allowed to execute the loop if and only if I equals the value of the ordering semaphore N. After completion of the execution, the process relinquishes control over the loop after incrementing the value N, allowing other processes to continue with the execution.

Counting Semaphores: Counting semaphores are very flexible in scheduling the order of execution, and are found useful in the management of messages buffers and queues. For instance, counting semaphores can ensure that a process which has been waiting the longest would be assigned the use of the resource. These semaphores allow the management of the utilization of several instances of the same resource. The counting semaphore is given a value N. This value can be interpreted as follows:

$N > 0$: N is the number of instances of the resources available, when $N = 1$, and there is a single resource, it is unlocked and available.

$N \leq 0$: No instances of the resource are available, and $-N$ represents the number of competing processes awaiting release of the resource.

To acquire the use of the resource, a process has to decrement N. If N is less than 0, it queues its identity in the $-N$ th slot and awaits it turn.

To release the use of the resource, the process examines if N is less than zero, and then notifies the first process in the queue. It then increments N.

2.1.2. Barrier Synchronization

In a typical parallel program, a number of parallel tasks are forked out to different processes by the job scheduler. After completion of their assigned tasks, processes mark themselves as present at a barrier. After all the processes in the computation have marked themselves at the barrier, a new phase of computation can start using the results of the preceding phase. Barrier synchronization is commonly used in both shared-memory systems and distributed-memory systems.

Barrier synchronization can be implemented in a number of ways. A class of computation called Static Computation Graphs (to be discussed in the latter sections of this chapter) employ this method of synchronization. The obvious method for implementing barrier synchronization is to assign to one process the responsibility of making sure all processes reach the barrier. Another common method is to contruct a tree-type interconection between the processes, where the task completion signals reach the root through a number of intermediate levels in the tree. Use of a hardware bus for synchronization would be especially efficient.

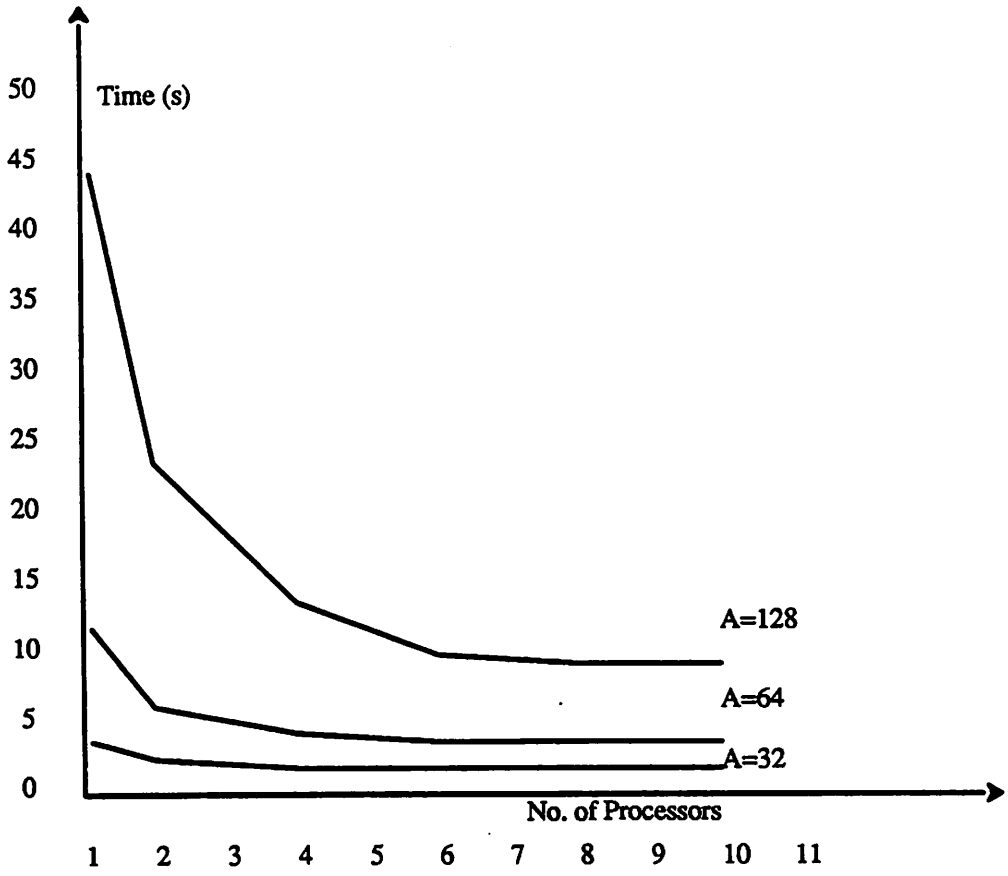


Figure 1: Computation on Single Bus Shared-Memory System

2.1.3. Performance

While mechanisms for synchronization in shared memory systems have been understood to a large degree, current issues that need to be solved concern the scalability of these mechanisms to a larger number of competing processes, and also issues in ensuring coherence of data in private caches. Ease of programming shared-memory machines has been crucial to their wide acceptance in the commercial market today, and speedup of application programs by an order of magnitude is typical with very little additional effort in synchronization.

In Figure 1, we illustrate the performance of a single-bus shared memory Sequent system on a seismic migration computation. As can be observed, increasing the number of processors up to six decreases the computation time correspondingly. The performance quickly saturates after that, due to bus contention. Increasing the size of the problem (in this figure, A represents the size of the data array)

improves the initial speedup.

2.2. Synchronization in Distributed-Memory Systems

In a distributed-memory system there are no shared variables. Synchronization is implemented via messages communicated between processors. Time, being a distributed variable, implies that each processor, i, has a local logical clock, $C_i(t)$ at a real time t, representing its local progress (in simulated time) in the computation. $C_i(t)$ evolves according to the dynamics of the computation. At this point we will formalize the notion of a distributed computation. A distributed computation, or a *physical system* consists of N subsystems, where the state of subsystem i evolves in discrete time as a function of its past state at time $t-1$ and its interactions with other processors j. At each time step (the step size can be fixed or random), the subsystem updates it state, and sends messages to other subsystems participating in the physical system. The physical system is then mapped onto a set of M processors. This mapping is said to represent the logical nature of the computation, or the *logical system.*

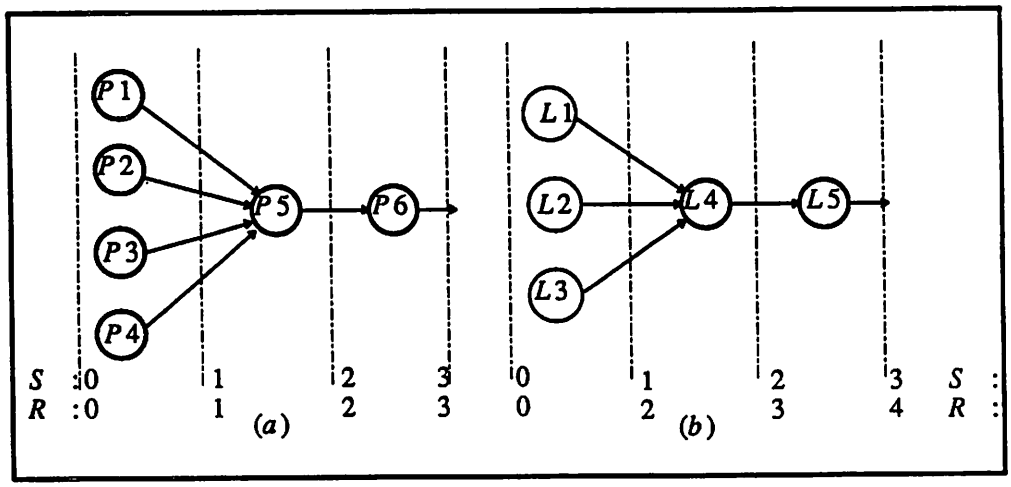


Figure 2 .(a) Physical System, and (b). Logical System.

S represents simulated time, and R represents real computing time. In the physical system, both simulated and real times progressed at the same rate. In the logical implementation on three processors the simulated time moved slower relative to the real time. This is because $L3$ was assigned two tasks, $P3$ and $P4$, instead of one.

As an example, let us consider a distributed computation represented by the data flow graph illustrated in Figure 2a. This represents the physical structure of the distributed computation independent of any implementation. The vertices can represent tasks or subprograms participating in the distributed computation. For the sake of simplicity let us assume that each task takes exactly one simulated time unit and communications are instantaneous. The task start times are shown in the figure as well. A notion of global time exists in the physical system and simulated times, S, are equal to the real computing times, R (which represent the real time spent in the CPU on the computation). The implementation of this *physical* system (on a distributed computing system) is defined as a *logical* system.

In Figure 2b, the physical system is shown mapped on to a logical system of three processors. Two of the physical tasks from Figure 2a were mapped onto one logical process in Figure 2b. Note that logical process $L3$ now is assigned two tasks $P3$ and $P4$. Therefore, there is no single global time representing the distributed computation and $L2$ and $L1$ remain idle for one real computing time unit awaiting completion of tasks $P3$ and $P4$ on $L3$. Each logical process has its own local logical clock measuring the progress in simulated units with real computing time. $L1$ and $L2$ take one computing time unit ($R=1$) to reach global simulated time $S=1$, while $L3$ takes two real computing time units ($R = 2$) to reach global simulated time $S=1$. The local times are measured in simulated time units while the computing time is measured in real time units (wall clock times).

This particular mapping between physical and logical systems is not unique. The selection of the logical system depends on the the number of processors available in the distributed system, the efficiency of a particular mapping, and the cost of the communications (relative to computation). In our example, if the logical system of Figure 2b were implemented on four processors instead of three, the local simulated times $C_i(t)$ in the logical system and the global times S in the physical system would correspond to each other.

Interactions between the subsystems in the physical system are faithfully carried over to the logical system, and in addition the logical system introduces a number of synchronization and control messages which ensure the correctness and efficiency of the distributed computation. The state information

in the logical system is correspondingly larger. The connectivity of the logical system can subsume that of its associated physical system largely to accommodate efficient signaling for synchronization and control purposes. Obviously, the logical system is of interest only if the state of the physical system can be recovered from it.

2.2.1 Models for Distributed Computation

The dynamics of the *physical system* can be represented by a set of state transition equations. Two cases of distributed computation that are especially important are developed further in this thesis. One of them evolves in discrete time (time-driven) while the other evolves at discrete points (event-driven) in continuous time. Time-driven distributed computation includes a wide set of engineering problems in the solution of systems partial differential equations and those which involve substantial iterative manipulation of numerical expressions at each time step until completion. Event-driven systems are of importance in the modeling and analysis of distributed systems in a variety of applications, in distributed simulation of dynamical systems and for efficient implementation of computation where activity is localized to certain discrete (and not necessarily evenly spaced) points in time.

Time-Driven Physical System

The physical system consists of N subsystems, S_1, $S_2, \ldots, S_N$ with states $x_1(t)$, $x_2(t)$, $x_3(t), \ldots, x_N(t), t = 0, 1, 2, 3 \ldots$ that evolve according to the set of dynamical equations

$$x_i(t) = f_i(x_i(t-1), \{m_{ji}(t-1), j \neq i\}, \omega_i(t), t)$$
$$m_{ij}(t) = g_{ij}(x_i(t), \omega_i(t), t)$$

$\omega_i(t)$ represents the stochastic component of the dynamics of the distributed computation. $m_{ji}(t-1)$ represent communications (or messages) from subsystem j to i in time step $t-1$. The function, $f_i(\cdot)$ updates the state while $g_{ij}(t)$ determines which messages are to be transmitted to other processors, j, from i.

The implementation of the time-driven physical system on a set of M processors is called the time-driven logical system.

Time-Driven Logical System

The logical system consists of M processes, $L_1, L_2, \ldots, L_M$. These systems evolve dynamically in $T = 0, 1, 2, \ldots,$ where $T \triangleq T(t)$ and $t = 0, 1, 2 \ldots$ is the real computing time. The augmented states, $X_i(T)$ and the messages $M_{ji}(T)$ are represented by

$$X_i(T) = F_i(X_i(T-1), \{M_{ji}(T-1), j \neq i\}, \omega_i(T), T)$$
$$M_{ij}(T) = G_{ij}(X_i(T), \omega_i(T), T)$$

In this implementation the local times of the processors $C_i(t) = C_j(t) = T(t)$,

Event-Driven Physical System

In an event-driven physical system N subsystems, $S_1, S_2, \ldots, S_N$ with states $x_1(t), x_2(t), \ldots, x_N(t)$, evolve in $t \in (0, \infty)$.

There also exists a sequence of times $\tau_i^1 \leq \tau_i^2 \leq \ldots \tau_i^n$ such that $\tau_i^m - \tau_i^{m-1}$ is a random interval of time, when the system *jumps* from one state to another. For example, the state could change in response to a message from another processor. The arrival time of this message is itself a random variable. The state jumps to a new state when this message is ultimately received.

Then the system can be described by the following set of dynamical equations:

$$x_i(\tau_i^m) = f_i(x_i(\tau_i^{m-1}), \{m_{ji}(\tau_j^{m'}), j \neq i\}, \omega_i(\tau_i^m), \tau_i^m)$$
$$m_{ij}(\tau_i^m) = g_{ij}(x(\tau_i^m), \omega_i(\tau_i^m), \tau_i^m)$$

with $\tau_j^{m'}$ satisfying $\tau_i^{m-1} \leq \tau_j^{m'} \leq \tau_i^m$. $x_i(\cdot)$ is a right continuous function of t. $m_{ji}(\tau_j^{m'})$ refers to the messages received by i from other systems j in between two successive jumps at τ_i^{m-1} and τ_i^m respectively.

Event-Driven Logical System

The implementation of an event-driven physical system on a concurrent computing system is called the event-driven logical system. The state $X_i(\cdot)$ of subsystem L_i is a right continuous function of the local time $C_i(t)$. The real computing time, t, can be continuous or discrete (as in any computer system, continuous time is modeled as discrete time steps). In the former case, the local clock $C_i(t)$ is right

continuous in t. $C_i(t)$ takes values $T_i^1, T_i^2, \ldots, T_i^m$ such that $T_i^m - T_i^{m-1}$ is a random variable for all m and i. The jumps in the values of $C_i(t)$ represent the stochastic nature of the discrete event dynamical system. The evolution of the augmented states in the logical system can then be described by

$$X_i(T_i^m) = F_i(X_i(T_i^{m-1}), \{M_{ji}(T_j^n), j \neq i\}, \omega_i(T_i^m), T_i^m)$$
$$M_{ij}(T_i^m) = G_{ij}(x(T_i^m), \omega_i(T_i^m), T_i^m)$$

The progress of the computation can then be expressed in terms of rate of growth of $\min_i C_i(t)$, since the smallest simulated time in the logical system represents the time up to which the distributed computation has been progressed.

2.2.2. Causality Conditions

The distributed computation is *correct* iff $x_i(\cdot)$ and $m_{ji}(\cdot)$ can be recovered from $X_i(\cdot)$ and $M_{ij}(\cdot)$ for all i, j. The necessary conditions for correctness, called the **causality** conditions, that should be satisfied are,

(a). $T_i^{m-1} \leq T_i^m \leq T_i^{m+1}$ for all m and i.

(b). $T_i^{m-1} \leq T_j^n \leq T_i^m$ for all i, j and for appropriate m and n (satisfying the state update equation).

Formulating sufficiency conditions that are more general and depend both on the dynamics of the systems as well as on the correctness within the programs themselves remain the responsibility of the user.

A **synchronous** synchronization algorithm ensures that the conditions (a) and (b) are satisfied at the end of *each* computing time step increment $\delta t = \varepsilon, 2\varepsilon \ldots$

An **asynchronous** synchronization algorithm ensures that the conditions (a) and (b) are satisfied *eventually* at $t \rightarrow \infty$ (or equivalently when $C_i(t) = \infty$ for all i).

A synchronous algorithm ensures that the conditions are satisfied at every time step and that the local clocks move forward in lock-step. In the synchronous algorithm the absence of a message has

also to be explicitly communicated to every other processor in the system. This is a high penalty in communications and does not scale well with an increase in the number of processors. In addition, the faster processors remain idle at every time step waiting for the slower processors to catch up. This situation then favors an asynchronous synchronization algorithm.

The asynchronous algorithm does not incur the overhead of enforcing the causality conditions at each time step. Whenever an error is discovered, the synchronization algorithm rolls back the simulated time clocks until the last time they were correct and the computation begins anew. The asynchronous algorithm thus enforces the conditions (a) and (b) only when it discovers a discrepancy. It is of much interest to evaluate the performance of these two algorithms.

Some observations can be made at this point. There are two notions of time in the logical system. The first notion is that of simulated time, which represents the point of time in the execution of the physical system. The next notion is that of real computing time which represents the time spent on computation in clock cycles (or ticks) on the implementation of the computation. In a logical system, therefore, the simulated time evolves as some function of the real computing time. Some processors participating in the logical system can update their simulated time much faster (in real computing time) than other processors in the system. This leads to a disparity between local simulated times in different processors at the same point in real computing time. In our example shown in Figure 2, the simulated times of logical processors $L1$ and $L2$ kept pace with the real computing time until $R=1$ and then remained at 1 until the real computing time reached $R=2$, when processor $L3$ reached simulated time 1 as well. Processor $L3$ is slower in reaching the barrier because it had two tasks of size 1 unit assigned to it. In a logical implementation of a physical system on an infinite number of processors, the simulated times can closely follow real computing times when tasks have comparable granularity.

Let us now define $G(t)$ as follows

$$G(t) \triangleq \{(i, j, m_{ij}(t)) \mid m_{ij}(t) \neq \phi\}$$

In a time-stepped physical system where t evolves in discrete time, $G(t)$ describes the precedences of tasks and messages communicated at each time. If these precedences and messages $m_{ji}(t)$ are known

apriori, i.e if $G(t) = f(t)$, where $f(t)$ is a known function of t, then the physical system is defined to represent a **Static Computation Graph (SCG).**

On the other hand, if $G(t) = g(t, x_1(t), x_2(t), \ldots, x_N(t))$ is a random function of time, as in the event-driven system, the physical system is then defined to be **Dynamic Computation Graph (DCG).**

Note that $G(t)$ for $t = \tau_i^m$ for some i and some m, is unknown apriori in a stochastic event-driven system. Thus discrete event systems fall into the class of DCGs. Some time-driven systems can be described by DCGs as well, when the message transmission times and the messages themselves (e.g in data-dependent computation) at each time step are unknown before the computation starts.

Static Computation Graphs, also called *Class* **S**, imply that the structure of the computation is known and statically determined. This knowledge can be used to synchronize such graphs efficiently. Dynamic Computation Graphs, also called *Class* **D**, represent a general model for distributed computation where the task precedences and messages are unknown prior to the computation and depend on the stochastic evolution of the states of the physical system.

In this description of the distributed system, we have assumed that the state of each subsystem is influenced by interactions with every other subsystem. In practice, this interaction is often local, each subsystem interacting locally with just a few other subsystems.

In a logical system the system states evolve in real computing time at different rates in simulated time. Hence some of the interactions present in the physical representation may not be available to the faster logical process in the computation of the next state. The logical system then has two alternatives to avoid making an error in the forward computation. In a synchronous implementation, the next state is not computed unless all the subsystems have a common simulated time and only then the lowest time stamped event is processed. In an asynchronous implementation the logical process assumes that it has all the relevant information and computes the next state. The synchronization algorithm ensures that the logical system is able to recover from this error. This error occurs due to the fact that some of the messages $m_{ji}(T_j^{n*})$ were not received by processor i as it had reached simulated time T_i^{n+1} (such that

$T_i^n \leq T_j^m \leq T_i^{n+1}$) much earlier in computing time.

Since our primary objective is to understand the performance of synchronization algorithms, we will restrict formalism for its own sake to a minimum, focusing more on the efficiency issues.

We will now examine how the problem of ensuring global synchronization of logical systems (representing event-driven physical systems) is solved in this distributed environment.

2.2.3. Synchronous Methods

The local state of a subsystem i at T_i^{n+1} depends on the previous state (at some discrete point T_i^n), as well as on its interactions with the other subsystems that are between simulated times of T_i^n and T_i^{n+1}. The local clocks of the different physical subsystems are synchronous to a global clock. However, when mapped to the logical system for the purposes of distributed computation, the clocks tend to grow at different rates, and furthermore, a physical subsystem may not interact with other subsystems at some points in time. This implies that in a logical system the *absence* of an interaction also has to be explicitly communicated. The absence of interaction between two subsystems in the physical system is carried over to the logical system as a *null message*. Obviously, if the logical system were simulating a physical system where such interactions were infrequent, the overhead in communicating these null messages can be very high. Synchronous methods find use in Chapter 3, when the structure of the computation is predictable.

2.2.4. Asynchronous Methods

When logical processes are allowed to update their local clocks on the basis of partial or incomplete information, the clocks can move independently of each other, except when interactions take place in the corresponding physical system (as described in the event-driven system). These methods have the advantage that processes can move ahead in computation time using available information without awaiting messages. The logical system corresponding to the event-driven system is itself event-driven. Some of the events occurring in an asynchronous logical system can be false, as they were scheduled on the basis of partial information. The price paid for this optimism is that processes

need be capable of recovering from errors in computation that could have occurred on the basis of incomplete local information (when a message from a subsystem which is at a smaller simulated time bearing new information is not received at that real computing time) These methods are discussed in some detail in Chapter 4.

2.2.5 Comparison of Synchronization Methods

One approach that distributes computation in a discrete-event system identifies the events that have the smallest time stamps in the system and schedules them for execution. We will illustrate this for the case of an example of an acyclic queueing system shown in Figure 3. If in addition, we make the assumption that processes can execute events only in increasing time stamp order, the efficiency of the algorithm improves, with most of the processors doing some kind of work. The introduction of null messages improves efficiency further. Subsequently, we will also describe how an asynchronous synchronization mechanism performs on the same network. Both the synchronous and asynchronous strategies appear to work well for this example.

The physical system to be simulated is described by five first-come-first-served (FCFS) queues connected as shown in Figure 3. The logical system on five processors assumes the same topology as the physical system. The initial condition of the network is described in Figure 3, where events are each identified with a unique identity (in small case alphabet) and a time stamp in simulated time. The (simulated) time stamp is updated as the event traverses through the network. To simplify study further, we also assume that executing each event takes one time unit in both simulated and real computing times (ticks). Therefore, the time stamp (in simulated time units) of each event is incremented by the service time (one unit) and by the waiting time at each processor. Each successive snapshot represents the state of the network after one computing time tick.

The first strategy for synchronization is implemented in two phases, a synchronization phase followed by a computation phase. In the synchronization phase, the events in the system are rank ordered on the basis of their time stamps. At the end of the synchronization phase, the computation phase begins

with the identification of a process that can begin processing its input queue. The process chosen is the one with rank = 0. Once the event has been processed the synchronization phase resumes and the events are rank ordered again.

One way of rank ordering the events would be to interconnect the processes together in a virtual ring. Each process communicates a message containing its identity and its current timestamp to its neighbor in the ring. The rank rk of each process k, is initialized to 0. Each process receiving a message, increments its rank by one if the event in its input queue has a larger time stamp than the time stamp of the event in the received message. The message is then forwarded to the next process in the ring. If there are N processes in the system, this will take N communication times. The synchronization phase ends when each process receives its own message. The computation phase then begins, with the process assigned rk=0 processing its input queue. This computation is shown in Figure 3. The entire computation is terminated after 10 steps.

In the algorithm just discussed, there were no additional synchronization messages between processes. The concurrency of the system is limited to the processes with the smallest rank. With an increase in the number of processes, most processes remain idle resulting in an inefficient implementation of the logical system. We will now describe how null messages can speed up the execution. Null messages are not real events, but are messages that inform the processes when it is safe to update their local clocks. The knowledge that the service time in each process takes one time unit is used with advantage in letting recipient processors know when they could expect event messages. If time stamps can be predicted in advance, the efficiency of a distributed computation improves as a function of this lookahead.

In a synchronous strategy (See Figure 4), the algorithm ensures that events are scheduled for execution in time stamp order. At the first time tick, processes 2 and 5 are enabled, and process jobs (e , 6) and (b , 7) respectively. Job (e , 6) exits with the identity (e , 8). The processing time on process 1 is one simulated time unit, but the local clock on process 2 was previously 7 (it had processed (b , 6) previously). Therefore, the time stamp of job e is 8 instead of 7.

At this instant, process 2 has information that it will not send any task to process 5 with a timestamp less than 8, hence it transmits a null message $(\pi, 8)$ to process 5. This allows process 5 to schedule event $(c, 8)$ for execution in the next time tick. Similarly, in the next time step, null message $(\pi, 11)$ assures process 4 that it can schedule event $(f, 11)$ for execution.

The computation therefore proceeds much faster with the introduction of these messages explicitly for synchronization. The sequence of snapshots in Figure 4 illustrates that the termination occurs in 6 time ticks.

The performance of asynchronous synchronization is illustrated in Figure 5. Here, processes are not blocked, and each process executes whatever task is available at hand. However, as can be seen in figure 3, process 3 realizes it has made an error in time step 1 when it had scheduled event $(d, 9)$ for execution without awaiting the arrival of $(e, 8)$. Fortunately, the effect of this error was not propagated by process 5, as it had scheduled event $(c, 8)$ for execution. The process 3 recovers from the error and sends an *antimessage* to process 4 asking it to delete event $(d, 8)$ from its event list. The remainder of the distributed computation proceeds as shown in the sequence of figures, and terminates in 6 time ticks.

This example serves to introduce the different methods of synchronization used in message passing distributed systems. The aim of this thesis is to provide the foundation for the design of *self-synchronizing* distributed systems. In the models we present and in subsequent analyses, we will often find it rewarding to compare the performance of self-synchronizing distributed systems with the performance in conventional distributed systems. Our models are simple and robust and allow us to study performance of synchronization algorithms when the conditions of asynchrony are severe. We will point out some limitations to our analyses as well, so that the reader will be advised as to when they are not applicable.

Synchronous methods incur a high overhead in idle times when blocked processes are unable to advance their clocks because global information is not available locally. The introduction of information or null messages partly ameliorates this problem but adds to the communication overhead. Each

process must now keep informing every other process in the system updated estimates of its local time. For obvious reasons, such an approach would scale poorly with an increase in the number of processes, when the cost of communicating updates takes up a significant portion of the computation time. Pathological examples, involving feedback, have been constructed with poor efficiency (0.05 - 0.1) of implementation.

Asynchronous synchronization allows the local clocks to drift apart rather than blocking them from doing so. This achieves the maximum potential of concurrency available in the computation. The disadvantage to this method is that occasionally the process has to undo the results of erroneous computation. The effects of secondary error propagation have also to be considered. The performance of such optimistic methods has not been clearly understood, but efficient implementation has been shown for some balanced applications. Another advantage to using asynchronous methods arises from the fact that the user does not have to synchronize her logical processes for efficiency. Some efficient strategies for such synchronization are studied in Chapters 4, 5 and 6.

Consider a system with a few thousand processors, and a larger number of logical processes, used in a distributed computation. It is unusual and often impossible for any system designer to be able to synchronize this distributed system part by part. The onus of synchronization should rightfully rest on the distributed computing system, heralding the introduction of *self-synchronizing concurrent computing systems* which are responsible for handling the synchronization responsibilities for a distributed application. The application expert subdivides the algorithm into parts and assigns processes to processors. The system then synchronizes the distributed application. The synchronization strategies are to be implemented both in the hardware and software. The separation of synchronization and control hardware from computation appears to be a viable alternative to traditional intra-computation synchronization methods.

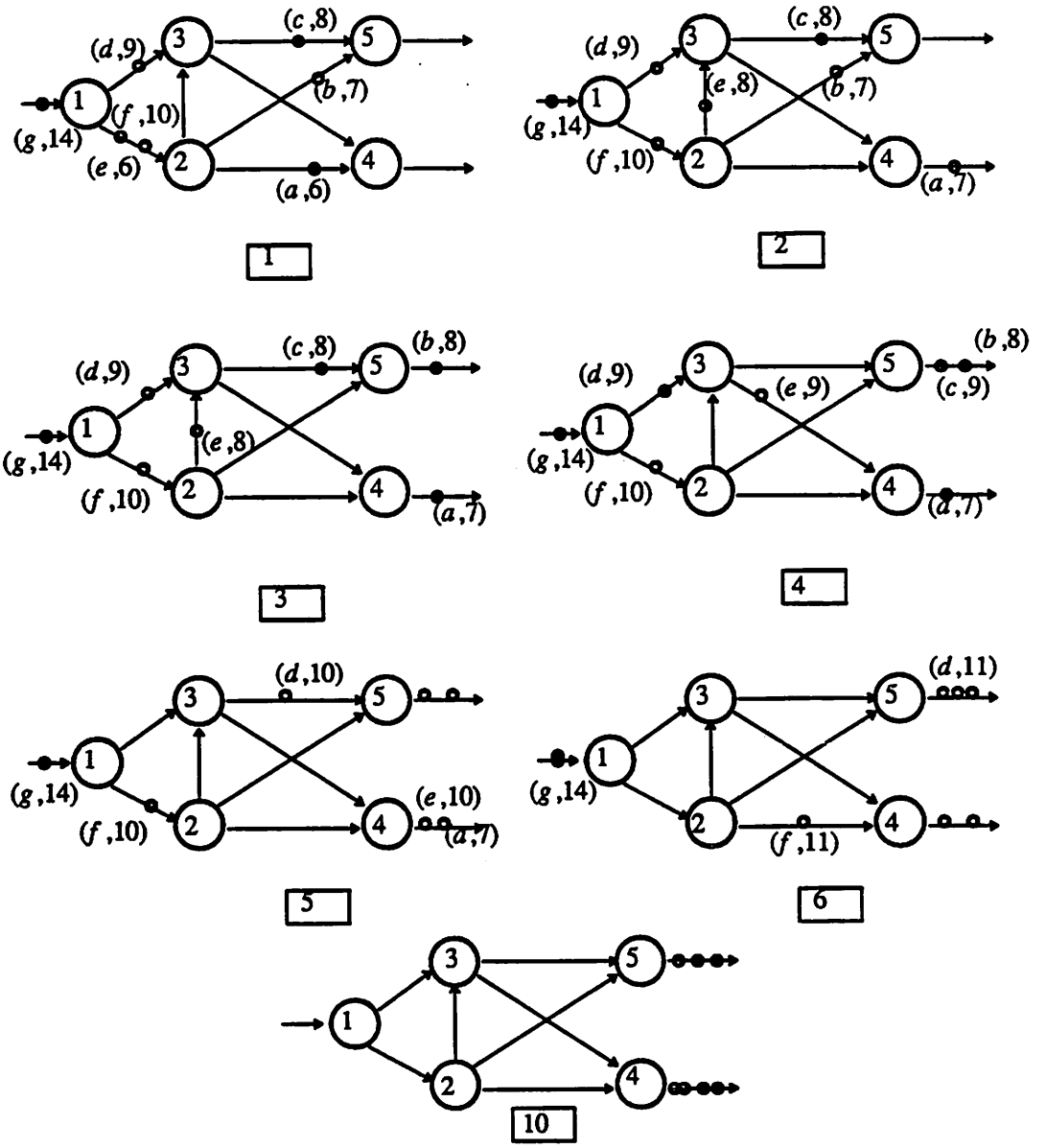


Figure 3: Synchronization of an Acyclic Network

This figure depicts the successive snapshots in the distributed computation of the events in a logical system consisting of five processors. The entire computation takes 10 time steps to complete, interleaved with time steps required for resynchronization. A ring type synchronization would be inefficient, and a broadcast communication mechanism would improve performance.

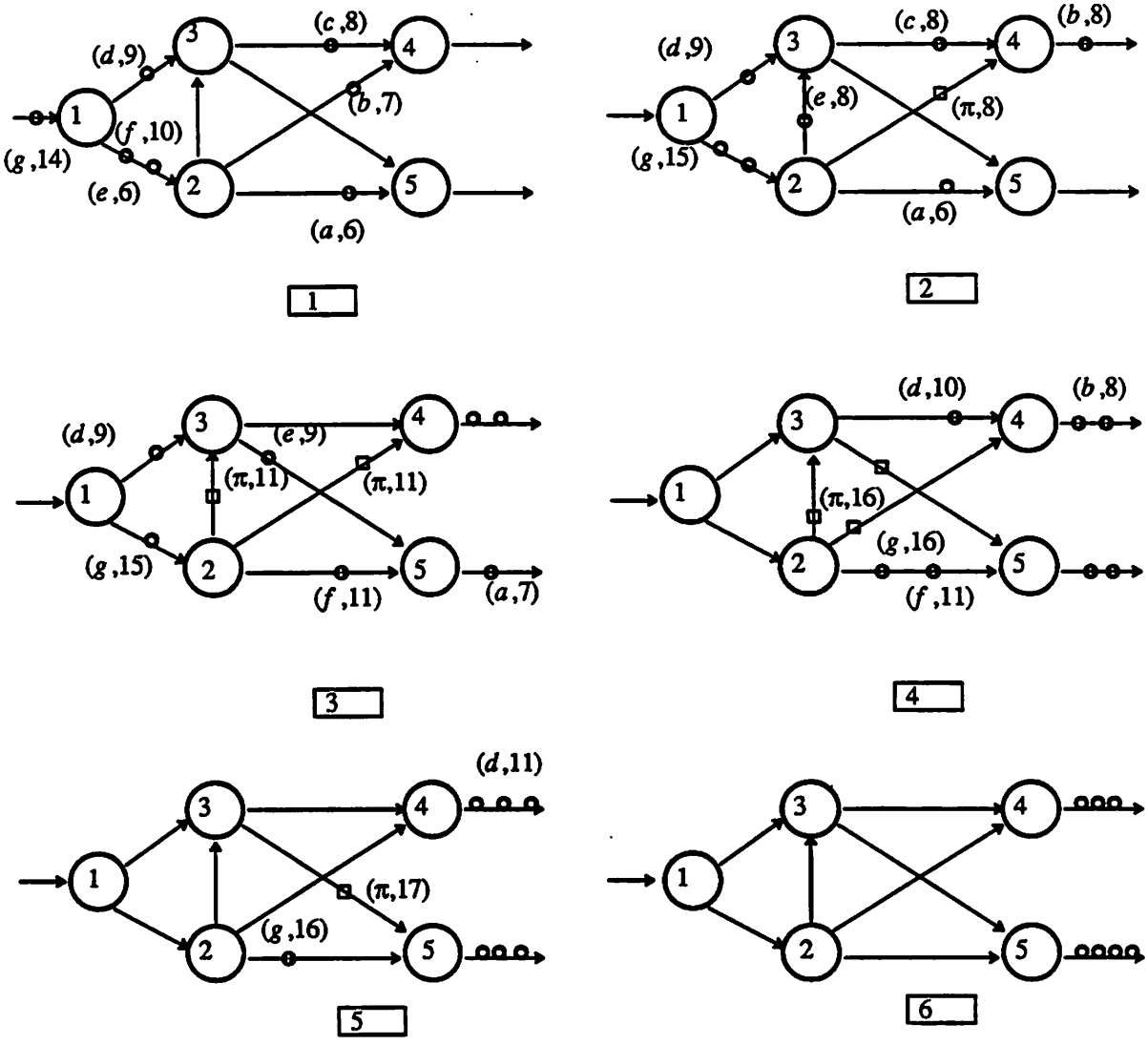


Figure 4: Synchronous Synchronization

The same computation is repeated using a synchronous computation algorithm, with specific messages used for synchronization. Here the introduction of "null" or information messages, improves the concurrency available in the system. Deadlock can also be shown to be avoided in cyclic systems. The computation proceeds faster as a result, completing in 6 time steps. These classes of methods are popularly known as conservative methods.

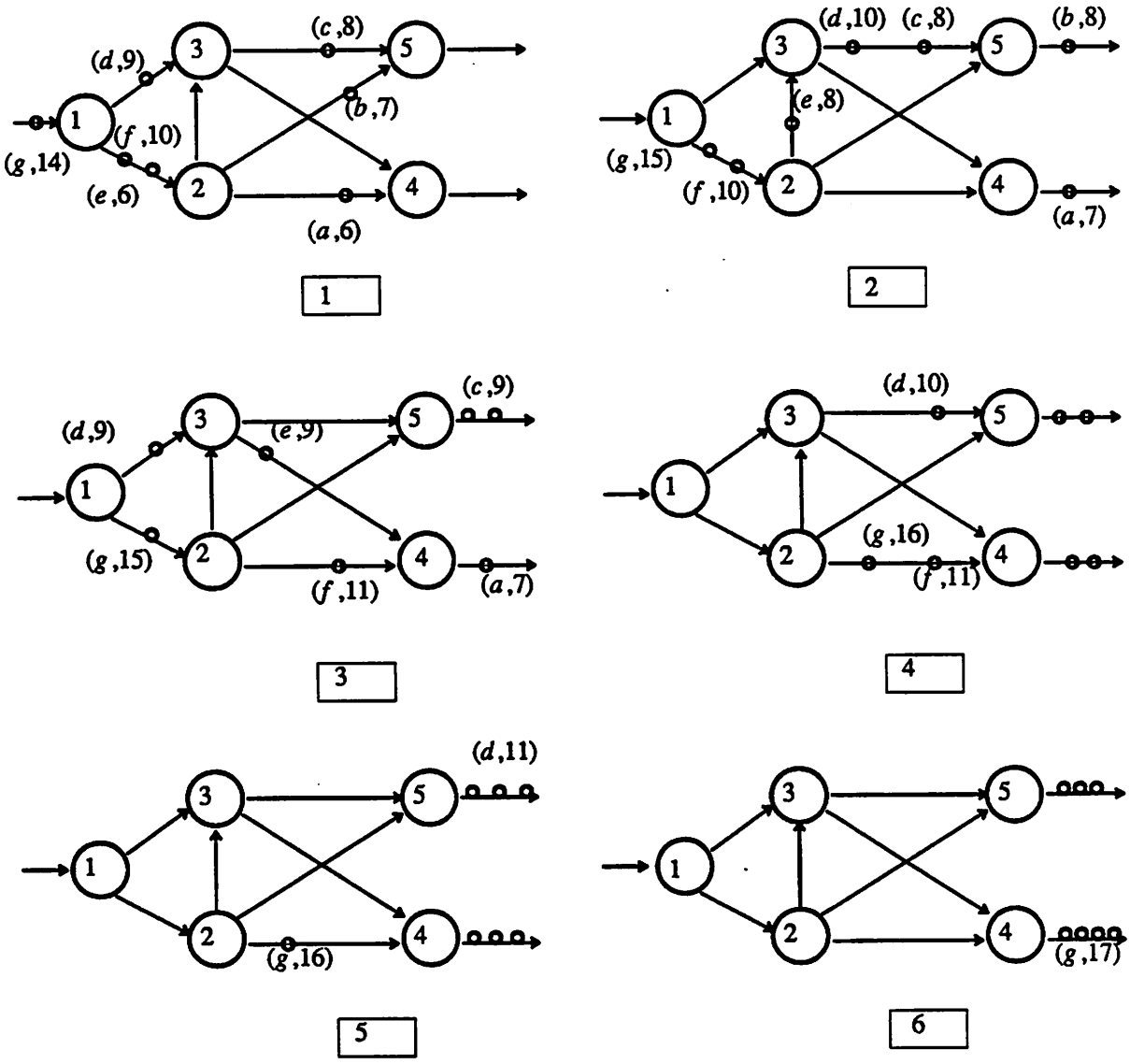


Figure 5: Asynchronous Synchronization

The simplicity of an asynchronous computation is shown in this diagram. There is no synchronization overhead, but there is additional processing whenever an error is detected. The computation proceeds, with processors executing tasks whenever a task is present in its input queue. The possibility of error is ignored, but corrective action is enforced once an error in execution is detected. In this particular example, during time step 2 and 3, the system recovers from error (the error occurring when event (*d*,9) was processed before event (*e*,8). The computation in this case also is terminated quickly. While maximal concurrency is extracted, recovery from error can penalize efficiency. Therefore, synchronization algorithms need be very efficient.

2.3. Self-Synchronizing Concurrent Computing Systems

A Self-Synchronizing Concurrent Computing System (SESYCCS), (pronounced, *say-six)*, is distributed computation on message-passing systems, where the computing system provides means for the synchronization of the distributed computation. The roles of the computation and the synchronization (i.e. control) are clearly **separated** from one another. This synchronization could be provided either through hardware (special busses, etc) or through software (in the operating system) means.

For obvious reasons, it would be of interest to identify those classes of scientific computation that would execute efficiently in a SESYCCS environment. Our studies indicate that two important classes of scientific computation lend themselves conveniently to an efficient implementation on a SESYCCS.

2.3.1. Why SESYCCS ?

As described in Section 2.2, the physical system representing the computation evolves stochastically either in discrete-time or at discrete points in time. Whatever be the nature of the physical system, the efficiency of the implementation depends on the logical system representing the computation.

In the general case where the physical system can be represented by a discrete-event physical system, we have a number of ways available to select a corresponding logical system. In a synchronous system, the logical clocks on the system move synchronously and in lock-step. The overhead in ensuring that clocks move in lock-step can be quite high and performance poor as a result [ChMi79].

In an asynchronous logical system, the logical system follows the dynamics of the subsystems contained by the physical system. The overhead in resynchronizing clocks that diverge rapidly and the excessive memory requirements for flow control can lead to poor performance [JeSo83].

We propose a theory for self-synchronization where the concurrent computing system provides the synchronization facilities. In this approach, we relax the stringent communication requirements of the synchronous case. We propose that the logical processors (or processes) in the logical system interact in a Bernoulli environment (see Section 2.3.3) Each logical processor (at some points in time) probabilistically decides whether it wants to communicate local clock information to another logical

processor in the logical system.

A number of rewards accrue from this approach. First, efficient algorithms for self-synchronization have been formulated that promise an improvement in performance over conventional methods of intra-computation synchronization. Secondly, the buffer sizes (memory requirement) are also guaranteed to be bounded in an *asynchronous* environment. The excessive communication overhead imposed on synchronous methods is also eliminated.

In Chapters 3 and 4, we derive concrete results for the performance of self-synchronization algorithms. In Chapter 5 we establish that memory requirements in a SESYCCS environment are bounded as well.

2.3.2. Self-Synchronization for Static Computation Graphs (SCG)

The first class of concurrent computation discussed in this thesis is one where the data flow in the computation can be expressed by a **fork/join** type network. A number of parallel tasks are created at a *fork* and the results merged at a *join*. $G(t)$ is a deterministic function of time, implying that these computations can be expressed as Static Computation Graphs and the synchronization mechanism can be synchronous in nature. The concurrency available in the system, therefore, varies deterministically with time, and efficiency of the computation can be poor if resources are assigned to processes without taking this temporal variation into consideration. Even if the task times are assumed deterministic, manual synchronization of such computation on the basis of *local* information alone soon becomes intractable, partly because each processor in the distributed system has to schedule resources to its processes, keeping in mind the requirements of other processors and other users as well. Our solution includes the development of an algorithm which identifies idle processors, and assigns "useful" tasks to them. These new tasks are culled from pool of tasks associated with independent programs executed by other users of the SESYCCS.

In Chapter 3, we will have the occasion to study in some detail a subset of fork/join type computation where the task execution times are assumed deterministic. However, the execution times are unknown both to the user and the SESSYCS. We will describe how the concurrent computation can be

efficiently performed when the computation is repeated over a number of instances of the input data. A perturbation algorithm allows the SESYCCS to determine iteratively the *static* structure of the computation graph based on input-output considerations alone. This knowledge of the SCG is then utilized in time-sharing the system with other independent users.

Concrete results on the performance of SESYCCS on Class **S** type computation are presented in Chapter 3, along with results from the simulation of the algorithms on test examples. The interested reader is referred to [MaMe88] for details on a Class **S** seismic migration computation implemented on an NCUBE multicomputer.

2.3.3. Self-Synchronization for Dynamic Computation Graphs (DCGs)

A typical example in the second class of scientific computation suitable for execution in a SESYCCS framework, is best described by the computation involved in distributed simulation of discrete-event systems (Chapter 6). Here both task precedences and execution times are random in nature. Efficient synchronization of such computation is very difficult even in systems with a few processors. Indeed most present algorithms scale poorly with the number of processors. We present a new clock model for such dynamical asynchronous computation, and specify how the computation evolves with time. The model presented in Chapter 4 is sufficiently general to describe a number of computational problems in this framework. We now make precise the notion of asynchronous communication, and its effect on the clock synchronization. The model presented is a specific case of the logical system for the event-driven DCG described earlier, but it is sufficiently general for the purposes of the analysis of self-synchronizing concurrent computing systems. We now wish to model the behavior of logical clocks with computing time in the distributed asynchronous implementation. C_n^i represents the logical clock of processor i at computing time step n. This model lays the foundation for the models for logical system that are developed in the chapters that follow.

In an N processor logical system, $\{ C_n^i \ , \ n \geq 0 , i = 1 , \ldots , N \}$ is defined as follows;

$\{T_m^{ij} , m = 1 , 2 \ldots \}$ are Bernoulli times

That is,

$$Prob\ [\ T^{ij}_{m+1} - T^{ij}_{m} = k\ ] = p_{ij}(1-p_{ij})^{k-1}\ ,\ k = 1\ ,2\ ,3\ \ldots$$

If

$$n \neq T^{ij}_{m}\ ;\ C^{j}_{n+1}\ =\ F_j(C^{j}_{n}\ ,\phi\ ,a^{j}_{n}\ )$$

Otherwise, if

$$n = T^{ij}_{m}\ ;\ \ C^{j}_{n+1} = F_j(C^{j}_{n}\ ,C^{i}_{n}\ ,a^{j}_{n})$$

Qualitatively, the logical system for a Dynamic Computation Graph (DCG) consists of a set of N interacting processors (or processes) whose local clocks evolve asynchronously with real time n. Each processor i executes some assigned tasks, and in addition, communicates and receives results from other processors j at random points in time, T^{ij}_{n} and T^{ji}_{n}, respectively. In our description of the logical system for a dynamic computation graph, we require that the interactions be Bernoulli in nature. This is a condition where each processor i can communicate with any processor j at the end of each time step with a probability p_{ij}. Bernoulli communications is a weak requirement on the structure of the logical system. This is because of the fact that it can model a number of other communication patterns as well (by choosing p_{ij} appropriately). In addition, it is easy to implement on a distributed computing system and guarantees finite buffer sizes in the SESYCCS (Chapter 6). C^{j}_{n} represents the state (local time) of the processor j at time step n in the computation. a^{j}_{n} is a random variable would represent the increment in the local time at time step $n+1$ if no message were received from other processors between n and $n+1$. $F_j(\cdot)$ determines the evolution of the local clock as a function of its previous local clock, C^{j}_{n}, the received remote clock information, C^{i}_{n} (or the lack of it, ϕ), and a^{j}_{n}. It is likely that processor j receives more than one message at some point, and the function $F_j(\cdot)$ incorporates all the new information available into computing the next state.

2.4. Summary

It is clear from our discussion so far, that in a system where processors differ in their rates of forward growth, inefficiency manifests itself as idle times in the faster processors in a *synchronous* environment, and as resynchronization of local times in an *asynchronous* environment. The SESYCSS, therefore, has to provide efficient synchronization to ensure that the dynamic computation graph is executed correctly and with a minimum of direct or indirect overhead.

In Chapter 4 we derive new results for the estimation of the progress of computation on a SESYCCS with two processors. Effect of communication delay is also captured in our analysis. Closed form results derived were then confirmed from detailed simulation. Algorithms for the two-processor SESYCCS were then extended to the case of a multiple-processor SESYCCS. A multiple-processor SESYCCS has a number of unique problems associated with its synchronization. We propose two algorithms for self-synchronization. The first, *Concurrent Resynchronizations* (CR), is analyzed and its performance is compared with another algorithm, *Successive Resynchronizations* (SR). The main result from this comparison is that occasional resynchronization of all the processors is far more efficient in terms of efficiency of forward computation as well as in terms of memory required by the concurrent system, than for the case where synchronization is enforced as part of the computation itself. Chapter 4 discusses role of synchronization in Class **D** computation and is self contained.

SESYCCS for both Classes **S** and **D**, as described in Chapters 3 and 4, use deterministic algorithms to synchronize computation that often had a *temporal* (deterministic or random) variation as well. A natural extension of the algorithms would be to introduce a "learning" component into SESYCCS. The cases for which this is possible are enumerated in Chapter 5, and closed form results are derived to illustrate their efficiency.

Another important result was derived in this framework. In Chapter 5, we proved that asynchronous communication is possible in a bounded memory environment. Algorithms for distributed synchronization, especially in the context of distributed simulation, assumed that communications were synchronous, that is both the sending and the receiving processors had to be ready to send and receive,

respectively. This restriction leads to deadlocks and subsequently expensive methods to detect and resolve them. We show that Bernoulli communications ensure bounded message buffers. This guarantees that SESYCCS are in fact realizable in practice.

We then study the application of these results to the problem of distributed simulation. Performance of distributed simulation of discrete event systems has been very poor, and our objective has been to make it efficient. In Chapter 6, we propose a new algorithm, Wolf, which builds upon the theory developed earlier in the thesis to provide an efficient distributed environment for distributed simulation on multiple-processor systems.

Chapter 3

Self-Synchronization for Static Computation Graphs

Scientific computation, especially in digital signal processing applications, can be represented in the form of a network of fork/join networks as depicted in Figure 1. During the "computation" phase, the fork assigns tasks in parallel to a number of processors, and then the results of these computations are "merged" appropriately by a join type network. The forks and the joins are logical processes and their functions can be carried out by designated processors in the distributed network.

As can also be observed in Figure 1, the concurrency in the computation varies with the global clocks in the system. In a distributed environment processors are self-timed. Unless efficient synchronization is provided scheduling algorithms are unable to extract the concurrency from the computation. Specifically, all the processors are not busy in every "phase" of computation. Idle processors are defined to be "inactive" in those phases of the computation. Moreover, all the processors participating in a phase of computation are not busy all of the time. Thus these processors are only partially active during the computation phase. The presence of both inactive and partially active processors represents an inefficiency in the concurrent computing system. Our objective is to schedule execution such that computing resources are not wasted, and such that these processors are able to execute tasks from other user programs as well, to offset this inefficiency.

While the objective of this chapter is straightforward, a number of issues complicate the self-synchronization of the distributed computation represented by Static Computation Graphs (Chapter 2). First of all, even if the user were willing to schedule a computation involving thousands of processors (and processes), the execution times of each task are not known a priori. In addition, the user does not know the requirements or the priorities of the other users in the same computing system.

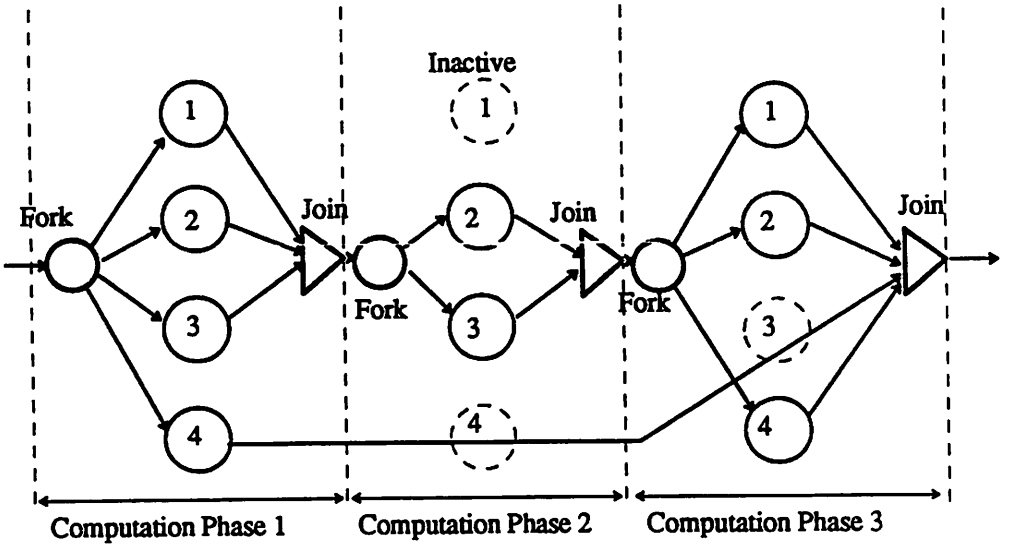


Figure 1: A Static Computation Graph

In the SCG illustrated above, the computation assumes a fork/join type structure. In each computation phase, a fork assigns tasks to a few processors in the system, while the other processors remain idle or "inactive." The objective of a self-synchronizing system is then be to identify the active and the inactive processors within each computation phase and assign useful work to them. Processor 4 is is marked active (complete circle) in Phase 1 and Phase 3, but marked inactive (dotted circle) in Phase 2. Processor 4 could just as well have been scheduled to be active in Phase 2 instead of in Phase 1. Further inefficiency can result if the task sizes within each phase are unequal, when the completion time in each phase is dominated by the worst case.

The onus of achieving the optimum use of computing resources lies, therefore, on the computing system itself. Our proposed solution involves the use of a distributed perturbation algorithm which allows the time-sharing the processors amongst the processes of a number of independent user programs. The only restriction is that these programs be represented by Static Computation Graphs (See Chapter 2).

Our approach in this chapter is as follows. In Section 3.4 we specify the structure of the Static Computation Graph (SCG), alternatively called Class **S** computation. It may be argued that Class **D** subsumes Class **S**. This is true. But the fact that Class **S** is more restrictive in structure allows us to synchronize these programs along with others in the same class in a way that allows optimum use of computing resources. We will tackle the challenging problems of synchronizing Class **D** problems in Chapter 4.

In Sections 3.7-3.9 we present a distributed algorithm that uses the traces of each execution run to identify processes that are active in each computation phase. Also identified are those processors that have additional computing resources (which can be utilized to execute independent user programs).

Later sections provide concrete results on the improvement that can be expected from using *Self-Synchronizing Concurrent Computing Systems (SESYCCS)* for solving static Class **S** problems. Specifically, we make precise the notion of critical path computation. We also describe the rewards that can be accrued by migrating processes within the same framework. The chapter concludes with a brief description of a seismic signal processing application which lends itself to computation on a SESYCCS.

3.1. Introduction

Much recent interest has been directed towards using parallel machines for fast and efficient scientific computing. Parallel machines built from inexpensive hardware can deliver performance comparable to supercomputers at a much lower cost to performance ratio for a large set of engineering problems. For best results, however, the computation has to have enough inherent concurrency to fully

utilize the parallel hardware available and reduce the *latency*.

Distributed computation is organized as a number of *cooperating* processes scattered over a number of computing processors. Processes constituting independent programs, however, *compete* with each other for resources. The computation is synchronized through a message passing network, and messages are used to communicate tasks, information and other control signals, all of which are used by processes to advance their computation. Processes typically execute a task, then communicate results to other processes and/or await results from other processes before resuming computation. Each processor has its own memory, and its processes share the resources within the processor. Each of these processes can either be "asleep" (ready to run) awaiting messages, resources or the scheduler's signal or can be "running" in memory and performing some useful task. In case more than one process is running at the same time within a processor, we assume that they share resources and have sufficient memory to execute. Most processes will be assumed to execute scientific computation; tasks which have a medium to large grain when compared to communication costs [See AtSe88]. This is because communication is expensive compared to a single instruction execution time, and this paradigm will not support fine grained applications efficiently for the same reason.

Most parallel machines at present support a *space-shared* approach to sharing processors. Each user or program is assigned a set of processors which do not overlap with the sets assigned to other processors. This approach has the advantages that scheduling tasks for execution becomes easy, considering task execution times are predictable, but the hardware utilization remains poor [See AtSe88]. Due to latency, however, the effective throughput in terms of MFLOPS could be reduced. In some applications in the distributed simulation of dynamical event-driven systems, efficiencies as low as 5% have been reported [Fu88]. We propose *time-sharing* the processors such that each user can use each and every one of the processors available in the machine. Note that time-sharing in this context refers to sharing the same processor with another set of users, and it does not necessarily imply that users use the CPU resources available in a round-robin fashion. The scheduling of this multiuser (and multiprogrammed) parallel machine can be very efficient if some algorithms presented in this chapter are used by the scheduling kernel. The scheduler works iteratively and automatically changes its schedule as the

execution of the distributed computation proceeds in a SESYCCS environment.

Each algorithm has an upper bound on its execution time, T_{seq}, that is required by sequential computation on a piece of hardware. This computation time can be reduced to T_{∞} if the concurrency in the computation is completely utilized (with an infinite number of processors solely executing the computation in parallel). However, in practice we have a finite number of processors and the presence of other processes (competing for resources) limits the execution time to at least T_N, where N is the number of processors available. Our aim is to schedule processes for execution such that the best use of the parallel resources is made while getting an acceptable execution time.

Processes communicate via messages; a process going to sleep when it has completed its active phase and transmitted updated information to successor processes. It sleeps until it receives messages with new information from all the processes that it expects to communicate with. In a synchronous synchronization environment, we expect a *monitor* process to ensure that a process is scheduled for computation only after it receives all the relevant messages from other processes. For sake of clarity of exposition we demarcate these phases in activity and sleep as the *active* and *inactive* phases of the process respectively.

The algorithm proposed in this chapter is a two-pass algorithm. In the first pass, the algorithm determines the information it needs from the data flow in the execution of the user program, and in the second pass it determines a dynamic schedule for all the user processes in each processor using the information extracted in the first pass. Our processes are assumed to execute scientific computation that can be described by a Static Computation Graph (SCG). The schedule is automatically modified to include the generation of new-user processes and termination of old-user processes.

3.2. Message Passing Parallel Machines

In this section, we describe a typical message passing parallel machine, which consists of a set of computing processors (P) interconnected via a message passing network (IN). Each processor is a specialized floating point processor capable of executing computation assigned by a user program in a high

level language (e.g. FORTRAN or C), and has its own private memory. The private memory is typically organized into the *user memory (UM)*, *system memory (SM)*, and the *message memory (MM)*. The system memory is used by the operating system on each processor to handle inter-processor message communications, scheduling, interrupts and other low level system functions.

The message memory (MM), is used to store messages for transmission to other processors in the system. Communications are usually implemented using an asynchronous handshake protocol. The user memory (UM) stores the user program and data. Typically the UM is very small (1/2 Mb), and secondary access may be necessary for large programs. The local variables are stored in the UM, and when messages are to be communicated with neighboring processors, the operating system is invoked to copy the variables into the MM, and then transmits the message to the MM of the target processor, where it is subsequently read by the destination process. The interconnection network (IN) is usually a hypercube or a torus routing network [See DaSe85]. Message passing between non-neighboring nodes, implemented using a virtual cut-through type algorithm, is very efficient, making message costs for multihop transmission comparable to those of nearest-neighbor communications.

Messages sent between processes are identified using the following fields. The control fields identify whether the message is to be treated as data or a migrated process region.

For *read* calls the following format is used;

[buffer, length, source, type, control]

For *write* messages, the format used is;

[buffer, length, destination, type, control]

Here **buffer** and **length** refer to the length and address of the message buffer which stores the message. The **source** and **destination** fields are process id's and **type** is a qualification to discriminate between dissimilar messages communicated between the same pair of processes.

3.3. Static Computation Graphs

The parallel program is developed with the architecture of the message-passing machine in mind. The algorithm is broken down into a number of smaller parts, each of which is defined as a process and assigned to a separate processor. These processors carry out tasks which are described in a high level user program usually written in FORTRAN or C, and proceed with computation. At the end of the first active phase, the process communicates the results of its computation to some of the other processes, and awaits the receipt of messages, if any, from other processes. These processes complete their own active phases before transmitting and receiving messages, but this is not necessarily the case. Most scheduling is done locally on each processor, the initial mapping being arbitrary. This, coupled with the fact that parent processors may be loaded down with other user programs, results in most processes "waiting to run in memory" for unpredictable amounts of computing time. As a consequence there is an underutilization of processing resources, unacceptable response times and loss in concurrency. The active phases themselves cannot be determined by each processor as they depend on the run times of other processes (which in turn depend on unrelated user programs).

Local information available in each processor is not sufficient to handle the scheduling of users with differing priorities, or to handle migration of processes to another processor when resources are available, or to minimize idle times when the computation is repetitive in nature.

It is for these reasons that conventional parallel machines have adopted the simpler space-shared approach to implementation of parallel computation. In this chapter, we propose an effective automated approach to determining an optimal schedule for time-shared parallel machines in a SESYCCS environment.

As mentioned in Section 3.2, the synchronization of the distributed computation follows a two-pass procedure. The first pass suspends execution of other user programs, and concentrates on a single target user program. The host then determines an activity graph for each process, and this information is used to determine *a* schedule for this program and then the same task is repeated with other user programs, each time scheduling a program taking into account the requirements of *other* user programs

already scheduled. A more comprehensive description of the advantages and limitations to our algorithms will be given in later sections (3.7-3.9). The next few sections describe some properties of the *fork/join* type of network that provides a concise description of the static computation graph.

Let us now describe the mechanisms which allow the implementation of the transition of a process from its inactive phase to its active phase. A process "sleeps" while awaiting a specific number of messages from a set of cooperating processes. However, the scheduler may delay scheduling of the active phase until such time it thinks it is appropriate to do so. Once the static structure of the computation is determined, the resources usually assigned to a user program in its inactive phases can alternatively be assigned to another independent user program that can utilize them.

3.4. Structure of Static Computation Graphs

Static Computation Graphs as described in Chapter 2 have a *static* precedence structure which distinguishes them from the class of Dynamic Computation Graphs. This additional restriction allows the development of a number of algorithms which exploit their structure to optimize computer resource assignment. The basic idea is as follows. If a computation graph has a static structure that requires it to restrict the use of computing facilities to only certain periods of time, the computing resource that is unused for the remaining period of execution may as well be utilized by another independent program. In a uniprocessor case this leads naturally to the concept of pipeline-interleaving. In the multiple processor domain , however, efficient algorithms need be developed so that a number of processors in the logical system are able to coordinate and time-share computing resources with other independent programs executing on the system.

We will make precise these notions by developing the structure of SCGs in the form of fork/join type networks. A join-type network collects the results of a number of independent computations from input processes (or sources) $S_1(t), \cdot, \cdot, S_n(t)$ and performs some computation on the aggregate before redirecting the result to another set of processes via a fork-type network. The join-type network has no control over the sources to its inputs, it needs buffers to store inputs until such time t_n^* when inputs from

all sources arrive at the join-type network. In other words, the join type network needs an input value from each of its buffers before it can produce any output. The join-type network then takes one piece of data from each of its buffers and merges them together into one output result. This implies that each of the buffer sizes in the join-type network will be reduced by one. Let us suppose for the sake of argument that one of the input sources $S_1(t)$ generates inputs to the join-type network at a much higher rate than other sources. This means that the buffer corresponding to $S_1(t)$ will grow very rapidly in size with a possibility of overflow. We would therefore like to know when buffer sizes can be guaranteed stable. We can then ensure that SESYCCS for SCGs are realizable in practice.

A Join-Type Network (J-TN)

Consider the network illustrated in Figure 2. Sources $S_1(t), S_2(t), \ldots, S_N(t)$ provide N inputs to their respective buffers $b_1(t), b_2(t), b_3(t), \ldots, b_N(t)$. The buffers operate like a first-in-first-out stack, where data is taken in and out in order of arrival times at the buffer. The buffer $b_i(t)$ at time t takes the value of its most recent input. For example, if the n^{th} input, a, to i^{th} buffer arrived at time t_n, then $b_i(t)$ takes the value a for the entire time interval until the arrival of the $n+1^{th}$ input at time t^i_{n+1}.

Let us assume that the inputs are Poisson. This assumption implies that the interval between successive data inputs are independent. This is routinely used in queueing analysis to model behavior of inputs when little else is known about their statistics. The sources S_i transmit at Poisson rates λ_i, with the n^{th} arrival at time $t^i{}_n$ to the i^{th} buffer. For $t \geq t^*_n$, each buffer has at least n input data points. We quantify our model of the J-TN as follows,

$$v_i(n) \doteq b_i(t) \cdot I\{t^i{}_{n+1} > t \geq t^i{}_n\}$$

$$\text{For all } n, i, \text{ if } t < t^i_n, \text{ then } v_i(n) \doteq \phi.$$

$$Y(t+\theta) = \sum_{n=0}^{\infty} (f(v_1(n), v_2(n), \ldots, v_N(n)) \cdot I\{t^*{}_{n+1} > t \geq t^*{}_n\}$$

where

$$t^*{}_n \doteq \{\min t \mid v_i(n) \neq \phi, i \in \{1, N\}\}$$

and $f(v_1(n), v_2(n), \ldots, v_N(n))$ is a program which operates on the data in the buffers and I is the indicator function which takes value one (zero) when the condition enclosed in the { } is true (false). Qualitatively, the output is produced only when all the input buffers have at least one input so that $Y(n) = f(v_1(n), v_2(n), \ldots, v_N(n))$. Here, $Y(n)$ is defined in a manner analogous to $v_i(n)$, and is the n^{th} output from the J-TN. θ is the processing time required by the J-TN to generate an output given the inputs. The standard multiplexer is a degenerate example of a J-TN.

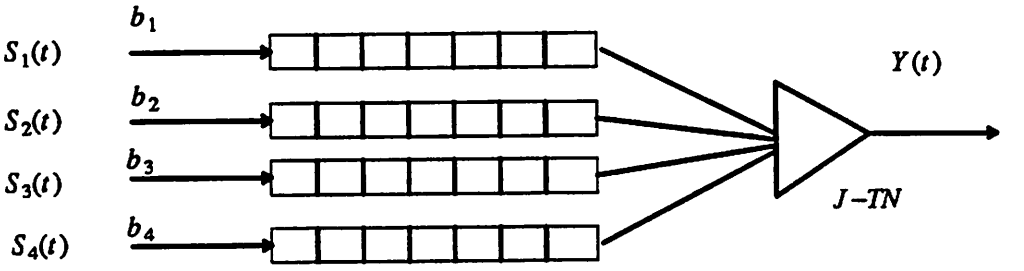


Figure 2: A Join-Type Network

A Fork-Type Network (F-TN)

A Fork-Type Network takes in one input and outputs N outputs. A Fork-Type Network is illustrated in Figure 3. A F-TN has a single input buffer $v_1(n)$ and N outputs $Y_i, i = 1, 2, \ldots, N$ and the outputs are related to the inputs according to

$$Y_i(t) = \sum_{n=1}^{\infty} f_i(v_1(n)) \cdot I\{t^*_{n+1} > t \geq t^*_n\}$$

In general, we could provide for a *selector set* S which determines which of the outputs $Y_i(t)$ would change, enabling a probabilistic routing, varying with time, from the inputs to the outputs. A standard demultiplexer is then a special case of the the F-TN, where each successive input (in time) is routed to a successive output link (in space).

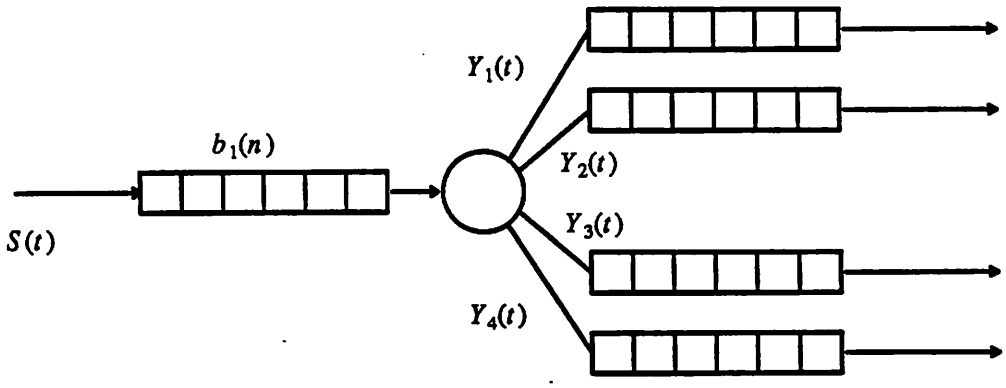


Figure 3: A Fork-Type Network

A Simple Process

A simple process p is one which accepts an input, x (n), and produces an output, y (n). If P is the processor which assigns c_p fraction of its CPU time to p then t_p is defined to be the time taken by p to produce an output. A simple process communicates with other processes only when it has completely processed the data assigned to it. (Figure 4).

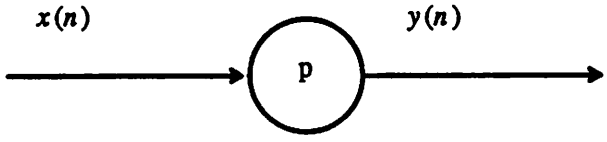


Figure 4: A Simple Process

A Compound Process

A compound process is composed of a number of processes interacting with each other. Each of these processes is spawned by an independent multitasking processor which assigns a certain fraction of its CPU time to its child process. An example of a compound process is shown in Figure 5 where processes $p_1, p_2, \ldots, p_n$ interact to process input data. The data flow allows for feedbacks, and multiple subsequent visits by the input to each process. The processes may generate shared variables which are transparent to the input-output specifications. We will discuss these compound processes further and develop their properties. Most multicomputing environments can be described with compound processes.

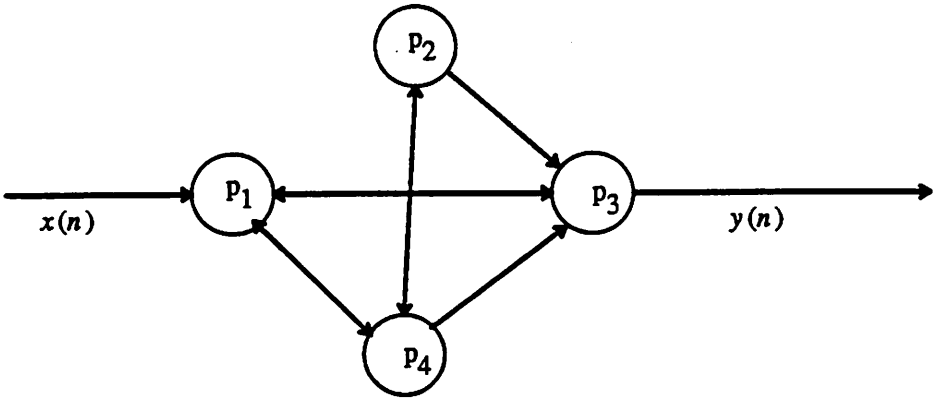


Figure 5: A Compound Process

A Simple J-TN (SJ-TN)

A Simple J-TN is a J-TN whose inputs are all simple processes. The SJ-TN processes the data from each of the individual processes to produce an output according to the equations for the SJ-TN.

A Compound J-TN (CJ-TN)

A Compound J-TN is one which has at least one input that is a compound process.

Pipes

Pipes allow for processes to communicate in a specified order. Output from one process is fed into the input of another process through a pipe.

At this point, we digress and develop a few properties of the J-TN that would be useful in subsequent analysis of static computation graphs. In effect, we will be deriving some stability results that will determine if SCGs are realizable in practice.

3.5. Properties of Static Computation Graphs

Static Computation Graphs would be realizable in an implementation if and only if the buffer sizes at the J-TNs are finite in size. We will now examine the dynamics of buffer sizes with different types of input. While it is not surprising to observe that buffers can overflow when the input sources emit data at unequal rates; it is also true that the J-TN is unstable even if the inputs rates are equal

(under some conditions). We will proceed immediately to establish this fact.

Stability: Consider a J-TN with two Poisson input sources, S_1 and S_2 as shown in Figure 6. If the buffer sizes in b_1 and b_2 are given by $|b_1|$ and $|b_2|$ then a J-TN is stable if and only if (b_1, b_2) is a **Positive Recurrent** Markov chain. Therefore, a J-TN is unstable if its expected buffer size is infinite.

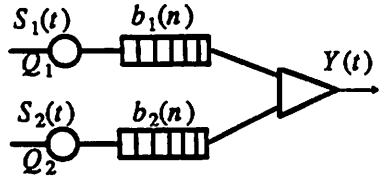


Figure 6: A J-TN with two inputs

Proposition 1:

Let the sources to a J-TN be S_1 and S_2. If the sources are independent and Poisson, then the J-TN is unstable for all positive rates $\lambda_i > 0$.

Proof: Let us assume that the Queues Q_1 and Q_2 have service rates, μ_1 and μ_2 and let the inputs S_1 and S_2 be independent Poisson sources.

Case 1: $\mu_1 = \mu_2 = \infty$ and $\lambda_1 = \lambda_2 = \lambda$

The proof is as follows: If b_1 and b_2 are the buffers from sources 1 and 2, then $|b_1| - |b_2|$ follows a one dimensional random walk with the following transition probabilities.

$$P_{k,k\pm 1} = \frac{1}{2}$$

The underlying Markov chain is, therefore, **null recurrent** [See Wa88]. This also implies that $||b_1| - |b_2||$ grows without bound for all positive rates. (The probability of $||b_1| - |b_2||$ becoming zero is one, however, the expected time for this to happen is infinite.)

Case 2: λ_1 is not equal to λ_2

In this case, the underlying Markov chain is transient and hence, the J-TN is unstable.

Therefore, we can conclude that independence of the two Poisson inputs is a sufficient condition for instability. **Q.E.D.**

This interesting result was derived with very few assumptions other than Poisson flow. Deterministic flows, for example, do not result in instability if their average rates are same.

Proposition 2:

The J-TN with inputs routed probabilistically from the output of a M/M/1 queue is unstable.

Proof: From the sampling theorem for Poisson processes [Wa88], the sampled output processes are independent and from Proposition 1, this implies instability.

Q.E.D.

Theorem 1:

The J-TN fed by N Poisson sources with rates λ_i is unstable for all positive rates if any two inputs are mutually independent.

Proof: Follows from Propositions 1 and 2.

3.6. Discussion on Join-Type Networks

Theorem 1 implies that a J-TN is not realizable in practice with finite buffer sizes if its inputs were independent and Poisson. It is conjectured that the J-TN is unstable for all random inputs with independent inter-arrival times (independent increments). This can be visualized by a trivial example. Consider a simple J-TN with two inputs. The interarrival probability density function has mean a and variance σ^2. A sequence of arrivals can then be constructed, $(a-\sigma), 2(a-\sigma)\ldots$ from input 1 and a sequence of arrivals at times $(a+\sigma), 2(a+\sigma)\ldots,$ from input 2, which leads to unbounded buffer sizes. It would, therefore, be impossible to assign buffer sizes for J-TNs with independent inputs. It is because of this that we require that task execution times be deterministic or atleast predictable to a some extent (see later sections for experimental results).

3.7. Synchronization of a Simple J-TN

A J-TN consists of a join of the outputs of a number (say, N) of simple processes. Each of these processes, p, is spawned by a processor which assigns a fraction of its computing time, c_p, to it. Heavily utilized processors can assign small c_p, if at all, while lightly loaded processors can devote a higher fraction of their computing time to processes. In a time shared multicomputing environment, the number of processes per processor is also variable. An efficient assignment algorithm would have the capability to track such variations in load in the context of the concurrent program. If the constituent processes (in a J-TN) produced outputs at varying rates, buffering would be difficult, and efficiency would be lost due to enforced idle times (Sections 3.4-3.5). The algorithm for resource allocation in a J-TN is straightforward. We will briefly outline it.

let us assume that the process p, consumes c_p and takes a processing time t_p to complete its task. Let $T = (t_1, t_2, t_3, \ldots, t_p, \ldots, t_N)$, and $C = (c_1, c_2, \ldots, c_p, \ldots, c_N)$ and there exists a vector $\Lambda = (\alpha_1, \alpha_2, \ldots, \alpha_p, \ldots, \alpha_N)$, such that

$$\Lambda = C \cdot T$$

We now define $\rho = \max_k(\frac{\alpha_k}{c_k})$ and the reassigned processor fractions, c_p^{new}, for each process will then be given by

$$c_p^{new} = \frac{\alpha_p}{\rho}$$

ρ keeps track of the "bottleneck" process, and the J-TN can be dynamically scaled up or down depending on the value of ρ. Process migration can further enhance the speed of completion, if the processes with the largest α_j are assigned to processors which can assign the highest c_j's. In other words, the computing fractions of the processes that have a small computing time requirement are reduced so that the task completion times (in all the processes) are equal to each other.

3.8. Synchronization of a Compound J-TN.

In this section, we consider the case of a general CJ-TN which is a join of compound processes (Section 3.4). A *compound* process was defined as one having a number of cooperating and concurrently executing processes. Regardless of the interconnection, the output of the compound process is joined with the output of other compound processes in a compound J-TN (CJ-TN). The CT-JN allows us to consider medium-grained model of computation useful for most scientific applications. We assume no specific knowledge of the tasks carried out by each process is available to the CJ-TN, other than the input/output constraints and task completion times. Our proposed stabilization algorithm is independent of the programs executed subject only to some assumptions which will be described below. Each of the sub-processes p executes a different task and the dependencies between the sub-processes make calculations for optimal CPU allocation intractable.

Our algorithm is best described by the *Activity Plot*, Figure 7a, which plots the Static Computation Graph versus time. The shaded regions, Figure 7b, indicate when a process is active. The dependencies are depicted by arrows. In principle, all active regions of a compound process are not observable from the execution completion times alone.

3.8.1. Synchronization Algorithm for Compound Process

In this section we describe the *checkpoint* algorithm which optimizes the processor utilization amongst the processes constituting the compound process. Each process while executing its assigned task communicates with other processes within the same compound process. Each process within the compound process can be assigned a maximum fraction of CPU by its parent processor depending on its multiprogramming requirements. The algorithm is an adaptive one, and involves repeated invoking of a subroutine called the *checkpoint* which provides information regarding the efficient allocation of processor fraction for a *certain fraction* of the total execution time. The outcome of the checkpoint algorithm is a list of *active* processes for nonoverlapping intervals of time (t_i, t_{i+1}), $A_{t_i, t_{i+1}}$ for $i = 1, 2, 3, \ldots, Z$ such that the union of the intervals is $(0, T)$, where T is the total time required for execution.

The idea behind the use of the checkpoint algorithm can be summarized as follows. Each process p is assigned a certain computing fraction c_p by the parent processor. We wish to determine if c_p is really necessary for the entire execution time. If the resource were unused for some known intervals of time, then taking it away from the process would not affect the total computation time. So the checkpoint algorithm choses an interval of time (q_1,q_2) that is a subset of the total execution time, and then changes (decreases) the service fraction c_p of each process p by a fraction σ for the duration of that interval. The change in total computation time T is then observed. This change called ΔQ_p, should equal $\sigma \cdot \Delta Q$ if the process actually needed the computing fraction c_p during the interval of time (q_1,q_2) (where $q_2 - q_1 \doteq \Delta Q$). In other words, if the total execution time is unaffected by the decrease in assigned resource to a certain process for a certain interval of time, it is likely that the process is inactive in that interval. The unused resource can instead be assigned to some other process. This idea is now made precise as follows.

subroutine Checkpoint (q_1 , q_2 , A , IA , PA , ε , T)

/* The initial processor assignment is $\underline{C} = (c_1, c_2, \ldots, c_N)$

/* Defining the operator ν_p $\underline{C}$ as follows*/

$$\nu_p \cdot \underline{C} \doteq (c_1, c_2, \ldots, (1-\sigma)c_p, \ldots, c_N), \text{ for } t \in (q_1, q_2) \cap (0, T)$$
$$\doteq \underline{C}, \text{ otherwise}$$
$$\Delta Q_p \doteq \alpha \nu_p \underline{C} - \alpha \underline{C}$$
$$\Delta Q \doteq q_2 - q_1$$
$$\gamma_p \doteq \frac{\Delta Q_p}{\Delta Q}$$

Each process, p, is assigned to the set of active states A_{q_1,q_2}, the to set of inactive states IA_{q_1,q_2}, or to the set of partially active states, PA_{q_1,q_2}, according to the following rule:

$$\sigma - \varepsilon < \gamma_p \leq \sigma, p \in A_{q_1,q_2}$$
$$\varepsilon < \gamma_p \leq \sigma - \varepsilon, p \in PA_{q_1,q_2}$$
$$0 \leq \gamma_p < \varepsilon, p \in IA_{q_1,q_2}$$

ε is a postive fraction that is small compared to σ.

The Checkpoint Algorithm

Step 1: The time interval $(0, T)$, determined from the initial trace execution with processor assignment C is divided into two intervals $(0, \frac{T}{2})$ and $(\frac{T}{2}, T)$. The *checkpoint subroutine* is invoked for both these intervals. The sets A, PA and IA are returned by the subroutine. If the cardinality of PA, $|PA|$ is non-zero for either interval, it is further divided into intervals, $(0, \frac{T}{4})$ and $(\frac{T}{4}, \frac{T}{2})$, or, $(\frac{T}{2}, \frac{3T}{4})$ and $(\frac{3T}{4}, T)$ as the case may be. The checkpoint subroutine is invoked for each of these intervals and the process of iteration repeated if $|PA|$ for any of these intervals is non-zero. The algorithm continues until all the PA s are zero, and we have an active set A for each interval. The algorithm is guaranteed to converge, with the least possible checkpoint interval being $(0, \varepsilon \Delta Q)$.

Step 2: At the end of Step 1, the CJ-TN has the Activity Sets for a number of non-overlapping intervals whose union is $(0, T)$. For each interval the *active* processes for that interval would cooperate in finding out the bottleneck. Each processor p, scales its c_p to suit the bottleneck process for that interval. As few processes are active in any interval, we expect good performance in the sense that the effect of a bottleneck process at a certain time does not penalize the entire execution. The CJ-TN (which could be modeled by a host processor) uses the data available from the parent processors, to calculate the new assignments. Once the efficient assignment of CPU to the compound process in a CJ-TN is completed, the entire CJ-TN is controlled using the scaling algorithm prescribed for a Simple J-TN.

Step 3: After the steps 1 and 2, the algorithm identifies only those regions of a process that are *on* the critical path. Tasks executed off the critical path are not identified, and these tasks can only be observed by reducing their processor assignment fraction until they lie on the critical path (as well). Once all active tasks are scheduled to lie on the critical path, resource allocation becomes simplified.

At the end of the steps 1 and 2, the checkpoint subroutine has demarcated the total execution time $(0, T)$ into *active* intervals (q_1, q_2), (q_2, q_3), ..., (q_{N-1}, q_N) each with a set of processes that are active within each interval. The processes, which are not active could either have no assigned tasks, or they could be executing off-critical path (but useful) computation. The algorithm for scheduling the *latter* processes begins with the examination of the intervals closest to the termination of the execution, (q_{N-1}, q_N).

The interval (q_{N-1}, q_N) is assigned the *level* 0. The next interval (q_{N-2}, q_{N-1}) is assigned the level 1 and so forth for the other intervals. The computing fraction, c_p, for $p \in IA_{q_{N-1}, q_N}$ is set to zero. If the change in total execution time, ΔT, is also zero, then the next interval with level 1 is examined. The computing fraction for this interval is again set to zero and the resulting effect on T is observed. If ΔT is zero, the next higher level is examined, until level i is reached where setting the processor computing fraction to zero results in $\Delta T > 0$. At this point, the computing fractions of p in all levels 0 through i are slowly increased from zero until ΔT again exceeds zero. The checkpoint subroutine is then used to determine the active region of p in the levels 0 through i.

The procedure is continued again as before, but this time the level 0 is assigned to interval (q_{N-i-1}, q_{N-1}). and the checkpoint subroutine is again used to demarcate the active regions.

This procedure is carried out for all the inactive processes ensuring that they compute efficiently.

This procedure is illustrated in Figure 7, where (a) illustrates the activity of different processes over time. The arrows represent asynchronous communications. At the end of the checkpoint algorithm, only those process segments which were active on the critical path are demarcated (by the shaded regions). The other regions are left blank, as they have not yet been identified by the scheduler. Step 3, then identifies process 5, as being active, when decreasing the initial computing fraction extends the region J into the level 1 of the procedure. The checkpoint algorithm then identifies the active regions of process 5 as depicted in (c). In the next step, process 4 is examined, where the execution of region G is completed only when it is needed by segment F. This delay puts G on the critical path as well. The checkpoint algorithm then finds the active processes for the entire execution graph, (d). The

inactive regions can then be assigned to other independent users.

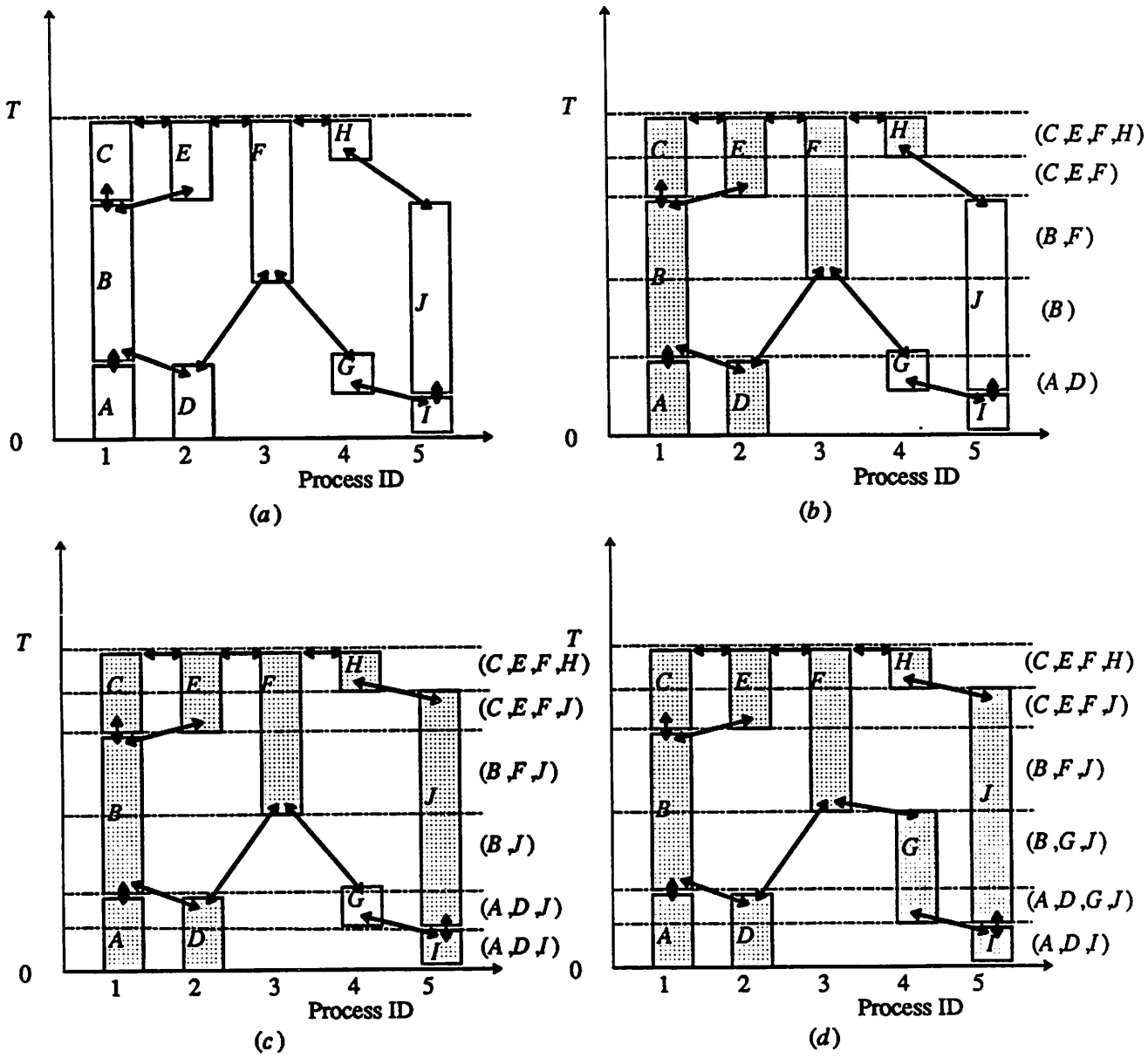


Figure 7: Description of the Checkpoint Algorithm

Proposition 1:

Let the events executed by the active processes in the interval (q_2, q_1) be J_{q_2,q_1}. The time required for processing these events is the $f\ (J_{q_2,q_1}, \underline{C}) = q_2 - q_1$, where $\underline{C}$ is the resource allocation vector and $f: \mathbf{Z}\ X\ \mathbf{R}^N \rightarrow \mathbf{R}$. Let $\underline{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_N)$, then a sufficient condition for

$$f\,(J_{q_2,q_1}, \underline{\alpha}.\underline{C}) < f\,(J_{q_2,q_1}, \underline{C})$$

is that $\alpha_i > 1$ for all i.

Justification:

The proof of sufficiency is immediate. However, the fact that it is not a necessary condition may not be obvious.

Let $p_1, p_2, \ldots, p_5$ execute events J_{q_2,q_1} in the interval (q_2, q_1). Let the processes p_1, p_2, p_3, p_4 execute events $J_{q_2,q_1} - E$. E is the event assigned to process p_5 which records the progress of the execution of events among the other processes. Therefore any vector $\underline{\alpha}$ with $\alpha_i > 1$ for $i \in (1, 4)$ and $\alpha_5 = 1$ will lead to a smaller $f\,(J_{q_2,q_1}, \underline{C})$.

Proposition 2:

An event a which belongs to the set of inactive processes for an interval (q_2, q_1) can be scheduled to occur independently of other events in the interval which are associated with processes that are active, i.e. $p \in A_{q_2,q_1}$.

Justification:

If an event a which is scheduled by an inactive process could affect an event scheduled by an active process, then perturbing the service rate of the inactive process, would affect the total execution time. Hence, the events associated with inactive processes are independent of other active processes during the interval (q_2, q_1).

An important consequence of this observation is that, the inactive processes in each interval, can be assigned tasks which are from another independent program or user. The new user program is also subjected to the same algorithm to determine its active phases, and resources assigned such that the independent programs time share the common resources.

Theorem 2:

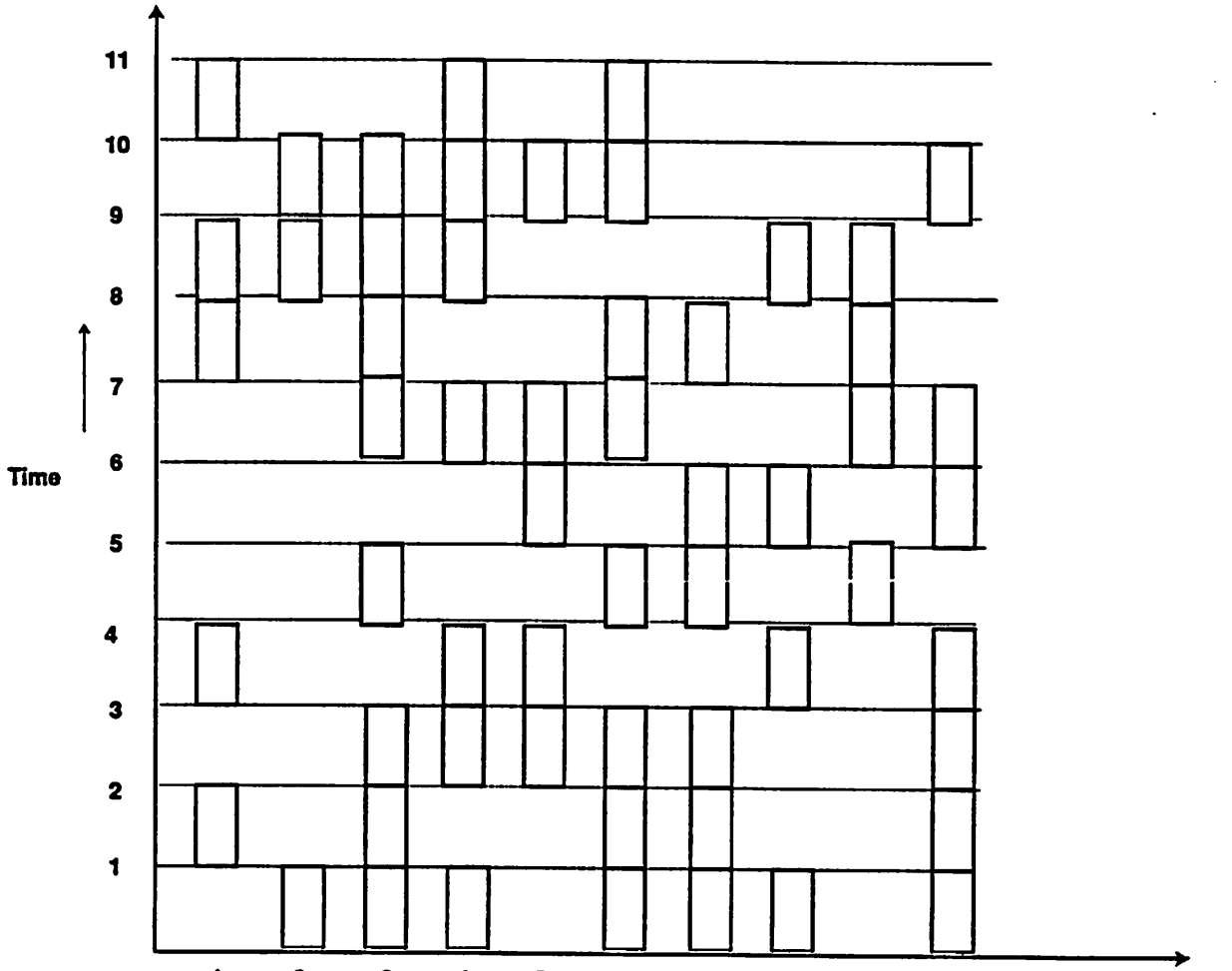
Every process p, when active for an interval (q_1, q_2), $p \in A_{q_1, q_2}$, is on the critical path for that interval.

Justification:

By construction, a process is classified as active if perturbation of its assigned fraction results in a perturbation of a like amount in the total execution time. Therefore *every* process is on the critical path whenever it is active. It shares the critical path with all other processes which are also active within the same interval.

This is one of the main results of this section. The checkpoint algorithm does not complete a task until it has been declared active (See Figure 8). Processes are scheduled to run only when they are active, and concurrent processes which are scheduled to run at the same time are shifted along the time axis until they are on the critical path. Therefore, after scheduling with a checkpoint algorithm, every process is on the critical path, slowing down one process can slow the total execution time. Likewise speeding up a single process cannot, in general, speed up the execution, unless all concurrent processes in the active interval are also scaled up proportionally. Therefore, the programmer need only specify the interconnection within a compound process, leaving optimal allocation and execution to the checkpoint algorithm.

We analyze the performance of our algorithms for a simulated activity plot shown in Figures 8-12. The test pattern was generated using a random number generator and represents the activity in the static computation graph when each process was assigned a $c_p = 0.1$. With a probability 1/2, the active regions extend into the next checkpoint interval. The process fractions for two cases are assigned from a uniform distribution $U(0, 1)$. The successive graphs depict the performance on a ten processor system, with and without process migration and with and without checkpoint optimization. The execution times are assumed deterministic. The optimization proceeds interval by interval, when the processors can upgrade their individual c_p's to the largest value such that the cardinality of the set of active processes for that interval remains unchanged.



Case 1:	0.3	0.2	0.8	0.3	0.5	0.8	1.0	0.1	0.4	1.0
Case 2:	1.0	0.3	0.2	0.8	0.3	0.5	0.8	1.0	0.1	0.4

Figure 8: A Test Example

This figure shows the activity graph for a test example described in the text. The active phases of each processor are plotted against the total execution time. The active regions were generated using a Bernoulli distribution with $p = \frac{1}{2}$. The maximum processor computing fractions available for each process was assigned (for both the cases) using a uniform probability density function. The activity plot shows the active phases when the computing fraction assigned to each process was 0.1.

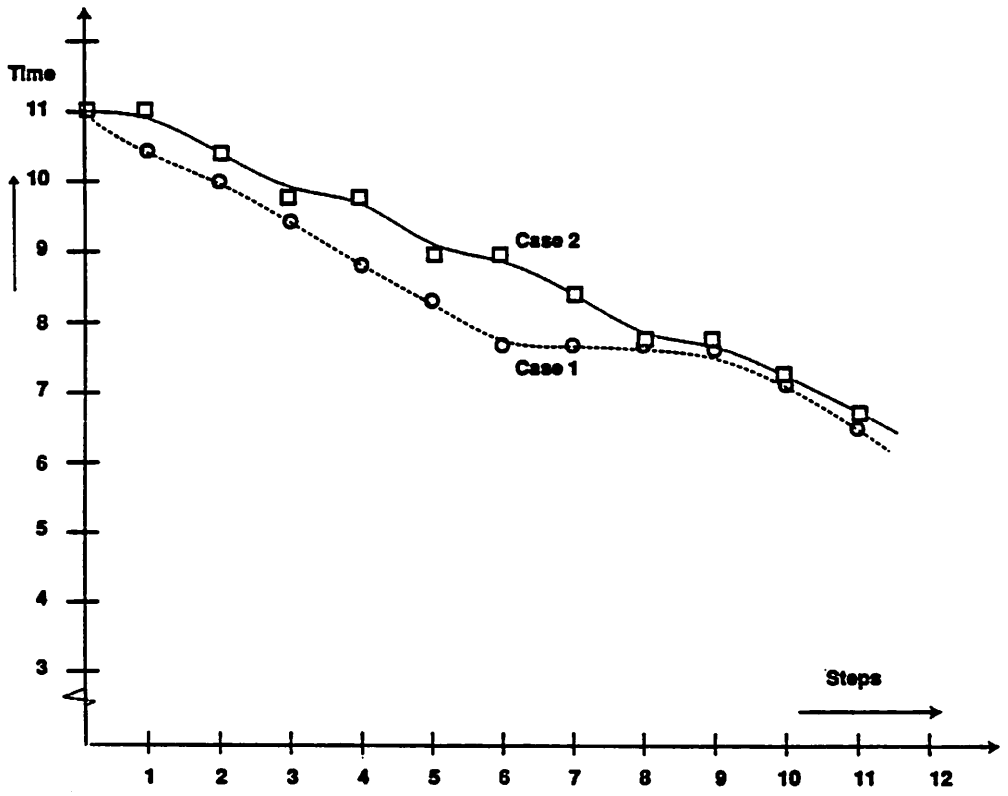


Figure 9: Optimization Using the Checkpoint Algorithm

This figure shows the decrease in execution time possible for both cases in Figure 8, using the scaling methods described in Chapter 3.

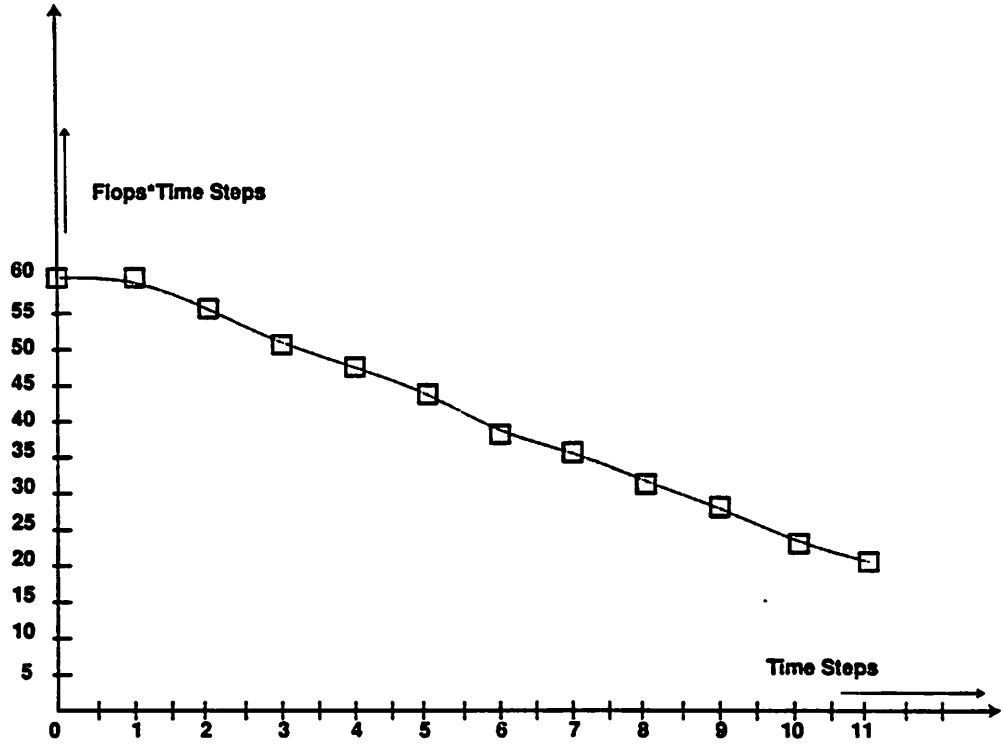


Figure 10: Optimization of Computing Resources

This figure represents the computing resource, measured in computing fraction saved times the time that it is available, versus the time steps of the iterative algorithm. As observed, the savings are significant.

Nonhomogeneity in SCGs

We now introduce the concept of nonhomogeneity in multicomputing systems. Let the checkpoints determined by the algorithm be $q_1, q_2, q_3, \ldots, q_M$ such that $\bigcup (q_{i+1}, q_i) = (0, T)$.

Let the initially assigned CPU fractions be $c_1^I, c_2^I, \ldots, c_N^I$ respectively, and the maximum fraction available be $c_1^{max}, c_2^{max}, \ldots, c_N^{max}$.

Defining

$$\kappa_p = \frac{c_p^I}{c_p^{max}} \quad , \quad \kappa_p^* = \max_p \kappa_p$$

$$\kappa_{p \in A_{n,\varphi}} = \frac{c^I{}_{p \in A_{n,\varphi}}}{c^{max}{}_{p \in A_{n,\varphi}}} \quad , \quad \kappa^*{}_{p \in A_{n,\varphi}} = \max_p \kappa_{p \in A_{n,\varphi}}$$

If γ_n denotes the ratio of the change in total execution time after the checkpoint algorithm to the unoptimized time T then

$$\gamma_n = \frac{\sum_{i=1}^{M} (q_{i+1} - q_i)(\kappa^*{}_p - \kappa^*{}_{p \in A_{n,\varphi}})}{\sum_{i=1}^{M} (q_{i+1} - q_i)\kappa^*{}_p}$$

γ_n is defined as the **degree of nonhomogeneity** and it is an indicator of the extent of the disparity between the tasks and the process CPU assignments. The larger γ_n is, the greater the gains from our optimization algorithms, since a smaller γ_n would imply a homogeneous partitioning of tasks before preprocessing with the checkpoint algorithm. In some cases, calculation of allocation may be simplified if all the $c^I{}_p$ were equal for all p.

It must be noted here that the checkpoint algorithm extracts enough information from the constituent processes in the concurrent program to determine the processes which are active for a large number of subintervals. Obviously these processes need the largest fractions of CPU and a suitable scheme can be devised which reassigns processes to processors such that the computation intensive processes get assigned to processes with larger $c^{max}{}_p$. (See Figures 9-10).

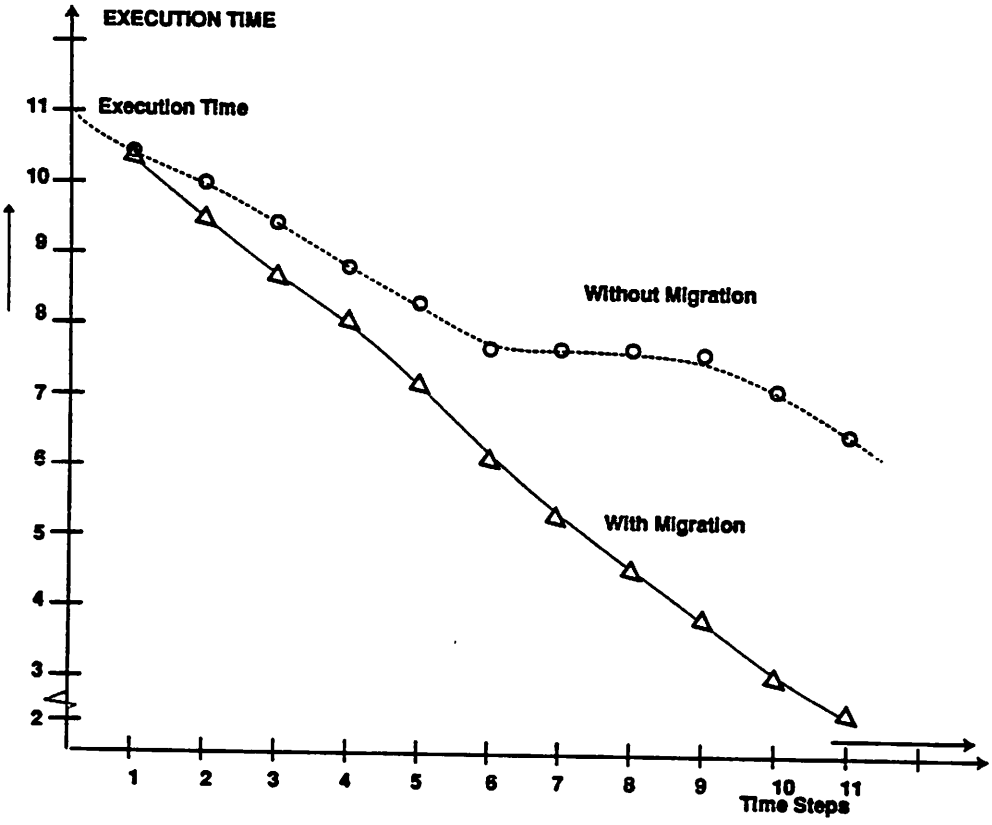


Figure 11: Performance of Process Migration

Migration involves the reassignment of tasks to lightly loaded processors. The checkpoint algorithm can efficiently migrate processes, as seen from the figure. The results for Case 1 in Figure 8, show a decrease in execution time from 11 to 2 time units.

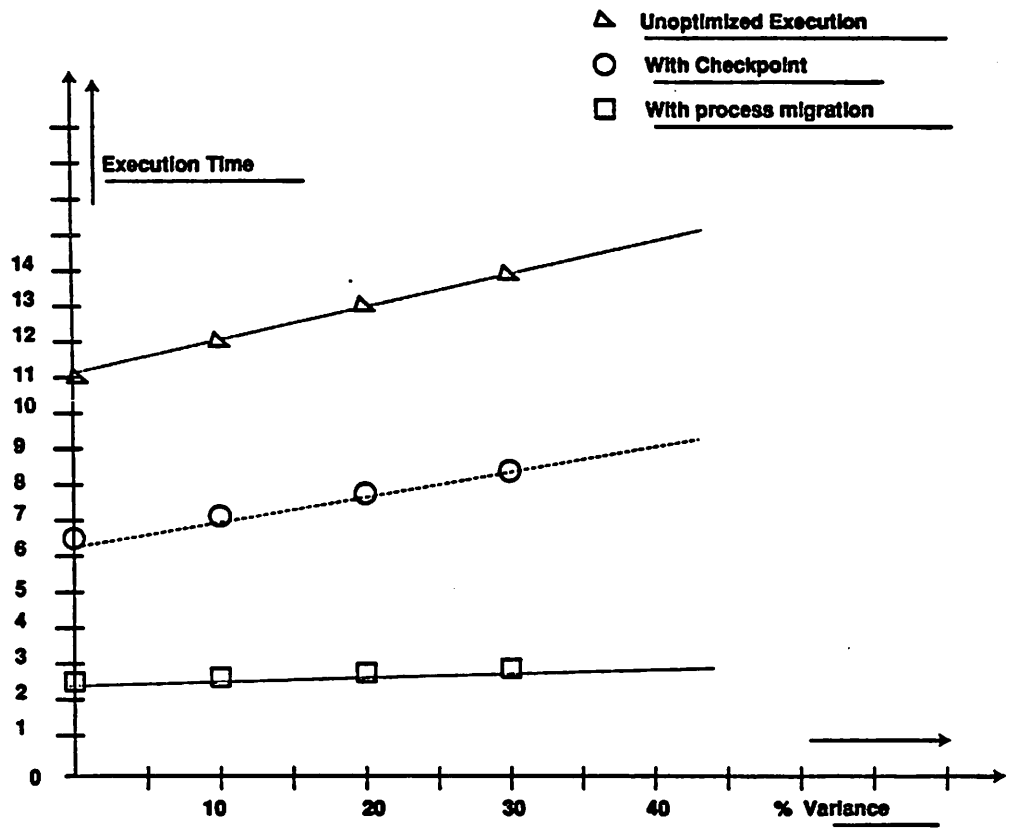


Figure 12: Data Dependencies

If the task times are slightly stochastic, or they have a small dependency on the data, the total execution time varies proportionally to the execution time. Since the checkpoint algorithm reduces the total execution time by a significant amount, the variation is much smaller as a result. This improvement is seen in this figure.

3.8.2. Process Migration

In this section, we examine some strategies for migrating processes to other processors in the system.

Consider N processes $1, 2, \ldots, p, \ldots, N$ spawned by processors $P_1, P_2, \ldots, P_N$ each with a maximum available fraction of CPU $c^{\max}_{P_1}, c^{\max}_{P_2}, \ldots, c^{\max}_{P_N}$.

The checkpoint algorithm generates sets of active processes $A_{q_i, q_{i+1}}$ for intervals (q_i, q_{i+1}) with an initial assignment of CPU fractions equal to the maximum available per processor. Let the cardinality of the active sets $|A_{q_i, q_{i+1}}| = Z_i$, and the processors be ordered in the decreasing fraction of CPU available, $P_1, P_2, P_3, \ldots, P_N$. We consider two cases for process migration which enhance processor utilization.

Case 1: Process migration

At the end of the checkpoint algorithm, each of the processes belonging to the active set $A_{q_i, q_{i+1}}$ is reassigned to one of the processors $P_1, \ldots, P_{Z_i}$. Each process then has its CPU assignment scaled up by $\frac{c^{\max}_{P_y}}{c^{\max}_{x}}$, where y and x are chosen such that the ratio is the minimum, say β_i^{-1}, for that interval. In practice we can scale up the CPU assignment by a larger fraction but this requires a larger amount of computation to ensure that each process is assigned to a processor which can scale its CPU fraction by the same amount. The speedup factor γ_m can then be represented as

$$\gamma_m = \frac{\sum_{i=1}^{M}(q_i - q_{i+1})\kappa^*_p - \sum_{i=1}^{M}(q_i - q_{i+1})\beta_i + Z_i \cdot C}{\sum_{i=1}^{M}(q_i - q_{i+1})k^*_p}$$

Since $\beta_i \leq \kappa^*_{p \in A_{q_i, q_{i+1}}}$ process migration can result in improvement in speed if the overhead, $Z_i \cdot C$, associated with process migration is assumed negligible. This is only justified if the intervals (q_i, q_{i+1}) are large compared to the costs of context switching and communications associated with process migration. Therefore, the processes are migrated to available processors with larger $c^{\max}_p$ if and only if the interval (q_i, q_{i+1}) is sufficiently large (See Figure 11). This overhead is significant only for the first

few data samples consumed by the CJ-TN.

The evaluation of A and IA at each step, allows the reduction of the complexity of the algorithm at each successive subinterval. This is because of the fact that if a process is active during an interval it maintains that state for all subintervals.

Case 2: Data-dependency:

In the above analysis, we have tacitly assumed that the data dependency of the process segments within the checkpoints can be expressed as some function of ε. The efficiency of the checkpoint algorithm is then a function of ε. If epsilon were small enough, which is true in most coarse-grained signal processing applications the algorithm is quite efficient. However, a large variation in execution time, can result in processes in other processors terminating much earlier or much later than the errant process. As discussed above, the check point algorithm moves process segments on to the critical path, however processing them earlier has no effect on the total run length. A **monitor** on each processor keeps a list of process segments which can be executed over a number of checkpoint intervals without affecting the total execution time. Consequently, whenever, a processor completes execution of its program within a checkpoint interval much ahead of other processors, it processes one of these *floating* off-critical path tasks. A finite algorithm, similar to the checkpoint perturbation algorithm, can then be formulated to determine which of the processes can be moved away from the critical path, for execution as a fill-in task for processors (to utilize their computation fraction efficiently) in the case of large ε. This modification enables an enhancement in the efficiency at the cost of a floating process queue of processes (which can be executed with some latitude in the time of their completion) (See Figure 12).

3.9. COSPROL : Rules and Syntax

COSPROL is proposed as a concurrent system programming language which enables the algorithm developer to define a parallel paradigm for execution of an algorithm. The COSPROL preprocessor, does the following tasks.

1. It maps the COSPROL source program onto a distributed processor architecture, such that the algo-

rithm constraints, precedences and data flow characteristics are not violated. This is similar to the specification used in languages such as in Blosim (See [Mess79]), but targeted towards multiple processor machines.

2. The Join-Type Networks of processes are then optimized for efficient CPU utilization, as will be described in the following sections.

3. By adaptive perturbation techniques, the COSPROL distributed architecture consisting of asynchronously executing processors is synchronized.

4. By keeping a current description of the critical sections of the network, the throughput can be dynamically varied with fluctuating load.

The individual processes are pieces of code which are defined by the programmer. COSPROL optimizes resource allocation solely on the basis of input-output characteristics of the output flow. The execution times, process fractions, and communication protocols within processes are not assumed to be known beforehand, making such a description of concurrent programs very attractive. The task execution times are assumed deterministic. In this section we describe the COSPROL syntax and rules, which enable a concurrent system developer to specify an algorithm for implementation on a distributed computing architecture.

The algorithm specified in COSPROL can then be analyzed by a preprocessor to allocate the architecture, assign resources, and optimize performance. We will first describe the syntax and then discuss the processing algorithms used by the COSPROL preprocessor.

Syntax

In the following discussion we denote processes by capital letters A, B, C and so forth. J-TN denotes a Join-Type Network. Unless otherwise noted we assume each process is spawned by separate parent processor.

a. $A = SP\ (A)$

b. $X = CP\ [\ AB, AC, BC, CD\ ]$

c. $S1 = SJ\text{-}TN\ [\ z_1 A, z_2 B\ ]$

d. $C2 = CJ\text{-}TN\ [\ z_1 A, z_2 B, z_3 X\ ]$

e. $C3 = n\ CJ\text{-}TN\ [\ z_1 A, z_2 B, C2\ ]$

f. $C5 = F\text{-}TN\ [\ A, S, z_1]$

g. $C5\ /\ C4$

(a) defines A as a simple process. X is a compound process, comprising of a cluster of processes A, B, C and D processes A and B, A and C etc. communicating with each other. S1 is a Simple J-TN of z_1 processes described by A and z_2 processes described by B. C2 is defined as a Compound J-TN of processes, A, B and X of which at least one is a compound process. C3 is defined to be a CJ-TN of order n which implies that it is the J-TN of at least one process of order n - 1. For example if n = 1 then it is the CJ-TN, if however, n = 2 then, it is second order hierarchical CJ-TN of first order CJ-TNs. C5 is a Fork-Type Process, with A as the input process, S is the selector process which determines which of the outputs are available to C5 and z_1 is the maximum number of output buffers. (g) defines the sequential connection of two processes C5 and C4. The output of C5 is piped into the input of process C4. This definition is similar to the UNIX pipes.

We now present a few typical examples to illustrate use of COSPROL to specify distributed computation algorithms in a SESYCCS environment.

3.10. Concurrent Programming in COSPROL

In this section we will present the results of a pilot implementation of two computation intensive problems (expressed as SCGs) on a commercially available NCUBE Multicomputer.

3.10.1. Phase Shift Migration

Migration of seismic data involves repositioning the measured data to determine accurately the

topology of the subsurface reflectors. Migration is an inverse process in which the recorded waves are propagated back to their source by systematically solving the wave equation for each successive layer. There has been considerable study of stable algorithms for efficient solution to the wave equation, and seismic migration has been routinely used for interpreting seismic data for over two decades. Migration techniques range from simple finite-difference techniques to the more sophisticated frequency domain methods. Regardless of the technique used, migration greatly facilitates accuracy in seismic interpretation and identification and in some cases is indispensable.

Seismic migration algorithms are computationally very intensive and require processing large amounts of data. A frequency domain parallel *phase shift* migration algorithm was analyzed in [MaMe88a] and the performance indices for that algorithm are discussed. A detailed discussion of these algorithms and their implementation is beyond the scope of this chapter and the reader is referred to [MaMe88] for further details. Migration as a SCG can be described by the block diagram illustrated in Figure 13.

The COSPROL representation of phase shift migration can be elegantly described as follows.

Seismic Migration Algorithm

S1 = SJ-TN [a FFT]

C2 = S1 / A1

P1 = F-TN [C2, ALL, a]

C3 = SJ-TN [a P1 / FFT, E]

C4 = C3 / M

P2 = F-TN [C4, ALL, a]

P3 = P2 / IFFT

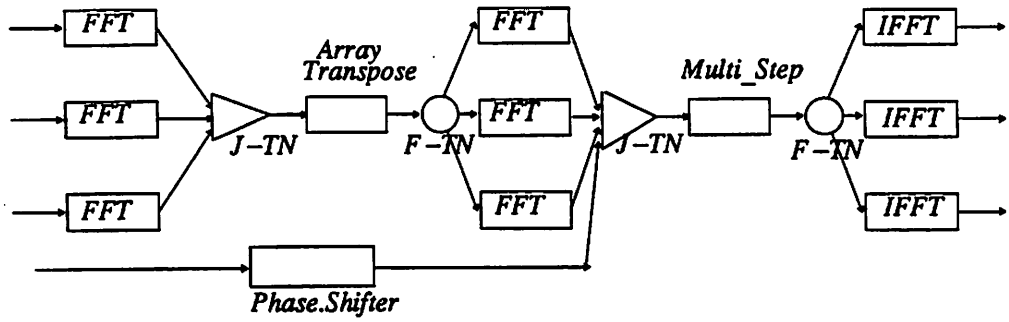


Figure 13: Seismic Migration as a SCG

In our above program, A1 is the CJ-TN representing the matrix transpose involving all the processors in log N time steps, E is the process generating the exponential multipliers, M is the multi-step algorithm. P3 represent the processes containing the result, which is a migrated layer. ALL implies that all the outputs (up to a maximum of a which is the data matrix size) are selected. FFT and IFFT are the processes computing the Fast Fourier Transforms of the data. Since the processes are inherently load balanced by the simple structure the optimization of the code is straightforward (Figure 13).

Programming for the distributed application in COSPROL involves the following subtasks:

a). Partitioning the sequential algorithm, to distribute the computational load uniformly over the independent processors. This partitioning is relatively straightforward in the case of migration since the concurrent tasks were identical.

b). A communication protocol to enable efficient data and information transfer between the processors to yield results consistent with the sequential program.

Very often the communication overhead determines the bottomline in performance to be expected. This is where the perturbation approach proposed in this chapter wins over other methods. The checkpoint algorithm does not assume a priori knowledge of the tasks or the communication times. It optimizes on the implementation and *not* on the model of the computation. The costs of communications are therefore automatically included in the performance. Further optimization of the performance is done by balancing the load dynamically, overlapping computation with communication, pipelining I/O and computation and by increasing the memory per processor.

The NCUBE/Ten Multiprocessor system interconnects 1024 32-bit processors each with 128/512 Kbyte private memory in a hypercube configuration [NCUBE86]. Host processors are available to enable loose global control and synchronization. I/O channels lead directly to the processing nodes and the host through multiported memories.

The parallel phase shift algorithm was implemented on the 64 processor model. Solution for data sizes for higher dimensions was implemented by breaking down the problem into smaller manageable tasks, and the performance then projected for a higher number of nodes. The modularity and the regularity of the node programs enables rapid porting onto higher order hypercubes.

The first communication algorithm we use is the the *sequential communication protocol (SCP) [MaMe88a]*. The host is the center of the star of processors, each node communicates only with the host in receiving and sending data. The host therefore implements the forks and the joins. The sequential algorithm thus penalizes the performance by not utilizing the parallelism inherent the highly interconnected hypercube configuration. The second communication algorithm the *parallel communication protocol (PCP)* overlaps communication between sets of nodes, reducing the *total* communication time. This implies that the forks and the joins are implemented by the nodes themselves in the logical system. Both the protocols move the same amount of data and the times are proportional to A^2 where (A, A) is the size of the data array, though the constants of proportionality are an order of magnitude apart.

Our basis of comparison is a hypothetical sequential machine (HSP) with infinite cache size, and the computation of the algorithm on this machine is the total computation time on the different parallel processors. Processor utilization of this machine is assumed 100 %.

Our experience with seismic signal processing has been encouraging in that it gave us a robust testbed for testing the performance of SESYCCS on SCGs. Table 1 gives the speedup, I, for an implementation with a data size of $A = 256$. The reader may contrast these figures with the performance of a shared-memory type implementation described in Chapter 2 where the speedup was only an order of magnitude.

Table 1

Performance of Parallel Phase Shift Migration on NCUBE						
Index	# of Nodes	HSP (sec.)	SCP(sec)	PCP(sec)	I	I
1	64	252	9.05	6.1	27	41
2	128	252	7.2	4.2	35	60
3	256	252	6.3	3.36	40	75
4	512	252	5.6	2.76	45	91

3.10.2. Optimization of a CJ-TN.

Our next example is the synchronization of a compound process on the NCUBE multicomputer. Consider the simple four node network shown Figure 14. The data path is from node 0 to node 3, with a loop at node 1. The execution times of the node programs for each data input is assumed to be distributed as an exponential random variable with mean λ_i.

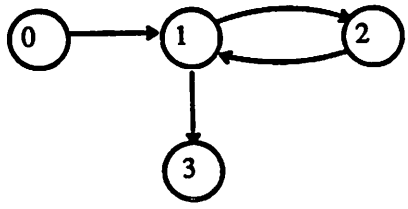


Figure 14: Synchronization of CJ-TN

Table 2

Synchronization of the CJ-TN				
Index	λ_1	λ_2	λ_3	Execution_time
1	20	20	5	114
2	25	20	5	150
3	20	25	5	128
4	20	20	6.25	116
5	15	25	5.0	107

We illustrate the use of the checkpoint algorithm for speeding up stochastic systems (its utility in deterministic systems was illustrated in Sections 3.8-3.9 above). Node 0 generates all the inputs to the processes in the compound network. We perturb the mean service times of these nodes to determine the bottleneck process. The results are then used to speed up throughput. The results on NCUBE are shown in Table 2 above.

As observed from Table 2, the process 1 is the bottleneck. Its role in the total execution time is quickly recognized and it is assigned the maximum use of resources (implying that its mean time is reduced from 20 to 15 in column two of Table 2). The algorithm does not require a knowledge of the data flow within the network and assigns resources on the basis of the input/output characteristics which makes it so attractive for multiple processor systems.

Closer (mean) service times can penalize the resolution of the checkpoint algorithm. However, this is not serious complication since then both the processes would be on the critical path.

3.11. Summary

Self-Synchronizing Concurrent Computing Systems have been proposed for efficient distributed computation. In this chapter, we described computation that can be described by a Static Computation Graph.

We first examine the conditions under which a SESYCCS can be realizable in practice (with bounded buffer sizes). We then present an algorithm, called the checkpoint algorithm, which demarcates the active regions of each process in the SESYCCS. Since the computation is described using a SCG, this knowledge is then utilized in time-sharing the processors with other user programs.

Quantitative estimates of the advantages of process migration are also derived. The mismatch between the SCG and the SESYCCS is quantified by a coefficient of nonhomogeneity, γ_n, which provides an estimate of the improvement resulting from synchronization.

All the algorithms described in this chapter are designed to be implemented on the system itself, and require neither user participation nor prior knowledge of the computation times by the system. The SESYCCS adaptively takes into account the presence of users of differing priorities in the same system, and is able to modify the synchronization schedule on demand.

The framework introduced in this chapter provides a concrete base for the design of SESYCCS for high speed signal processing applications. An example of a seismic migration shows the flexibility of this approach as well the ease in implementation.

Future work is focussed on implementing the algorithms on a commercially available multicomputer, and providing for a powerful user interface with the SESYCCS.

Chapter 4

Self-Synchronization for Dynamic Computation Graphs

In a distributed computation, the compute-bound algorithm is divided into a number of smaller pieces and each of these pieces is assigned to be executed on a separate processor whenever possible. The amount of concurrency in the system can be very large in applications such as in the distributed simulation of discrete event-systems. Most present day concurrent computing systems use a modest number of processors (a few hundred) with the intention of reducing chip counts, backplane connections and the complexity of communications. However, recent advances in VLSI and in new methods of routing messages have encouraged the development of more ambitious multicomputers. At the time of writing, a commercial vendor has announced the development of an 8192-processor multicomputing machine with a peak computing capability of 27 GFLOPS [See NCUBE89].

Software for programming and efficient use of such machines has yet to catch up to the rapid pace in the development of computing hardware. The multiprogramming and multitasking nature of parallel and distributed computation has hindered the tasks of manual synchronization and load balancing. With the application often breaking up into thousands of smaller pieces, synchronization of the distributed computation is not feasible unless efficient algorithms for automation are developed to meet the challenge.

In the previous chapters, we introduced Static Computation Graphs (SCGs) and Dynamic Computation Graphs (DCGs) as describing two important classes of scientific computation. Chapter 3 described efficient synchronization of SCGs in a SESYCCS environment. In this chapter, we wish to model computation described by Dynamic Computation Graphs, and then propose algorithms for their efficient synchronization. Most of the material presented in this chapter is new. Closed-form results are derived for a number of synchronization algorithms, and their performance is compared.

For a variety of reasons, the techniques introduced in Chapter 3 cannot be used for synchronizing computation that can be described by Dynamic Computation Graphs For instance, a Static Computation Graph has a temporal variation in concurrency that is a deterministic function of time. On the other hand, the precedence relations among events in a Dynamic Computation Graph are unknown a priori, and lookahead in the computation is very poor. The concurrency available in a Dynamic Computation Graph is thus random in nature. Consequently, algorithms for time-sharing resources lose their elegance and simplicity. Very few concrete results on the performance analysis of synchronization schemes have been reported in literature. We provide a number of new analytical results which describe the behavior of Dynamic Computation Graphs, and present novel algorithms for their self-synchronization (i.e synchronization is.not explicitly provided as part of the computation).

Synchronous methods of synchronization based on worst-case analysis pay a high price in efficiency and the performance of these methods appears to be dependent on the application. These methods do not scale well with an increasing number of processors in the distributed system. The challenge, therefore, lies in the development of asynchronous algorithms for automated distributed synchronization of concurrent processors that scale well with the number of processors and yet are easy and efficient to implement. It is also desired that the synchronization algorithm be capable of utilizing the knowledge of the behavior (at run-time) of the distributed computation to further bootstrap its performance.

As discussed in Chapter 2, the **causality conditions** are necessary to ensure that the logical system provides results that are correct. This chapter discusses *asynchronous* synchronization methods for

DCGs. DCGs are modeled as a system of self-timed processors communicating to each other via messages. The stochastic nature of the coupling between local clocks is captured in our model. Detailed simulations confirm theoretical results quantifying the efficiency of the self-synchronization.

The situation of greatest interest is when the asynchronous clocks progress at different and possibly time-varying rates; the computation being inefficient owing to the high overhead in synchronization. Fortunately, as will be shown in later sections, the analysis is tractable for this case.

We will now provide exact results on the performance of asynchronous synchronization mechanisms. To clarify our exposition, we will first focus our efforts on an analysis for a logical system that has two processors cooperating in a distributed computation. The two-processor case is very useful in introducing the techniques and performance indices that we shall have occasion to use throughout the remainder of the chapter.

4.1. The Two-Processor Logical System

In a two-processor asynchronous computation there are two clocks, C_n^1 and C_n^2, associated with processors 1 and 2 respectively. These clocks evolve with real time n according to the set of dynamical equations given below. As discussed in Chapter 2, processor 1 (2) can communicate with processor 2 (1) at every time step n with a probability of p_{12} (p_{21}). The communications in the logical system take place at a sequence of Bernoulli times $n = T_m^{i,j}$ for $m = 1, 2, 3 \cdots$, and $i, j \in \{1,2\}$. If processor j communicates with processor i when $C_n^i > C_n^j$, then C_n^i is resynchronized (or reset) to the value C_n^j. This resynchronization is assumed to take one real computing unit of time. On the other hand, if the $C_n^i < C_n^j$ then the progress of C_n^i is given by a_n^i, also known as the rate of forward computation of i. We assume that all the simulated times are multiples of some positive quantity ε. This ensures that the clocks move in a countable state space. This property will be used in later chapters to establish some stability results. The dynamics of the local clocks are given by following set of equations.

$\{C_n^i, n \geq 0, i = 1, 2\}$ is defined as follows,

$\{T_m^{i,j}; m = 1, 2, 3 \ldots\}$ are Bernoulli times such that

$$\text{Prob}[\, T_{m+1}^{ij} - T_m^{ij} = k \,] = p_{ij}\,(1 - p_{ij})^{k-1}, \; k = 1, 2, \ldots$$

$$\text{If } n \neq T_m^{ji} \text{ for all j, m} \quad \text{then,} \quad C_{n+1}^i = C_n^i + a_n^i$$

$$\text{Else, if } n = T_m^{ji}, \text{ for all m, j, then,} \begin{cases} C_{n+1}^i = \min\{C_n^i, C_n^j\} + a_n^i \, I\{C_n^j < C_n^i\} \\ C_{n+1}^i = C_n^i + a_n^i \end{cases}$$

Assume also that; $a_n^1 > a_n^2$, for all n.

$$\frac{a_n^i}{\varepsilon} \in \mathbb{Z}_+ - \{0\}$$

$$\frac{C_0^i}{\varepsilon} \in \mathbb{Z}_+$$

$B_1 < a_2^i < B_2$, for some constants $B \in \mathbb{Z}_+$ (the set of integers).

[*Note:* The assumption that $a_n^1 > a_n^2$ can be relaxed to the case where $a_n^1 > a_n^2$ for all n such that $C_n^1 \leq C_n^2$. The restrictions on the rates, therefore, are very weak and sufficiently general to be widely applicable.]

4.1.1 Discussion of the Two-Processor Model

We will now offer a qualitative description of self-synchronizing computation on two processors, 1 and 2. Processors 1 and 2 communicate with each other via time-stamped messages. Messages are time-stamped (in discrete real computing time) with the local simulated times of the transmitting processor. The progress of each processor can be depicted graphically by a *profile* that plots the simulated computation time on the y axis and the real computer time (wall-clock ticks) on the x axis (See Figure 1). Let us assume that processor 1 computes forward with a deterministic rate of A units per computing time tick, while the corresponding rate for processor 2 is B units per tick. Without loss in generality, let us assume $A > B$. Processor 1 can communicate with processor 2 at each time tick with a probability p_{12}. Likewise, processor 2 can send a time-stamped message to processor 1 after each time tick, with a probability p_{21}. Figure 1, describes the progress of the clocks if they did not communicate with each other. Each processor then progresses in simulated time at its forward computation rate (A or B). The

analysis of the two-processor case becomes interesting when the processors are allowed to interact with each other.

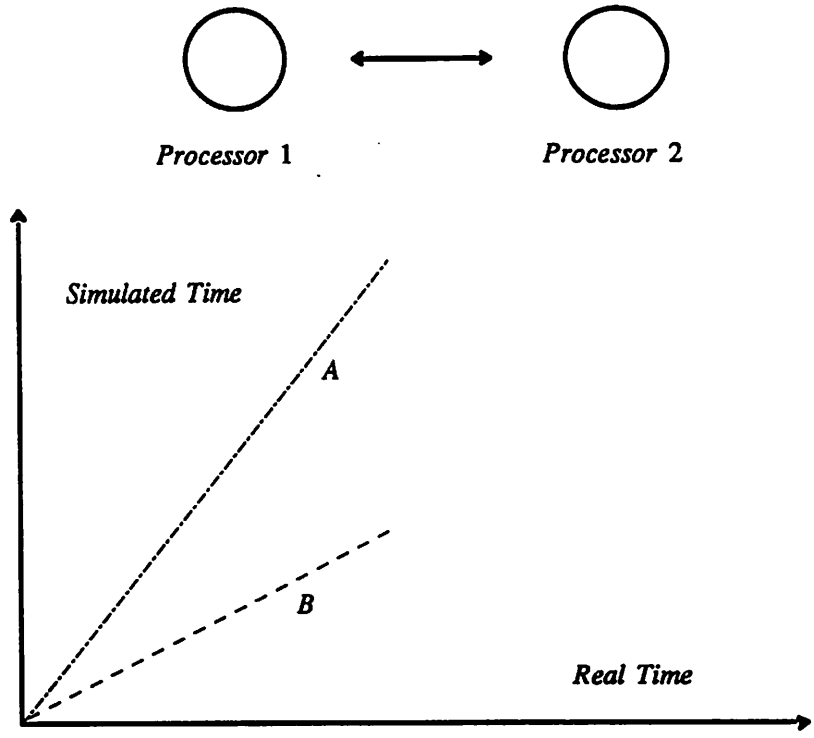


Figure 1: Profile for Two-Processor System

A message can be represented by a vertical line in the profile if we assume that the real time required for communication is very small. A processor is said to "rollback" or "resynchronize" from time T_1 to time T_2 when it is at local simulated time T_1 and receives a message time-stamped T_2 and $T_1 > T_2$. The processor, therefore, has to reset its own local clock from T_1 to T_2. We further assume that this resynchronization takes a fixed amount of real computing time to complete (say, one clock tick). The processor can then resume forward computation. (Note that all messages do not trigger resynchronization). A typical execution profile is given in Figure 2.

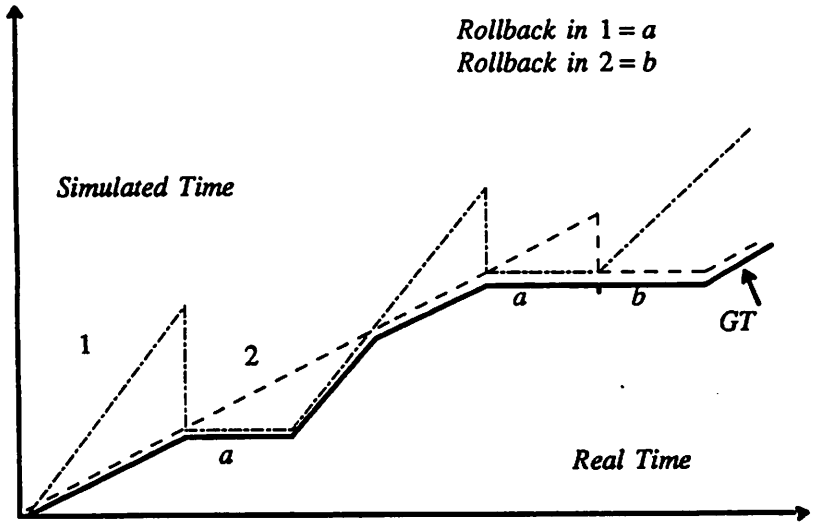


Figure 2: Typical Profile for Asynchronous Computation

In the scenario illustrated in Figure 2, processors 1 and 2 begin computing forward with rates A and B respectively, and at some point in time processor 2 triggers a rollback in Processor 1 (Case 1). Processor 1 then resynchronizes to the simulated time of processor 2, and then spends one computing time unit recovering from the resynchronization. Two possible events can occur at this point. In the first, processor 1 triggers a rollback in processor 2 with a probability p_{12} (Case 2), and in the second, processor 2 resynchronizes processor 1 (Case 3) again. These cases are illustrated in Figure 3.

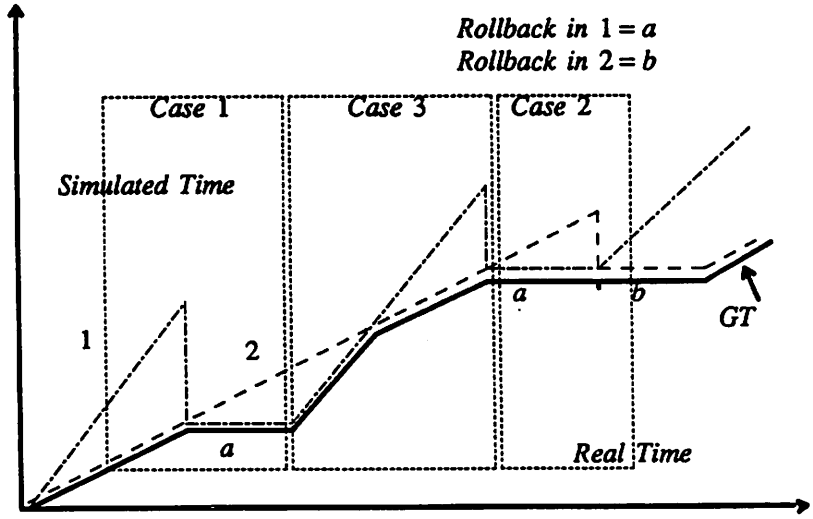


Figure 3: Possible State Transitions.

In the ideal case, local clocks on both processors march ahead at the equal rates, $A = B$, and the rate of growth of the computation achieves its maximum value, B simulated time units per real computing time unit.

4.1.2 Associated Markov Chain Representation

The dynamical system represented by equations in Section 4.1 can now be represented by a Markov chain. If we assume that a_n^1 is at least twice as large as a_n^2, it can be easily shown that the discrete-time Markov chain representation of the two-processor asynchronous computation would be given by Figure 4. If the system is in state $S_i = 1(2)$, this implies that processor 1(2) is ahead in local time, and received messages can induce a *resynchronization or rollback* if C_n^i is ahead of C_n^j (measured in simulated time). Let the invariant probabilities of states S_1 and S_2 be denoted by $\pi(1)$ and $\pi(2)$ respectively.

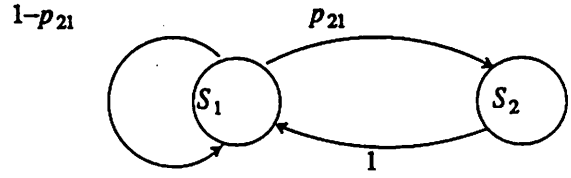


Figure 4: Two-Processor System

The balance equations are,

$$\pi(1) = (1-p_{21}) \cdot \pi(1) + \pi(2)$$
$$\pi(2) = \pi(1) \cdot p_{21}$$

Solving these equations, one finds

$$\pi(1) = \frac{1}{1+p_{21}} \quad , \quad \pi(2) = \frac{p_{21}}{1+p_{21}}$$

The Global Time (GT_n) is defined to be the smallest local time (at time step n) in the entire distributed system. It is the time up to which the distributed computation can be guaranteed correct. In other words, $GT_n = \min\{C_n^1, C_n^2\}$. The average rate of growth of GT_n, $\alpha \doteq \lim_{n \to \infty} \frac{GT_n}{n}$. An analytical expression for α, after N computing time ticks can be derived as follows:

$$\alpha \cdot N = (1-p_{21}) \cdot \pi(1) \cdot BN + (1-p_{12}) \cdot \pi(2) \cdot 2BN$$

Hence,

$$\alpha = \frac{(1-p_{21})}{(1+p_{21})} \cdot B + p_{21} \cdot \frac{(1-p_{12})}{(1+p_{21})} \cdot 2B$$

α, therefore, provides an useful estimate of the progress of the self-synchronizing computation. The progress of the distributed computation depends on the probabilities of interaction and also on the rates of the individual processors.

The fact that the multiplier for the second term on the right hand side of the expression above is $2B$ should be obvious from Figures 3 and 4. If A were less than $2B$, the value of α would be an upper bound, and not an equality. Note also that α will be affected only if the slower processor is resynchronized. We will sometimes have occasion to refer to α as the "Wolf Coefficient," with regards to its use in an algorithm for distributed synchronization, Wolf, to be discussed in Chapter 6.

We will now try to interpret our analytical results to get further insight into the dynamics of asynchronous computation. For this purpose, the clock model discussed in this section was simulated to reveal the transient nature of the asynchronous computation (our analysis in earlier sections derives steady-state values).

In Figure 5, we study the progress of asynchronous computation, when the probabilities p_{12} and p_{21} are given the values 0.1 and 0.8 for Case 1, and 0.7 and 0.8 for Case 2, respectively. The values of a_n^1 and a_n^2 are 1 and 3 respectively. Analytical values for the Wolf Coefficient, α, are calculated (using the expressions derived above) to be 0.911 and 0.377 respectively. The former coefficient is much larger because the probabilities of interaction are small. In Case 1 the system spends a smaller amount of real computing time recovering from resynchronization. In the second case, however, the communi-

cations between processors are frequent, and the Wolf Coefficient is penalized as a result. Observing the dynamics in the value of the Wolf Coefficient in Figure 5, we note that the value settles down to its steady state value quite early in the computation.

Figure 6, shows the dynamics of the clocks on the processors themselves for the two cases. The slower processor 2 frequently sends messages to processr 1 in the course of the computation. Most of these messages force processor 1 to resynchronize. The average global growth is, however, not penalized unless p_{12} is large. This is because of the fact that if p_{12} is small, then processor 1 would then overtake processor 2 leaving α unaffected. On the other hand, if p_{12} were large (as in Case 2), the progress of the slow processor is further impeded, as shown in the Figure 6, and the coefficient α is affected significantly.

In Figure 7, we present the results of another interesting experiment, where the two cases of asynchronous computation have varying communication to computation ratios. In Case 1, communications between processors are infrequent, and in Case 2 communications are relatively frequent. As expected, the former case progresses faster. We also note that the memory requirements are much larger as well. The intuitive explanation for this is as follows. If the processors communciate infrequently with each other they tend to drift apart further before resynchronization. Processors in this environment needs correspondingly larger buffer sizes to store the state information.

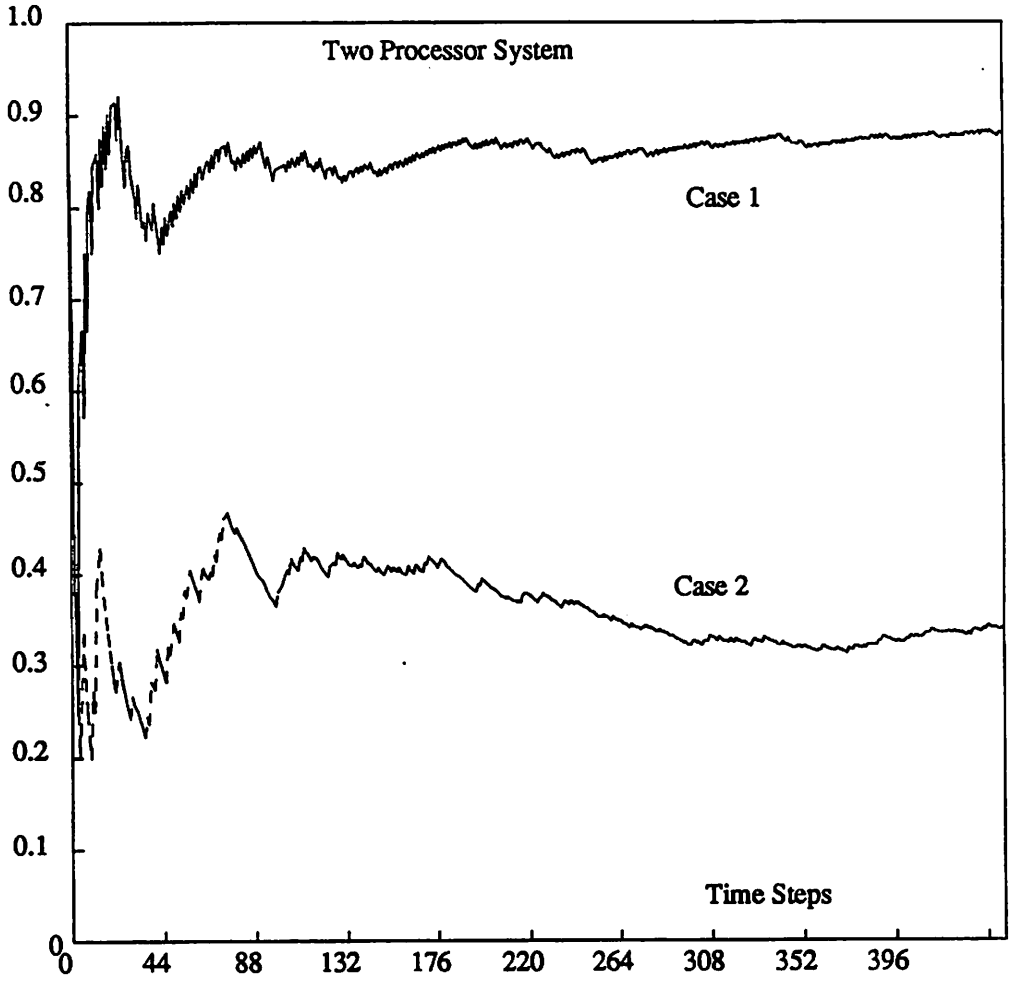


Figure 5

Case 1: $p_{12} = 0.1, p_{21} = 0.8$
Case 2: $p_{12} = 0.7, p_{21} = 0.8$

In this figure α is plotted against time steps. In Case 1, α was analytically calculated to be 0.911, correspondingly in Case 2 it was 0.377. As observed from the figure, α rapidly settles to its asymptotic value. The values of a_n^1 and a_n^2 were chosen to be 3 and 1 respectively. If $a_n^1 = a_n^2$, then $\alpha = 1$ if the processors 1 and 2 both had started at the same initial clock time. Once out of synchrony, however, the value of α drops down to 0.64.

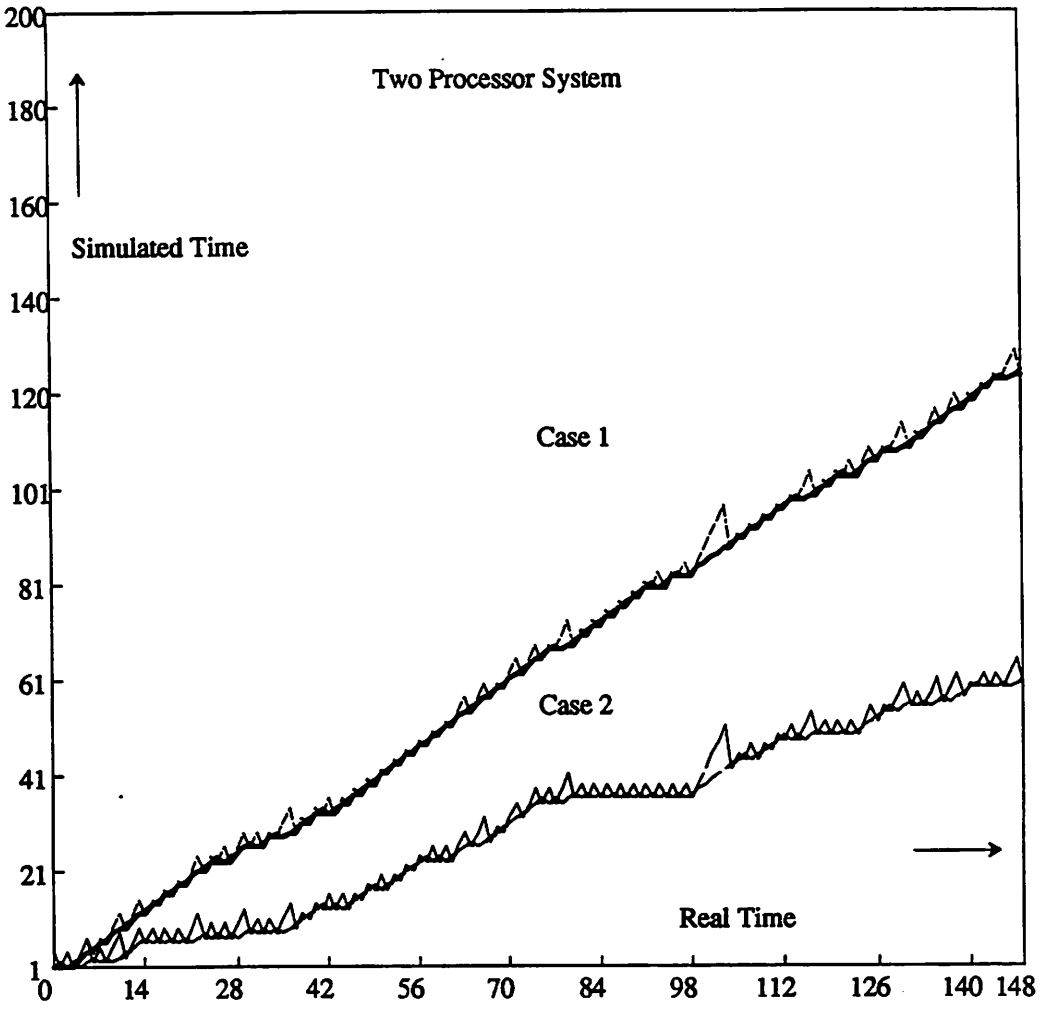


Figure 6

Case 1: $p_{12} = 0.1, p_{21} = 0.8$
Case 2: $p_{12} = 0.7, p_{21} = 0.8$

This figure describes the evolution of the local clock times and the global time (GT) of the two-processor system. Both Case 1 and Case 2 have about the same number of messages sent from the slow to the fast processor, however in Case 2 p_{12} is larger, contributing to a significant drop in the rate of growth. The memory required on each processor is roughly the same for both cases. Frequent communications imply smaller drift between local clock.

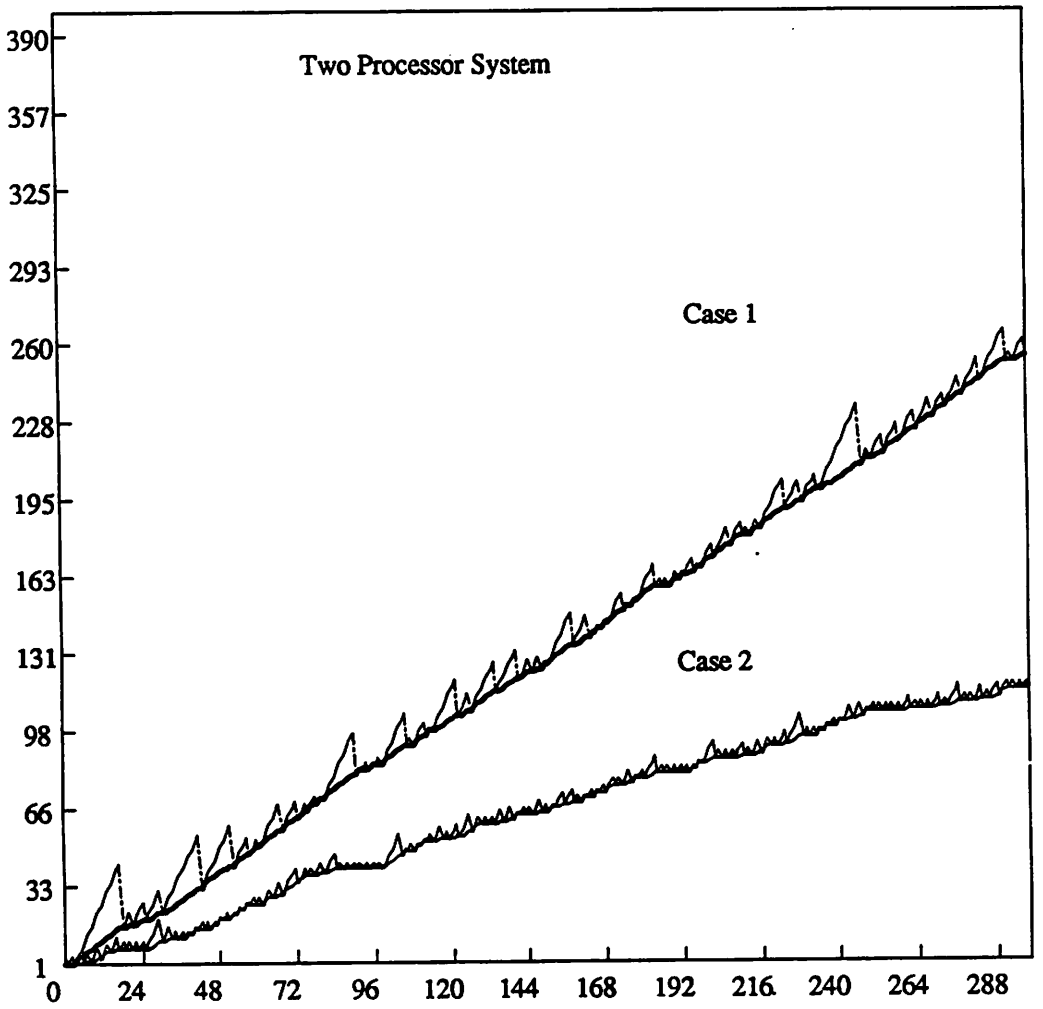


Figure 7

Case 1: $p_{12} = 0.3, p_{21} = 0.3$
Case 2: $p_{12} = 0.7, p_{21} = 0.7$

This figure studies the two-processor synchronization under two communication scenarios. In Case 1, with "infrequent communications" the rate of progress the memory requirements were proportionally higher. In Case 2, with "frequent communications," the progress is slow, however, the memory requirements are also smaller. We believe this memory-speed tradeoff is fundamental to asynchronous distributed systems. Asynchronous communications ensure that processors need not wait to communicate, and Bernoulli interactions ensure that the buffer sizes remain bounded.

4.2. Computation in Presence of Communication Delay

Let us consider once again the model of an asynchronous distributed computation on two processors, as developed in Section 4.1. Our present objective is to integrate the effect of communication delay into our model for the dynamics of the local clocks. Let us also assume that the communication delay in the transmission of the message from processor 1 to processor 2 is a real time units, and b for a message on the return path. Without loss of generality, let us assume that a is less than one computing time tick and b is between one and two computing time ticks. Other cases can be handled analogously.

The dynamics of the self-synchronized system will be given by the following set of equations. (It is assumed that there are no messages in transit when the system is initialized.)

$$\text{If } n \neq T_m^{ji}, \text{ for all } m, j, i, \text{ then, } \begin{cases} C_{n+1}^1 = C_n^1 + a_n^1 \\ C_{n+1}^2 = C_n^2 + a_n^2 \end{cases}$$

$$\text{If } n = T_m^{12}, \begin{cases} C_{n+1}^2 = C_n^2 + a_n^2 \\ C_{n+2}^2 = \min\{C_n^1, C_{n+1}^2\} + a_n^2 I\{C_{n+1}^2 < C_n^1\} \\ C_{n+1}^1 = C_n^1 + a_n^1 \end{cases}$$

$$\text{If } n = T_m^{21}, \begin{cases} C_{n+1}^1 = C_n^1 + a_n^1 \\ C_{n+2}^1 = C_{n+1}^1 + a_{n+1}^1 \\ C_{n+3}^1 = \min\{C_n^2, C_{n+2}^1\} + a_n^1 I\{C_{n+2}^1 < C_n^1\} \end{cases}$$

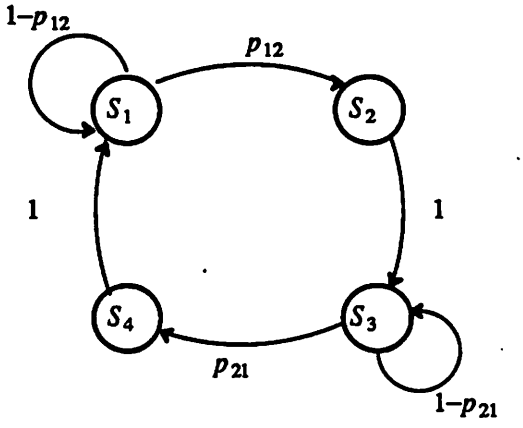


Figure 5: Two-Processor System with Communication Delay

Equations for the analysis of systems with arbitrary delays can be constructed in a similar manner. We will pursue the analysis for one special and interesting case, where we assume that both processor 1 and processor 2 progress at the same rate, $a_n^1 = a_n^2 = B$ for all n. We also assume that a and b are both less than one computing time tick. We believe that these restrictions will isolate the effect of communication delays on the progress of the computation.

4.2.1. Markov Chain Representation of Communication Delay

The discrete-time Markov chain now consists of 4 states. The effect of a message is hidden for one computing time tick (while the message is in transit). This results in two additional states. The system is in state S_1 (S_3) when processor 1 (2) is ahead. State S_2 represents the case where processor 2 sends a message to processor 1 when $C_n^1 > C_n^2$. This message is, however, not read until the next computing time unit. At that point, processor 2 gets ahead and will remain ahead of processor 1 until it receives a message from processor 1 (with probability p_{12}). The Markov chain is shown in Figure 8.

The balance equations are

$$\pi(1) = (1-p_{21})\pi(1) + \pi(4) \ , \ \pi(2) = \pi(1)p_{21}$$
$$\pi(3) = (1-p_{12})\pi(3) + \pi(2) \ , \ \pi(4) = \pi(3)p_{12}$$

Solving for the invariant probabilities,

$$\pi(1) = \frac{1}{1 + 2p_{12} - p_{21} - p_{21}^2 - p_{12}p_{21}^2}$$

$$\pi(2) = \frac{p_{21}}{1 + 2p_{12} - p_{21} - p_{21}^2 - p_{12}p_{21}^2}$$

$$\pi(4) = \frac{1 - p_{21} - p_{21}^2}{1 + 2p_{12} - p_{21} - p_{21}^2 - p_{12}p_{21}^2}$$

$$\pi(3) = \frac{1 - p_{21} - p_{21}^2}{(p_{12})(1 + 2p_{12} - p_{21} - p_{21}^2 - p_{12}p_{21}^2)}$$

The coefficient α will now be given by

$$\alpha = (1 - 2\pi(2) - 2\pi(4)) \cdot B$$

We will close this section with a pertinent observation. Finite communication delay between processors can also be incorporated quite easily into the model for the clock dynamics in asynchronous

systems. The effect of the delay is to further reduce the Wolf coefficient.

4.3. The Multiple Processor Logical System

In the previous sections, we had analyzed distributed asynchronous computation on two processors. Our analysis was rewarding for a number of reasons. The simplicity and generality of the model was elegantly illustrated by the two-processor system. The effect of resynchronization can be described by the associated Markov chain balance equations, and the coefficient α provides a closed-form description of the progress of asynchronous computation. This is a new result.

It could be argued that asynchronous distributed computation can only progress as fast as the slowest processor in the system. This is incorrect. The progress of asynchronous computation depends both on the rates of the faster and the slower processors and also on the probabilities of their interaction. There is one interesting fringe benefit, garbage collection algorithms become simple to implement. As the system converges very fast to the rate α, each processor (especially the faster one) can wipe its memory clean of any value below that indicated by the Wolf Coefficient for that system. One could argue further that the probabilities of interaction are not known a priori. However, this is not a problem (when implementing a distributed system) as the rate can be easily measured locally once the computation proceeds. The convergence to asymptotic rate of growth is very quick (this was verified in simulation results).

Our interest now lies in the analysis of an asynchronous computation mapped onto a system with a number of processors. Typically, multicomputing involves the synchronization of a few hundred processors, and performance can be very poor if a synchronization algorithm does not scale gracefully. Poor efficiency results when some processors are able to compute forward with a greater speed than other processors. A slower processor is one where the local state update chews up a significant portion of processor cycles. A faster processor is usually one with a small amount of state information at each instant.

Let us once again examine a case where synchronization can do the most good. This is the "unbalanced" system, where fast and slow asynchronous processors coexist in the distributed system.

Section 4.3.1 introduces a new algorithm for synchronization. Here, the processors are classified into one of two sets; fast or slow. Whenever a slow processor communicates with a faster processor, all the faster processors are resynchronized. The idea then is to reduce the possibility of a cascade of resynchronizations that can occur. The distinction between fast and slow is not restrictive. A slow processor can change its dynamics and join the set of faster processors and vice versa. In an inefficient implementation of a logical system there are processors that could belong to either class. It is not necessary that a particular processor remain in either of them for the entire course of the computation. We introduce without preamble the notion of $W(j, n_j^*)$, also known as the sphere of influence of processor j. For the purposes of the next section, the reader is requested to assume that this sphere of influence denotes a set of processors that need to be resynchronized whenever j is resynchronized. This is done to ensure that the causality conditions of Chapter 2 are not violated. We will make this notion precise in later sections.

4.3.1. Concurrent Resynchronization

$\{C_n^i, i=1,2,\ldots,N\}$ are defined as follows,

$[T_m^{i,j}, m = 1,2,3\ldots]$ are Bernoulli times such that

$$\text{Prob}\,[T_{m+1}^{i,j} - T_m^{i,j} = k] = p_{ij}(1-p_{ij})^{k-1}, k = 1,2,3\ldots$$

$$\text{When } n \neq T_m^{ij},\ C_{n+1}^i = C_n^i + a_n^i$$

The a_n^i are partially ordered as follows,

$$a_n^2, a_n^3, \ldots, a_n^N > a_n^1 \text{ , for all } n$$

$$\text{Let}\{2,3,4,\ldots,N\} \in F$$

$$\text{and}\{1\} \in S$$

We can now describe the dynamics of the computation when processors within the "fast set" F interact between themselves, and when processors in the "slow set" S interact with the processors in F and vice versa. Detailed explanations follow the equations.

The probability that one or more of the faster processors communicate with a slower processor is given by p_S. p_F gives the probability that the slow processor sends a message to one of the faster processors.

Case 1: Fast Processors Interact within F

$$\text{For } i,j \in F, \text{ if } n = T_m^{i,j}, \quad \begin{cases} C_{n+1}^k = \min\{C_n^i, C_n^k\} + a_n^k, k \in W(j, n_j^*) \; U \; j \\ C_{n+1}^i = C_n^i + a_n^i \end{cases}$$

Let times $T_m^{S,F}$ and $T_m^{F,S}$ be defined as follows,

$$\text{Prob}\,[T_{m+1}^{S,F} - T_m^{S,F} = k] = p_F(1-p_F)^{k-1}, k = 1, 2, \ldots$$

$$\textbf{where} \quad p_F = 1 - \prod_{j \in F}(1 - p_{1j})$$

and

$$\text{Prob}\,[T_{m+1}^{F,S} - T_m^{F,S} = k] = p_S(1-p_S)^{k-1}, k = 1, 2, \ldots$$

$$\textbf{where} \quad p_S = 1 - \prod_{j \in F}(1 - p_{j1})$$

Case 2: Slow Processor Interacts with Fast Processors

Then for $i \in S$, and $k \in F$,

$$\text{If } n = T_m^{S,F}, \quad \begin{cases} C_{n+1}^k = \min\{C_n^i, C_n^k\} + a_n \cdot I\{C_n^k < C_n^i\}, k \in F \\ C_{n+1}^i = C_n^i + a_n^i \end{cases}$$

Case 3: Fast Processors Interact with Slow Processor

$$\text{If } n = T_m^{F,S}, \begin{cases} C_{n+1}^i = \min\{C_n^i, C_n^k\} + a_n \cdot I\{C_n^i < C_n^k\}, k \in F \\ C_{n+1}^k = C_n^k + a_n^k \end{cases}$$

In Case 1, we discuss the interactions among the ''fast'' processors themselves. When a processor $i \in F$, communicates with another processor $j \in F$, the processor j rollsback only if it is ahead in

simulated time. In case it does rollback, it also resynchronizes other processors with whom it had communicated in the recent past, while its local clock was above that of processor i. These processors which were indirectly affected by the message from i to j, are said to belong to the *sphere of influence*, $W(j,n_j^*)$, of processor j. We will discuss the effect of $W(j,n_j^*)$ on the distributed system in later sections. For now, suffice it to say that the fast processors recover from messages and recompute forward without losing any real time in the process.

Case 2 describes the dynamics of the system when a "slow" processor communicates with a "fast" processor. When a slow processor i sends a message with a lower time stamp to processor $k \in F$, then the processor F rollsback to the time stamp of the message. In addition, all other processors in the set F *concurrently* rollback to the time stamp of the message sent from processor i. It can be trivially shown that the computation will still be correct. Our objective now would to quantify the performance of such a synchronization method.

A qualitative description of the performance would be useful in arriving at a good model for the state transitions. The dynamics of the system can again be described by two states. In state S_F, all the processors in set F are ahead (in local times) of the processor 1 in S. In state S_S, processor 1 in the set S, is ahead of all the processors in the set F. When no communication occurs between the processors in the sets F and S, the system is in the state S_F. Whenever, communications occur among the processors in the set F, the system is still in the state S_F, as very little time is spent on the resynchronization (if any). The system jumps from state S_F to state S_S, whenever there is a communication from S to F. At the next step, the system again jumps back to the state S_F if a_n^j for $j \in F$ are at least twice as large as a_n^1 (if this assumption is relaxed then the bounds derived are upper bounds on the performance). The Global Time in the system is not affected if there were no communication from processors in set F to the processor in set S when the system is in state S. If there is indeed such a message as in Case 3 above, then the Global Time is penalized. The values of p_F and p_S, therefore, are both responsible for determining the rate of progress of the distributed computation. However, interactions within the processors in the set F themselves have little effect (if any) on the progress. The performance analysis would have

the advantage that it is not overly sensitive to the individual communication probabilities, but instead depends on some function of their values. The indices of performances would then also be able to indicate if physical processes need be lumped together into one logical process if they communicate too often in relation to the rest of the system. We will discuss this issue later, in the context of distributed simulation of dynamical discrete event systems.

4.3.2. Associated Markov Chain Representation

In this section, we will analyze the performance of concurrent rollbacks as a method of synchronization in distributed systems. The discrete time Markov chain is shown in Figure 9, with two states. The transition probabilities are p_F and p_S. The effect of p_S is to reduce growth of the Global Time. The analysis and the balance equations are very similar to those of the two-processor case.

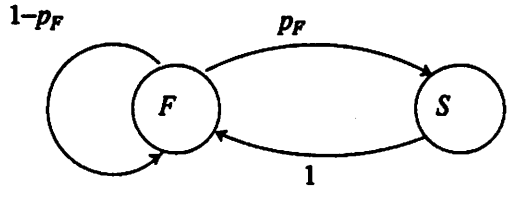


Figure 9: State Representation for Concurrent Resynchronizations

The balance equations are,

$$\pi(F) = (1 - p_F)\,\pi(F) + \pi(S)$$
$$\pi(S) = p_F \cdot \pi(F).$$

Solving the equations together with the fact that $\pi(F) + \pi(S) = 1$ we have

$$\pi(F) = \frac{1}{1+p_F} \quad , \quad \pi(S) = \frac{p_F}{1+p_F}.$$

The Wolf Coefficient α_1 which estimates the average growth of the Global Time GT_n is given by

$$\alpha_1 = \frac{1-p_F}{1+p_F} \cdot B + \frac{p_F(1-p_S)}{1+p_F} \cdot 2B \ ,$$

where $a_n^1 = B$ for all n. Similar bounds can be derived for other ranges of a_n^1. If the faster processors are not at least twice as fast, α_1 the upper bound on the attainable performance.

Example 1: Consider the three-processor system shown in Figure 10, where processors 1, 2, and 3 cooperate in an optimistic computation. Let 1 be the slowest processor in set S with rate of forward computation being R_1, the rates of 2 and 3 in set F are R_2 and R_3 respectively. After each time step, the processors can communicate with each other with probabilities given by p_{ij} (where i is the transmitting processor and j is the receiving processor). Consider the profile of the computation as shown in Figure 2. The forward computation time (or simulated time) is plotted against the real computing time in wall clock ticks. Processor 1 sends a message to Processor 2 at real time t_1. This message is (instantaneously) received by Processor 2 which rolls back to $L(1)$. In our aggressive protocol, Processor 3 also rolls back its local clock to $L(1)$. At this point both Processors 2 and 3 rollback for one time unit, while Processor 1 moves ahead in simulated time. At this stage either of the Processors 2 or 3 in F could send a message to roll back Processor 1. The typical profile illustrates this case. The Global Time (GT) of this system follows the trajectory given by the bold line.

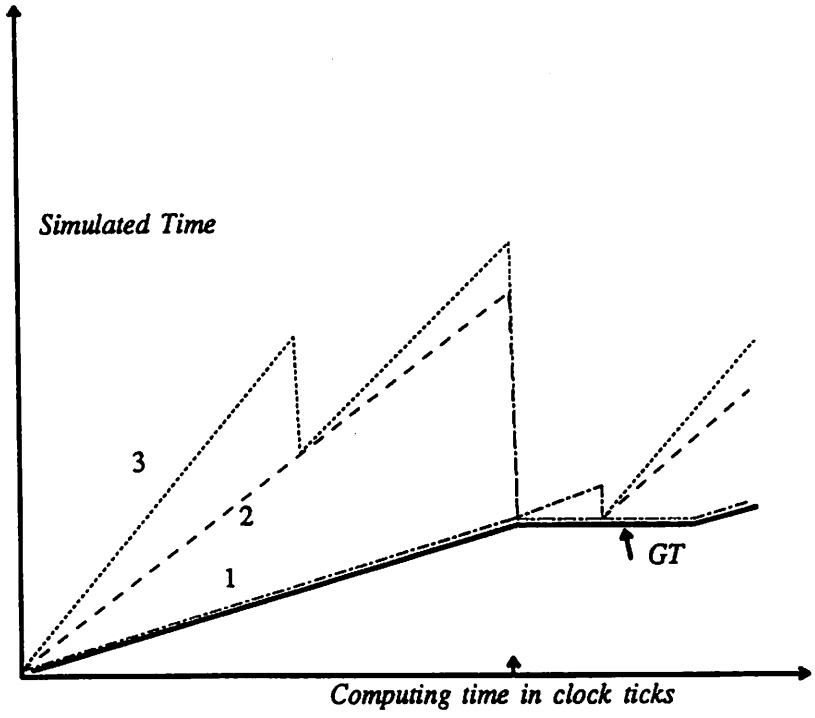


Figure 10: Typical Profile for Concurrent Resynchronization

To study the performance of concurrent resynchronizations further, we present the results of a simulation of the dynamics of the clocks in a four-processor system (See Figure 11). The four processors have rates, $a_n^1 = 1$, $a_n^2 = 2$, $a_n^3 = 4$ and $a_n^4 = 8$. The probabilities of interaction in Case 1, are low ($p_{ij} = 0.3$ for all i , j). In Case 2, the probabilties of interaction are relatively higher ($p_{ij} = 0.7$ for all i , j). Analogous to the two processor case, Case 1 with fewer communciations between processors progresses much faster than the case where processors communicate frequently. The buffer sizes in Case 1 are also larger. The Wolf Coefficients, plotted in Figure 12, also show that frequent communications penalize forward growth.

To show that the progress of the computation is quite sensitive to the rate of growth of the slowest processor Let us examine Figure 13. Here the slowest processor in the system was boosted from a forward computation rate of 1 to 3 simulated time units per computing time unit. The impulsive increase in the forward computation is reflected in the the progress of the overall computation itself.

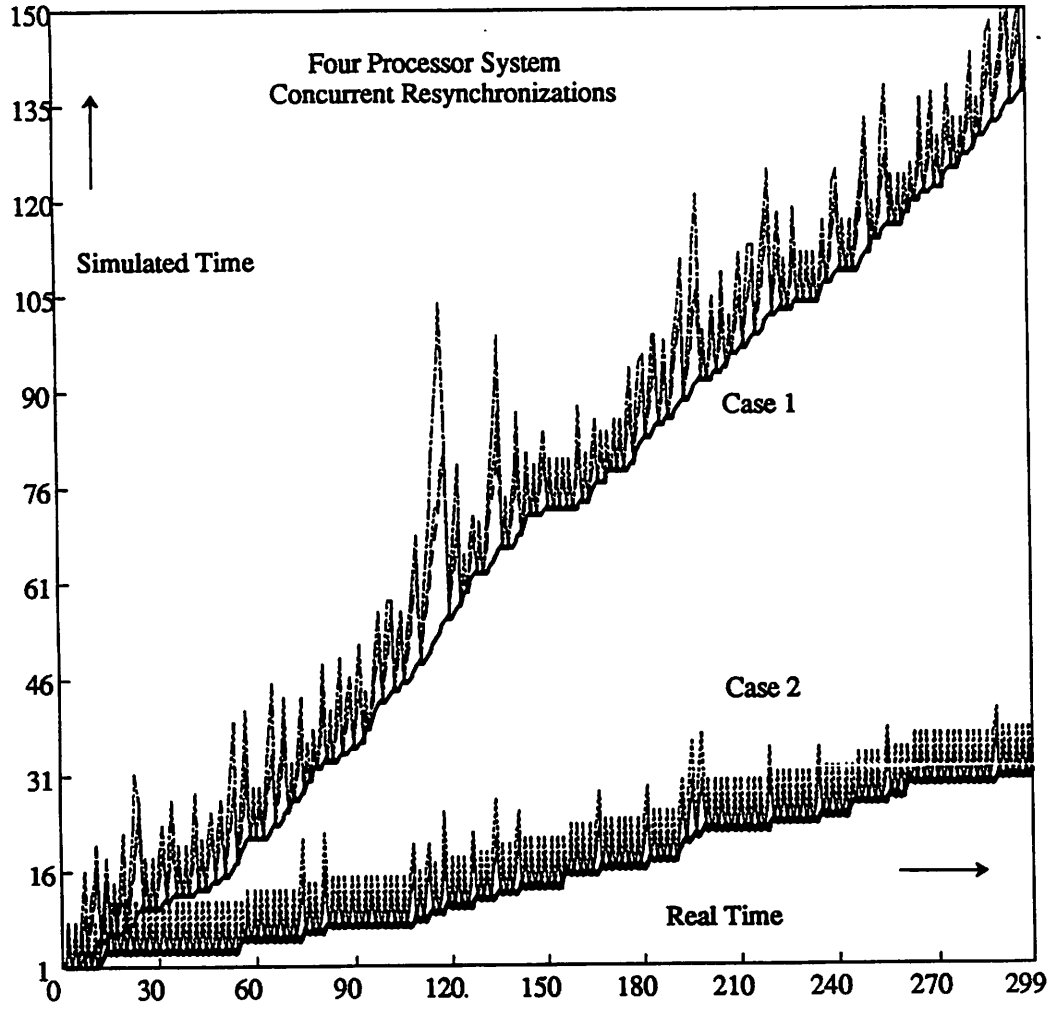


Figure 11

Case 1: Infrequent Communications.
Case 2: Frequent Communications.

This simulation extends the performance analysis to the case of multiple processors. Frequent communications again penalize progress. It may be more efficient to use the results of the simulation to "lump" processes together to reduce the communications between processes. The memory-time tradeoff is again observed. The performance of concurrent synchronizations is illustrated in this plot.

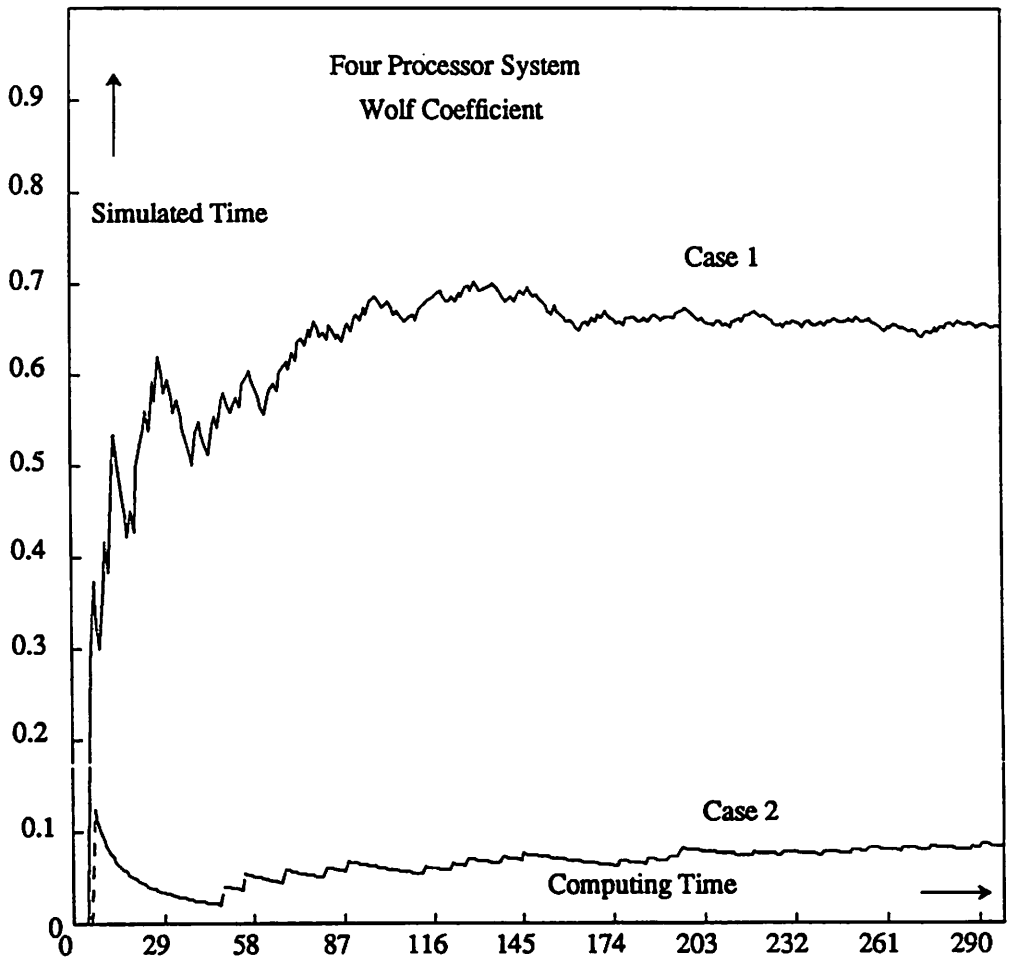


Figure 12

Case 1: Infrequent Communications.
Case 2: Frequent Communications.

The rate of growth index, or the Wolf Coefficient, α_1, is studied for concurrent resynchronizations in the figure above. The plot confirms the conclusions drawn in Sections 3 and 4. The analytical value of the coefficient is useful in designing efficient garbage collection algorithms. The overhead in concurrent resynchronization lies in the fact that in a distributed system consisting of a few thousand processors, broadcast can be costly unless special hardware is provided.

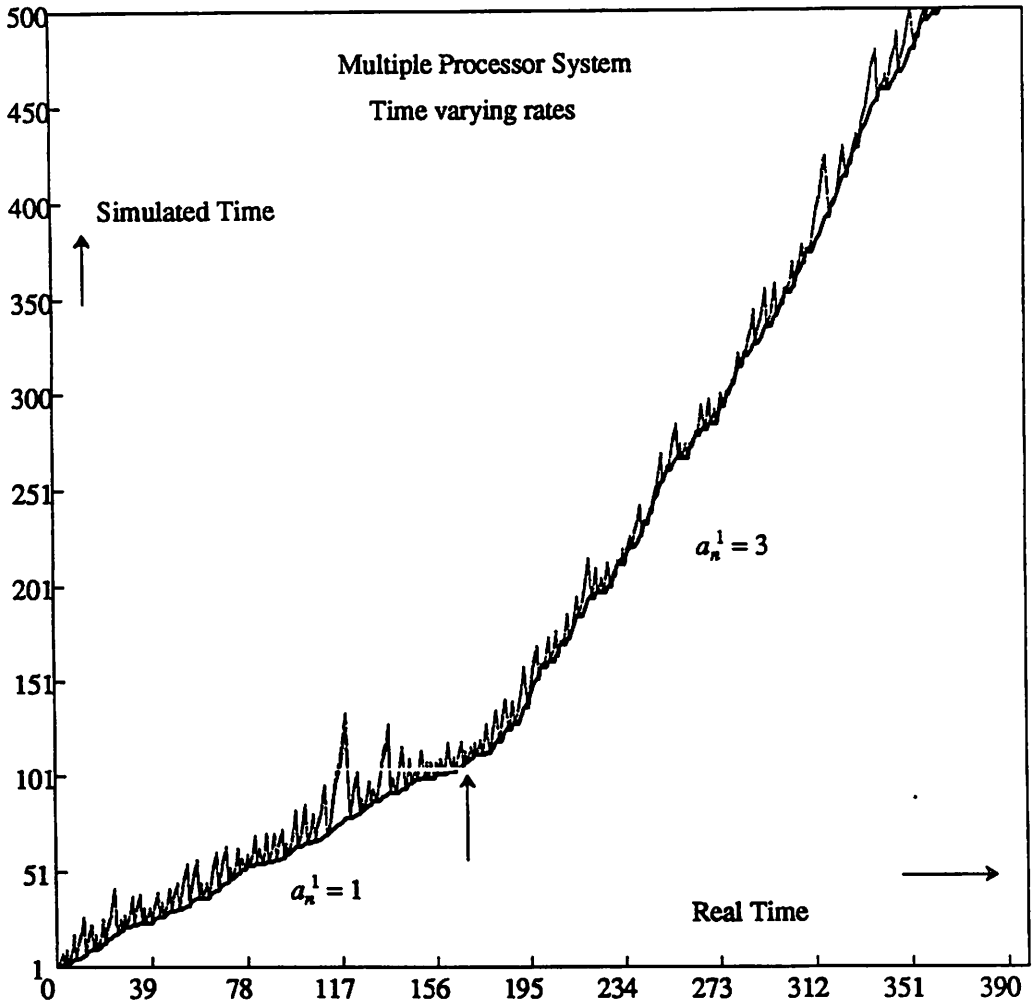


Figure 13

The progress of concurrent resynchronizations is sensitive to the rate of the slowest processor. In this example, the slowest processor was assigned a faster rate after time t=180. The immediate change in the profile is reflected in the plot.

Synchronization using concurrent rollbacks has the advantage that the processes are resynchronized very often, whenever a slow processor communicates with a fast processor. The memory requirements for this algorithm will be minimal. The overhead manifests itself in the broadcast mechanism needed to communicate with all the fast processors. If the distributed system consists of a few hundred processors, this overhead can be significant. Broadcast by each processor to a few select other processors in the system will find favor from the view point of efficiency.

We will now consider another synchronization algorithm. In this algorithm, processors communicate with each other in the same pattern as before, but with one important difference. There is no broadcast, and communication between processors results in the resynchronization if the message is received by a processor with a larger local time. Therefore, it is only the receiving processor that could rollback. If the receiving processor had communicated intermediate results to other processors, (which it now finds were erroneous), those processors are then rolled back to consistent states in *subsequent* time steps. In response to a single message, a cascade of secondary synchronizations could result. Exactly how this affects performance is the subject of the next section. We will then have the opportunity to show that Successive Resynchronization, often performs poorly with respect to Concurrent Resynchronizations. In addition, we will *simplify* concurrent rollbacks to introduce another efficient rollback algorithm [See MaWaMe88], Wolf, which boosts the efficiency of implementation further.

4.4. Successive Resynchronization

In this section, we will propose and examine another algorithm for the synchronization of asynchronous distributed computing on multiple processors. The effect of propagation of error on the progress of distributed computation is quantified.

At this point in our discussion, let us reexamine our premises. We have assumed that rollback (or resynchronization) takes exactly one time step. This is a conservative estimate of the time required to resynchronize. A realistic assumption is that the time required to resynchronize is proportional to the magnitude of the time rolled back in local times. This would imply that the performance bounds we

derive here are upper bounds on the performance. A *successive synchronization* scheme, such as one which will be discussed in this section will be penalized even further, for reasons that shall soon be apparent.

Let us now define the notion of a **Valid Error Path,** V_{ij}, from processor i to processor j. The length of a valid error path, $|V_{ij}|$ is defined as an integer measuring the number of processors in the valid error path. A valid error path, $V_{ij}^{n_1,n_2}$ is defined to exist between processors i and j iff there is a sequence of times $T_x^{i,k}, T_y^{k,l}, .., T_z^{n,j}$ such that $n_1 \le T_x^{i,k} \le T_y^{k,l} \le \ . \ \le T_z^{n,j} \le n_2$ for some $x, y, .., z$ and for some processors $k, l, .., n$. We further assume that

$$\text{Prob}\{|V_{ij}^{k,k+T}| = n\} = p_W^{n-1}(1-p_W),\ p_W \in [0,1], \text{ if } n < T$$

$$\text{else, Prob}\{|V_{ij}^{0,T}| = n\} = 1 - \sum_{i=0}^{T-1}(1-p_W)\cdot p_W^i, \text{ if } n = T$$

Furthermore, if $j \in F$ then if $T_m^{1,j}$ is the real time when processor $1 \in S$ communicates with j, then let $n=n_j^*$ be the smallest time such that $C_{n_j^*}^j \ge C_{T_m^{1,j}}^1$. The set of all k such that, $V_{j,k}^a$, where $a \doteq n_j^*, T_m^{1,j}$, is a valid error path is known as the **Sphere of Influence,** $W(j, T_m^{1,j} - n_j^*)$.

The Sphere of Influence can be determined by a number of methods. A conservative approach would be to use minimum communication times and execution times to determine the maximum range of propagation of the error message from node j. Alternatively, it could be iteratively determined at run time, using a suitable dynamic programming algorithm (i.e. Bellman-Ford Algorithm, [BeTs89]), where each processor routinely determines its own Sphere of Influence using marker messages. The Sphere of Influence can then be stored in the form of a look-up table, and used in an algorithm called the Wolf, which will be discussed in the next chapter. The probability p_W is also sometimes called the *Send Factor* by some researchers (See [AtSe88]).

Having introduced the notion of a sphere of influence, it is convenient to introduce the notion of *radius* of the sphere of influence. A node k in the sphere of influence of node j, is defined to be at a radius of influence of r_{jk} iff at least $r_{jk}-1$ processors are in between processors j and k for any valid error path. Here R denotes the maximum radius of the sphere of influence.

Whatever the nature of the computation, the effect of the sphere of influence on the computation is substantial. A succession of rollbacks can arise as a result of a single message from the slow processor to one of the faster processors. The effect of the rollback of the faster processors *temporarily* reduces the slope of the Global Time, GT_n. The average rate of growth α_2 is only reduced if one of the faster processors sends a message to a slower processor (which is now ahead) during resynchronization. Since there are a number of likely opportunities for a faster processor to send a message to a slower processor especially when the sphere of influence is large, the net effect on the coefficient α is large if p_0 is large. Here p_0^j is defined as the probability processor j can send a message to the slow processor at each time step. For the sake of simplicity, let us assume that $p_0^j = p_0^k = \cdot = \cdot \doteq p_0$ implying that each fast processor is equally likely to send a message to the slower processor. The faster processor can only affect the slower processor if the slower processor is ahead.

Let us now examine the dynamics of the distributed asynchronous computation in this framework.

Dynamical Equations

$\{C_n^i, i=1,2,\ldots,N\}$ is defined as follows,

$[T_m^{i,j}, m = 1,2,3,\ldots]$ are Bernoulli times such that

$$\text{Prob}[T_{m+1}^{i,j} - T_m^{i,j} = k] = p_{ij}(1-p_{ij})^{k-1}, k = 1,2,3\ldots$$

$$\text{When } n \neq T_m^{ij},\ C_{n+1}^i = C_n^i + a_n^i$$

The a_n^i are partially ordered as follows,

$$a_n^2, a_n^3, \ldots, a_n^N > a_n^1 \text{ , for all } n$$

$$x_j^n = \infty \text{ , for all } j \text{ , } n.$$

$$\text{For } i \in S \text{ , } j \in F \text{ , if } n = T_m^{ij} \text{ , } \begin{cases} C_{n+1}^j = \min\{C_n^j, C_n^i\} + a_n^j I\{C_n^j < C_n^i\} \\ x_{n+2}^k = \min\{C_n^j, C_n^i\} \text{ , for all } k \ s.t.\ r_{jk}=1 \\ \cdots \\ x_{n+1+R}^l = \min\{C_n^j, C_n^i\} \text{ , for all } l \ s.t.\ r_{jl} = R \\ C_{n+1}^i = C_n^i + a_n^i \end{cases}$$

$$\text{For } i \in S,\ j \in F,\ \text{if } n = T_m^{j,i},\ \begin{cases} C_{n+1}^i = \min\{C_n^i, C_n^j\} + a_n^i\, I\{C_n^i < C_n^j\} \\ C_{n+1}^j = C_n^j + a_n^j \end{cases}$$

$$\text{For } i,j \in F, \text{if } n = T_m^{i,j},\ \begin{cases} C_{n+1}^k = \min\{C_n^i, C_n^k\} + a_n^k,\ k \in W(j, n_j^*)\ U\ j \\ C_{n+1}^i = C_n^i + a_n^i \end{cases}$$

$$\text{For all } i,\ j,\ n \neq T_m^{i,j},\ \begin{cases} C_{n+1} = \min\{C_n^i, x_n\} + a_n^i \end{cases}$$

The dummy states $x_n^j, j \in F$ represent the memory in the system, when a single rollback can cause a succession of rollbacks in the system, keeping the GT_n temporarily at a slope zero.

Once again, the condition that a_n^j for $j \in F$ are greater than a_n^1 for all n can be relaxed further to the case where this inequality holds only for those n, where $C_n^j \leq C_n^1$.

4.4.1. Markov Representation of Successive Resynchronization

We will now analyze the performance of the asynchronous system with successive resynchronizations. We will simplify our model further, by letting p_W alone represent the effects of the sphere of influence (tantamount to assuming there are a very large number, possibly infinite, number of processors in the system). This assumption implies that we derive a conservative upper bound on the progress of the distributed computation. The analysis is much simpler, and the bound will be accurate if p_F is small. This is also the case of most interest.

If a slow processor has infrequent communications with the faster processors, a very long cascade (possibly infinite!) of successive resynchronizations can result. Fortunately, in a real system, the number of resynchronizations would be restricted to the number of processors in the system.

The Markov representation of the system consists once again of two states. In state S_F all the "fast" processors are ahead (in local times) of the "slow" processor. In state S_S, the "slow" processor is at least as large a local time as the local times on one or more of the "faster" processors. In state

S_S slope of the global time is zero. The system remains in state S_S with a probability p_W at each time step. This Markov chain is described in Figure 14.

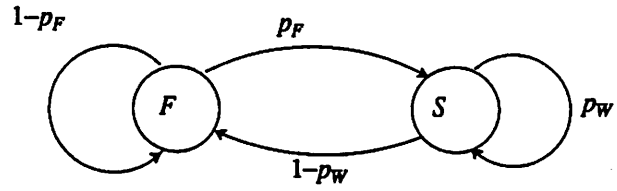


Figure 14: Successive Resynchronizations

The balance equations for this system are:

$$\pi(F)\cdot(1-p_F)+\pi(S)\cdot(1-p_W)=\pi(F)$$
$$\pi(S)=\pi(S)\cdot p_W+\pi(F)\cdot p_F.$$

The invariant probabilities will then be

$$\pi(S)=\frac{\frac{p_F}{1-p_W}}{1+\frac{p_F}{1-p_W}},\quad \pi(F)=\frac{1}{1+\frac{p_F}{1-p_W}}.$$

The Wolf Coefficient, α_2 is then given by

$$\alpha_2 \le \pi(F)\cdot B+\pi(S)\cdot(1-p_O)\cdot 2B.$$

The effect of the communications between the faster processors is captured by p_W, while p_0 describes the effect on α_2. Let us now look at an example of a five-processor system whose profile is depicted in Figure 15. From the examination of Figure 15, one can easily verify that α_2 is indeed an upper bound. Here α_2 would be accurate for small values of p_W, or alternatively, large values of p_0. Note α_2 can be larger than 1.

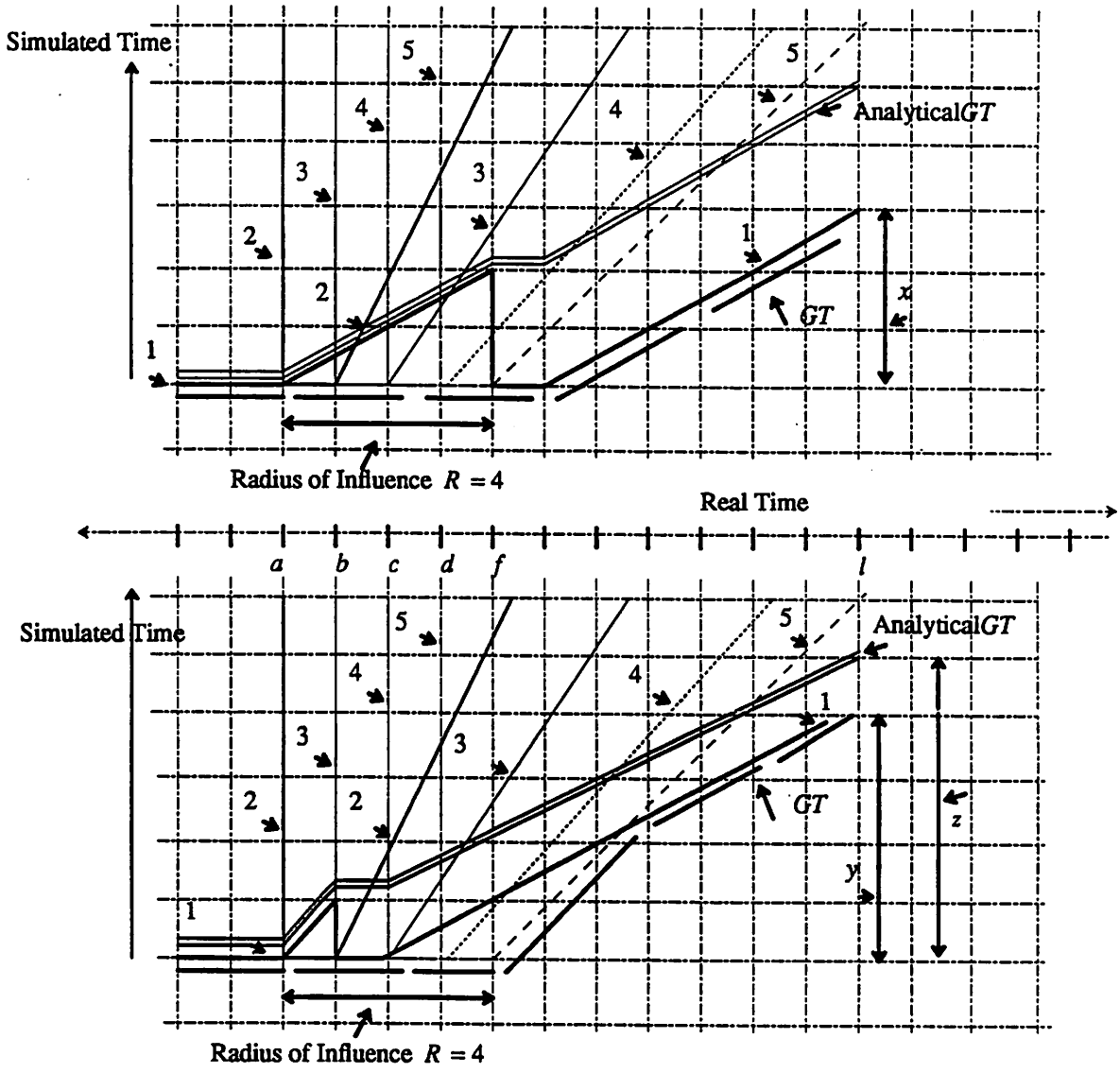


Figure 15: α_2 is an upper bound.

In this figure we describe asynchronous computation using SR on five processors. 1 is the slowest processor, and processors 2 , 3 , 4 , 5 are in its sphere of influence. x and y are the true estimates of the global time at the end of the computation for two possible cases. z is the upper bound predicted in our analysis.

In Figure 15, we reconstruct two of a number of possible cases which can occur in the case of Successive Resynchronizations. In every case, however, the analytical expression derived in the text is an upper bound. To show that this is the case, we examine Case 1 above, where a set of five processors cooperate in an asynchronous computation. Processor 1 resynchronizes at point f in real time, after the entire sphere of influence has resynchronized. The actual GT_n follows the trajectory shown in the figure. However, the analytically derived GT estimates the growth in simulated time as z, as opposed to actual growth of x reached.

In the second plot in Figure 15, another possible sequence of events is traced. Here, Processor 1 resynchronizes early in the computation, at point b itself. The actual GT reaches y which, though less than z is closer to it than x. Therefore, it should be clear that our upper bound is accurate for cases of small p_W and large p_0.

Example 2: Consider again a three processor system with Processors 1, 2 and 3. A typical execution profile is shown in Figure 16. Processor 1 sends a message to Processor 2 at time t_1 and local time $L(1)$, causing Processor 2 to rollback from $L(2)$ to $L(1)$. However, Processor 2 had earlier communicated with Processor 3 in the interval $L(2) - L(1)$. In the general model introduced in the earlier part of the section, this interaction was captured by a probability p_W. As seen in the profile, the Processor 2 first rollsback and then Processor 2 rolls back. Note that this is in contrast to the case of Example 1, where both 2 and 3 rolled back concurrently. The slope of the Global Time (GT) remains zero for a longer period which could lead to a smaller growth in forward computation, because each of these faster processors could resynchronize the slower processor (with probability p_0). However, in the case of Example 1, since more processors roll back concurrently, there is a greater likelihood that one of them could resynchronize Processor 1 (implying $p_S \geq p_0$), resulting in a reduction in the forward rate of computation in the next time step.

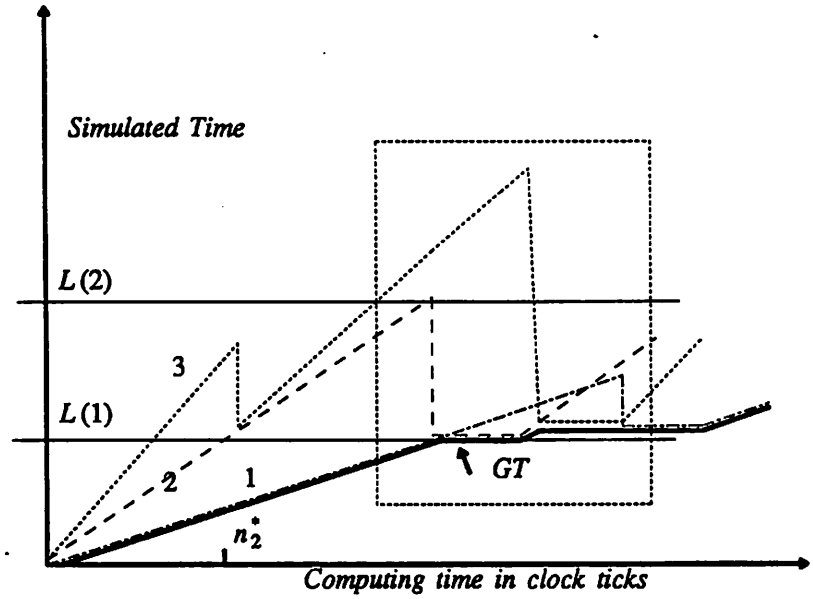


Figure 16: Profile for Successive Resynchronizations

We will now examine the results of the simulation of the clock dynamics of a four-processor system synchronized using Successive Resynchronizations. Here, two new factors come into play. The first is the probability, p_W, that a faster processor communicates with other processors in the system before being resynchronized by the slower processor. The second, p_0, is the probability that models the fact that the fast processors can communicate with the slower processor while recovering from resynchronization.

In Figure 17, we study the dynamics of two cases where p_W is kept constant, and the value of p_0 is varied. With a larger p_0, the probability that the slowest processor is resynchronized (thus penalizing the average rate of growth) is enhanced, and the performance of Case 2 illustrates this. In Case 1, the performance is better. The buffer sizes are much larger than the case of Concurrent Resynchronizations because the processors communciate with each other (or resynchronize) less frequently. The smaller values of the Wolf Coefficients in Figure 18, also confirm our analytical results, that CR requires less memory that SR and also progresses much faster.

Figure 19 illustrates another important result. The fact that p_W is large is not suffcient to penalize the α. It is necessary that p_0 be large too.

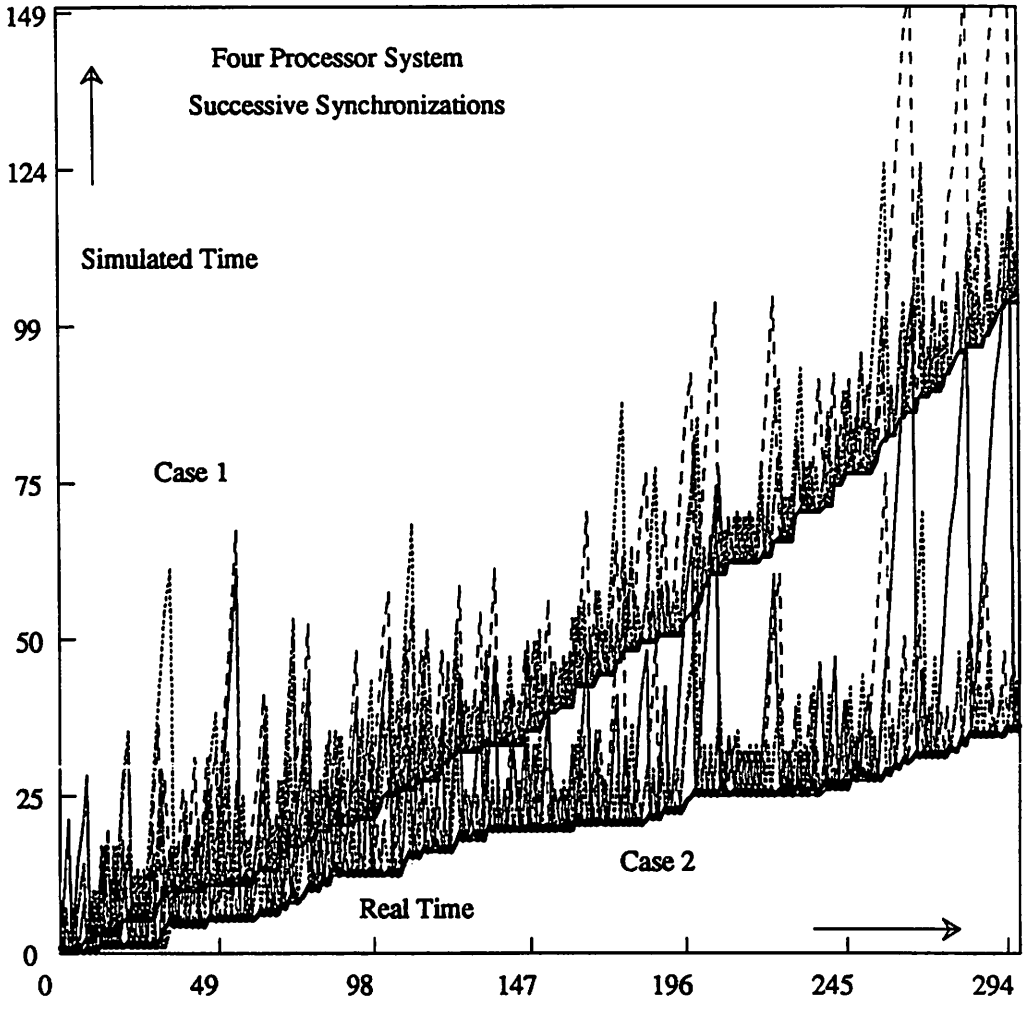


Figure 17

This simulation studies the effect of p_0 on two runs of Successive Resynchronization on a system with 4 processors. The value of p_W was kept constant at 0.7, while p_0 was 0.4 in Case 1, and in Case 2 p_0 was kept at 0.8. Note the slower progress of the distributed computation in Case 2. Both computations run slower than the previous examples with Concurrent Resynchronizations.

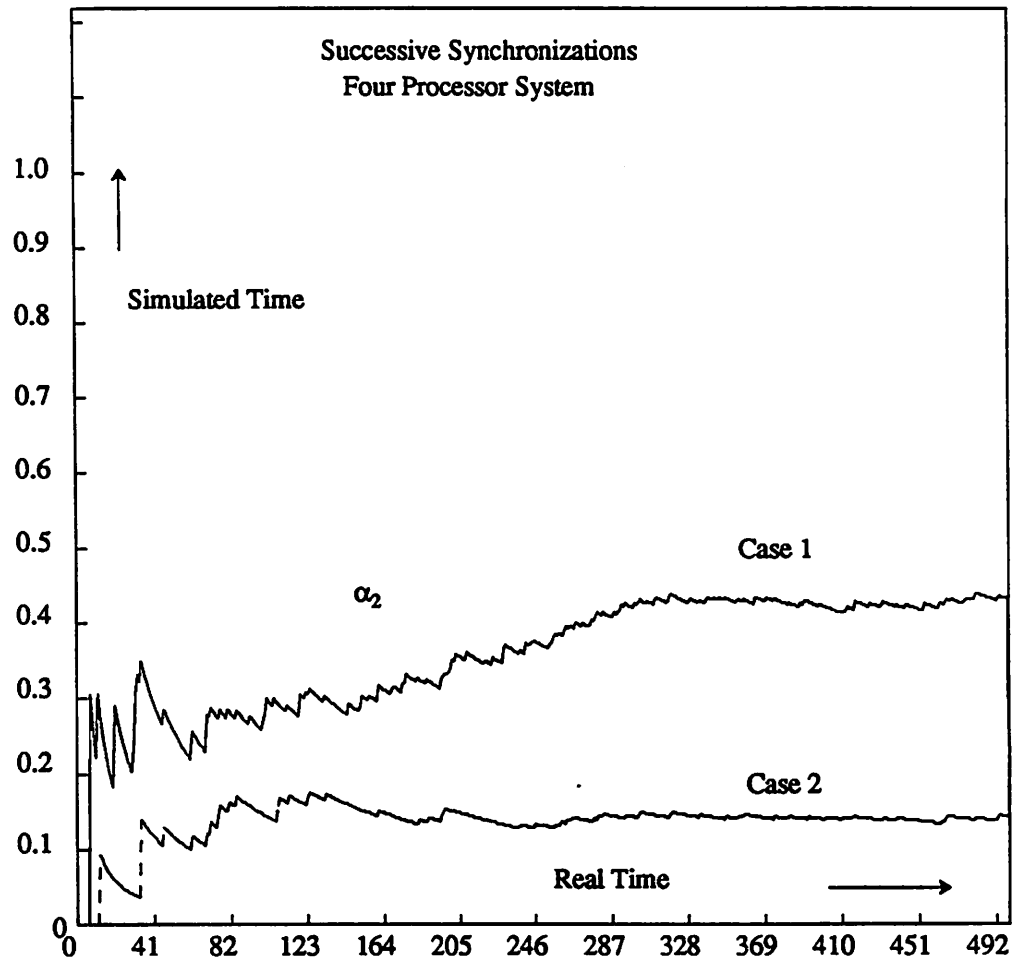


Figure 18

The observed values of α are plotted for the case of distributed computation with successive resynchronizations, for two different cases. A higher value of p_0 in Case 2, (p_0=0.7 , $p_W = 0.7$) penalizes the performance of this case relative to Case 1, ($p_0 = 0.4$, $p_W = 0.7$).

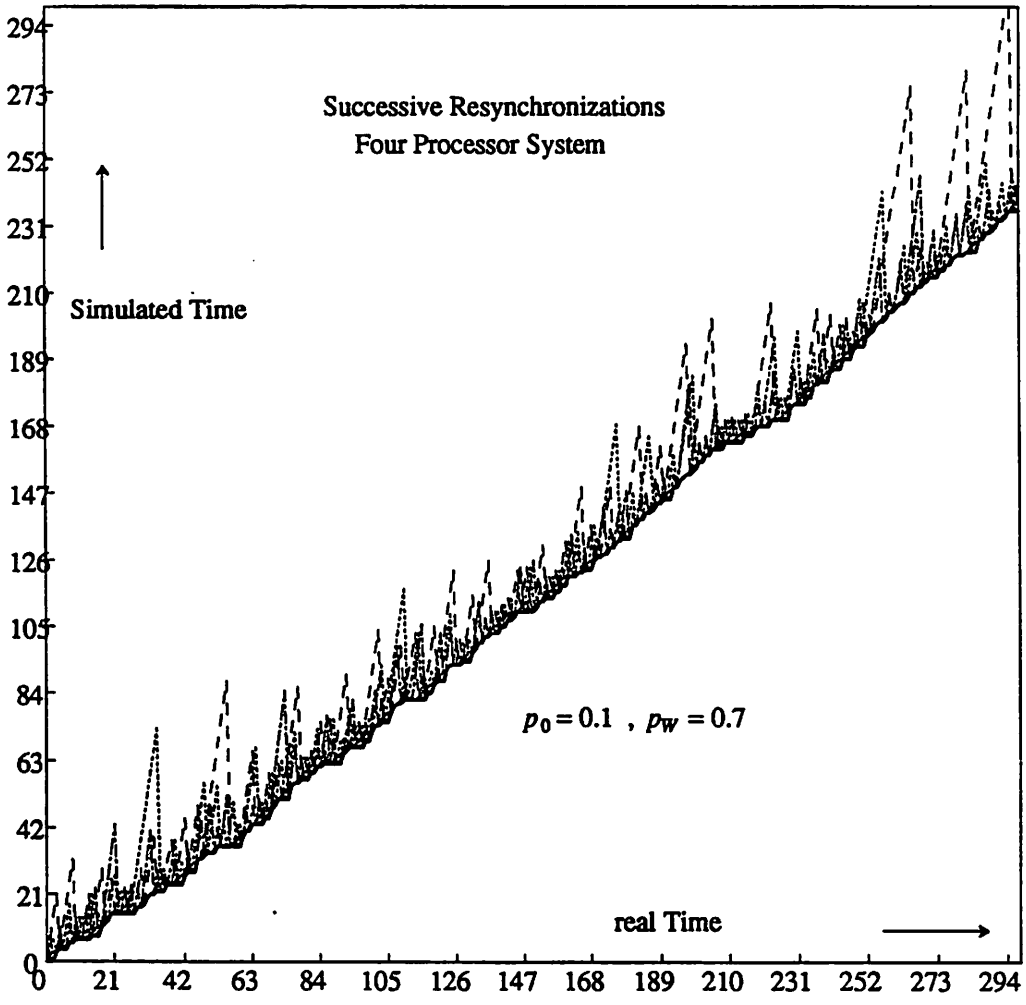


Figure 19

This simulations shows that a large p_W can affect GT_n but not α, unless p_0 is large as well.

Comparing Successive Resynchronizations with Concurrent Resynchronizations we have from the analytical expressions for the Wolf Coefficient, the Figures 20 and 21, where their relative values are plotted.

Figure 22 and 23 illustrate the performance of SR and CR under similar computation to communication ratios. It is seen that CR outperforms SR both in terms of rate of forward progress as well as in the terms of buffer sizes (as these are proportional to drift in the clocks.)

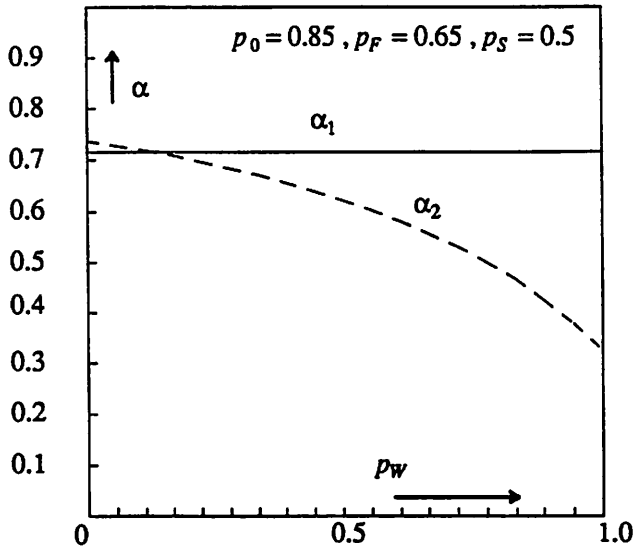


Figure 20: Comparison of Wolf Coefficients

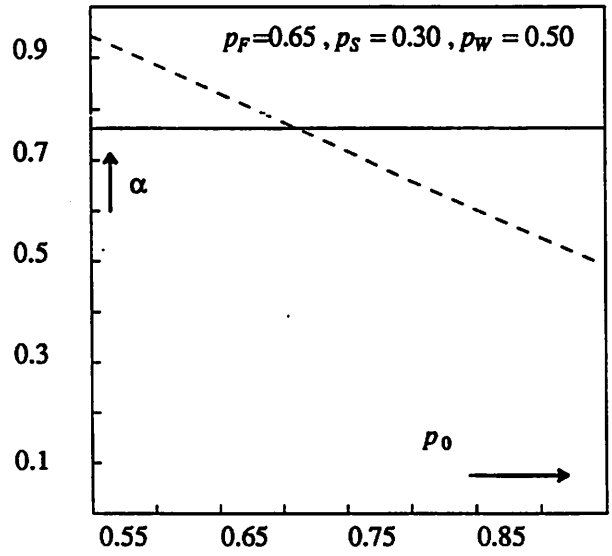


Figure 21: Comparison of Wolf Coefficients

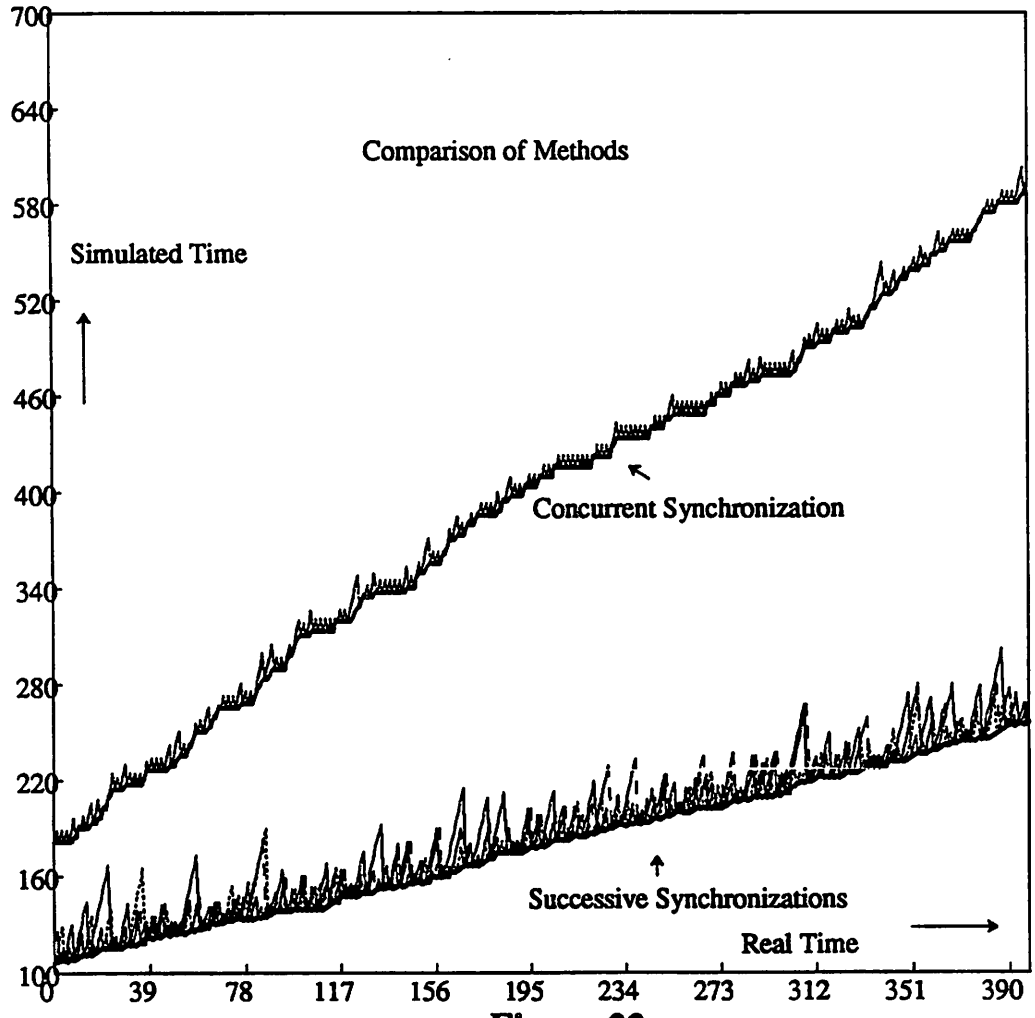


Figure 22

This study shows the performance of two synchronization algorithms, under identical conditions. The probabilities of communications were the same for the two cases, and the forward computation rates were also matched. In the case of "Successive Resynchronization," p_W or the *SendFactor* was assumed to be 0.7. Note also the smaller memory requirements for the case of concurrent synchronizations.

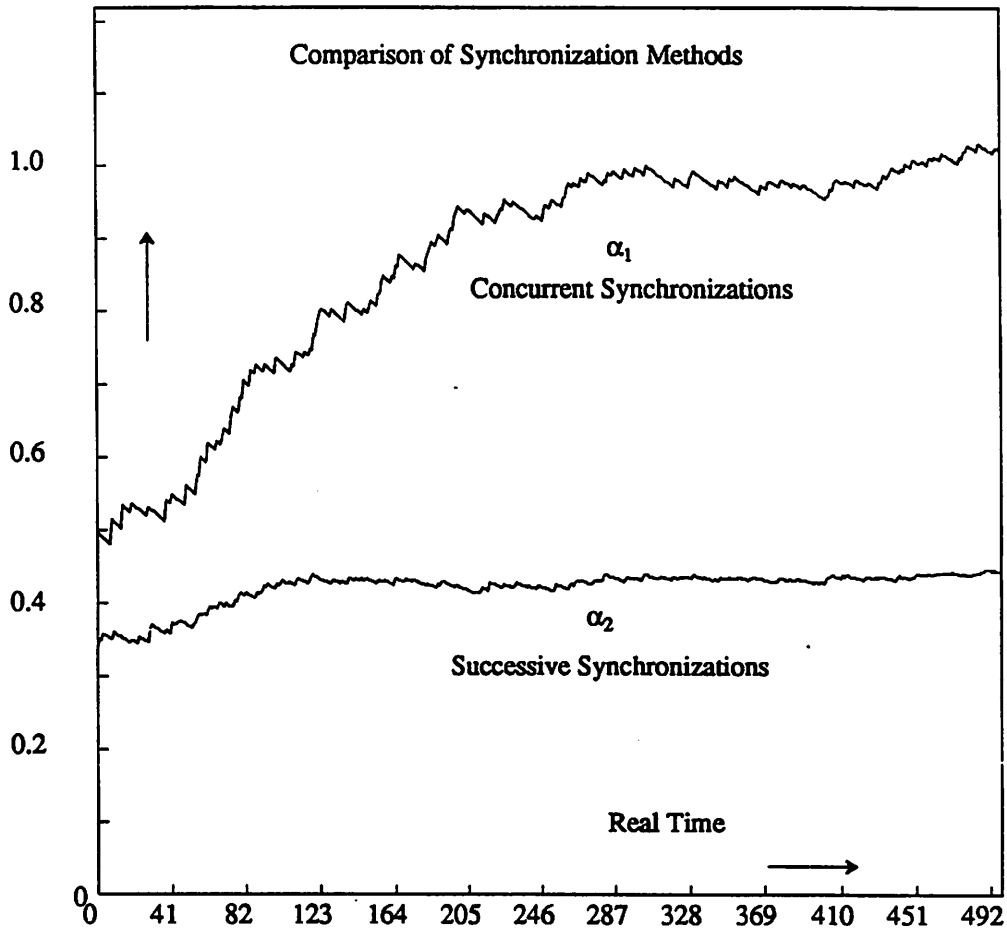


Figure 23

The Wolf Coefficients are compared for CR and SR under identical conditions. Performance of CR is clearly superior.

4.5. Summary

In this chapter, a new model has been proposed for the analysis of synchronization mechanisms in self-synchronizing concurrent computing systems (SESYCCS). Such systems are important in the efficient solution of large scale distributed computation that can be described by Dynamic Computation Graphs.

In our study of the two-processor logical system, we have derived an analytical estimate of the progress of distributed computation. The rate of growth of Global Time is not the rate of

growth of the slowest processor in the system, but has been shown to depend on both the rates of the two processors as well as the probabilities of their interaction. Frequent communications between processors result in a slower growth in the computation. The Wolf Coefficient, α, which represents the average growth in the Global Time (GT_n) in addition to describing the efficiency of the self-synchronization, is also useful in the design of efficient and simple garbage collection algorithms.

Simulation results confirm that the rate of growth is indeed that given by the analytical result.

Communication delay increases the memory of the system. The effect of the communication delay is to introduce new states into the performance analysis state representation, and this effect was quantitatively determined in the chapter. Closed form results have been derived for a few cases. Communication delay can cause a further degradation in efficiency of a SESYCCS, especially if communications are frequent. Results of our analysis can be used to "lump" processes together to reduce this interaction.

The results were then extended to the case of the multiple processor SESYCCS. A new algorithm, Concurrent Resynchronization (CR), has been proposed to synchronize a SESYCCS, where synchronization is enforced separately from the computation. Concrete and exact results of the performance of this scheme have been derived. Concurrent Resynchronization has the advantage that the rate of growth is not penalized by communications between the larger number of processors in the computing system. This is because all processors are resynchronized whenever an inconsistency in the local clocks is discovered. The memory requirements are also much smaller. However, the price paid is in the additional communications required to enforce the synchronization. Use of special synchronization hardware will, therefore, be of merit.

The second algorithm proposed, Successive Resynchronization (SR), describes the performance of a synchronization algorithm where the synchronization is implemented as a part of the computation. Exact results are derived in the performance analysis. The effect of propagation of

error amongst communicating processes is particularly highlighted, and the drop in the rate of growth of the Global Time is attributed to an increased likelihood of a succession of resynchronizations in the system. Closed form results provide a concrete basis for comparison between SR and CR.

CR is then extended to include only those processors in the resynchronization which belong to the *Sphere of Influence* of each processor. The efficiency is shown to be improved as a result. In Chapter 6, we will describe the implementation of this algorithm in the distributed simulation of discrete-event systems.

The theory presented in this chapter is new. The results and analysis are robust and general enough to allow the analysis of a number of other synchronization mechanisms for SESYCCS as well.

We conclude that the separation of synchronization from computation in the logical system has the advantages of an increased efficiency of implementation, smaller memory requirements, and a reduction in the burden imposed upon the user of the concurrent computing system.

Current work consists of an extension of these results to the analysis of systems where the time required for resynchronization is proportional to the amount resynchronized in local times and also to incorporate the effect of finite memory.

Chapter 5

Randomized Algorithms for Self-Synchronization

This chapter introduces and develops a new class of self-synchronization algorithms called the Randomized Algorithms (RA). It is envisaged that these algorithms would take advantage of the knowledge of the dynamics of the computation, and thus provide a basis for systematic adaptive synchronization of distributed computation.

The algorithms for self-synchronization for Static Computation Graphs (Chapter 3) and for Dynamic Computation Graphs (Chapter 4) were proposed for two distinct classes of scientific computation. Static Computation Graphs are proposed with a specific fork/join type computation in mind. In a number of scientific applications, especially in digital signal processing, algorithms can be constrained to fit this model. The static structure of the computation is first identified by the synchronization algorithm and the resource allocation schemes are subsequently optimized. Our results from Chapter 3 demonstrate that it is possible to develop an efficient time-shared SESYCCS environment for this class of computation.

Dynamic Computation Graphs, on the other hand, are very broad in their scope. The structure of the computation is random and varies dynamically with time. Our solution to the problem of self-synchronization was to develop a theory of self-synchronization that separated the roles of computation

from that of synchronization. Our analysis in Chapter 4 provides concrete results on the performance of several algorithms for self-synchronization. Provision of explicit synchronization facilities was shown to be advantageous in the SESYCCS environment.

However, no attempt was made to "learn" the dynamics of the distributed computation itself to improve upon the efficiency of self-synchronization. It can be argued that for a sufficiently large class of scientific computation, the behavior of the computation can be adaptively identified as the computation progresses. Nevertheless, since the computation is itself random, it would be unusual to expect to be able to predict the its behavior exactly (as in Chapter 3). But it would not be unreasonable to expect that the some knowledge of the computation can be used to optimize the performance of the distributed implementation.

This chapter addresses this very question. The question is reformulated as follows. Can the problem of self-synchronization be couched in the terms of statistical estimation ? The input to the adaptive algorithm would be the sequence of observations representing the evolution of the local times on other processors in the distributed system. Each processor then tries to estimate the local clocks on the remote processors on the basis of these observations. If each processor were to communicate with other processors at local times that are comparable, the penalties of resynchronization or additional memory storage would not be as severe.

This chapter also addresses another important question. Are SESYCCS realizable for DCGs ? In other words, are the memory requirements finite? Flow control and memory requirements constitute an important part of the design of any distributed computing system. It is essential that algorithms for flow control and garbage collection be as efficient as possible. Otherwise, this would negate the benefits accrued from self-synchronization.

Hitherto an important requirement in a distributed message passing environment was to ensure that communications between processors were *blocking*. This implied that both the sending and the receiving processors had to be ready to send and receive the message respectively. This condition ensured that buffer sizes were finite. In the *non-blocking* communication environment there is no such

restriction. Thus there is a likelihood that message buffers would overflow when the distributed computation was inappropriately implemented.

Relaxing the assumption of *blocking* communications allows a practical and efficient realization of a SESYCCS, precluding the necessity of providing large buffers for communication purposes. We show that under some patterns of communications between processors *non-blocking* communications can be implemented in a stable and efficient manner with finite memory requirements.

Bernoulli communications imply that the different processes in the logical system interact with each other at some point in time. This notion of irreducibility leads directly to the stability of buffers. Note the similarity in flavor with results derived in Chapter 3, where we proved that non-interacting (e.g. independent) processes can affect stability in buffer sizes.

5.1. Randomized Self-Synchronization

Our previous results in analyzing the performance of SESYCCS showed that the progress of the computation depends on the rates of the forward growth in the constituent processors. The computation is particularly inefficient if the divergence between the rates of individual processors is very high. In addition if the time taken to undo the effects of erroneous computation are taken into consideration, it is necessary to ensure that the local clocks of individual processors do not diverge "quickly." Indeed, most experimental evaluations of optimistic computation have been based on balanced realizations. How this balance can be achieved is the subject of this chapter.

Each processor in the system makes some decisions on how the clocks of other processors evolve in time. When a communication between processors is imminent, the transmitting processor ensures that some randomized algorithm is followed that estimates the local times of the remote processors. The transmitting processor may then decide to wait to allow other processors to catch up, or assign more resources to its own computation should it appear that it were slower than the other processors in the system. We will analyze randomized algorithms (RA) for the case of two processors. In a more

general framework, the algorithm has to be implemented to estimate the clocks of all processors in the immediate sphere of influence.

The problem can now be formulated as follows: Two processors participate in the computation. The clock of Processor 1 is given by C_n at a discrete time n. The evolution of the local clock in the $n+1$ step is then given by (where ζ_n is a random variable)

$$C_{n+1} = C_n + \zeta_n$$

Processor 2 tries to estimate the clock at instant $n+1$, given that the clocks synchronized earlier at instant n. The evolution of the *estimator* U_n can, therefore, be given by

$$U_{n+1} = C_n + V_n$$

ζ_n and V_n are the decision variables. It must be noted at this point that V_n is the estimator of the remote clock (in Processor 2). We must also assign a cost function to the estimation algorithm. A fair cost function would penalize Processor 2, when U_{n+1} is either ahead or behind C_{n+1}. To derive some analytical results we proceed to a few specific cases.

Case 1:

Let us assume that ζ_n's are i.i.d on $(0, \infty)$, and V_n can take values in $[0, V]$ Therefore, given

$$C_{n+1} = C_n + \zeta_n$$
$$U_{n+1} = C_n + V_n$$

The problem then is to come up with a sequence $V_1, V_2 \ldots$ which minimizes a discounted cost function, W_n which consists of the contributions of a distance function $f(\,)$ over a number of intervals in discrete time. With $\beta \leq 1$ a constant, and $E[\,]$ denoting expectation we have,

$$W_n = E[\sum_{i=1}^{n} \beta^i \cdot f(U_n, C_n)]$$

Since the ζ_n's are i.i.d the estimation procedure at each step is independent of error in the previous step. So we can rephrase our desired goal as a single step optimization, where we have to select a v such that,

$$\underset{v}{\textit{minimize}}\; E[f(v, \zeta_1)]$$

Let us also assume that $f(v, \zeta_1)$ can be written as

$$f(v, \zeta_1) = a(v-\zeta_1)^+ + b(\zeta_1 - v)^+$$

Here a and b represent the costs associated with overestimating and underestimating the remote clock respectively, and $(x)^+$ equals x if x is positive and equals 0 if x is negative. These costs directly relate to the costs of resynchronization, lost computation, and additional memory required for storage of messages ahead in local times.

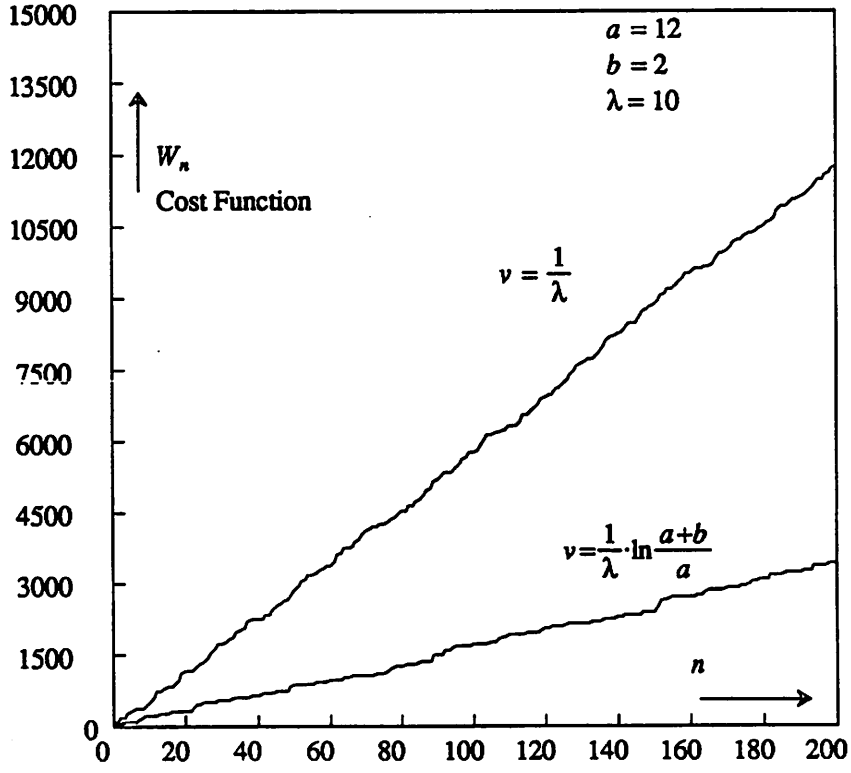


Figure 1: Clock Estimation in an Asynchronous Environment

In the first strategy, the mean $\frac{1}{\lambda}$ is used to estimate the remote clock v_n (which has an exponential pdf), in the second case, $v = \frac{1}{\lambda} \cdot \ln\frac{a+b}{a}$. The latter gives the best performance.

If ζ follows a probability density function given by $g(\zeta)$, the cost function can then be written as

$$Ef(v, \zeta) = a \int_0^v (v - \zeta)\, g(\zeta)\, d\zeta + b \int_v^\infty (\zeta - v)\, g(\zeta)\, d\zeta$$

Let us consider the case where $g(\zeta)$ describes an exponential density function. The cost function can then be written as

$$Ef(v, \zeta) = a \int_0^v (v - \zeta)\lambda e^{-\lambda\zeta} d\zeta + b \int_v^\infty (\zeta - v)\lambda e^{-\lambda\zeta} d\zeta$$

Carrying out the simplification

$$Ef(v, \zeta) = av + \frac{a}{\lambda} e^{-\lambda v} - a - \frac{b}{\lambda} e^{-\lambda v}$$

Minimizing over v gives

$$v^* = \frac{1}{\lambda} \ln \frac{a+b}{a}$$

Therefore, at each step, the V_n is assigned the value v^*. Similar analysis can be carried over to other probability density functions. The performance of this strategy is observed in Figure 1, where two different strategies are compared. If the processors communicated at irregular intervals this analysis can be extended to estimate the sum of a number of random variables.

Case 2:

Consider now the case where ζ_n's form a Markov sequence of numbers with a transition matrix A. The solution to the estimator problem is then given by the V_n which minimizes

$$W_n = E[f(V_n, \zeta_n) | \zeta_{n-1}]$$

This function can sometimes be easily minimized as illustrated in the following example.

Example 1: Consider the irreducible and aperiodic four-state Markov chain which describes the evolution of ζ_n with discrete time. The probabilities of transition are shown in the Figure 2.

Let us also assume that the system started in state 2, with value $\zeta_1 = 2$. The two possible transitions are to states $\zeta_2 = 1$ or to $\zeta_2 = 3$.

The cost function in terms of V_2 will be

$$W_2 = a(V_2 - 1)^+ \cdot p(1 \mid 2) + b(3 - V_2)^+ \cdot p(3 \mid 2)$$

V_n then is chosen such that the cost function is minimized. This depends on the probability transition

matrices and also on the values of a and b.

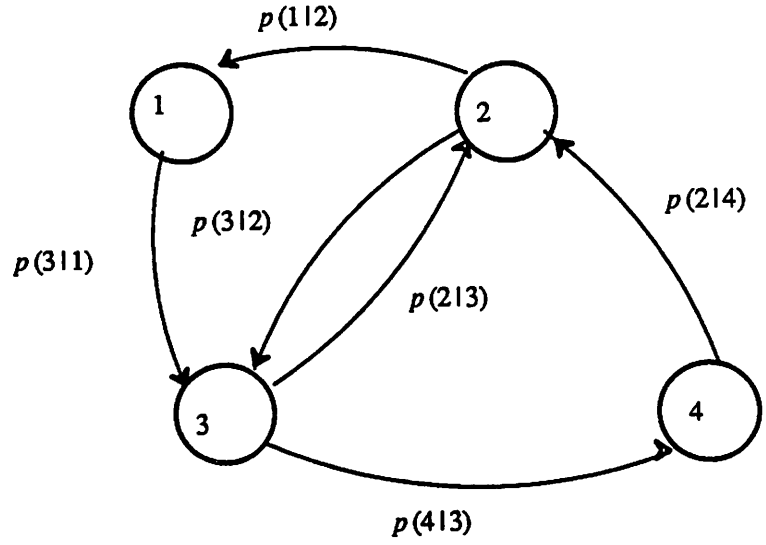


Figure 2: Example 1

Case 3:

Let us now consider a general case where ζ_n's are not a Markov sequence, however, there exists a Markov sequence $y_1, y_2 \ldots$ such that

$$\zeta_n = h(y_n)$$

The objective function then becomes

$$W_n = E[f(V_n, \zeta_n) \mid \zeta_{n-1}, \zeta_{n-2}, \ldots]$$

and if we define

$$\pi_n(i) \doteq P(y_n = i \mid \zeta_{n-1}, \zeta_{n-2}, \ldots)$$

Then we have

$$W_n = \sum_i f(v, h(i)) \cdot \pi_n(i)$$

$\pi_n(i)$ can be calculated using a suitable nonlinear estimator (e.g. Bayes' Rule).

In this section, we have derived specific algorithms for the estimation of clock increments in other processors. Each processor, would therefore, update its clock recovery algorithm which each new

message communicated to it by a remote processor.

5.2. Finite Memory Requirements

We now wish to examine the conditions under which a SESYCCS can be realizable in practice. While our conjecture was that a SESYCCS was realizable we have not yet established that asynchronous distributed computation with Bernoulli interactions and non-blocking communications can be executed in a bounded-memory computing system. This section establishes this fact. The notation follows that introduced in Chapters 2 and 4.

In a J processor system, $\{ C_n^i \ , \ n \geq 0 \, , i = 1 \, , \ldots , J \}$ is defined as follows;

$$\{T_m^{ij} \, , m = 1 \, , 2 \ldots . \} \text{are Bernoulli times}$$

That is,

$$Prob[\, T_{m+1}^{ij} - T_m^{ij} = k \,] = p_{ij}(1 - p_{ij})^{k-1} \, , k = 1 \, , 2 \, , 3 \cdot \cdot$$

If

$$n \neq T_m^{ji} \ ; \ C_{n+1}^i \ = C_n^i + a_i$$

Otherwise, if

$$n = T_m^{ji} \ ; \ C_{n+1}^i = \min\{C_n^i \, , C_n^j\} + a_n^i \cdot I \, \{C_n^i < C_n^j\}$$

with $0 \leq p_{ij} \leq 1$. In addition, we also assume that

$$\frac{a_i}{\varepsilon} \in \mathbf{Z}_+ - \{0\}$$

$$\frac{C_0^i}{\varepsilon} \in \mathbf{Z}_+$$

Define

$$y_n^i \doteq C_n^i - \min_j \{C_n^j\}$$

Theorem 1: If irreducible and aperiodic, then

$$y_n \ \doteq \ (y_n^i \, , i = 1 \, , \ldots , J)$$

is a Positive Recurrent Markov chain.

The condition that the increments a_i and C_n^i be multiples of ε ensures that we deal with a countable state space.

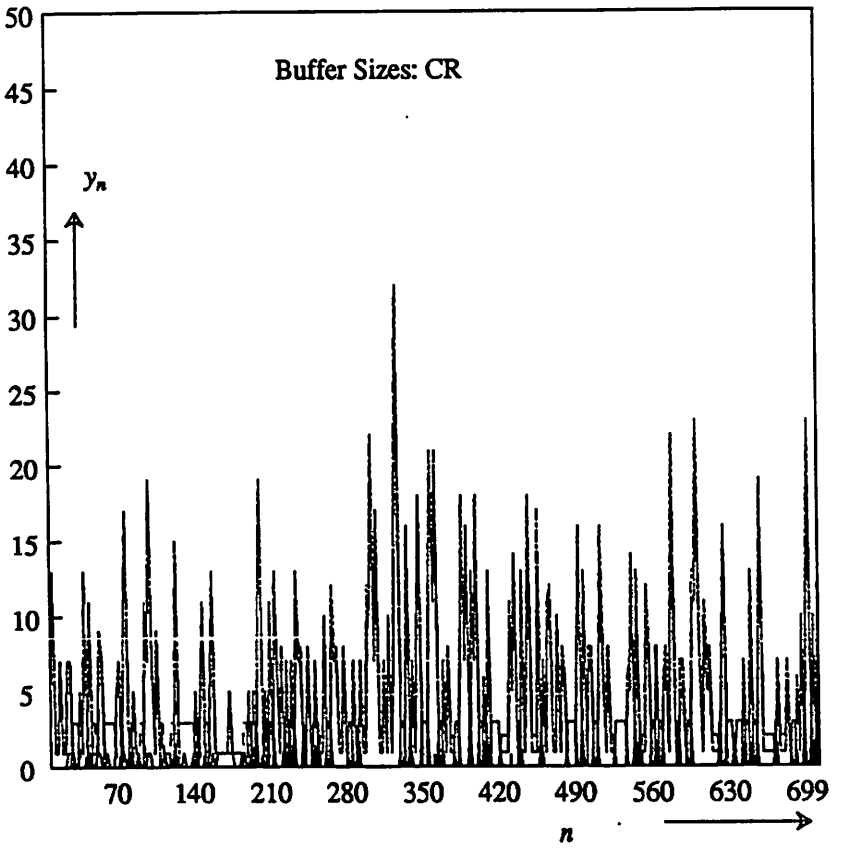


Figure 3: Buffer Requirements with Concurrent Resynchronization

This figure describes the requirements of message buffers in a four processor system synchronized using Concurrent Resynchronizations.

The main idea in this section is to show that the buffer sizes will be bounded under Bernoulli interactions. If it is guaranteed that all processors communicate with each other sooner or later, then it is easy to see that the buffers will always be bounded. This is because the local clocks will be in "synchrony" and will not drift too far apart from each other. We will now establish this notion formally.

The condition of irreducibility ensures that all the processors communicate with each other at some time, allowing all states to be reached. y_n^i represent the drifts of each local time C_n^i from the minimum clock in the system. While C_n^i could drift to infinity with increasing n, we wish to show that

the differences in the times remain bounded. To prove this, we make use of the following Lemma 1, which we will not prove. The reader is referred to [Wa88] for details of the proof.

Lemma 1: Let $V : C \longrightarrow \mathbf{R}_+$, then if there exists a K such that

$$E[V(C_{n+1}) - V(C_n) \mid V(C_n) \geq K] \leq -\varepsilon < 0$$

Then

$$E[V(C_n)] \leq A < \infty \text{ , all } n \geq 1$$

We can now use Lemma 1 to prove that if $V(\cdot)$ is such that $\{V(C_n) \leq B\}$ is a finite set for all B, then

$$Prob\ \{V(C_n) > B\} \leq \frac{E[V(C_n)]}{B},$$

so that

$$Prob\ \{V(C_n) \leq B\} \geq 1 - \frac{E[V(C_n)]}{B},$$

and

$$Prob\ \{V(C_n) \leq B\} > 0 \text{, for } B > A.$$

Therefore,

$$Prob\ \{\text{finite set}\} > 0.$$

The last condition implies that the Markov chain is Positive Recurrent.

Using the notation used in the model for asynchronous systems described earlier in this section, letting $V(y_n) = \max y_n^i$, we have

$$E[\max_i C_n^i - \min_i C_n^i] \leq A$$

or

$$E[\max_i y_n^i] \leq A$$

and

$$\{y_n \mid \max y_n^i \leq B\} \text{is finite}$$

Therefore, y_n is a Positive Recurrent Markov chain and Theorem 1 is proved.

To prove that the condition for Lemma 1 to hold is indeed satisfied, we proceed as follows. Let us assume, with loss in generality, that there is a processor i that is the largest in local times, and that $y_n = y_n^i = K$. Let y_n^j for $j \neq i$ be very small in comparison to $K \in \mathbb{Z}$, taking at most the value $K - L$. Let the probability that any one of the processors j communicates with i at each time step be given by $1-p$. The next-step transition diagram is then given by Figure 4. We normalize the value of y_n to be zero and consequently the rest of the system is at local clocks of at least $-(K-L)$. Here, y_n can increase by an increment $a_n^i \doteq a$ in one time step with probability p, alternatively it can decrease to at most $-L$ with probability $1-p$. Then it is easily derived that

$$E[y_{n+m} - y_n \mid y_n \geq 0] \leq m \cdot a \cdot p^m + (1-p^m) \cdot (-L)$$

so that,

$$E[y_{n+m} - y_n \mid y_n \geq 0] \leq p^m(a \cdot m + L) - L < 0 \quad \text{for } m, L \text{ large enough.}$$

Let us now show that the condition is true for the general case as well. At some point in time we have $y_n = K$. Let us chose an $\varepsilon < 1$, say $\varepsilon = \frac{1}{2}$. Since the system is irreducible (all processors can communicate with each other), we can find N such that after N time steps all the processors have communicated with each other (or resynchronized) with probability ε. If a is the upper bound on the forward computation rates of the processors this implies that the upper bound on y_n is Na with probability ε and $K + Na$ with probability 1–ε. Then

$$E[V(C_{n+1}) - V(C_n) \mid V(C_n) \geq K]$$

is

$$\leq aN(1-\varepsilon) + (K+aN)\varepsilon - K$$

This implies that

$$aN - K(1-\varepsilon) < 0 \text{ for } K \geq \frac{aN}{1-\varepsilon}$$

This proves that the condition is satisfied, and the Lemma 1 is applicable.

Finding the invariant distribution of y_n allows one to derive bounds on the memory requirements.

Figures 3 and 5 plot the size of the buffers with computing time n for the cases of self-synchronization using CR and SR respectively. The memory requirements are assumed to be

proportional to the drift in local times on each processor from the minimum.

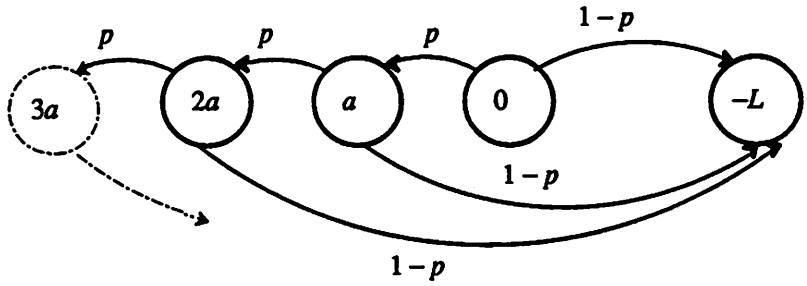


Figure 4: Transition Diagram for Buffer Increment

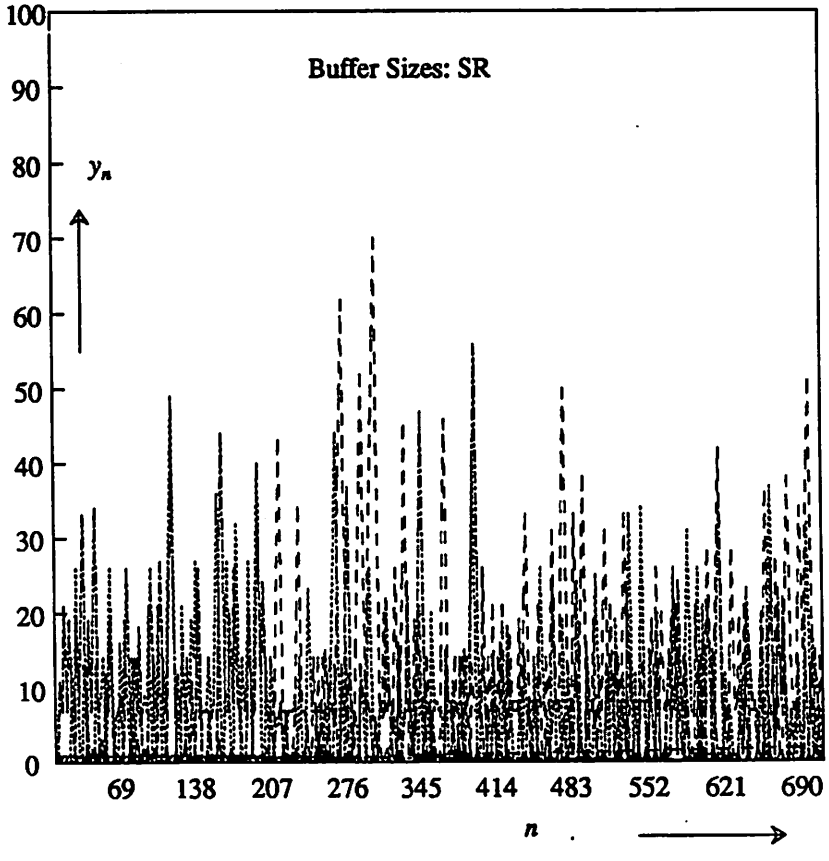


Figure 5: Buffer Requirements with Successive Resynchronizations

This figure describes the buffer requirements in a four-processor system synchronized using Successive Resynchronizations (SR). Note that the buffer sizes needed are larger than the corresponding sizes in Figure 3 where CR is used for synchronization.

In the analysis derived above, the communication costs were assumed to zero (i.e. communications are instantaneous). The state could, therefore, be captured by the local times C_j^i alone. In the case that the communication times are non-zero, the new state vector would be characterized by (C_n^i, M_n^{ij}), where M_n^{ij} represents the messages in transit at time n originating in processor i and destined for other processors j. It is expected that the Markov chain for the differences would be ergodic as well, (if there is an upper bound on the message propagation time).

To summarize the results of this section, we have have presented a model of asynchronous distributed computation, which requires a small synchronization overhead. The cost lies in memory requirements and the computing time needed to undo some erroneous computation. We have proved that under some weak assumptions of irreducibility, the system is stable (ergodic) with finite memory requirements. We have not proposed any garbage collection algorithm for our system, but we do not expect such an algorithm to have any stringent efficiency requirements. The efficiency of the Randomized Algorithms can be evaluated by measuring the Wolf Coefficient. The invariant distribution of the y_n^i would be the guideline for memory design.

5.3. Summary

In the previous sections, we have introduced the framework and methods involved in describing a new class of synchronization methods called the **Randomized Algorithms (RA),** which promise a further improvement in performance. We have discussed a few concrete cases where the improvement can be quantitatively measured. Our approach reformulates the problem as a statistical estimation problem. The preliminary analysis was rewarding in that we have been able to derive specific closed form results.

The theory developed in this chapter fits hand-in-glove with the theory developed for the self-synchronization of dynamic computation graphs. This happy marriage of self-synchronization with adaptive clock estimation promises much in terms of efficient implementation of distributed computation.

We have also proved that computation in a SESYCCS environment is realizable in practice using *nonblocking* communications and *finite* memory whenever the logical system is irreducible. Bernoulli communications in the physical system imply that this is naturally the case. If the physical system were not irreducible, bounded buffers may still be guaranteed by using messages explicitly for transmitting clock information such that the logical system is irreducible. This is a weak restriction on the communications, and augurs well for an efficient implementation. The assumption of infinite buffers, the bane of the non-blocking environment so far, can be relaxed as a result.

It is hoped that in such a framework, the efficiency of distributed asynchronous computation will be then be better understood.

Chapter 6

Efficient Distributed Simulation

Distributed simulation systems, by definition, eliminate the globally shared event list used in the sequential (uniprocessor or shared-memory multiprocessor) simulation systems. The physical system being simulated is modelled by a set of logical processes (LP), each of which is provided with a local clock. Each LP can communicate with other asynchronous LPs through messages. The concurrency in the events in the physical system is captured by the logical system and it is hoped that the simulation progresses much faster in consequence. However, a number of problems typical to distributed systems arise. They include verifying global correctness of simulation, detecting and resolving deadlock, task partitioning among processes, and the overhead incurred in message passing, to name just a few.

The fact that a multiple processor machine can achieve greater computational power at a more attractive price in terms of design cost, comparted to a single processor design, has spurred much recent interest in distributed computing as a solution to compute-bound sequential problems. It is proposed that such machines offer a viable alternative to traditional supercomputers at a higher performance to cost ratio. (See [AtSe88]).

There, however, remain a number of interesting research issues in concurrent computing. First, is the development of distributed algorithms suitable for execution on a network of asynchronous com-

puting nodes. Parallel algorithms for such machines differ from serial ones, in the sense that both the algorithm and the data have to be partitioned into smaller parts for concurrent execution. Secondly, efficient operating systems for concurrent multiple processor systems need be developed. Algorithms for process scheduling and allocation are decidedly primitive and limited in scope. Load imbalances and communication penalties routinely cripple distributed computing applications. The third major concern, is data management, I/O facilities and the user interface.

A number of event driven distributed simulation algorithms have been examined in literature. (See [ChMi79], [JeSo83], [MaWaMe88b]). These methods can be classified into 1) Conservative (Synchronous) Methods, 2) Optimistic (Asynchronous) Methods, and, 3) Randomized Methods (See Chapter 5). In Chapters 4 and 5, we have analyzed the performance of these methods. In this chapter we will focus mainly on the implementation issues in the context of distributed simulation.

6.1. Structure of Simulation

We will summarize qualitatively some related concepts introduced in Chapter 2 that are germane to the distributed simulation of discrete-event dynamical systems (DEDS). The Distributed Simulation System (DSS) consists of the following subsystems.

The *Physical System* being simulated is first described by a set of communicating physical processes (PP). A few assumptions on the behavior of these processes are usually made. A process can send a message at any time $t > 0$ to any other process. The contents of this message depend only on the information available to the process up to time t. (See Chapter 2 for a more formal description of the causality conditions.)

The *Logical System* is then derived from the Physical System by simulating each constituent process by a *logical process (LP)*. The simulation of each PP by an LP is independent of the simulation of the rest of the DSS. In addition, the interactions between PPs in the Physical System are faithfully carried over to the LPs in the Logical System as well (the converse may not hold true!). Therefore, LP_i communicates with LP_j if the corresponding PPs communicate in the Physical System being simulated.

Each logical process has associated with it a logical clock $C_i(t)$ or C_n^i that evolves in continuous or discrete-time respectively.

The *Message System* synchronizes the different clocks in the Logical System. Time stamped messages allow the logical processes to execute asynchronously at differing rates. For instance, if PP_i communicates a message m to PP_j at physical time t, then the equivalent logical system schedules a message (t , m) from LP_i to LP_j. Messages can be of a number of types. Some of them are used for the purposes of simulating the physical system, while others are used to look after the aspects of distributed control and clock synchronization. In addition to the time stamp and message identification fields, it is often advantageous to append statistics collection fields as well, largely to simplify interpretation of the distributed simulation. Each LP_i has a local clock value T_i, which implies that LP_i has simulated PP_i till time T_i. This also implies that if the distributed synchronization is correct, LP_i will not receive any messages timestamped earlier than T_i. Ensuring that such loss of causality will not occur is central to conservative simulation schemes.

We now make precise the underlying structure and mechanisms through which elements (processes) in a dynamic discrete event system interact with each other. The efficiency of a distributed implementation of a simulation is dependent on a number of factors: 1) The concurrency inherent in the system being simulated, 2) The potential parallelism that can be extracted (through the use of lookahead), and, 3) The communications overhead in passing data and control variables within the system. Very often, as in most optimistic simulation algorithms, the structure of the system being simulated is not utilized in planning the simulation.

6.2. Vectored Simulation

In this section, we will describe how a few common networks can be simulated with greater efficiency than that which can be achieved in a conventional distributed realization. The enhancement in the utilization of the processors and reduction in the communication overhead will be illustrated graphically for these networks and expected values for efficiencies will be derived.

Simulation of any large scale dynamical systems involves the simulation of a number of busy cycles to obtain statistically consistent estimates (See [He86],[GIIg87]). For example, a large communication network with about 40 nodes, could have a busy cycle of about 200 years (with about 2^{40} jobs processed!). Most networks are, however, used only for a small period of that time span. Therefore, simulating transient characteristics of the system is important. Vectored simulation interleaves B independent simulation runs of the same physical system on the same network of N computing processors (and the results are subsequently averaged). By permuting the mapping of the system to the distributed computing system for each simulation run (as described below), the efficiency of the implementation is greatly enhanced. As B is usually much smaller than N, speedup is possible. Vectoring also has the positive effect of amortizing the overhead associated with communication setup times over a number of simulation runs. Concisely, we propose that B independent (and identical) simulation runs be distributed over the same N independent processors. Therefore, B independent simulation runs, share the same real-time distributed computing system while maintaining orthogonality in simulated time. We propose that such a "vectored simulation," does in fact enhance efficiency, by reducing the communication overhead, and increasing the amount of useful computation.

These examples show how distributed simulation can be used with advantage in simulating large scale networks while maintaining a high processor utilization. A single simulation run on a network of N processors can be inefficient both due to the high overhead in synchronization and due to the fact that most nodes remain idle for the lack of useful tasks to execute. Conservative simulations are required to enforce causality locally at each node at evey time step, resulting in poor efficiency. On the other hand, optimistic asynchronous simulation algorithms can exhibit a higher processor utilization, but there is a substantial penalty paid while recovering from a loss of causality (most simulation studies in literature present results for well "balanced" optimistic simulations). To complicate matters, the efficiency of the distributed realization in addition very often depends on the structure of the physical network. This additional information is seldom incorporated into the simulation itself to improve its' efficiency. A partially asynchronous simulation algorithm may often outperform a *totally* asynchronous simulation, utilizing the knowledge of the system (being simulated) available to it. This knowledge can be

incorporated in the simulation algorithm without any loss in generality and is transparent to the user. We will classify most of the algorithms in this section as *partially asynchronous* simulations.

In the following, we introduce vectoring in the context of a few typical building blocks for large scale systems. They include, 1) Tandem FCFS Queues, 2) FCFS Queue with Feedback, 3) A Merge Network, 4) A Fork Network, 5) A Central Server Network.

Example 1: Tandem FCFS Queues: Consider the network shown in Figure 1, consisting of N FCFS queues strung in tandem. This physical system can be simulated on N logical processes, each of which is represented by a computing processor (node). Each computing node typically receives an input, simulates a service time by invoking a suitable random number generator, and updates its statistics collection routines, and then reroutes the job to the next node in the network. Let us assume that the time

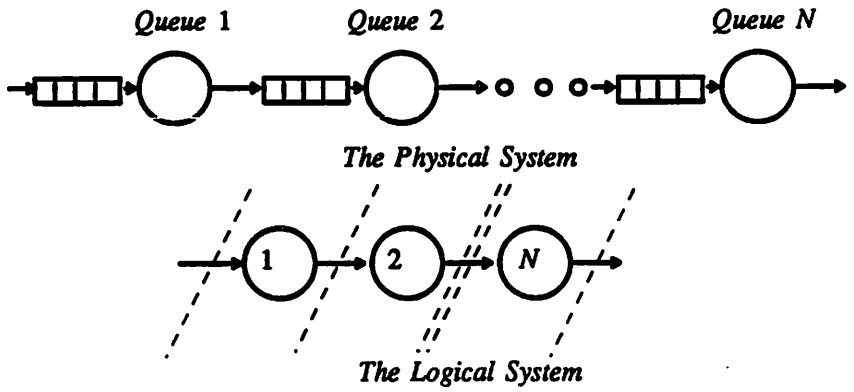


Figure 1: Tandem FCFS Queues

taken by processor i to execute these tasks be given by $t^i_{comp} = a$ seconds and the time for non-overlapped node to node communications be given by $t^i_{comm} = b$. The total computation time will then be $\sum_{i=1}^{N} t^i_{comp}$ and the total time spent in communication and computation will be $\sum_{i=1}^{N} (t^i_{comm} + t^i_{comp})$. The ratio of the time spent in useful computation to the time spent on communicating and computing is defined as the efficiency, η. In case every processor does the same work, the efficiency is given by [MaMe88a],

$$\eta = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{1 + \frac{t^i_{comm}}{t^i_{comp}}} = \frac{1}{1 + \frac{b}{a}}$$

The speedup would then be defined by $N\eta$. In the ideal case, t_{comm} would be zero and speedup would be N.

Let us assume that we process messages from two simulation runs at the same time. Each message between any two nodes consists actually of two jobs from two identical (i.e. simulating the same physical system) but independent simulation runs. The t_{comp} then increases to $2a$. The time for communication between two nodes, however, does not become $2b$ as may be expected. This is because

$$t_{comm} = t_{setup} + \alpha \cdot B$$

Here, t_{setup} is the time required to initialize communication between processors, $\alpha \cdot B$ is the component which increases linearly with the number of bytes (B) transmitted between processors. Typically, t_{setup} dominates communication costs. The resulting communication time for a message of vector length $B = 2$, is therefore only slightly larger than b, and this boosts the efficiency of the implementation even further.

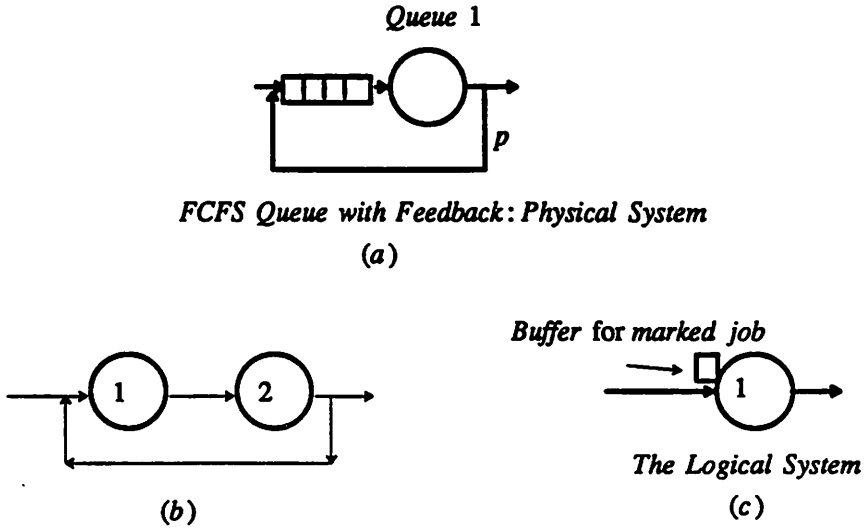


Figure 2: Queue with Feedback

Example 2: FCFS Queue with Feedback: The FCFS queue to be simulated is shown in Figure 2a, and each job has a probability p of being fedback for reprocessing. If simulated on a logical processor network depicted by Figure 2b, processor 1 cannot process jobs until it receives a message from processor

2, informing it whether or not a job will in fact be fedback, along with the simulated time on link C. The efficiency will then be poor, specially if p is small, as this implies that though very few jobs will infact be fedback, processes 1 and 2 have to communicate for each job processed. Instead of this approach, we propose that the jobs which need to be fedback be "marked" probabilistically (Figure 2c) before the simulation begins and system then be simulated just as in the case of the tandem queue. The difference arises when a marked job is received and a special processing routine is invoked (see below).

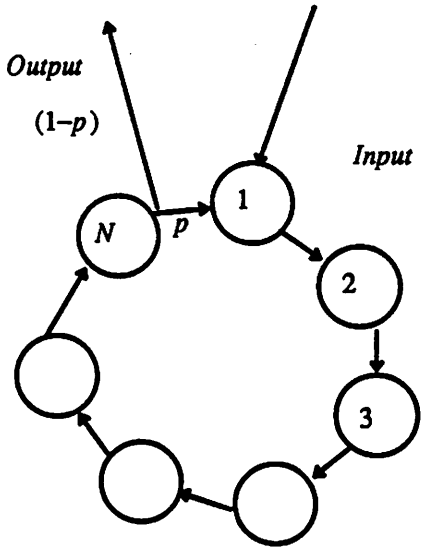
Let the input arrivals occur at simulated times $A_1, A_2, A_3, \cdot, A_p, \ldots, A_n$, and let the p^{th} job be the one which is to be fedback. Let the real times when the processor Q begins processing these jobs be $t_1, t_2, \ldots, t_p \ldots$ respectively. When the p^{th} job arrives at simulated time A_p and its processing begins at real time t_p, the local simulated time at the processor will be given by $s(t_p)$ or equivalently, $s(t_{p-1} + a)$. After the p^{th} job is processed, the local simulated time is given by

$$s(t_p + a) = \max\{s(t_p), A_p\} + x_p$$

Here x_p is the simulated service time for the p^{th} job. The processor Q then processes and reroutes the jobs in the order $p+1, p+2, \ldots, p+k, p$ where

$$A_{p+k} < \max\{s(t_p), A_p\} + x_p$$

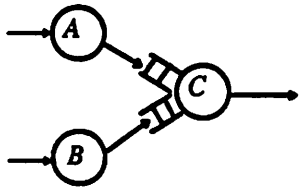
Consider now the general case when N queues are connected in tandem. It is easily observed that whenever a job is fedback in any individual queue the successor computing nodes have to remain idle for a time period $(b + a)$. For the case of a chain of queues $1, \ldots, N$, where jobs could return to queue 1 from queue N, (Figure 3), the same strategy adopted will enforce an idle time of $(N - i)(a + b)$ at node i each time a marked job is received (with probability p) at node 1. As $p \to 0$, the efficiency of our scheme tends to one while conventional conservative distributed simulations would require that only one job be processed at a time by the N logical processes.



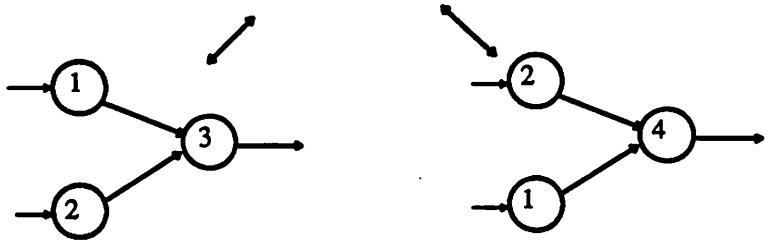
A Feedback Loop

Figure 3: Ring of Queues

Example 3: A Merge Network:



Merge Network: Physical System
a



Logical Mapping 1
b

Logical Mapping 2
c

Figure 4: A Merge Network

Let us now consider a merge network, as shown in Figure 4a. A, B and C are three subsystems interacting via time stamped messages. This system can be mapped on the nodes of the computing

system as shown in Figure 4b. Subsystems A, B, and C are respectively mapped onto processors 1, 2 and 3. We denote this mapping as:

$$(A,B,C) \rightarrow (1,2,3)$$

For the sake of simplicity, let us assign a deterministic task schedule for these processors. Let $A_1, A_2, \ldots,$ and $B_1, B_2, B_3, \ldots$ be the jobs leaving subsystems A and B respectively, system C then processes these jobs in the order $A_1 B_1 B_2 A_2 B_3 B_4$ and so on. Therefore, C processes two jobs from B for each job from A. The *timing diagram* for the simulation is described by Figure 5. The rectangular boxes represent useful computation, while the dotted lines represent communication. The horizontal axis represents real computing time (and not simulated time). It is easily seen that

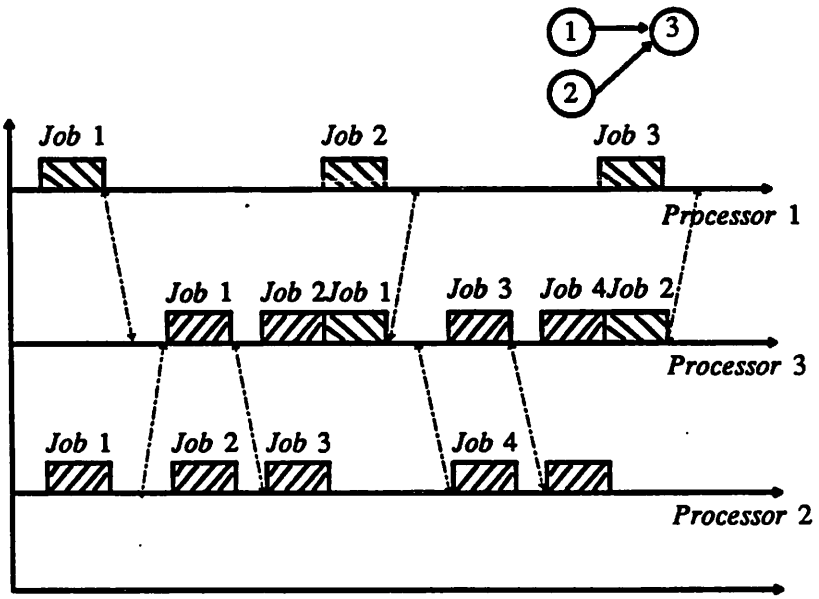


Figure 5: Timing Diagram

processor A is only utilized $\frac{1}{3}$ of the time, while processor B is utilized just $\frac{2}{3}$ of the time. This would imply that the efficiency of conventional distributed implementations would be $\frac{2}{3}$, (note that idle time was counted as a communication cost). We will try to see how efficiency could be boosted to reduce idle time to zero in the ideal case. For this purpose (Figure 4c), we map

$$(A, B, C) \rightarrow (2, 1, 4)$$

Note that the mapping of systems A and B is a permutation of the previous one (intentionally), while a new processor 4 simulates subsystem C. The timing diagram is shown in Figure 6. As may be expected this s very similar to Figure 5, but different in that the idle times of processors 1 and 2 are reversed. By simple superposition of the two timing diagrams, we have the mapping

$$(A, B, C)^2 \rightarrow (1,2,3)+(2,1,4)$$

resulting in an efficiency of 1 (assuming communications costs were zero!).

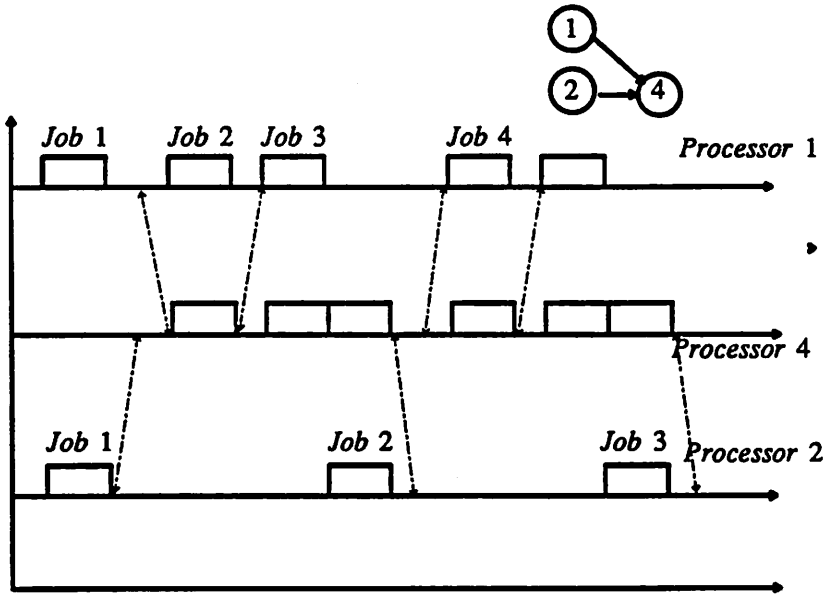


Figure 6: Timing Diagram

Our example was simple, in that it assumed deterministic processing of jobs in a prespecified order. The buffer requirements, therefore, were minimal. Let us consider a more general case, where A and B have exponential service times with rates, μ and 2μ respectively. On the average, system C processes 2 jobs from B for each job from A. Let the mapping be $(A, B, C) \rightarrow (1,2,3)$ as before. Processor 3, however, receives at least two inputs from 2 before receiving one input from 1. This implies that processor 1 may have to operate in real time at half the rate of processor 2, (just as before). If the simulated times are compared, their difference is very small leading to a small storage buffer requirement. However, for the case where both processors 1 and 2 process inputs at the same real time rate, the simulated times drift apart very rapidly leading to unbounded buffers. The vectoring is again

carried out as before, where the second simulation run II permutes the mapping of A and B to processors 2 and 1, with the addition of a new processor 4. Processor 1, then processes one job of simulation run I, before processing two jobs for simulation run II, while processor 2 does exactly the reverse, resulting in efficient utilization of the processors in the long run. Because of the inherent randomness there will be a certain slack and the buffer sizes may be chosen appropriately. This simulation would be called a *partially asynchronous* one, in the sense that the processor 1 and 2 are conservative to the extent of using mean values to determine relative speeds of processing. Buffer sizes can then be designed to smooth the flow accordingly.

Example 4: A Fork Network:

This network is shown in Figure 7. Process A sends a message to only one of the processes B , C , D . When modeled by a (logical system) computing network, only one processor receives the task while the other processors receive information messages which update their local simulated times. Processing these information messages takes very little time (if any) and when these information messages outnumber the number of computing jobs that a logical processor receives, efficiency of the implementation is poor. The advantage of vectoring is the greatest for the fork network. Qualitatively this is because of the fact that permuting the system to processor mapping for independent simulation runs, allows even partitioning of the load over the recipient processors. For a back-of-the-envelope estimate of the improvement in efficiency we proceed as follows.

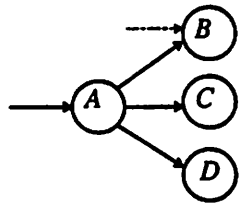


Figure 7: A Fork Network

Let p be the probability that a message is a information message (and consequently $(1-p)$ that it is a "real" job). The effective useful computation per message is then

$$t_{comp} = p\ t_{information} + (1-p)t_{real}$$

The ratio of communication to computation for scalar simulation ($B=1$) would, therefore, be

$$\frac{t_{comm}}{t_{comp}} = \frac{t_{set} + \alpha}{pt_{information} + (1-p)t_{real}}$$

While that for "vectored simulation" would be

$$\frac{t_{comm}}{t_{comp}} = \frac{t_{set} + \alpha \cdot B}{Bpt_{information} + B(1-p)t_{real}}$$

and

$$\lim_{B \to \infty} \frac{t_{comm}}{t_{comp}} = \frac{\alpha}{pt_{information} + (1-p)t_{real}}$$

Efficiency, therefore, is enhanced for a well chosen vector length B. Practical considerations such as finite message buffer sizes limit B. Feasible values for B lie in the range of 10-1000. A few of the vector components can be devoted to either statistics collection, or for other control mechanisms and I/O.

The ratio of information messages to the real messages determine the value of p, and experiments conducted on distributed machines show that this value can be has high as 20, giving p a value of 0.95. The structure of the system being simulated often determines the number of information messages generated. For example, presence of a number of forks and branches in a queueing system can result in a high information message overhead (See [Fu88], [ReMa88]).

The graphs in Figures 8a and 8b describe the efficiencies for first and second generations of tightly coupled multicomputer networks, for varying ratios of information messages to real messages. The knee of the curve would determine a "good" choice for B. In the case of Figure 8a, where the setup time is very high (indicative of the multicomputers presently available commercially), the knee is reached at high vector lengths of 50-100. In Figure 8b, the knee is reached earlier at vector lengths between 5-20.

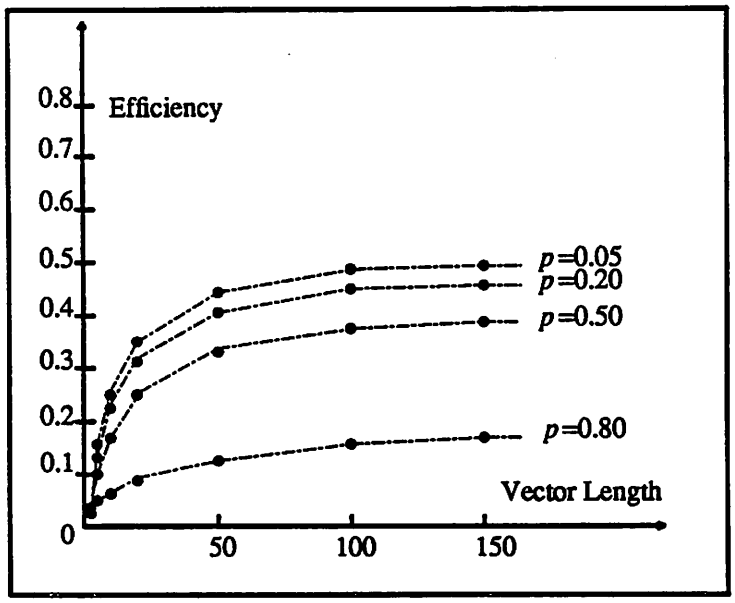


Figure 8a: Efficiency of Vectored Simulation

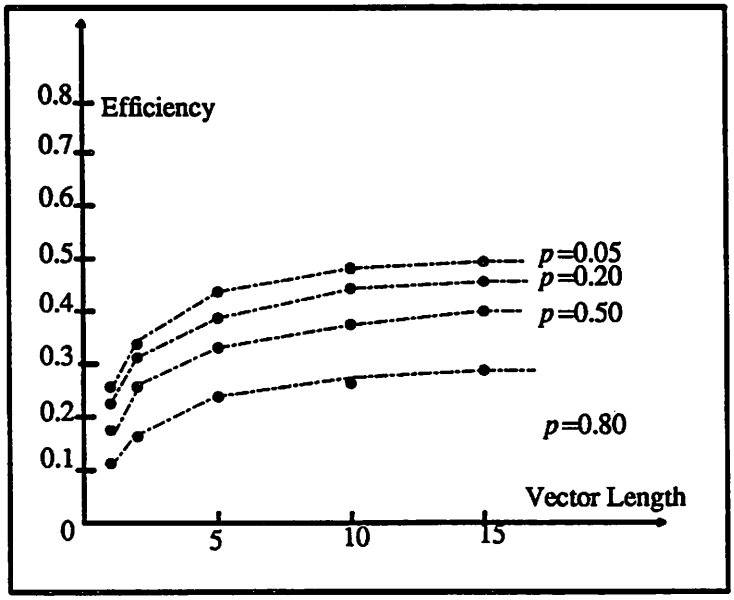


Figure 8b: Efficiency of Vectored Simulation

It is also observed that the initial efficiencies are much higher in the latter case. In generating the plot for second generation machines, we have assumed that the communication setup times are reduced by two orders of magnitude while the computation capabilities of the nodes are enhanced by an order in

magnitude (See [AtSe88]). With further expected improvements in the message communication protocols in future generations of multicomputers, it would be reasonable to limit B to a few tens of independent simulations.

The effect of "vectoring" on the performance of a distributed simulation was simulated on a model of a simple communication network on the NCUBE system.

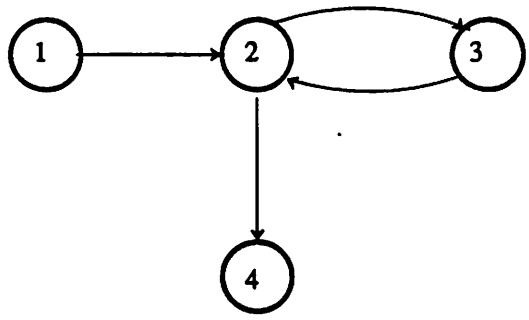


Figure 8c: 4-Processor Simulation

Figure 8c, shows the network consisting of 4 nodes (1-4). Node 1 is the source of messages, and node 4 is the sink. The messages follow the deterministic path 1-2-3-2-4. The message communication times, and processing times per message (in mseconds) were given values such that each node had the same ratio of communications to computation.

Two different initial values of the efficiencies were chosen, one with an efficiency $\eta_1 = 48/64$, and the other relatively lower with an efficiency of $\eta_2 = 37/64$. The communication setup times were given the values existing in commercially available multicomputers, and the effect of vector length on the performance was observed. The communication time (μs) versus message-length (bytes) characteristic is shown in Figure 9d.

From Figure 8e, it can be observed that η_1 was boosted upto a value of 60, and η_2 to a value 58. The enhancement in efficiency being substantially larger for η_2. The efficiency drops again briefly

between vector lengths 16-24 (this is because of the *step+ramp* communication characteristic adopted for the experiment), and then stabilizes to its asymptotic value (which is higher than the initial value for vector length =1). Figure 8e, extrapolates the performance from 4 to 64 nodes.

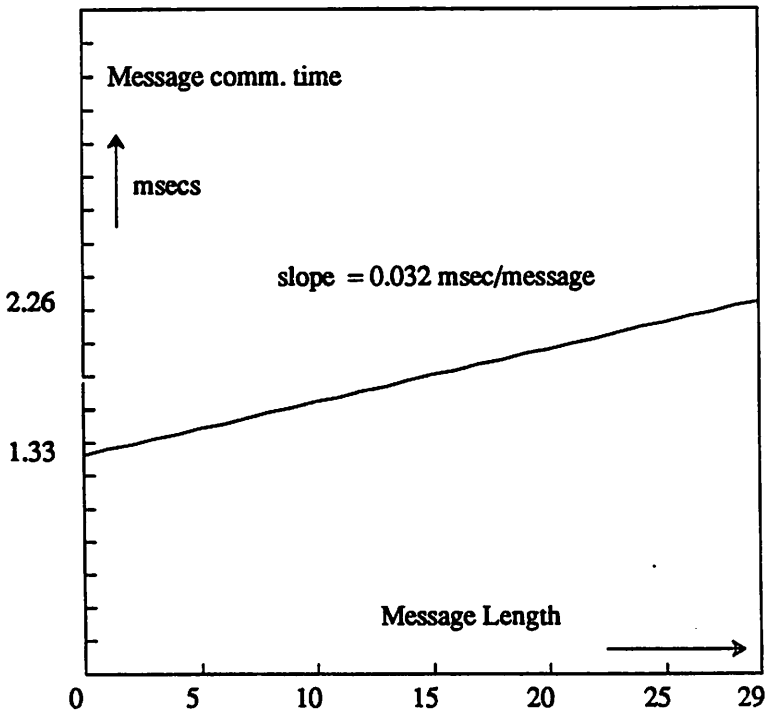


Figure 8d: Communication Time Characteristic (NCUBE)

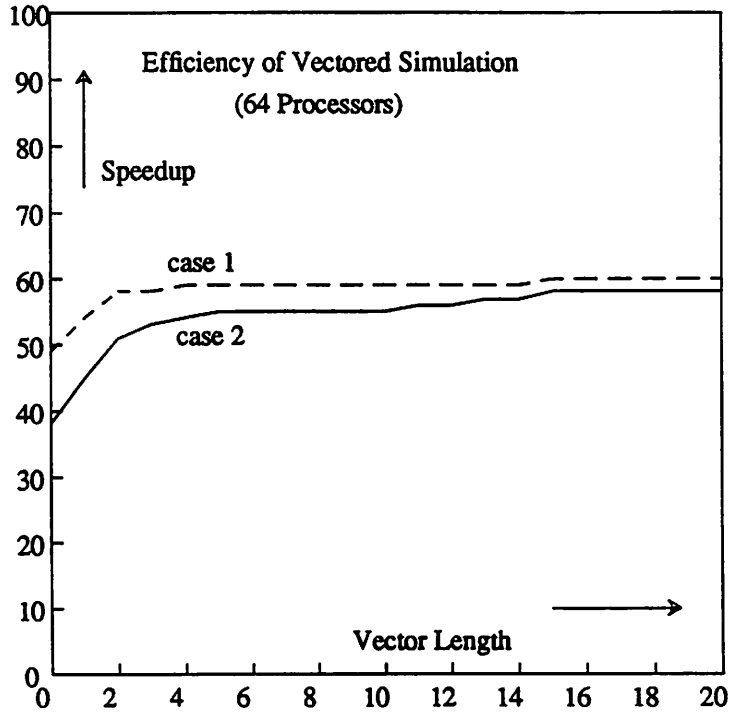


Figure 8e: Efficiency of Vectored Simulation

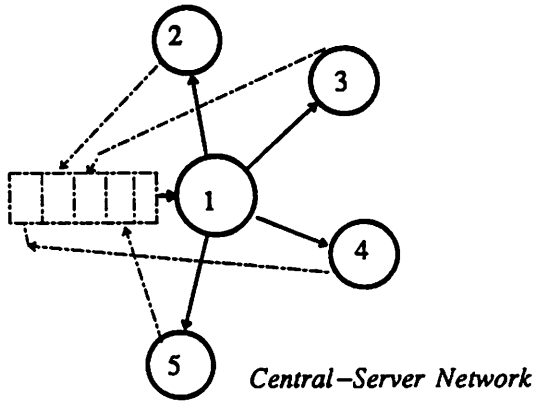


Figure 9: A Central Server Network

Example 5: A Central-Server Network:

The central server network as shown by Figure 9, models an important class of computer systems. Conventional distributed simulations have exhibited a very poor efficiency. The timing diagram shown in Figure 10, shows why this is the case.

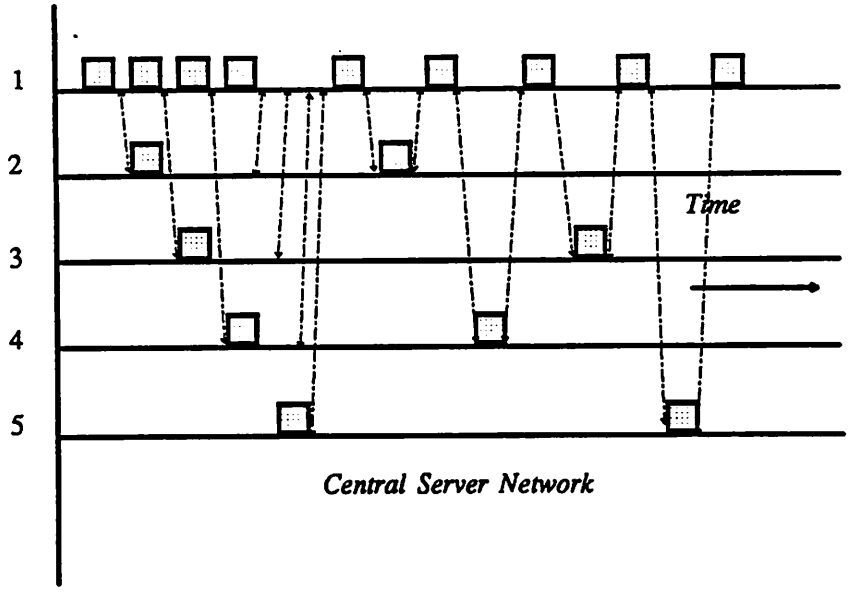


Figure 10: The Timing Diagram of Central Server Network

The central server network operates as follows (let us consider four tasks circulating around the system), the central server processes a task and assigns the task to one of the satellite servers, which upon processing it returns it back to the central server. The simulation is slowed down because the central server and the other satellite servers are unable to process any further tasks until first task returns to the central server. This implies that the portion of the network simulating the central server system, will be unable to extract the natural concurrency available from the central server type of structure. The processor simulating central server has to await the arrival of time stamped tasks at all of its input links before assigning one of them to a satellite processor. Most of the satellite processors are forced to remain idle. Fortunately, vectoring simulations again provides an easy way out. Here again, we consider first the case of two independent simulation runs I and II. The processor simulating the central-server first processes a task from simulation run I, and then assigns it to one of the satellite servers. It then processes a task from simulation run II and assigns it to one of the three satellite servers which are idle. This will keep the processor simulating the central server busy all the time, and two satellite processors occupied. By adding another processor to simulate the central server the efficiency can be boosted further by having four independent simulation runs. Observing the timing diagram (Figure 11),

we can see that a vector length of eight can achieve ideal efficiency!

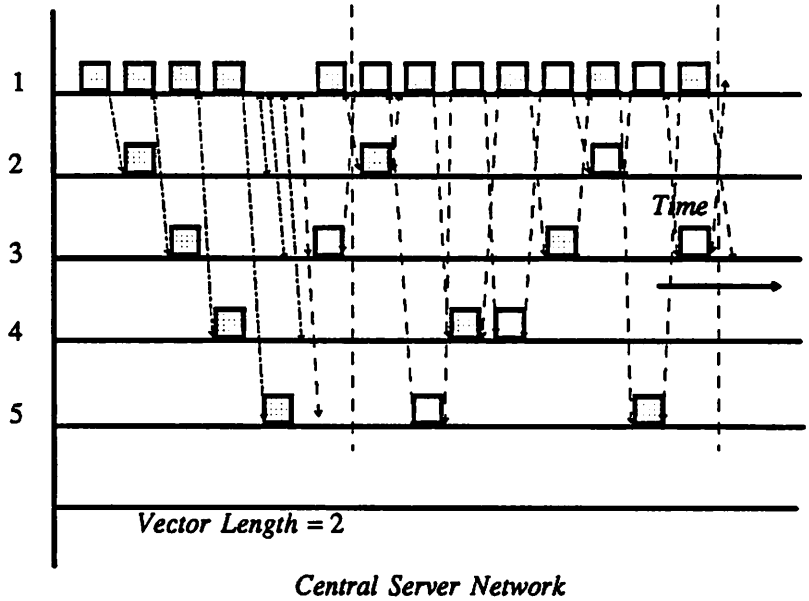


Figure 11: Timing Diagram for ideal efficiency

As the speedup would then be unity (ignoring communication overhead) , it may be far more efficient to simulate the entire central server network on a single processor. This is consistent with the experimental results of previous studies. To summarize, with a deterministic feedback cycle, the concurrency of the real time system cannot be captured by conservative schemes, and simulation on a single processor would be attractive. However, if the network were open, with tasks capable of leaving the system, the concurrency in the physical system can be captured simply by marking those tasks which would be fed back, and taking precautions only for these tasks. The distributed computing system can then be used with advantage to simulate this system.

Conservative simulation schemes are penalized by the high overhead incurred in the communication of information messages, in addition to the enforced idle times of the computing nodes awaiting tasks. Vectored simulations provide both an increased amount of useful work while reducing the communication costs per simulation run. In optimistic simulation (as in the merge network), knowledge of the rates of the input links can be used with advantage in synchronizing their simulation times (or by

duality, scaling buffer sizes according to flow rates for each link).

The Central Server Network can be simulated with greater efficiency and speed using an optimistic algorithm. The conservative simulation algorithm was penalized by the fact that the processor simulating the central server was stopped from processing inputs until the job from the satellite servers returned back to the buffer. In an optimistic simulation, the central processor would process the lowest time stamped job, conditioned on the event that the return of a satellite job would not roll back the forward computation. In the worst case, it could perform only as poorly as the conservative scheme.

We will now discuss some efficient optimistic algorithms with an objective to model their performance and propose techniques to improve upon their efficiency. A new rollback algorithm[MaWaMe88b], Wolf, will be presented, which promises quick recovery from errors in the simulation.

Wolf arises directly from the theory of synchronization, as developed in Chapter 4. Concurrent Resynchronizations is enforced, but only over a set of processors within the sphere of influence.

6.3. A Synchronization Algorithm: WOLF

In this section, we will propose a new algorithm that separates the synchronization from the simulation aspects of the distributed computation. The logical system implementing the distributed computation is synchronized in an asynchronous environment estblished in Chapter 4. A modified version of Concurrent Resynchronizations will be introduced in this section. As outlined in Chapter 4, it is the Sphere of Influence of a processor i that need to be resynchronized whenever any discrepancy in the causality conditions is discovered by i. We will give methods for determining this sphere of influence. The focus of this chapter is on the implementation of the ideas discussed in Chapters 4 and 5, therefore, eficiency of implementation is the main objective.

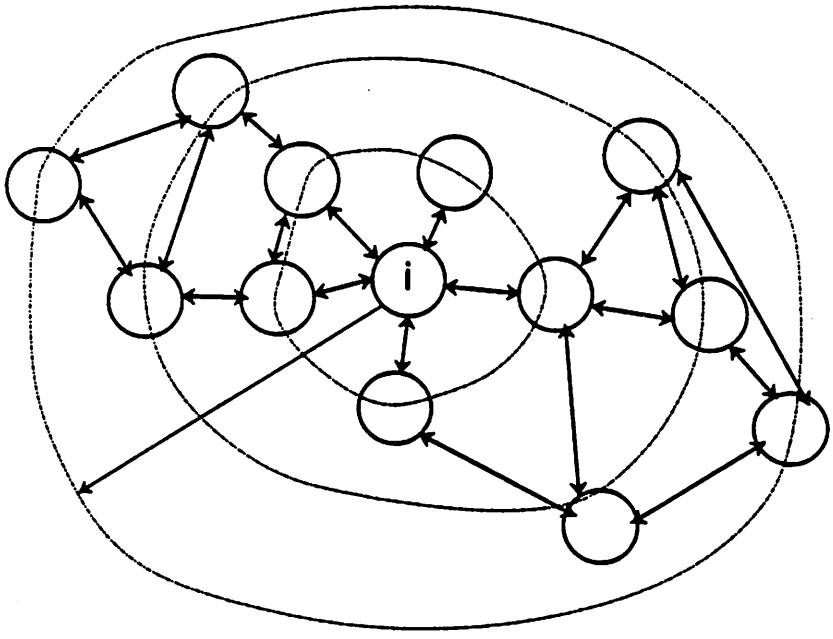


Figure 12: Sphere of Influence

6.3.1 Sphere of Influence, $W(i, t)$

In this section, we wish to quantify the notion of error propagation within an optimistic distributed simulation system. Let b be the communication time between two neighboring nodes, and a be the processing time required to process a single message by any computing node. If a message of class c completes its processing at time t=0 in node i, then we define the set $W(i, t)$ to be the set of nodes that can be influenced by that message in time t. More specifically,

$$j \in W(i, t) \iff p_{kj}^{\alpha\beta} > 0, \text{ for some } \alpha, \beta, \text{ and } k \in W(i, t - a - b)$$

As the communication times b and computation times a are in general random, replacing them by the minimum communication times and processing times leads to a conservative estimate of the sphere of influence. A finite algorithm allows one to compute $W(i, t)$ for any finite t. This can be done off-line. This information can be stored in node i, possibly after some compression by approximation in the form of a lookup table. The sphere of influence could also be adaptively updated to moni-

tor the changes in the network. The radius of propagation, $R\ (i\ ,t\ )$, of the sphere of influence , $W\ (i\ ,t\ )$, is the distance in the number of nodes, which a message transmitted by node i could propagate in the time t. For a conservative estimate, the radius of propagation would be $\frac{t}{a+b}$. The sphere of influence (in the conservative sense), can be looked upon as consisting of a number of shells of increasing radii $(1-R)$, each shell consisting of nodes reachable from i within a certain time span. The sphere of influence thus enables the simulation designer to take advantage of the structure of the system being simulated. For example, even if a certain computing node could communicate (directly and indirectly) with another, the requirements of the simulation could preclude such communications, and this knowledge can be used to prune the sphere of influence. (Figure 12)

The sphere of influence has another dimension when simulation is used as a design tool, specially in the Computer Aided Design of large scale circuits. The circuit is designed iteratively to satisfy certain output specifications. Each time a circuit component is modified, a subset of the entire circuit will only be affected by this change. The simulation algorithm then conserves its resources by simulating only those circuit components which lie within the sphere of influence of the modified component. This would increase the speed of the simulation, as the distributed computing system is able to focus on a smaller circuit than the original one. The design is then iteratively completed. At present, some attempts are being made to study the feasibility of this approach in developing a distributed version of a hardware simulation CAD tool.

6.3.2 Wolf for Resynchronization

Wolf, is an algorithm that is invoked by the synchronization mechanism whenever it is discovered that the causality conditions are not satisfied. Wolf, is a broadcast algorithm that resynchronizes the entire sphere of influence of the computing process that discovers it had made an error. This section discusses the structure of the synchronization mechansim.

Notation:

OQ_k is the Output Queue of Node k.

IQ_k is the Input Queue of Node k

S, is a message which is the Straggler.

LVT is the Local Virtual Time at a node.

E is the Error Message detected at time t_S after it was processed.

$A+$ denotes the messages which were processed after A was processed.

$i \Rightarrow j \in W \mid A$, implies that i broadcasts message A to nodes $j \in W(i, t_S)$

S_R consists of those nodes , $k \in W(i, t_S)$, which are at a radial distance of R away.

The following is the algorithm for the node, i, which initiates Wolf to rollback the effects of error message , E, processed at real time of t_S seconds earlier.

Wolf Algorithm

Node i:

$i \Rightarrow j \in W(i, t_S) \mid V(E, T_E)$

Rollback LVT to T_i

Await ACK_j from all $j \in S_R$

Initiate Forward Compute Phase

End

/* $V(E, T_i)$, is the Wolf-call, containing the identity of the error message, E, and the timestamp, T_E, at which it was received by i. */

/* Each receiving node, j, processes $V(E, T_E)$. We illustrate the algorithm for node $j \in W(i, t_S)$ when it receives a Wolf-call, $V(A, T)$. */

Node j:

Read $V(A, T)$.

While (LVT_j < T) *do* continue processing *enddo;*

If ($A \cap OQ_j \neq \phi$) *then* $j \Rightarrow k \in W(i,t_S) \mid V(\Omega+,T_{\Omega+})$

/* Here $A \cap OQ_j = \Omega$.*/

/* Here T_A denotes the LVT when j received message A. */

Rollback to T_A

If $j \in S_R$ transmit ACK_j to i

else await $V(A,T)$

endif

else await $V(A,T)$

endif

The algorithm ensures that the effects of the error message, E, are limited to the sphere of radius R, $W(i,t_S)$. In addition, only P broadcasts are necessary to complete the rollback as opposed to at least R communication steps required by timewarp. P is usually much smaller than R. In the next section, we outline how P can be determined.

We need next to estimate the communication overhead for Wolf. If the time required for a single broadcast is ε then for P broadcasts we need time $P\varepsilon$. At each step of the algorithm, a few nodes (on an error path) broadcast Wolf-calls concurrently, incurring overhead of only one broadcast. Therefore P, the number of steps the algorithm takes to terminate, determines the overhead in communications. In practice, the time required for a multihop broadcast is approximately equal to that of a single hop communication step. This performance is achieved using a virtual cut-through routing algorithm.

Let us now consider a few examples, to illustrate rollback using Wolf. In Example 6 (Figure 13), we have a simple queueing network with a sphere of influence, $W(i, t_S)$ consisting of the 17 nodes as shown. Node 1 initiates Wolf when it detects the straggler. Let us assume that the error message, E, is now at node 8. Therefore, nodes 1-8 form the *primary* error path, or the error path of *order 1* as we will call it. When node 1, initiates the rollback, it

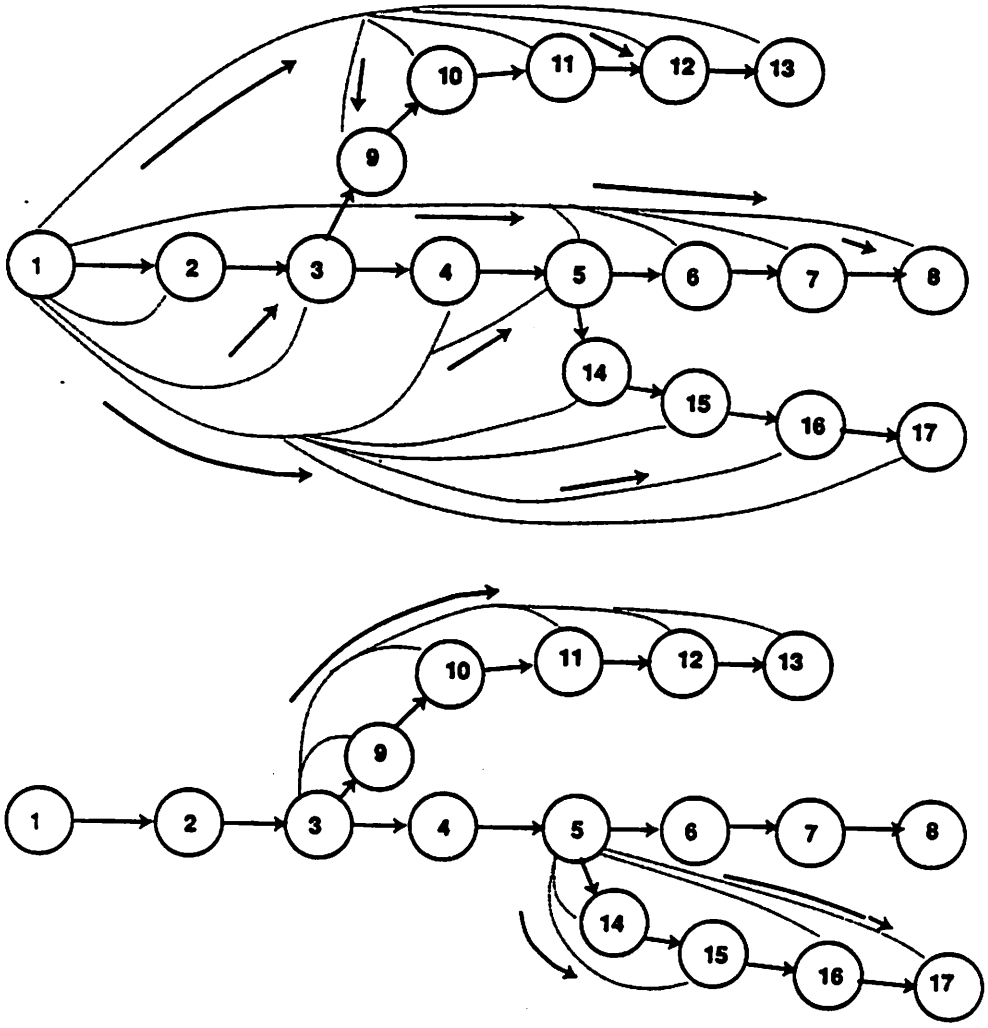


Figure 13: Example 6

broadcasts information about the error message and the its Local Virtual Time when it received the error message, E. Each receiving node stops processing its Input Queue if its LVT is greater than that time, and in addition, if it has actually processed E then it is on the primary error path and rolls back (to the LVT at which it received E). It then broadcasts to the remaining nodes (off the primary path) in $W(i, t_S)$, information regarding the messages it has processed subsequent to processing E. In this case, nodes nodes 3 and 5 are the only nodes on the primary error path (order 1) which have transmitted messages along paths 3-13 and 5-17 (order 2). These broadcasts take place simultaneously, with all the nodes rolling back to the LVTs which are consistent with the available information. In short, the primary path 1-8 rolls back after the first broadcast, and then the paths of order 2, 3-13 and 5-17, roll back after the second broadcast.

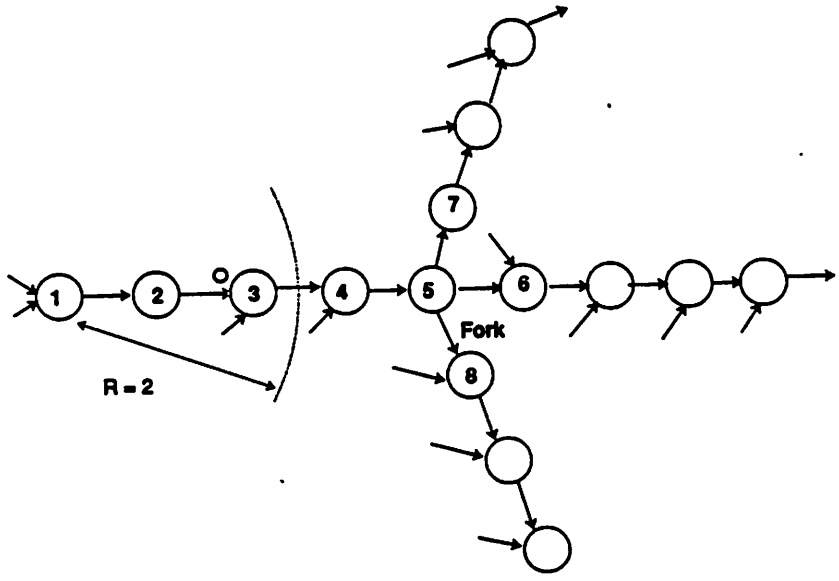
If standard timewarp (without broadcast and "blocking") were used, a minimum of 7 communication hops would be required, as the antimessage is propagated along the nodes 1-8 in succession and likewise along the paths 3-13 and 5-17. If the the nodes were lightly loaded then the number of communication steps required by the antimessage to meet with and *annihilate* the error message would be much larger. Forward computation is then initiated by Node 1, and the nodes restart forward computation. Forward computation after rollback can use with with advantage the facts that lazy cancellation can be used (to save on retransmission of messages) and that random numbers previously generated at the nodes can be reused to simulate service times.

We have introduced some new notation in the previous example; that of the *order* of the error path. Paths of order 1, contain those nodes which had processed the primary error message E. Paths of higher order are those which originate from the paths of lower order, when nodes on lower order paths process messages subsequent to processing an error message (and hence those messages are also errors and need be corrected). This notion allows us to determine the number of broadcasts which would be required by the Wolf algorithm.

The number of broadcasts required by Wolf equals the highest order of any error path in the sphere of influence.

In the case of Example 6, the highest order of error paths was 2, hence 2 broadcasts were required. It can be easily shown that the maximum number of broadcasts would be limited by the radius of propagation of the sphere of influence. In Example 6, this radius was 7.

Our next example (Figure 14) depicts the configuration of a manufacturing system. Let us consider a simulation on a very large grid (in two or three dimensions) of processors. To illustrate the difference in costs , we assign values for the minimum communication time between nodes, $b = 1$, seconds and the minimum processing time , $a = 1$ seconds. If the error was detected by node 1 at time t=4 seconds, and the network were lightly loaded, it is *unlikely* that the antimessage transmitted by node 1, would meet with the error message to annihilate it. If the error message reached the fork, secondary error messages would contaminate the entire network. Wolf, however, with a combination of broadcast and "blocking" guarantees that effects of the error message will be confined to the sphere of influence (with radius 2). Besides, all the nodes in the error path roll back in simulated time concurrently. Since, only a few paths span the sphere of influence the rollback phase with Wolf is very short, implying a short recovery period from the loss of causality.



Based on this discussion, we formalize the notion of primary and secondary error paths. We consider two paths differing in order by one, the one with the lower order being the primary path and the one with the higher order, the secondary error path.

6.3.3 Embedded-source Model for Rollback

Let node i process a message E at real time t=0, and at time t=t_S a *straggler* is detected, informing node i that message E was an error message. Message E, in the mean while, has been processed by nodes j , k , $l \cdot r$ when the straggler arrived at i. This path $i - j - k - l \cdot r$ is defined as an error path of **order 1.** Each node along the error path of order 1, processes other messages subsequent to processing the error message, these messages are also error messages and directly influence the rest of the network. In a dynamical discrete event system, with random routing, the path of order 1, gives rise to a number of secondary error paths of order 2. For the purpose of analysis, each of the nodes, n, on the path of order 1 is embedded with a source of rate μ_n (messages/sec), which generates error messages to the rest of the network. The messages from i are processed as they arrive. Therefore, between every two error messages processed along the primary error path, a number of secondary error messages resulting from interaction with the rest of the network are generated. (Figure 15) The reader may recall, that we had analytically modeled source by the probability p_W in our analysis in Chapter 4.

In the path of order 1, nodes j , k , l , ... , r are at radii of 1 , 2 , 3 ...., R respectively. The number of error messages generated by these R nodes in the time t_S would be

$$(t_S - b - a)\,\mu_j + (t_S - 2b - 2a)\,\mu_k + (t_S - 3a - 3b)\,\mu_l + \ldots + (t_S - Ra - Rb)\,\mu_r$$

These messages represent those which are generated prior to the time the straggler is detected. When Wolf is invoked, additional error messages are generated in the time it takes for the broadcast to reach the nodes, and their number is

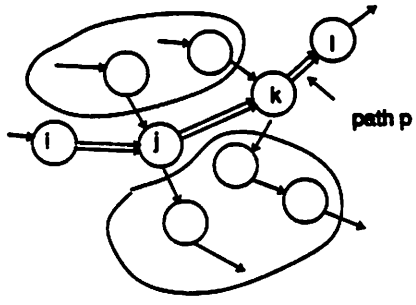
$$\varepsilon\,(\mu_j + \mu_k + \mu_l + \ldots + \mu_r)$$

However, if there were no Wolf-call and antimessages are propagated from node to node, then number

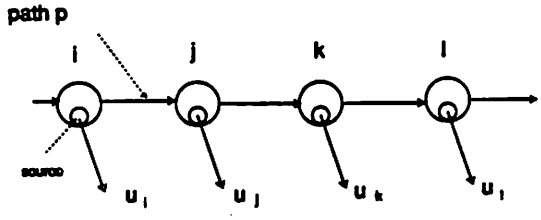
of additional error messages that remain to be cancelled are given by

$$(b+a)(\mu_j + 2\mu_k + 3\mu_l + \ldots + R\,\mu_r)$$

The number of additional error messages to be cancelled depends on the rates of the embedded sources, and is particularly sensitive to the sources on nodes at increasing radii of propagation (owing to a linear multiplier). Therefore, a long error path is capable of generating a large number of secondary error messages relative to a short error path (small t_S). Wolf, however, is insensitive to the path length. In this simple example, we have considered two paths; the analysis can be extended to paths of higher order. If all nodes in the sphere of influence were not informed about the straggler then the numbers given above represent only the lower bound on the additional messages that need be cancelled.



(a) General Network



(b) Embedded-source model

Figure 15: Embedded-Source Model

In this chapter, we remain content to describe the structure of the distributed simulation and comment on its efficiency. The rigorous analysis is, however, covered in Chapters 4 and 5.

6.3.4 Pipelined Forward Computation and Rollback

Spheres of influence allow an elegant representation of forward simulation and rollback in an optimistic distributed simulation. The sphere of influence of an error message grows with time, until the error is detected and Wolf is invoked. The the broadcast Wolf-call "freezes" the sphere of influence while the rollback phase begins.

In Wolf, nodes in more than one shell can rollback concurrently. The signal to rollback does not depend on the radius of the shell a particular node is in, but on the order of the error path of which it is part. This is unlike standard timewarp, where the shells rollback in succession, starting with the shell at radius 1 and ending with the rollback of shell at radius R. In Wolf, we have P broadcasts and hence P phases in the rollback. However, both the schemes allow pipelining of forward computation with rollback. In Wolf, a phase could include rollback of more than one complete shell. We must note, however, that restarted forward computation though faster is still sequential. Feedback paths may slow down forward computation. The overhead in communications for the node initiating Wolf, can be greatly reduced by using an iterative determination of the sphere of influence, where a number of smaller spheres of influence are influenced by the error (See later section).

After the nodes in a shell have rolled back, they restart forward computation, while the nodes in the shells of greater radii begin to roll back in simulated time. This pipelining then progresses until all the shells in the sphere of influence have rolled back to a state consistent with the available information. By then, the entire sphere will have restarted forward computation. This sequence is shown in Figure 16(a)-(f). In (a) the sphere of influence is growing to that in (b). At the instant (b), the node i detects the straggler, and the rollback starts as shown by (c). In (d), shells 2 and 3 rollback, while shell (1) computes forward. In (e), shell 4 rolls back, while shells 1, 2 and 3 compute forward. The Wolf phase is completed in (f).

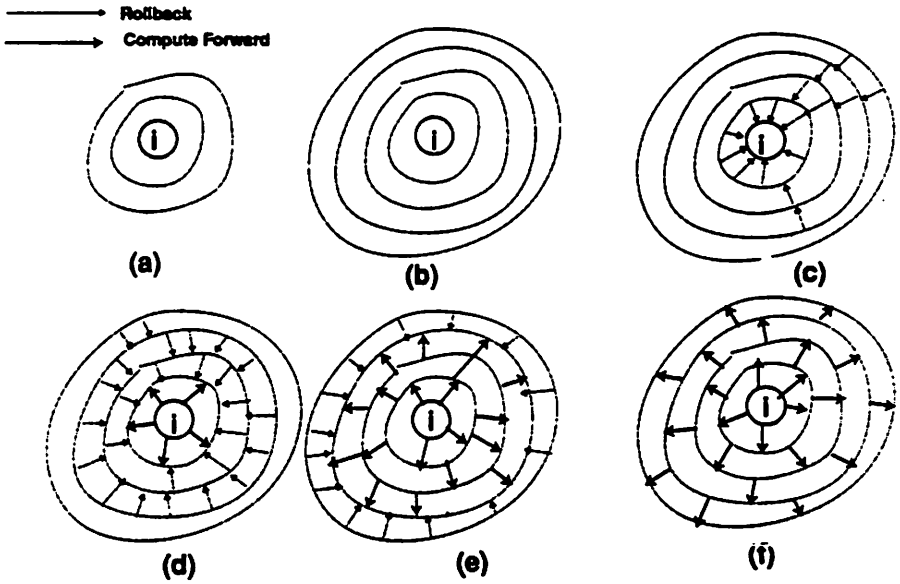


Figure 16: Pipelined Forward Computation and Rollback

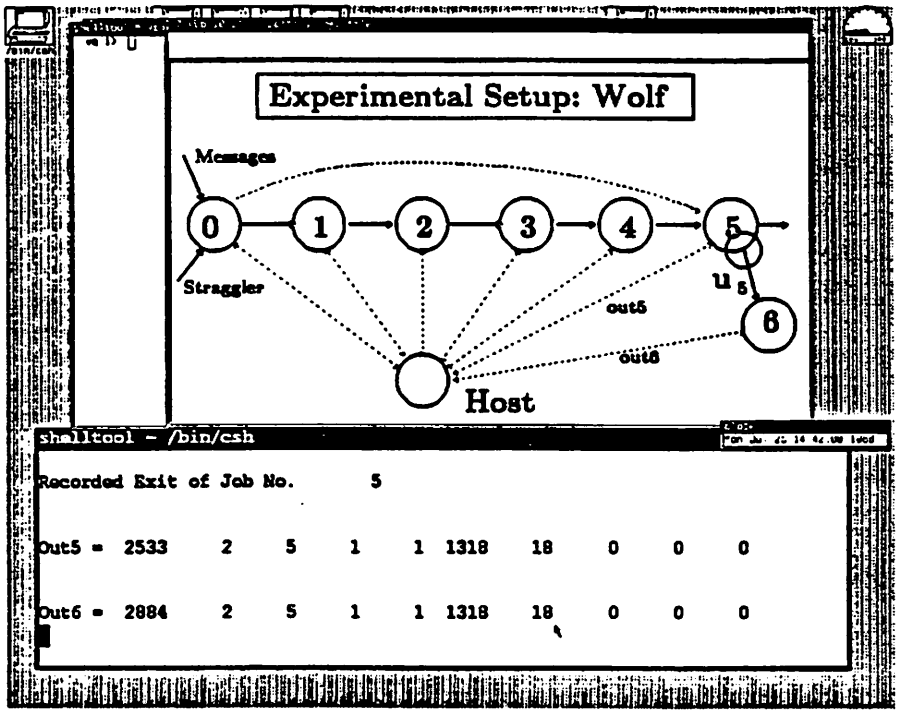


Figure 17: Experimental Setup on NCUBE

6.4. Design of Simulators

In this section, we delve briefly on some strategies to plan and implement large scale simulations of dynamical discrete event systems on a network of computing processors. Such systems contain a large number of computing nodes (40-50), flexible routing facilities, service disciplines, and monitoring and statistics collection routines.

The initial phase of the simulation involves formulating a model for the real life system being simulated. The onus of designing a precise model of the dynamical discrete event system rests largely on the application expert. The next step is to distribute the model onto a set of logical processes. Then the processes are mapped onto a distributed computing system with a number of computing elements. Since the concurrent computing environment consists of a number of concurrent and, in general, asynchronous processes sharing a few common processors, an efficient task assignment and load balancing algorithm must be used to schedule the distributed simulation. We have developed a some algorithms to adaptively assign computing resources in a distributed multitasking, multiprogramming environment (See MaMe88b.)

Both conservative and optimistic distributed simulation methods can improve upon the efficiency of their implementation by minimizing the communication overhead. The overhead in conservative methodologies arises from the information messages that are communicated between processes to maintain causality. In optimistic schemes, communications are necessary to rollback to a state consistent with the available information. We have proposed an algorithm through which a number of messages from B independent simulations are processed on the same distributed simulation testbed. Instead of a single message, a "vector" consisting of B messages from B independent simulations are transmitted between processes. All the messages in a "vector" share the same communication overhead and also help keep processors busy. For example, the information messages of one simulation are sent with the real jobs of another orthogonal simulation run. In the case of optimistic Wolf, the processors can compute forward in one simulation run, while they are blocked by rollback in another independent simulation run. The two main advantages are the reduction in the communication overhead, and

the increase in the utilization of the computing elements. We define this approach as "vectored simulation" and call B as the Buckshot vector.

Once a suitable "vector" length is chosen, the next step is the identification of Wolf nodes and determining their spheres of influence. Both of these depend on the structure of the distributed model being simulated.

Wolf nodes can be usually identified as those with more than one input. The sphere of influence can be determined by a number of methods. In the algorithms discussed in this chapter, we have assumed that the sphere of influence is known, and the efficiency of the algorithm depends on how well this sphere can be estimated in practice (hence the name, Wolf). A conservative method, in which the minimum communication and processing times are used is described in the chapter. This may overestimate the radius of the sphere, especially when the network is heavily loaded. Other approaches we are considering, include a dynamic algorithm (a Bellman-Ford type algorithm) which iteratively evaluates smaller spheres of influence and extrapolates to a large one. For example, to evaluate $W(i, T)$, at first $W(i, T_1)$ is determined for $T_1<T$, and then the nodes, k in $W(i, T_1)$ determine $W(k, T-T_1)$. The union of these spheres determines the required sphere of influence. This approach can be extended to evaluate the sphere of influence as accurately as possible. This approach has the additional benefit that the rollbacks will then be confined to a few spheres of influence of smaller radii and there will be a reduction in the communication cost. However, occasional errors can result due to changes in the states of the queues, in dynamic systems. Another method, would be to release marked messages occasionally and determine their trajectories with time. Very often, especially in static systems, the entire trajectory of a message through a system can be probabilistically determined before the message enters the network, simplying evaluation of $W(i, t)$.

6.5. Experimental Results

In this section, we present some results from some pilot implementations of rollback algorithms on the NCUBE distributed memory multiple processing system. The NCUBE multiple processor system, connects upto 1024 computing elements each with its own local memory (The latest commercial version interconnects 8192 processors with a peak performance of 27 GFLOPS!). The processors communicate via asynchronous message passing. Each of the nodes is capable of about 0.5 MFLOPS peak performance and a strong hypercube type interconnectivity allows concurrency in communications. Latest models of commercial multiprocessors provide very efficient multi-hop communication protocols (See [Da87]), with the time taken for a multi-hop message communication step only slightly larger than that of a single hop. In addition hardware broadcast facilities are provided, enabling a single node to broadcast to a number of other neighboring nodes through dedicated links.

Wolf and Timewarp ([JeSo83]) algorithms were implemented on a network of seven computing nodes. The network topology is illustrated in Figure 17. The objective of the experiment was to experimentally verify the embedded-source model and the sensitivity of the messages to be cancelled to the source rates and the path length. Extensive simulation of Wolf, on a large network using about 40 nodes is currently being implemented to study the performance tradeoffs in practice. In this example, the nodes were strung in tandem, and node 5, is embedded with a source of rate μ_5 modeling interaction with the rest of the network. The performance of Wolf and Timewarp was then measured with varying processing times. One such set of observations is illustrated in Figure 18. The numbers represent the number of error messages that need to be cancelled by either scheme. The straggler was separated from the error message by 6 time units. When two embedded sources were used (the other on node 4), the numbers of error messages were an order of magnitude higher. Our results are encouraging, indicating an order of magnitude reduction in the number of messages to be cancelled by Wolf relative to timewarp, in addition to a significant enhancement in the speed of rollback, when two paths are considered. (the sum $(a+b)$ along the path is normalized to 1 and the the source rate is varied between 1 and 2.

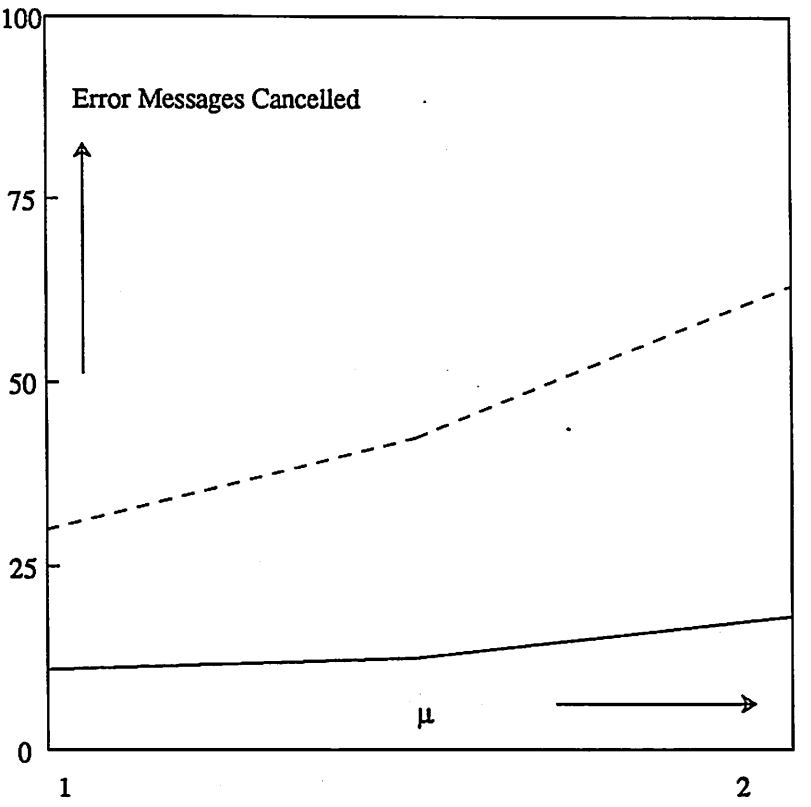


Figure 18

The number of error messages generated is proportional to the source rate as observed experimentally.

6.6. Summary and Future Work

This chapter examines an application of self-synchronizing concurrent computing systems in the distributed simulation of discrete event dynamical systems.

We have proposed some algorithms for efficient distributed simulation on a network of asynchronously executing processors. We have described methods for efficient rollback and recovery from error, vectoring simulations to balance load, and have illustrated the advantages to these techniques by means of examples.

Wolf, an algorithm for efficient self-synchronization, that had been quantitatively analyzed in Chapter 5, is proposed for efficient synchronization. Separation of synchronization from computation is shown to be efficient in this distributed framework.

Application and further development of such distributed algorithms is our current goal in the performance evaluation of large scale discrete event systems.

Chapter 7

Conclusions and Future Work

In this thesis, Self-Synchronizing Concurrent Computing Systems (SESYCCS) have been proposed for high speed scientific computation and their performance analyzed. SESYCCS provide synchronization for distributed computation applications within an efficient and powerful concurrent programming environment.

SESYCCS provide automatic synchronization of distributed programs that can be described by Static and Dynamic Computation Graphs. Chapter 2 provides a formal description of the types of computation-bound problems that fit into this framework. New models for clock behavior in an asynchronous distributed environment are also proposed.

Self-Synchronization for Static Computation Graphs is the theme in Chapter 3. A number of theoretical properties of SCGs are derived and discussed. The computer designer is made aware of the need and the algorithms for efficient program design that guarantee finite buffer sizes in the concurrent computing system. With systems currently providing upto a 100 GFLOPS in total computing power, it is of paramount importance to ensure that this resource is not squandered in an inefficient implementation. A novel perturbation algorithm for self-synchronization is proposed that optimizes resource allocation algorithmsin a multi-user environment. We show that it is possible to assign resources

efficiently and in a simple enough manner among a number of users of the concurrent computing system.

Chapter 4 tackles the problem of self-synchronization in Dynamic Computation Graphs. A robust model is developed for potraying the dynamics of clocks in the SESYCCS environment. Analysis is presented which provides new insight on the progress of distributed asynchronous computation. Efficiency in the dynamic environment is quantified systematically and a number of new algorithms for self-sycnhronization are proposed. It is proved that synchronization separated from the computation gives the best results in terms of performance and in terms of limiting the memory requirements in the processors. Closed form results describe the different design constraints and strategies in the design of SESYCCS. Detailed simulation of multiple processor systems validate the analytical results. Hardware synchronization facilities are proposed.

Chapter 5 extends the results of Chapter 3 and 4 to analyze the performance of SESYCCS in an "adaptive" environment, where the analysis of the traces of the distributed computation provides valuable insight into the behavior of some compute bound problems. These interesting results provide a further enhancement of the efficiency of implementation.

These algorithms are examined in the context of distributed simulation of discrete event systems. A number of new techniques for efficient distributed simulation are discussed and experimental results are presented in Chapter 6. Distributed simulation is seen as a test bed for self-synchronization and further experiments are being planned to determine the engineering tradeoffs involved in the design of high speed parallel computing systems.

Present research efforts are being directed towards using this rich body of theory and experimentation to develop a powerful and efficient concurrent computing environment for the 1990's.

REFERENCES

[Ag86]

G. A. Agha, (1986), *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Mass., 1986.

[AtSe88]

W. Athas, C. L. Seitz, (1988), "Multicomputers: Message-Passing Concurrent Computers," *IEEE Computer*, August 1988, pp 9-24.

[Ba73]J. L. Baer, (1973), "A Survey of Some Theoretical Aspects of Multiprocessing," *Computing Surveys*, vol. 15, pp 31-80, March 1973.

[BeTs89]

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

[ChMi79]

K. M. Chandy, J. Misra, (1979), "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs", *IEEE Transactions on Software Engg.* VOL SE-5, No. 5, September 1979, pp440-452.

[ChMi81]

K. M. Chandy, J. Misra, (1981), "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Comm. of the ACM, Volume 24, No. 11, April 1981, pp 198-206.*

[DaSe87]

W. J. Dally and C. L. Seitz, (1987) "Deadlock-Free Routing in Multiprocessor Interconnection Networks," *IEEE Trans. on Computers*, Vol. 36, No. 5, May 1987, pp 547-553.

[Da87]

W. J. Dally, (1987), "Wire-Efficient VLSI Multiprocesor Communication Networks," *Proc. of Stanford Conf. on VLSI*, pp 390-415.

[Ga88]

A. Gafni, (1988), "Rollback Mechanisms for Optimistic Distributed Simulation Systems", *Proc. of SCS Distributed Simulation Conference*, San Diego January, 1988.

[Fu87]R. Fujimoto, (1987), "Performance Measurements for Distributed Simulation Strategies," *Tech., Report, UUCS-87-026a, Univ. of Utah. 1987.*

[GlIg88]

P. W. Glynn, D. Iglehart, (1987), "Importance Sampling for Stochastic Simulations," Tech. Report 49, Dept. of OR., Stanford University, 1987.

[Hi85]W. D. Hillis, (1985), *The Connection Machine*, MIT Press, Mass., 1985.

[He86]

P. Heidelberger, (1986), "A Statistical Analysis of Parallel Simulations", *Proc. of Winter Simulation Conf. December 1986.*

[Ho78]

C. A. R. Hoare, (1978), "Communicating Sequential Processes," *Communications of ACM*, vol 15, pp 171-176, 1972.

[HwBr84]

K. Hwang and F. Briggs, (1984), *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, NY 1984.

[KaMi66]

R. M. Karp, R. E. Miller, (1966), "Properties of a Model for Parallel Computations: Determinacy, Termination and Queueing, ", *SIAM Journal of Appl. Math.*, Vol 14., November 1966, pp 1390-1411.

[JeSo1983]

D. Jefferson, and H. Sowizral, (1983), "Fast Concurrent Simulation Using and the Time Warp Mechanism", Part 1: Local Control, A Rand Note N-1906-AF; The RAND Corporation, Santa Monica, CA, June 1983.

[Je85]D. Jefferson, (1985), "Virtual Time", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, July 1985.

[Ko75]

W. E. Kohler, (1975), "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems," *IEEE Trans. on Computers*, December 1975, pp 1235-1238.

[Ku78]

D. J. Kuck, *The Structure of Computers and Computations*, Volume 1, John Wiley and Sons, New York, 1978.

[LaMuSa83]

S. Lavenberg, R. Muntz, B. Samadi, "Performance Analysis of a Rollback Method for Distributed Simulation," *Performance, '83*. 1983.

[Mi86]

J. Misra, (1986), "Distributed Discrete-Event Simulation," *Computing Surveys*, Vol. 18, No. 1, March 1986, pp. 39-64.

[MiMi84]

D. Mitra, I. Mitrani, (1984), "Analysis and Optimum Performance of Two Message Passing Processors Synchronized by Rollback, *Proc. of 10th, Int. Conf. on Comp. Perf. Modeling*, 1984.

[MaMe88a]

V. Madisetti, D. Messerschmitt, "Seismic Migration Algorithms on Parallel Computers," *Proc. of 3rd Hypercube Concurrent Computing Conference*, January 1988,

Pasadena.

[MaMe88b]

V. Madisetti, D. G. Messerschmitt, (1988), "Distributed Computation on Concurrent Processors", *Proc. of Allerton Conf.*, September 1988.

[MaWaMe88]

V. Madisetti, J. Walrand, D. Messerschmitt, "Wolf: A Rollback Algorithm for Optimistic Distibuted Simulation Systems," *Proc. of Winter Sim. Conf.* San Diego, December 1988.

[MaWaMe89]

V. Madisetti, J. Walrand, D. Messerschmitt, "Efficient Distributed Simulation," *IEEE/ACM/SCS Annual Simulation Symposium*, Tampa, Florida, March 1989.

[Me79]

D. Messerschmitt, (1979), "Blosim Simulation Program." *U. C. Berkeley*, 1979.

[MuCo69]

R. R. Muntz, E. G. Coffman, (1969), "Optimal Preemptive Scheduling on Two-Processor Systems," *IEEE Trans. on Computers*, Vol C-18, November, 1969,

[NCUBE89]

NCUBE User's Manual, *NCUBE Corp. Beaverton, Oregon.*

[Pa81]J. H. Patel, (1981), "Performance of Processor-Memory Interconnections for Multiprocessors", *IEEE Trans. on Computers*, Vol. C-30, October 1981, pp 771-780.

[PeWoMa79]

J. K. Peacock, J. W. Wong, E. G. Manning, (1979), "Distributed Simulation Using A Network of Processors," *Computer Networks*, Vol. 3, February 1979, pp 44-56.

[RaChGo72]

C. V. Ramamoorthy, K. M. Chandy, M. J. Gonzalez, (1972), "Optimal Scheduling Strategies in a Multiprocessor System," *IEEE Transactions on Computers*, Vol C-

21, pp. 137-146, Feb 1972.

[ReFu87]

D. Reed, R. Fujimoto, (1987), *Multicomputer Networks: Message-Based Parallel Processing*, The MIT Press, 1987.

[ReMaMc87]

D. A. Reed, A. D. Maloney, B. D. McCredie, "Parallel Discrete Event Simulation: A Shared Memory Approach," *ACM Sigmetrics Conf. on Meas. and Model. of Comp. Systems. pp 36-38, 1987. e*

[RiWa89]

R. Righter, J. Walrand, (1989), "Distributed Simulation of Discrete-Event Systems," *Proc. of IEEE*, Vol. 77, No.1 January 1989.

[RiWa88]

C. Rich, R. Waters, (1988), "Automatic Programming: Myths and Prospects", *IEEE Computer*, August 1988, pp 40-51.

[Se85]H. J. Seigel, 1985, *Interconnection Networks for Large-Scale Parallel Processing*, Lexington Books, Lexington, Mass. USA, 1985.

[Sc70]L. Schrage, (1970), "Solving Resource Constrained Network Problems by Implicit Enumeration-Nonpreemptive Case," *Oper. Res.* Vol. 18, pp 263-278, Mar-Apr 1970.

[Se85]C. L. Seitz, (1985), "The Cosmic Cube," *Communications of the ACM*, Vol.28, No. 1, Jan 1985, pp 22-33.

[Se84]C. L. Seitz, (1984), "Concurrent VLSI Architectures," *IEEE Trans. on Computers*, Vol 33, No. 12, December 1984, pp. 1247-1265.

[Ul73]J. D. Ullman, "Polynomial Complete Scheduling Problems," *ACM Operating Systems Rev.* Vol. 20, no. 3, pp. 96-101, Oct. 1973.

[St87]H. J. Stone, (1987), *High-Performance Computer Architecture*, Addison Wesley, Reading, Mass. USA. 1987.

[Wa88]

J. Walrand, (1988), *An Introduction to Queueing Networks*, Prentice-Hall, Englewood-Cliffs, NJ, 1988.

[Wi66]

N. Wirth, "A Note on Program Structures for Parallel Processing," *Communications of the ACM*, 9 May 1966, pp 320-321.

[ZeMo83]

J. Zeman, G. Moschytz, "Systematic Design and Programming of Signal Processors, Using Project Management Techniques," *IEEE Trans. on ASSP*, Vol ASSP-31, No. 6, December 1983.