University of California
Los Angeles

Competitive Execution in a Distributed Environment

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Sung Hyun Cho

1996

© Copyright by

Sung Hyun Cho

1996

The dissertation of Sung Hyun Cho is approved.

Bruce Rothschild

Andrew Kahng

Mario Gerla

David R. Jefferson, Committee Chair

University of California, Los Angeles

1996

Table of Contents

1 Introduction 1

1.1 Background 1

1.2 Speedup of a Sequential Process 3

1.2.1 Competition on Two Variable-speed Processors 3

1.2.2 Competition on Three Variable-speed Processors 6

1.3 Speedup of Communicating Processes 8

1.4 Differences from Related Work 11

1.5 Research Contributions 12

1.6 The Organization of this Dissertation 13

2 Related Work 15

2.1 Introduction 15

2.2 Using Idle Processors 16

2.3 Distributed or Parallel Computing on a SNOW 17

2.4 Replication 19

2.4.1 Replication to Improve Performance 19

2.4.2 Replication to Improve Fault Tolerance 20

3 Competition Protocols 22

3.1 Introduction 22

3.1.1 Variable-speed Processors 22

3.2 The Model 23
3.2.1 The SNOW 23
3.2.2 The Distributed Program 23
3.3 Critical Processes 25
3.3.1 Can We Detect Critical Processes at Runtime? 25
3.3.2 A Measure of Progress 27
3.4 Definition of Speedup for Competition 29
3.5 Maximum Progress among Peer Clones 30
3.6 A Propagating Competition Protocol 32
3.7 Performance Comparison of Competition Protocols 34
3.8 Comparison of Competition and Migration 36
3.8.1 The Advantage of Competition over Migration 36
3.8.2 The Advantage of Migration over Competition 36
3.9 When Can We Use Competition? 37

4 Competition Mechanisms 39

4.1 System Architecture 39
4.1.1 Internal Representation of a Process Clone 41
4.2 Competition Mechanisms 41
4.2.1 Implementation of a Competition Mechanism 43
4.2.2 Message Counters 46
4.2.3 State Propagation 48
4.3 Communication between Process Clones 49

4.3.1 Group Communication 49

4.4 Correctness of Competition Protocols 53

5 Analytic Study of Competition for a Sequential Process . . . 54

5.1 Introduction 54

5.1.1 Assumptions and Terminology 54

5.2 Stretching Ratio 56

5.2.1 Performance of a Variable-speed Processor 59

5.2.2 A Nonpropagating Competition Protocol 60

5.3 An Ideal Competition Protocol 62

5.3.1 Speedup of an Ideal Competition Protocol 62

5.4 Effect of Load Balancing 67

6 Competition and Migration for a Sequential Process 71

6.1 Introduction 71

6.1.1 A Migration Policy and a Competition Policy 72

6.2 Migration Protocols and Competition Protocols 74

6.2.1 Migration Protocols 74

6.2.2 Competition Protocols 76

6.2.3 Inherent Differences between Competition and Migration . 76

6.3 Performance Comparison of *MIG_0* and *COMP_0* 78

6.3.1 *MIG_0* Protocol 78

6.3.2 *COMP_0* Protocol 84

6.3.3 Simulation Results 87
6.4 Performance Comparison of *MIG_D* and *COMP_D* 89
6.4.1 *MIG_D* Protocol 89
6.4.2 *COMP_D* Protocol 92
6.4.3 Simulation Results 96

7 Competition and Migration for Distributed Programs 102
7.1 Introduction 102
7.1.1 The Application Layer 103
7.2 Some Issues in Our Simulation 104
7.2.1 Static and Dynamic Boundary for Migration 104
7.2.2 A Migration Policy and a Competition Policy 106
7.2.3 Assumptions in Our Simulation 106
7.3 Speedup of Competition Protocols 107
7.3.1 Programs with Full Connectivity 107
7.3.2 Programs with Mesh Connectivity 112
7.3.3 Programs with Ring Connectivity 115
7.4 Conclusion 116

8 Conclusions and Future Work 118
8.1 Conclusions 118
8.2 Future Work 119

References 121

List of Figures

1.1 Execution on Two Variable_speed Processors 5

1.2 Execution on Three Variable_speed Processors 7

1.3 Static Choices in Executing Communicating Processes 9

1.4 Competition and Migration for Two Communicating Processes . . 10

3.1 Execution Speed of Processors P_1 and P_2 31

3.2 Progress of Clones sp_1 and sp_2 32

3.3 State Propagation between Clones sp_1 and sp_2 33

3.4 Performance Comparison of Three Competition Protocols 35

4.1 System Software Layers 40

4.2 A Message M from a Process P to another Process Q 42

4.3 Message Communication in a Competition Mechanism 42

4.4 One-to-many Communication 52

4.5 One-to-one and Multicasting among Peer Clones 52

4.6 One-to-all Communication 52

5.1 Two State Model of a Processor 55

5.2 Execution Time on a Variable-speed Processor 56

5.3 Number of Clones to Have a 5% Diminishing Return 65

5.4 A M/M/2 Queue 67

5.5 The State Diagram of the M/M/2 Queue 68

6.1 Skeletons of Protocols 75

6.2 The *MIG_0* Protocol 78

6.3 The *COMP_0* Protocol 84

6.4 Simulation Results with Two Processors 88

6.5 Simulation Results with Three Processors 89

6.6 The *MIG_D* Protocol 90

6.7 The *COMP_D* Protocol 93

6.8 Speedup with Two Processors When $D = 0.2\tau_i$ 96

6.9 Speedup with Two Processors When $D = \tau_i$ 97

6.10 Speedup with Three Processors When $D = 0.2\tau_i$ 99

6.11 Speedup with Three Processors When $D = \tau_i$ 100

6.12 Speedup with Two and Three processors When $\tau_b = \tau_i$ 100

7.1 Static Boundary 104

7.2 Dynamic Boundary 104

7.3 10 Processes with Full Connectivity When D = 0 108

7.4 10 Processes with Full Connectivity When D = $0.2\tau_i$ 108

7.5 Speedup of Competition on Three Processors 110

7.6 Speedup of Migration on Three Processors 110

7.7 Comparison of Competition and Migration When D = $0.2\tau_i$ . . . 111

7.8 The Effect of the Number of Processes on the Speedup 112

7.9 The Effect of Granularity on the Speedup 113

7.10 The Effect of Load Imbalance on the Speedup 113

7.11 9 Processes with 2D Mesh Connectivity When D = $0.2\tau_i$ 114

7.12 4, 9, and 16 Processes with 2D Mesh Connectivity When D = $0.2\tau_i$ 115

7.13 10 Processes with Ring Connectivity When D = $0.2\tau_i$ 116

7.14 The Speedup with Different Topologies When D = $0.2\tau_i$ 117

List of Tables

Acknowledgments

I wish to express my gratitude to all the people who have helped me in going through my research and finishing one of the most difficult tasks in my life.

First of all, I am grateful to my advisor Professor David R. Jefferson for his patient advice during the final stage of this research. I am also thankful to Professors Bruce Rothschild, Andrew Kahng, and Mario Gerla for serving on my committee. Especially I like to thank Professor Andrew Kahng for allowing me to use one of workstations in his laboratory when the department computing environment was not stable.

I also like to thank Dirceu Cavendish Jr. for reading this dissertation and helping my simulation. Thanks to my past and present colleagues Art Goldberg, T. Ravi, Wentoh Liao, Liwen Chen, Kong Li, Adam King, Vikas Jha, Jeonghee Yi, and Byung-Ro Moon.

Verra Morgan, Valerie Aylett, and Shelly Backlar's help in dealing with the UCLA administration is sincerely appreciated. I am especially grateful to Verra for helping me during my stay at UCLA.

I would like to thank my mother, my wife, Gye Soon, and my two daughters, Sooah and Eunah. Without their constant prayers and loving support, I could not have gotten to this point.

My appreciation also goes to the Korean government which had provided me fellowship for four years.

Vita

1978	B.S., Seoul National University, Korea
1980	M.S., Seoul National University, Korea
1980–1983	Instructor, Korea Military Academy, Korea
1983–1987	Korean Government Fellowship
1987–1990	Research Assistant, Computer Science Department, UCLA
1990–1992	Research Assistant, Radiology Department, UCLA
1993–1995	Research Assistant, Computer Science Department, UCLA

Abstract of the Dissertation

Competitive Execution in a Distributed Environment

by

Sung Hyun Cho

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1996

Professor David R. Jefferson, Chair

We propose an alternative to process migration, called *competition*, to speed up distributed programs in the background on a network of variable-speed processors. Competition protocols are *transparent* operating system facilities that involve creating multiple instances (called *clones*) p_1, p_2, etc. of a process P on different variable-speed processors, and making clones "compete", i.e., attempting to guarantee that the output of the clone that is farthest "ahead" is fed to the rest of the computation, and that the entire application's performance tracks that of the clone which is farthest ahead.

One clone may be ahead of or behind others depending on the current foreground loads. If for any reason there is variation in the progress of the clones, so that one clone is ahead at some times, but another is ahead at other times, then a set of competing clones may outperform any single copy.

We present issues and mechanisms in competition, and show that competition protocols offer performance benefits that are as good as or better than migration

protocols for sequential or distributed programs under comparable assumptions. This claim is supported by analytical results and simulation results for sequential programs. Our simulation results for distributed programs also show that competition protocols offer even more speedup for distributed programs than for sequential programs.

CHAPTER 1

Introduction

1.1 Background

The cost/performance ratio of workstations has shown a dramatic improvement in the past, and this trend will probably continue in the near future. Accordingly, networks of workstations (NOW) [ACP95] have been increasingly prevalent in almost every computing environment. Much of computing capacity of a typical workstation has gone unused, and a NOW presents a large available source of computing power.

Since the middle of the 1980's, many researchers have tried to harness idle workstations, for example, the V system [TLC86], NEST [AE87], Butler [Nic87], Condor [LLM88], Benevolent Bandit [FSK89], REM [Sho91], and Stealth [KC91], and Utopia [Zho93]. The goal of those systems is to execute processes on remote idle workstations, maximizing the utilization of workstations. In this study, however, our goal is to execute *distributed* programs on a NOW as quickly as possible.

We may dedicate a subset of workstations in a NOW to execute distributed programs, or we may share the workstations with interactive foreground processes. These two NOW environments are called a *dedicated NOW* (DNOW) and a *shared NOW* (SNOW) respectively. The progress rate of processes on a DNOW is constant, but that on a SNOW may unpredictably vary due to processor shar-

ing. However, a DNOW with variable-speed processors is identical to a SNOW with background execution, and we will use both points of view in this study. We will focus on methods which are able to execute distributed programs on a SNOW in the background as quickly as possible.

One of characteristics in a SNOW is that the available computing capacity to background processes changes dynamically, probably fast or drastically. This unpredictable variability may drastically degrade the performance of background programs; an entire program can perform poorly when one of its processes is executing on a heavily loaded workstation.

One of key issues in a SNOW is how intrusive a background program is to foreground processes and vice versa. Owners may not be tolerant of interference from background programs. Workstation operating systems (WOS) have to guarantee that foreground processes have preemptive priority so that background programs do not interfere. Additionally, CPU scheduling, memory management, and file access must be prioritized to insulate foreground processes from background programs [KC91].

We assume that WOS manages resources such that background programs do not interfere with foreground processes, but foreground processes will interfere with the performance of background programs. We refer to workstations scheduled this way as *variable-speed* processors for background programs. We use the terms "workstations" and "processors" interchangeably in this study.

1.2 Speedup of a Sequential Process

1.2.1 Competition on Two Variable-speed Processors

Suppose that we have a non-interactive sequential process *sp* that we want to run in the background. In addition, we have two variable-speed processors $P1$ and $P2$ on which we might run *sp*. Here we cannot know what the future foreground activities are going to be on $P1$ and $P2$. What do we do?

We could statically choose one of them to execute *sp*. We could be wrong because we might choose the slower variable-speed processor. Even if we choose correctly, we may be able to do better than a static choice by migrating *sp* between $P1$ and $P2$ back and forth on the basis of load averages because the speed of variable-speed processors can change dynamically.

Alternatively, we could execute *sp* on both of $P1$ and $P2$ in parallel. The instances of *sp* on $P1$ and $P2$ are called *clones* of *sp*. Then we monitor which clone is ahead from time to time. If one clone is sufficiently far ahead, we pause and propagate the state of the clone ahead to the one that is behind. The clone that is behind uses the propagated state to continue execution. State propagation allows the clone that is behind to jump forward to the state of the clone ahead without executing instructions in *sp* itself. This is the essence of *competition*, where all clones 'compete' to execute as quickly as possible.

Under the same circumstances where migration protocols get speedup, we will show that competition protocols can get even better speedup by making use of more CPU cycles. However, competition protocols use only otherwise idle CPU cycles, so they do not really have a higher "cost" at all. But they presumably use more communication bandwidth.

Here is an example showing that a competition protocol can execute *sp* quicker than a migration protocol. In this example, we assume a simple policy for a migration protocol: when *sp* is preempted due to foreground processes, a migration protocol migrates *sp* immediately to an idle processor if one is available. Otherwise, the migration protocol migrates *sp* later as soon as an idle processor becomes available. This migration protocol is called a *MIG* protocol. On the other hand, a competition protocol propagates states only when a clone ahead is preempted. This competition protocol is called a *COMP* protocol. We assume that migration decision and state propagation decision is made instantly.

Consider two variable-speed processors $P1$ and $P2$ in Figure 1.1. Shaded rectangles represent foreground activities. Foreground activities on $P1$ and $P2$ are identical in Figure 1.1 (a), (b), and (c). Thick vertical arrows represent the fastest execution trace of *sp*, and dotted horizontal arrows represent process migration in Figure 1.1 (b) or state propagation in Figure 1.1 (c). Numbers in the left side of Figure 1.1 represent real time.

We assume that the latency for process migration or state propagation is 1 unit of time, and that $P1$ and $P2$ are physically identical. We also assume that it takes 10 units of time to run the sequential process *sp* on a dedicated processor. When we statically choose one processor, either $P1$ or $P2$, it takes 17 or 16 units of time on $P1$ or $P2$ respectively as shown in Figure 1.1 (a). When we migrate *sp* back and forth as shown in Figure 1.1 (b), the *MIG* protocol completes *sp* in 15 units of time in which process migration occurs five times. This example shows that the *MIG* protocol completes *sp* quicker than static choices do.

We assume that both of *sp*'s clones are not scheduled during the period of state propagation to prevent state modification. Figure 1.1 (c) shows those periods by

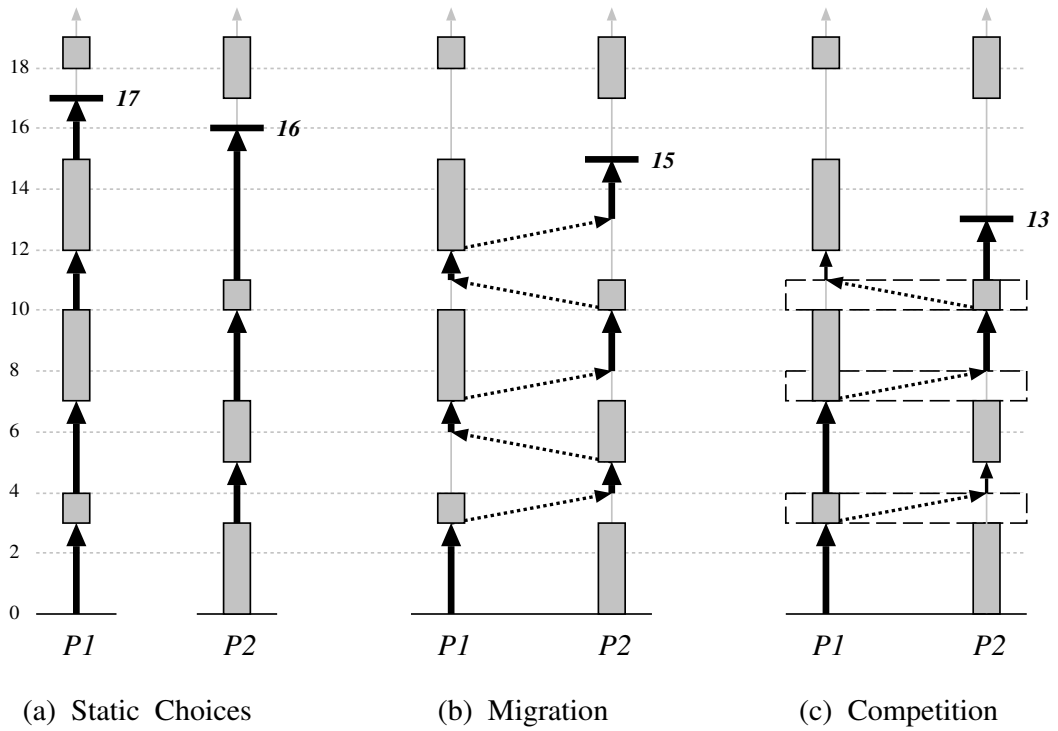


Figure 1.1: Execution on Two Variable_speed Processors

dashed rectangles, and execution traces of sp's two clones on $P1$ and $P2$. The faster trace of execution is shown by thick vertical arrows and the slower one is shown by thin vertical arrows. At real time 5 we consider that the clone on $P1$ is ahead because it is executing but the other on $P2$ stops execution. Similarly the clone on $P2$ is considered to be ahead at real time 12. Thus the *COMP* protocol does not need to propagate states at real time 5 and 12. The *COMP* protocol completes sp in 13 units of time in which state propagation occurs three times.

A competition protocol can easily simulate the behavior of the *MIG* protocol: whenever the *MIG* protocol migrates sp from a processor Pi to another Pj, the competition protocol also propagates the state of a clone on Pi to Pj. However, if the other clone on Pj is ahead, the competition protocol can easily suppress state propagation. Figure 1.1 shows that the *COMP* protocol suppresses state

propagation twice at real time 5 and 12 when *sp* is migrated. Thus the *COMP* protocol completes *sp* quicker than the *MIG* protocol by 2 units of time because the number of state propagation is smaller than that of process migration by 2.

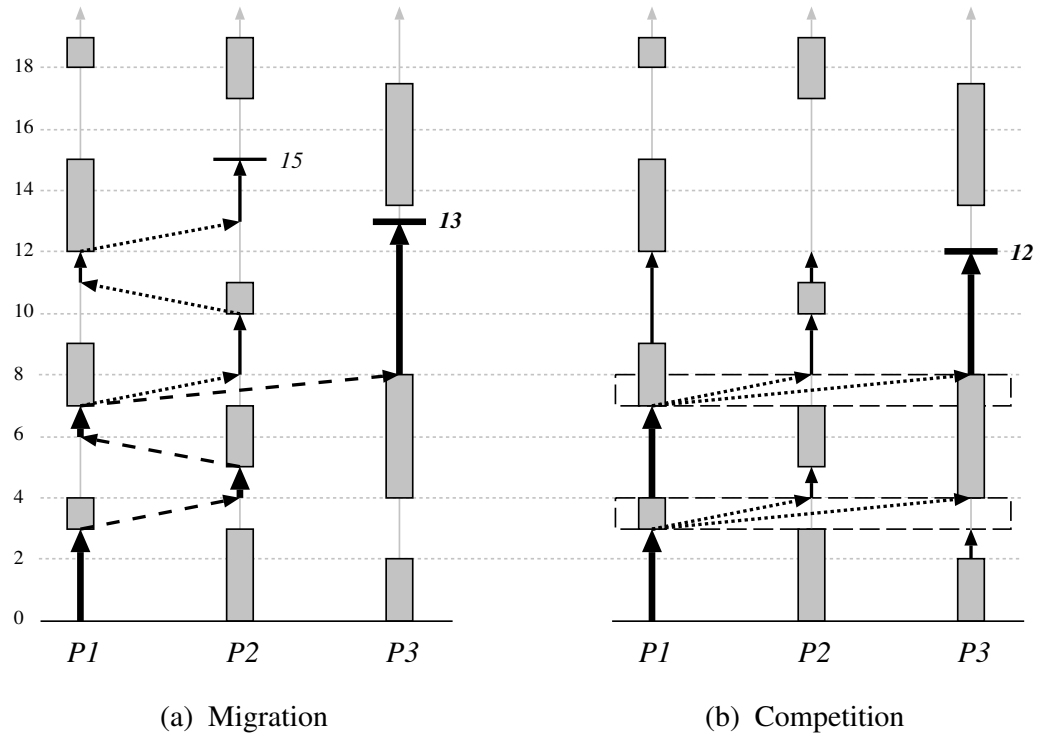
When there is no chance that the *COMP* protocol suppresses state propagation, the *COMP* protocol performs as well as the *MIG* protocol under the assumption that the overhead of state propagation is equal to that of migration. However, in case it is possible that the *COMP* protocol suppresses state propagation, the *COMP* protocol always performs as well as or better than the *MIG* protocol. The *COMP* protocol may perform significantly better than the *MIG* protocol if state propagation is suppressed many times when in the *MIG* protocol process migration would occur.

Competition speeds up *sp* by executing two clones of *sp* on two processors. However, competition does not increase the parallelism of *sp*, because *sp* is a sequential process and has no parallelism in it. We will show another reason why competition protocols are preferable to migration protocols in the next subsection.

1.2.2 Competition on Three Variable-speed Processors

Consider another example in Figure 1.2 where there are three variable-speed processors $P1$, $P2$, and $P3$. Assume that $P1$, $P2$, and $P3$ are physically identical again. In fact, foreground activities on $P1$ and $P2$ in Figure 1.2 are identical to those in Figure 1.1. Now we want to execute *sp* on $P1$, $P2$, and $P3$ in the background.

The dashed horizontal arrows in Figure 1.2 (a) show the optimal migration paths where migration completes *sp* in 13 units of time. But migration completes *sp* in 15 units of time if *sp* is migrated according to the dotted horizontal arrows.



(a) Migration (b) Competition

Figure 1.2: Execution on Three Variable_speed Processors

Thus it is necessary to migrate sp at real time 7 from $P1$ to $P3$ rather than $P2$ to obtain the shortest execution time. However, when migration decision is made at real time 7, it cannot be known that $P3$ is a better choice than $P2$ without the complete knowledge of the future foreground activities on $P2$ and $P3$. In general, this information will not be available. Thus it is impossible to achieve the optimal performance using migration.

Competition completes sp in 12 units of time as shown in Figure 1.2 (b) where the thick vertical arrows show the execution trace of the fastest clone at any given time. Unlike migration, competition does not need to know the future foreground activities on $P2$ and $P3$ at real time 7 to finish sp in the shortest time. Competition easily obtains this benefit at the cost of redundant execution of clones. Redundant execution may not cost anything in a SNOW because

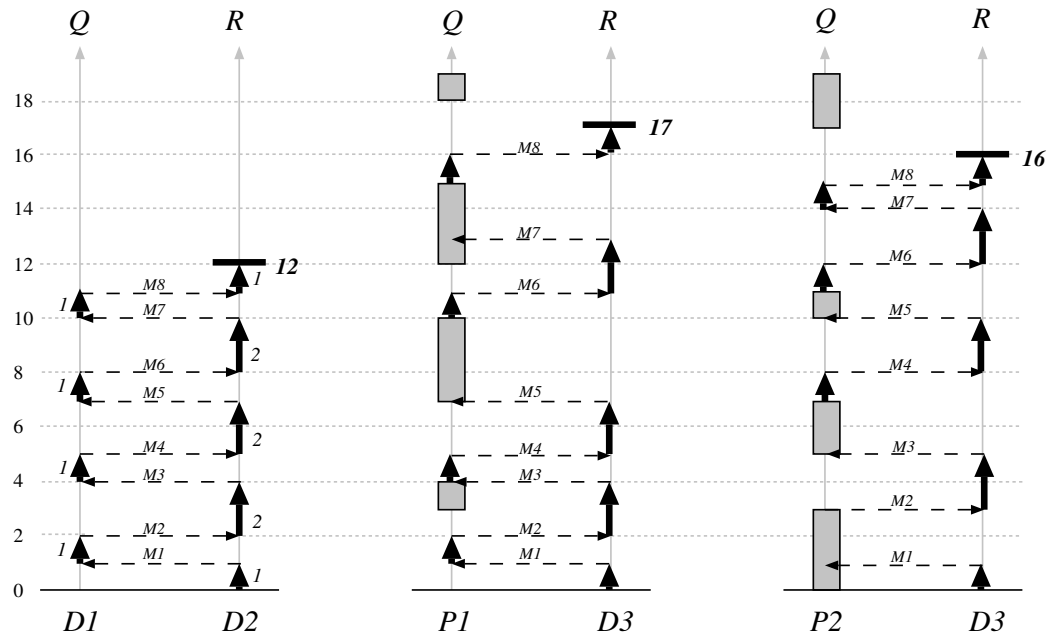
competition exploits processors that are otherwise idle.

So far we implicitly assume that competition always propagate states from the clone that is ahead to the others that is behind, but not vice versa. This assumption can be easily satisfied; based on the number of instructions that clones have executed, or the CPU times clones have used, we can determine which clone is most ahead at any given time. However, even if a heuristic decision is occasionally wrong, propagating states from a clone that is behind to one that is ahead won't violate correctness though it may degrade the performance of competition.

1.3 Speedup of Communicating Processes

Now we want to execute a distributed program on a set of workstations in a SNOW in the background. Let us consider a simple distributed program that consists of two communicating processes Q and R. Figure 1.3 (a) shows the behavior of Q and R when they run on two physically identical processors. We assume that message communication is instantaneous. Dashed arrows represent message communication between Q and R, and a message is shown above each dashed arrow. Numbers beside thick vertical arrows represent time to execute messages. R produces a message $M1$ in one unit of time and sends it to Q. Then Q executes $M1$, produces a message $M2$ in one unit of time, and sends it to R. Then R executes $M2$, produces a message $M3$ in two units of time, and sends it to Q, and so on. The distributed program completes in 12 units of time on two dedicated processors.

Now assume that we have two variable-speed processors $P1$ and $P2$ and a dedicated processor $D3$. We assume that $P1$, $P2$, and $D3$ are physically iden-



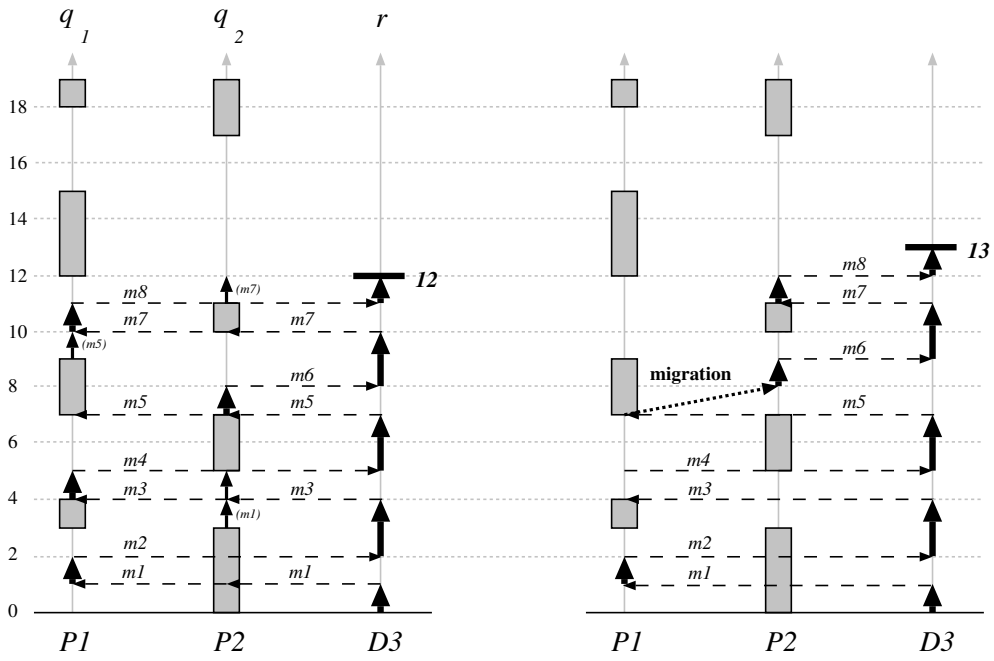
(a) Dedicated Processors (b) Q on P1 (c) Q on P2

Figure 1.3: Static Choices in Executing Communicating Processes

tical, and that message communication is instantaneous. How can we execute the distributed program in the background? First we consider static choices in Figure 1.3 (b) and (c) where the foreground activities on $P1$ and $P2$ are identical to $P1$ and $P2$ in Figure 1.1 or Figure 1.2.

One possible configuration is shown in Figure 1.3 (b) where Q and R run on $P1$ and $D3$ respectively. A message $M5$ ($M7$) arrives at $P1$ at real time 7 (13), but Q cannot execute $M5$ ($M7$) until real time 10 (15) because of foreground activities on $P1$. The distributed program completes in 17 units of time on this configuration. Figure 1.3 (c) shows another configuration where Q and R run on $P2$ and $D3$ respectively. Here the distributed program completes in 16 units of time.

Now we consider how much speedup the *COMP* protocol can achieve on $P1$,



(a) Competition (b) Migration

Figure 1.4: Competition and Migration for Two Communicating Processes

$P2$, and $D3$ as shown in Figure 1.4 (a) where process Q is represented by two clones q_1 and q_2 on $P1$ and $P2$ respectively while process R is represented by a single clone r on a dedicated processor $D3$. Foreground activities on $P1$ and $P2$ are identical to those shown previously in Figure 1.3.

Here we simply assume that when process R sends a message M_i to Q, the clone r sends two instances of M_i to q_1 and q_2 instantaneously. Instances of the message Mi are called *message clones* of M_i, and they are shown by m_i's. Dashed horizontal arrows represent communication of message clones. We will discuss this issue in detail in Chapter 4. Figure 1.4 (a) shows that r sends message clones m_1, m_3, m_5, and m_7 to both q_1 and q_2 instantaneously at real time 1, 4, 7, are 10 respectively.

The clone q_1 can execute m_1 immediately and send a message clone m_2 to r,

but q_2 cannot execute m_1 until real time 3 due to foreground activities on $P2$. This delayed execution is represented by a message clone m_1 in a parenthesis beside a thin vertical arrow. When q_2 produces m_2, the *COMP* protocol suppresses sending $m2$ because it was sent earlier by q_1. Figure 1.4 (a) shows that the thick vertical arrows trace the fastest execution of the distributed program. Both q_1 and q_2 produces m_4 at real time 5, but only m_4 produced by q_1 is shown in Figure 1.4 (a). We will discuss this issue in chapter 4. The *COMP* protocol completes the distributed program in 12 units of time.

Now consider how much speedup the *MIG* protocol can achieve on $P1$, $P2$, and $D3$ as shown in Figure 1.4 (b). Figure 1.4 (b) shows that initially Q and R start to run on $P1$ and $D3$ respectively, but shows that Q is migrated from $P1$ to $P2$ at real time 7. We assume that the latency for migration is one unit of time. The *MIG* protocol completes the distributed program in 13 units of time. The *COMP* protocol completes the distributed program quicker than *MIG* protocol by one unit of time because of one process migration.

1.4 Differences from Related Work

Many researchers have studied a set of homogeneous or heterogeneous processors mostly for load sharing or load balancing. However, they considered a *fixed* performance environment where the execution speed of processors do not change. A positive local change in an architecture (speeding up a processor, for example) might lead to a negative global effect for a distributed program. This phenomenon is called a paradoxical performance behavior. Chen [Che91] considered a *changing* performance environment to execute a distributed program on different architectures without a paradoxical performance behavior. In this study, we consider a

problem of executing a distributed program on a SNOW in the background, and thus our performance environment can change *dynamically* while executing the program.

The idea of replication has been investigated intensively for improving fault tolerance [AK84, Bir85], but we propose process replication for improving the performance of a distributed program on a SNOW. Previous work on process replication for increasing performance attempted to increase parallelism or reduce communication distance [AJ87]. Thus asymmetric strategies are employed in managing clones of a process, e.g., there is a "primary" clone, and only the primary clone can send a message [AJ87]. However, competition protocols employ a symmetrical strategy because any clone of a process can be the fastest at any given time. Competition protocols improve the performance of a distributed program by speeding up processes through making their clones compete, but *without increasing parallelism.*

1.5 Research Contributions

We investigate competition protocols in a SNOW where the execution speed of processors may change due to foreground processes. We study issues, mechanisms, and performance benefits of competition protocols for sequential and distributed programs.

The main contributions of this research are summarized below:

- Characterization of when competition may work, and how much it can improve performance, at what cost if any.
- Qualitative study of competition mechanisms.

- Performance analysis of competition protocols for a sequential process.
- Performance comparison of migration and competition.
- Performance analysis of competition protocols for distributed programs based on simulation.

We show that competition has its value on a SNOW. However, we also identify when competition protocols can be exploited on a DNOW as well as general distributed or parallel computing platforms. We investigate competition for sequential processes thoroughly based on mathematical analysis and simulation. But we have by no means finished the subject of competition in the distributed context.

1.6 The Organization of this Dissertation

We study competition for sequential programs primarily up to Chapter 6. We then extend competition to distributed programs in Chapter 7. This dissertation is organized as follows. In Chapter 2, the related work in current literature is discussed.

In Chapter 3, we introduce formal definitions of variable-speed processors and our speedup metric for competition. Then we introduce the model for a SNOW and that for a distributed program. We discuss which processes should be cloned, and how to detect them. Finally we introduce two competition protocols.

In Chapter 4, we present a competition mechanism and some issues including its correctness. Then we study the performance of two competition protocols for sequential programs in Chapter 5. We compare the performance of competition

with the performance of migration for sequential programs in Chapter 6.

We study the performance benefit of competition for distributed programs with different topologies such as ring, full connection, and 2 dimensional mesh in Chapter 7. Finally, in Chapter 8 we summarize the contributions of our work and present related open problems.

CHAPTER 2

Related Work

2.1 Introduction

In order to minimize the completion time of parallel or distributed applications, many researchers have taken different approaches under different circumstances. There has been a considerable amount of theoretical work on the assignment problem for parallel and distributed computing. Most of this work addresses the problems of mapping tasks to processors under the assumption that resource requirements of distributed or parallel applications are known *a priori* on a DNOW [Bok87]. These static approaches utilize the knowledge of problem characteristics to reach a global optimal, or near optimal, solution with well-balanced load [ELA94].

Load balancing research in operating systems also focuses on the mapping problems. It is assumed that resource requirements of applications is not known *a priori*, and heuristics play an important role. Eager et al. [ELZ86b] compare heuristics for task placement and migration under various system loads. They show that simple heuristics work well when scheduling independent processes in homogeneous distributed systems [ELZ86a].

We will review remote execution facilities and distributed computing on a SNOW, and replication techniques for performance and fault tolerance.

2.2 Using Idle Processors

Many researchers have tried to harness idle or underutilized workstations by using remote execution facilities; these include the V system [TLC86], NEST [AE87], Butler [Nic87], Condor [LLM88], Benevolent Bandit [FSK89], REM [Sho91], and Stealth [KC91], and Utopia [Zho93].

There are two important issues: one is when a workstation is determined to be available for background processes; the other is how to handle background processes when workstations are reclaimed. In earlier systems [TLC86, Hag86, AE87, Nic87], availability is defined absolutely; an idle workstation is the one not used at all. Refer to [KC91, ML91] for more study on the available capacity of workstations.

When a processor is reclaimed by a foreground process, we can take two approaches; one is to migrate background processes out of it [TLC86, AE87, LLM88, FSK89]; another is to use a priority-driven local scheduler, which preempts background processes and schedules the foreground process, and reschedules background processes when the foreground process finishes [KC91, Sho91, JXT93]. The advantage of a priority-driven local scheduler is that it can utilize fully available CPU cycles. However, if a process in a distributed program is running on a heavily loaded processor, the other processes may have to wait for the process because of communication or synchronization. Therefore a priority-driven local scheduler is not appropriate to execute distributed programs in a SNOW. Process migration is the only alternative out of the two approaches to be employed in a SNOW.

Process migration has also been a mechanism for load balancing or shar-

ing [LM82, PW85, NXG85, ELZ88, KL88, AF89, DO91, KK92, SZM94]. Refer to the survey on process or object migration in [Smi88, Nut94].

2.3 Distributed or Parallel Computing on a SNOW

In order to execute distributed programs on a SNOW, researchers have studied the following areas; developing software tools, increasing communication performance, and studying feasibility of distributed computing on a SNOW.

Several software tools have been developed to support SNOW environments; PVM [Sun90, Gei94], P4 [Boy87], Express [FKB91], Linda [CG89], Isis [BM89], and MPI [GLS94]. These tools allow programmers to treat a heterogeneous collection of processors in a SNOW as one large virtual machine. The virtual machine can be composed of a set of identical workstations. But since networked computers can have several other users on them running a variety of jobs, the machine load might vary dramatically from moment to moment. The result is that the effective background computational power of otherwise identical workstations can vary enormously.

The performance of distributed programs on a network of workstations is partly determined by the speed of the underlying communication network. Network speeds have been increasing rapidly, but the communication latency at the application level has not been improved much because of the overhead in preparing to send or receive a message. Many researchers have tried to eliminate the network I/O bottleneck [Cha94, LHD94, EBB95], and it was demonstrated that executing communication-intensive distributed programs over local ATM networks appears to be promising [LHD94].

The goal of the Berkeley NOW project is to harness all the computers in a building to satisfy the needs of both desktop computing and applications that require a hundredfold more computing resources than any single machine can provide. They improve communication performance by two orders of magnitude by using an ATM network and reducing communication overhead based on lean communication layers [EBB95]. They reported that they can overlay a typical 32-processor parallel workload on a 64-workstation cluster used for interactive foreground jobs without significantly sacrificing the performance of either.

At Lewis Advanced Cluster Environment (LACE) in NASA Lewis Research Center all machines are connected by two Ethernet networks: one is public (for file server and other activities) and the other is private (for multiprocessing) [Fat95]. The underlying communication network in a SNOW can be the bottleneck in executing communication-intensive distributed programs. When there are no alternatives in networks, unlike processors, the performance of competition protocols may heavily depend on the foreground message traffic on the network. Thus the environment like LACE is ideal for competition protocols because of alternatives in networks.

Korflage [KK89] investigated the performance of a distributed program on a SNOW under the assumption that workload of the program can always be divided evenly among all available processors. He ignored overheads that occur in a real system such as communication delay. Thus, his model provides a lower bound on the time to completion. In our study, we investigate methods to speed up a distributed program on a SNOW and their performance benefits. Leutenegger and Sun [LS93] studied whether distributed programs can be run in a SNOW. They demonstrated that this may be possible without a significant impact on

distributed programs.

2.4 Replication

2.4.1 Replication to Improve Performance

Goldberg [AJ87] has proposed process replication to reduce communication distance and increase parallelism. He distinguished read messages from write messages to minimize redundant execution and message communication bandwidth; read messages are executed in only one of the clones and no message clones are delivered to the others. When the clone receiving original read messages executes on a heavily loaded or slow processor, the slow execution of the clone can slow down the whole distributed program. In competition protocols, all clones compete to execute read and write messages, and the performance of the whole program tracks the performance of the clone that is farthest ahead at the cost of more message traffic and redundant computation.

Most research on replication for performance has two goals: reduce communication cost by applying read operations to a local copy of the object [Phi89] and increase parallelism by reading local copy (avoid sequential bottleneck) [AJ87, BKT92]. So an important factor is the ratio of read to write operations, and is useful primarily if a shared object is read relatively often because of the overhead to maintain consistency of multiple copies.

When there are alternative methods of computing a result, Smith [SG89] has proposed concurrent execution of those alternative methods to get the result quickly. However, programmers have to provide all alternatives to solve a problem. Competition protocols share a similar philosophy in exploiting multiple

alternative processors where clones of processes are executed in parallel. In order to use competition protocols, users do not need to provide any alternatives, and do not need to change their programs; competition protocols are *transparent*.

2.4.2 Replication to Improve Fault Tolerance

A replicated process group of deterministic processes, called a *troupe*, was proposed as a building block of replicated distributed programs for transparent fault tolerance [Coo85]. Processes are assumed to communicate via remote procedure calls, which are transformed to "replicated procedure calls" by the operating system. Replicated procedure call sends a message from each member of the calling troupe to each member of the receiving troupe, and the message cost is quadratic in the number of clones.

Process replication was proposed for fault tolerance and performance improvement in [CC94]. Their major concern was fault tolerance, but they also tried to improve performance by allowing only the first replica of a troupe generating a message to multicast the message to reduce communication traffic. Thus destination replica does not need to identify possibly redundant messages. However, a replica, upon generating a message, must check to see whether the same message has already been sent out by other replicas. If the outcome is negative, it would make a request for permission to send to its coordinator. Therefore, the first replica generating a message always needs to get a permission from its coordinator, and this can delay message delivery as much as the latency to get a permission.

In a SNOW, ideally the frequency that clones of a process send a message clone concurrently is rare, and thus competition protocols allow process clones to

send message clones to destination clones without the concept of the coordinator. When a clone sends a message clone, competition protocols may multicast the message clone to both destination clones and the other peer clones of the sender. If the other clones of the sender generate the message clone after the message clone that is broadcast has arrived at them, competition protocols suppress multicasting the message clone.

Our motivation for process replication is totally different from others, and communication of message clones in competition protocols differs from that in the other protocols. We are not aware of any previous study similar to competition protocols, and consequently any systematic study of competition protocols.

CHAPTER 3

Competition Protocols

3.1 Introduction

We first define a variable-speed processor, and discuss our model of a distributed program. Then we define speedup for competition, and present competition protocols. Finally, we discuss when it is appropriate to use competition protocols.

3.1.1 Variable-speed Processors

The foreground load of processors in a SNOW can vary unpredictably and thus the time available to execute background processes can also vary. This is why SNOW processors are referred to as variable-speed processors from the point of view of background processes. When there is no ambiguity, a variable-speed processor will be simply called a processor.

The progress of a background process can be measured in the number of instructions it has executed so far. In practice an operating system ordinarily does not have access to the number of instructions executed, but it does have access to the CPU time used. For simplicity we assume that all the processors are physically identical, and thus CPU time per second is linearly proportional to the number of instructions a processor executes per second.

The progress rate of a background process goes down on a processor when

its foreground load increases. Although the execution speed of a processor for a background process is not a continuous, but a step function at a very fine scale, the average execution speed at a larger scale can be considered as a continuous function at a coarser time scale. In this chapter we assume that the execution speed of a processor for a background process changes continuously.

3.2 The Model

3.2.1 The SNOW

We assume that a SNOW consists of M processors and a variable-delay communication network connecting processors. Each processor has its own memory, and processors communicates with each other via messages. Their execution speeds for background processes change unpredictably, and the execution speed of a processor is uncorrelated with those of other processors. In this study, we make only one assumption about execution speeds of processors for background processes: they all have the same average execution speed.

3.2.2 The Distributed Program

We assume that a distributed program is written in the form of a set of cooperating sequential processes that communicate only via asynchronous messages without shared memory. For simplicity we assume that processes are neither created nor destroyed dynamically. A process may execute **send** or **receive** primitives for communication or synchronization.

A message is sent or received by executing:

send (message, receiver_name),

receive (message, sender_name).

In this model, we assume *direct naming*, that is, messages are addressed directly to processes by name. There are no multicasting or broadcasting primitives.

A process running on a faster processor than its communication partner will reach the communication statement earlier. Synchronous message passing called rendezvous as in Occam [Wex89] delays a process sending a message until the other process is ready to receive the message. This can degrade the completion time of programs significantly in a SNOW. Thus we assume asynchronous message passing where the send primitive is non-blocking, that is, it does not cause a process to block. Thus sending and receiving processes execute independently, and a message might be received arbitrarily long after it has been sent. However, asynchronous message passing requires message buffering and queueing.

We assume that the communication traffic for the background processes have lower priority in the communication subsystems than that for the foreground processes. Consequently the communication delay is as variable as the speed of a processor. But in this study we model the communication delay as constant. We also assume that messages are reliably delivered, and that message order is preserved between two communicating processes.

We require the runtime behavior of each process to be deterministic; the sequence of states it passes through and the sequence of messages it sends depends only on the process's initial state and the sequence of messages received. Non-deterministic programming constructs are not allowed in our model. Thus all clones of a process execute the same instructions in the same order, and send and receive the same messages in the same sequence, and go through the same sequence of states.

3.3 Critical Processes

Competition can reduce the completion time of a distributed program by making all processes in the program compete. In general, however, all processes in the program probably do not need to compete, but the subset of processes that is critical for the completion time of the program does need to. The set of critical processes can change during the execution of the program. Formally we can define a *critical* process in a distributed program as follows:

Definition 3.1 *A process p in a distributed program is critical at real time t if and only if the completion time of the entire program would be longer at least by Δt if we pause the process p at t for a short period of Δt.*

Accordingly, at least one process in a sequential program is always critical. In a distributed program, whenever the last process becomes non-critical, some other process becomes critical unless the system is deadlocked. Therefore there will always exist at least one critical process in a program. The goal of competition is not to make critical processes to non-critical ones, but to speed up critical processes because by doing so, we can speed up the completion time of the whole program.

3.3.1 Can We Detect Critical Processes at Runtime?

The definition of critical processes described above is theoretic, and we cannot detect critical processes based on that definition at runtime. Most runtime scheduling algorithms do not utilize knowledge of the application programs [SW95]. Is there any heuristic to detect critical processes without using such knowledge?

Consider a distributed program that consists of two sequential processes P and

Q. Assume that process P is composed of seven million instructions and process Q is composed of four million instructions. Assume for illustrative purposes that there is no communication between them. Thus they can be executed fully in parallel, and the "application" is complete when both have terminated.

How do we execute those two processes P and Q on two dedicated processors with different execution speeds? If we know which is the longer process, we can execute P on the faster processor and Q on the slower processor. If P terminates earlier than Q does, we may migrate Q to the faster processor depending on the migration delay and the remaining time to finish Q on the slower processor. However, if it is not known which one is the long process, what do we do? Operating systems cannot make any reasonable guess about which is the longer process. Thus operating systems cannot execute them better than the average.

Now assume that we execute P and Q on two variable-speed processors. It is not sufficient to know which is the longer process. In the beginning P is critical and we execute P on a faster variable-speed processor by migrating it back and forth. At some point of time P may progress ahead of Q, which becomes critical. But there is no way to detect when Q becomes critical. In order to reduce the completion time of a distributed program on a SNOW, we should be able to detect which process is critical at any given time.

If we do not have this information, that is, we do not know which process is critical at any given time, then we may clone all processes so that we include critical processes. Therefore we get competition benefit just as if we knew what the critical process was. The price to pay is that we just need more excess background resources to get that benefit.

3.3.2 A Measure of Progress

In this study we are looking at the case where there are enough excess resources to clone all of the processes. However, we only have to clone processes that are critical. If there are good heuristics in determining which processes are critical at any given time, then operating systems can get away with much less cloning by cloning only critical processes. But the heuristics for deciding which processes are critical are beyond the scope of our study. Our examples in the later chapters are symmetric enough that all processes are equally likely to be critical.

It is not a part of our research to investigate how to detect critical processes, but we will discuss some issues related with critical processes. Distributed computation in general is just more complex than sequential computation due to new concepts such as communication, synchronization, or load balancing. We do not have such analogs in sequential computation. Thus new concepts should appear to do with competition in distributed computation. One of those is a measure of "progress".

We have an easy measure of progress in sequential programs. It's just CPU time or the number of instructions. The number of remaining instructions to complete a sequential program is the total number of instructions in the program minus the current progress in terms of the number of instructions. In a program consisting two processes discussed above, we need to know what is the total number of instructions in each process in order to know what is the number of remaining instructions to be executed. Otherwise, it is not possible to know which one is critical.

It is more difficult to define a measure of progress in a distributed program. We can consider a communication primitive or a block of sequential instructions

in a process as an event, and use Lamport's logical clock [Lam78] as a measure of progress to detect a critical process. We can also use virtual time which Chen [Che91] proposed as a measure of progress in a concurrent program; each event in a process is assigned the virtual time which the event should occur when it is executed on an imaginary architecture, called a virtual architecture.

In general, we need knowledge of a distributed program to assign virtual time values to events well. But the issue of what is a good way to do so is beyond the scope of our study. Virtual time compilers may insert some codes in each process to maintain their virtual times [Che91]. The progress of processes can be quantified and compared based on their virtual times. In the context of virtual time, we can define a critical process as the process with the smallest virtual time at any given time.

Our competition protocols could be adequate in Time Warp [Jef85] because virtual time is available and processes are deterministic in Time Warp. However, our competition mechanisms do not apply directly to Time Warp, because of the difference in the programming model of Time Warp from our model and because of optimistic synchronization in Time Warp. We would have to modify them to deal with antimessages and antimessage replication. But this issue is beyond the scope of our study.

It is not required that a progress indicator be perfect. Approximate progress indicators may degrade performance of competition protocols, but do not violate their correctness. We can tolerate setting a clone that is ahead back to the pace of a clone that is behind, if we do so rarely. Then the net result is speedup. In the parallel context, we may be just guessing as to which process is ahead and which is behind. But guessing has to do with which is the critical processes not

which of clones are ahead and which is behind.

3.4 Definition of Speedup for Competition

There are various kinds of speedup metrics, e.g., the speedup in Amdahl's law [Amd67], Gustafson's scaled speedup [Gus88], and memory-bounded speedup [SN93]. However, we cannot use any of those definitions because competition protocols need more processors than noncompeting cases and thus competition protocols cannot execute a distributed program in the same condition as non-competing cases can.

We are going to use the following statistical definition of speedup; consider a set of physically identical processors where the arrival of foreground processes at all processors forms a Poisson process with the same rate, and their execution times have a distribution with the same parameter. Then the mean foreground processor utilization ρ is identical at all processors, and thus the mean background execution speed at all processors is also identical. However, background execution speeds may fluctuate differently on processors over time.

For the sake of analysis, we assume that the underlying statistics for foreground processes on processors do not change for the duration of the entire background computation. Assume that we run a distributed program on K such processors in the background without competition, and that we run the program in the background with competition on N ($N > K$) such processors with the same mean foreground processor utilization ρ. Let $T_{nc}(K)$ and $T_c(N)$ denote the mean completion time of the program without competition and that with

competition respectively. We define the speedup of competition, $SP(K, N, \rho)$, as

$$SP(K, N, \rho) = \frac{T_{nc}(K, \rho)}{T_c(N, \rho)}. \tag{3.1}$$

The speedup of competition, $SP(K, N, \rho)$, is the ratio of the mean completion time without competition on K processors to that with competition on N processors under the assumption that the behavior of foreground processes on all processors is statistically identical. The new speedup metric shows the performance benefit of competition over noncompetition.

When foreground processes on all processors start and go idle at the same real time for the entire computation of a distributed program, competition protocols cannot reduce the completion time of the application at all. In this case the execution speed of processors for background processes changes in a lock step. Background applications only achieve speedup from competition when idle periods on processors occur at different times, and thus some processors are faster at some times while others are faster at other times. The performance of competition depends on the extent to which idle periods on processors are disjoint.

3.5 Maximum Progress among Peer Clones

Let us consider $f_1(t)$ and $f_2(t)$ in Figure 3.1 that shows the execution speed of two processors P_1 and P_2 respectively. A dotted line in Figure 3.1 shows the maximum possible execution speed when processors are completely idle. It is shown that P_2 has a greater execution speed than P_1 during the period from a to b, but the reverse is true at other times.

Assume that clones sp_1 and sp_2 of a process sp run on processors P_1 and P_2. If $f_1(t)$ and $f_2(t)$ change dynamically but $f_1(t)$ is always smaller than $f_2(t)$,

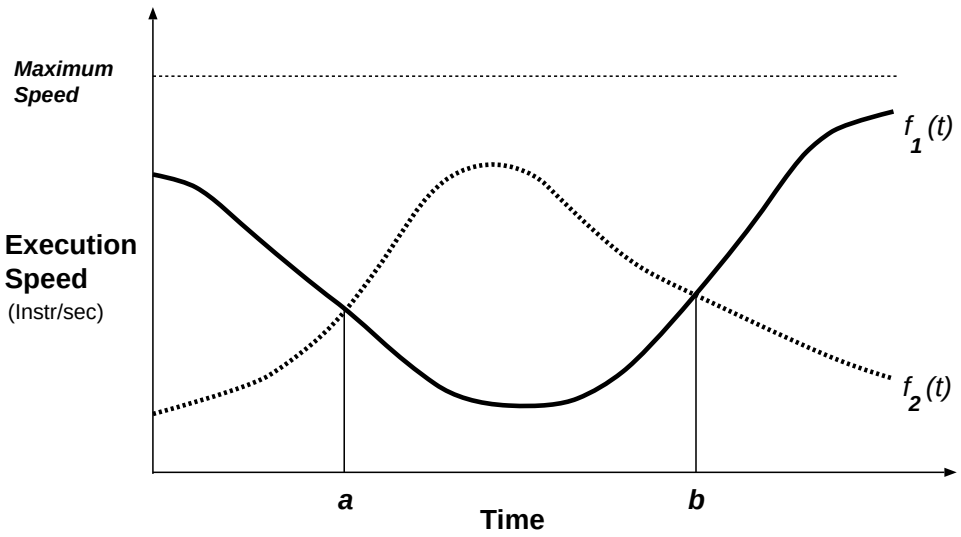


Figure 3.1: Execution Speed of Processors P_1 and P_2

then competition offers no advantage because sp_2 will be always ahead of sp_1. When the difference between $f_1(t)$ and $f_2(t)$ is always sufficiently small, it is not useful to keep two clones sp_1 and sp_2, either. In this study, we focus a SNOW environment where $f_1(t)$ and $f_2(t)$ change dynamically so that $f_1(t) > f_2(t)$ at some times and $f_1(t) < f_2(t)$ at others.

Let $g_i(t) = \int_0^t f_i(x)dx$ for i=1 or 2. Then $g_i(T)$ represents the progress that is the total number of instructions executed by a clone sp_i up to real time T. The progress for sp_1 and sp_2 is shown by $g_1(t)$ and $g_2(t)$ in Figure 3.2, which is drawn based on Figure 3.1. If $g_1(t)$ is always equal to or smaller or larger than $g_2(t)$, competition offers no advantage. This means that $g_1(t)$ and $g_2(t)$ must cross each other over time as in Figure 3.2 in order to have competition benefit. For a given time T, the competition benefit increases when

$$\int_0^T |g_1(x) - g_2(x)|dx$$

increases.

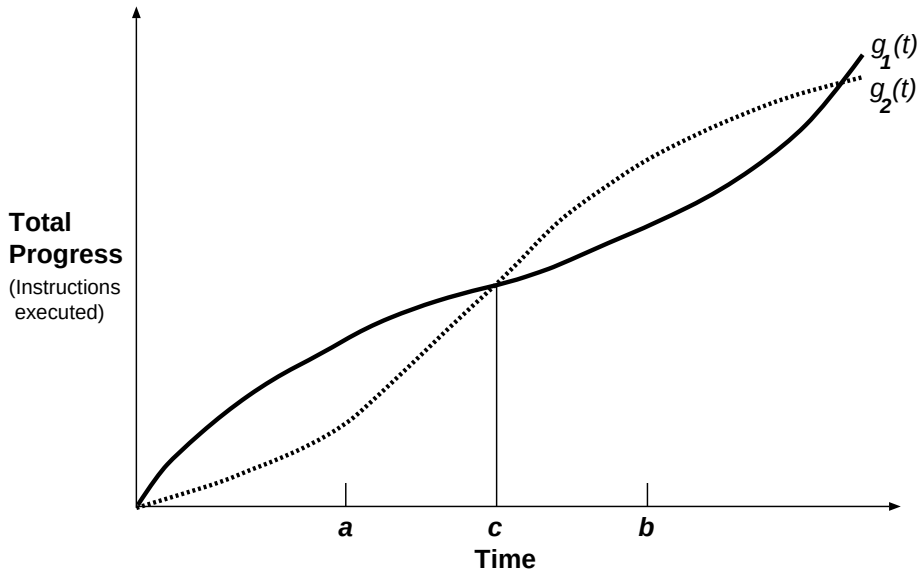


Figure 3.2: Progress of Clones sp_1 and sp_2

Let us assume ideal conditions; there is no delay in process switching, no communication latency, and no state propagation latency. We also assume that competition protocols can detect which processor is the fastest among the processors. Under these ideal conditions, competition protocols can achieve at all times the maximum execution speed of all processors, which is $\max(f_1(t), f_2(t))$ in Figure 3.1, and thus the maximum progress of all clones, which is $G_{max}(t) = \max(g_1(t), g_2(t))$ in Figure 3.2.

3.6 A Propagating Competition Protocol

It is desirable that competition protocols exploit the execution speed of P_2 from a to b because $f_1(x) < f_2(x)$ for $a < x < b$ as shown in Figure 3.1. Without state propagation, Figure 3.2 shows that competition protocols may not exploit the faster execution speed of P_2 until real time c because sp_2 needs real time $c - a$ to go ahead of sp_1. The time $c - a$ is called a *catching up* latency. We

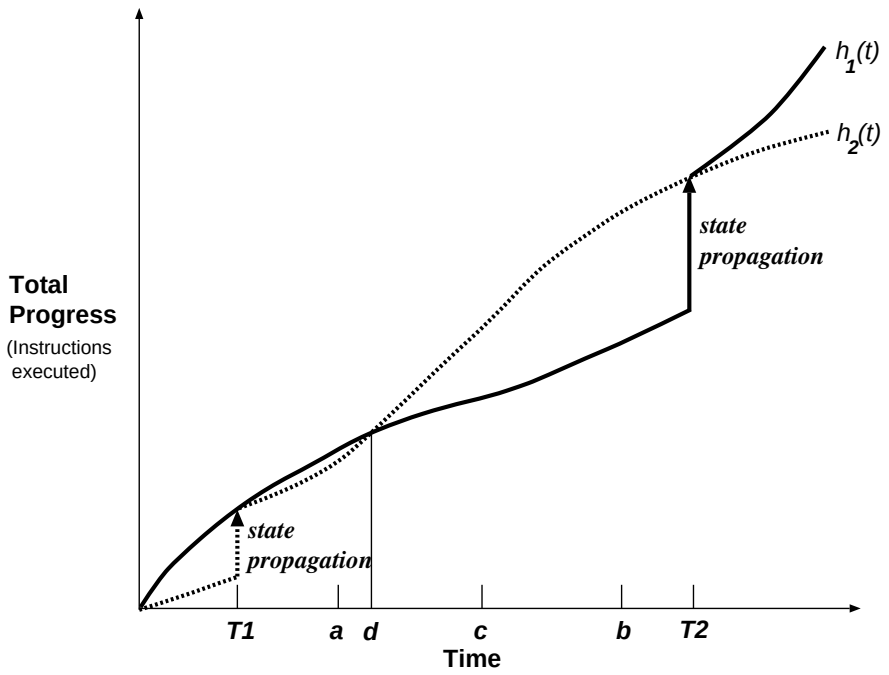


Figure 3.3: State Propagation between Clones sp_1 and sp_2

can reduce the catching up latency by propagating a state from sp_1 to sp_2 when the state propagation takes less time than the catching up latency. We refer to a competition protocol with state propagation and that without state propagation as a *propagating* competition protocol and a *nonpropagating* competition protocol respectively. But we consider a propagating competition protocol as a standard competition protocol in this study.

Assuming that state propagation is done at $T1$ and $T2$ instantaneously, Figure 3.3 shows that sp_2 jumps forward at $T1$ based on sp_1's state and then sp_2 can go ahead of sp_1 at real time d where $a < d < c$. In an ideal situation in a propagating competition protocol, the progress of the process can be at all times the maximum progress of all clones, which is $H_{max}(t) = \max(h_1(t), h_2(t))$.

Optimal points for state propagation are a and b rather than $T1$ and $T2$

because Figure 3.1 shows that $f_1(x) < f_2(x)$ for $a < x < b$ and $f_1(x) > f_2(x)$ otherwise. In general the optimal points are the time when the fastest processor changes among processors. However, if processors are determined to be idle when the CPU utilization of foreground processes is smaller than some threshold, there is no way to know the optimal points at any time.

If there is a clone that is far ahead but running on a slow processor and there is another clone that is far behind but running one a fast processor, it is desirable to propagate the state from the clone ahead to one that is behind. The purpose of state propagation is to minimize the catching up latency and thus make the clone that is most ahead run on the fastest processor. When the state propagation latency is larger than the catching up latency, however, state propagation gives no advantage. Competition protocols should propagate states infrequently when the state propagation latency is sufficiently large. In this case another processor may become faster than the currently fastest processor before the end of state propagation.

There are two heuristic decisions we must make in competition protocols; first, which processes are critical and should be cloned or uncloned. Second, which clones of a process is ahead or behind for the purpose of state propagation. The overhead of competition protocols should be small enough so that the cost of keeping multiple clones is offset by the advantage of competition protocols, or it is not worth doing.

3.7 Performance Comparison of Competition Protocols

Figure 3.4 shows three performance curves for competition protocols assuming zero overhead for state propagation: $O_{max}(t)$ of a propagating competition pro-

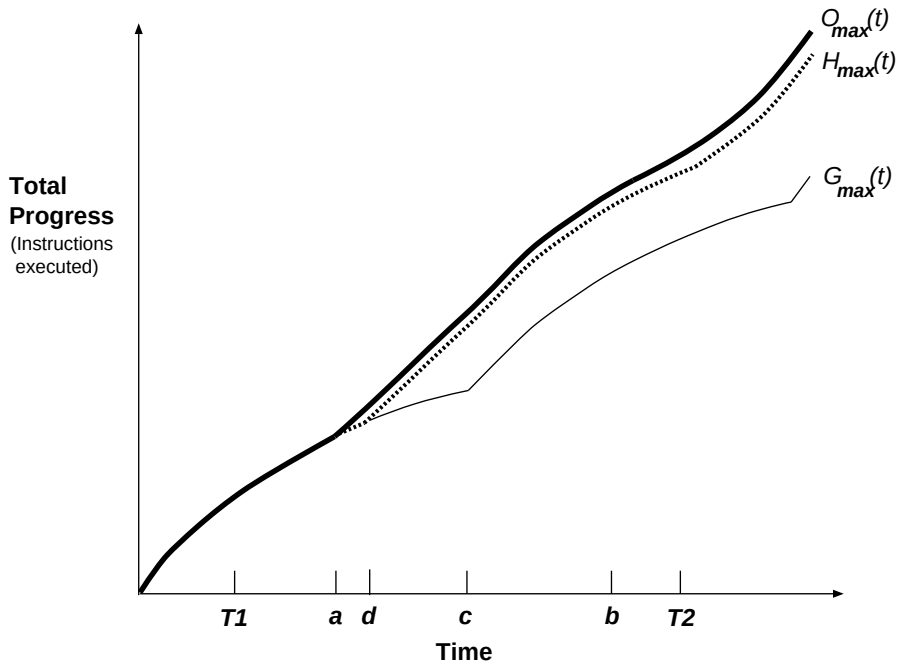


Figure 3.4: Performance Comparison of Three Competition Protocols

tocol with optimal state propagation, $H_{max}(t)$ of a propagating competition protocol with suboptimal state propagation, and $G_{max}(t)$ of a nonpropagating competition protocol. $G_{max}(t) = \max(g_1(t), g_2(t))$ is redrawn from Figure 3.2, $H_{max}(t) = \max(h_1(t), h_2(t))$ is redrawn from Figure 3.3, and $O_{max}(t)$ is drawn assuming that state propagations are done at optimal points a and b without the overhead of state propagation.

The performance of the propagating competition protocol with suboptimal state propagation, $H_{max}(t)$, is always at least as good as that of the nonpropagating competition protocol, $G_{max}(t)$. Furthermore, $H_{max}(t)$ can outperform $G_{max}(t)$ tremendously because the gaps between $H_{max}(t)$ and $G_{max}(t)$ increases over time. Obviously, $O_{max}(t)$ outperforms $H_{max}(t)$ because $O_{max}(t)$ propagates states at optimal points.

The concept of state propagation plays a key role in competition protocols, and the overhead of state propagation can be justified when it is small compared with the total processing time. In some special cases where the state of a process does not change after it processes messages, e.g., a process receives a message, calculates some function based on the message, and return the result, state propagation can be optimized.

3.8 Comparison of Competition and Migration

3.8.1 The Advantage of Competition over Migration

Migration requires that we predict the future; we must know which processor(s) will be faster in the near future. This requires measuring actual processor speed over a period of time. We must decide the time quantum for this measurement, and the measurement causes delay by at least one quantum. There must be an implicit assumption that the near future performance will be similar to that of the recent past over which the measurement was taken.

However, competition does not require any knowledge of the future. This requires measuring only which clone is ahead or behind (and how much) with instruction counter or CPU time clock. But this is an instantaneous measurement, and no delay is injected. More importantly, there need be no assumption that the future resembles the past on any time scale.

3.8.2 The Advantage of Migration over Competition

When future performance is predictable, then migration can use one processor, when competition would use more than one; in the extreme case of perfect knowl-

edge of which processor is fastest all the times, zero-overhead migration is optimal, and uses only one processor, whereas a competition protocol with zero-overhead state propagation is no faster, but takes more than one processor.

Message communication is simpler with migrating processes than with cloned processes.

3.9 When Can We Use Competition?

Whenever we make decisions in scheduling or resource allocation with incomplete information (about precedence constraints, the execution times, or the future resource demands) but with a large excess of resources, competition protocols may improve performance. We may get significant benefits by competition whenever the performance of clones varies widely for any reason. We can envision the following informal scenarios that satisfying the conditions favoring competition.

1. (Interference from Foreground Applications)

   Any process in a distributed program can become critical on a SNOW because of foreground processes. Competition protocols can speed up critical processes by cloning them and executing their clones on different processors in parallel. Competition protocols can also speed up a single sequential program.

2. (Excessive Parallel Capacity)

   The number of available processors in the SNOW is often larger than the number of processes in a distributed program. But there is no way to use additional processors without competition.

3. (Heterogeneous Performance Systems)

   Processors with binary compatibility in a system may have different processing power, memory space, disk storage, and so on. In general it is difficult to find the best configuration to execute a distributed program in this heterogeneous performance system. Competition protocols can try multiple alternative configurations concurrently.

In summary, when there are multiple alternative processors to execute a process and one processor is the fastest at one time but another processor is the fastest at another for any reason, competition protocols can try all of alternative processors in parallel for exploiting the fastest result. Competition protocols exploit excess resources which are otherwise wasted.

If it is known in advance that one processor is always faster than the others, competition protocols should not be used due to competition overheads. However, we believe that such information won't usually be available at runtime in distributed computing platforms, especially in SNOWs. Therefore, competition protocols can be a very useful operating system technique that can be applicable to many different cases to improve the performance of background distributed applications. Competition protocols can be applicable to both a DNOW and a SNOW statically or dynamically depending on the goal of the system.

CHAPTER 4

Competition Mechanisms

There are two issues in the implementation of competition protocols: competition policy and competition mechanism. In this study, we do not study competing policy issues; e.g., (1) what is the optimal number of competing clones for a process?; and (2) when and where to increase or decrease the number of competing clones? We consider simple policies only when they are necessary.

We will focus on competition mechanisms; a competition mechanism deals with message multicasting and state propagation among peer clones in addition to message communication between clones of communicating processes. A competition mechanism creates or destroys clones of a process depending on the requests of a competition policy. We assume that a competition mechanism interacts with a competition policy over a well defined interface.

4.1 System Architecture

A SNOW system using competition consists of 3 runtime software layers, the *Application* layer, the *Competition* layer, and the *Kernel* layer as in Figure 4.1. The application layer consists of user defined processes which communicate each other by messages to achieve a common task.

The competition layer consists of two components; competition policy and

layer	*functions*
Application	Communicating processes Application messages
Competition	Message Multicast State Propagation Clone creation and destruction
Kernel	Clone scheduling Message communication Progress measurement

Figure 4.1: System Software Layers

competition mechanism. The competition policy component collects performance measurement data and makes decisions about state propagation among peer clones, the number of clones of processes, the placement of clones on processors. In this study we focus on competition mechanisms. A process can be represented by any positive number of process clones at the competition layer. Similarly, a message can be represented by any positive number of message clones at runtime. We assume that process clones can run on any processor in a SNOW.

We present a competition mechanism which is responsible for message multicast among peer clones of communicating processes, and state propagation among peer clones of a process in addition to message communication between clones of processes. Since process clones communicate only via messages, the competition mechanisms are primarily contained in the implementation of the **send** and **receive** primitives in addition to state propagation. The creation and destruction of clones would be implemented in this layer, but we do not consider dynamic cloning and uncloning in this study.

The lowest level is the kernel that schedules foreground processes and back-

ground process clones and provides the basic message communication services used by competition mechanisms. We assume that the kernel schedules foreground processes with high priority; background process clones are preempted immediately whenever foreground processes are ready to execute. Thus background process clones exploits the otherwise idle CPU cycles.

We assume that the underlying message passing system provides asynchronous communication and that communication delays are finite but unpredictable. We also assume that messages are received reliably in the order they are sent between two processors, and they are not lost or duplicated at the kernel layer.

4.1.1 Internal Representation of a Process Clone

We assume that a process clone is represented by three components; two message queues and a state. Message queues consists of an input message queue to hold incoming messages, and an output message queue to hold copies of any outgoing messages that the process clone or its peer clones may send. A state includes an operating system structure that keeps track of the current activities of the system and holds pointers to the code necessary to run the process clone.

4.2 Competition Mechanisms

We consider message communication between process clones of communicating processes in this section. We make the following assumptions in competition mechanisms:

1. Each processor communicates one another by messages, and message communication between processors is FIFO and reliable.

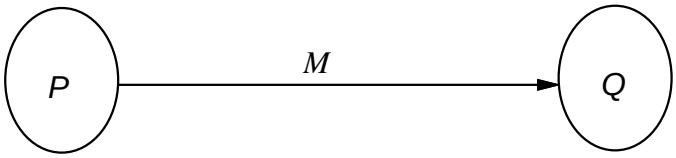


Figure 4.2: A Message M from a Process P to another Process Q

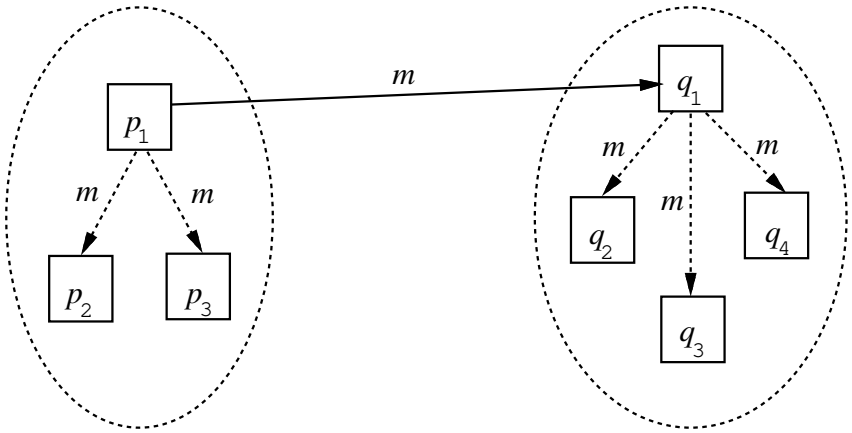


Figure 4.3: Message Communication in a Competition Mechanism

2. Each process is deterministic and hence its process clones are also deterministic.

3. Peer process clones do not run on the same processor, but they can run on any processor in a SNOW.

We consider two processes P and Q at the application layer as shown in Figure 4.2 where P sends a message M to Q. Let us assume that processes P and Q are represented by three clones and four clones respectively at the competition layer as in Figure 4.3. Clones p_1, p_2, and p_3 are peer clones of P, and clones q_1, q_1, q_3, and q_4 are peer clones of Q. A solid arrow represents message delivery between clones of communicating processes, and dotted arrows represent message delivery (called *multicast*) among peer clones.

All process clones of a process execute the same code. Competition mechanisms implement **send** and **receive** primitives so that the collective behavior of process clones of a process is logically identical to the behavior of a single clone of the process, and thus no executing process or clones can detect the cloning of any process.

4.2.1 Implementation of a Competition Mechanism

We now present a pseudocode implementation of a competition mechanism. We assume that a message clone at the competition layer has four more fields in addition to its message at the application layer as shown below;

```
Content of a message clone at the competition layer
    msg         /* message at the application layer */
    sender      /* sender of message */
    receiver    /* receiver of message */
    type        /* unicast or multicast */
    direction   /* input or output message queue */
```

When **send**(message, receiver) or **receive**(message, sender) is executed by a process clone, the following pseudocode is executed at the competition layer;

```
Procedure SEND(msg, receiver)
<1>  Msg.msg = msg;
<2>  Msg.sender = sender;   /* Assume that sender is known */
<3>  Msg.receiver = receiver;
<4>  If (msg is in the output queue of the sender)
<5>     do nothing;   /* message sending is suppressed */
<6>  else {
<7>     Msg.type = unicast;
<8>     Msg.direction = input;
<9>     UNICAST Msg to one clone of receiver;
<10>    Msg.type = multicast;
<11>    Msg.direction = output;
```

```
<12>    MULTICAST Msg to the other peer clones of sender;
<13> }

Procedure RECEIVE(msg, sender)
<1>  If (a message Msg from the sender is in the input queue)
<2>     msg = Msg.msg;
<3>  else
<4>     block receiver;

Procedure MESSAGE_ARRIVAL(Msg)   /* When a message arrives */
<1>  msg = Msg.msg;
<2>  If (Msg.direction == input) {
<3>     if (msg is in input queue of Msg.receiver)
<4>        do nothing;   /* its peer msg clone already arrived */
<5>     else {
<6>        ENQUEUE msg into the input queue of Msg.receiver;
<7>        if (Msg.type == unicast) {
<8>           Msg.type = multicast;
<9>           Msg.direction = input;
<10>          MULTICAST Msg to peer clones of Msg.receiver;
<11>       }
<12>    }
<13  }
<14> else {   /* output queue */
<15>    if (msg is in the output queue of Msg.sender)
<16>       do nothing;   /* its peer message clone already sent */
<17>    else
<18>       ENQUEUE msg into the output queue of Msg.sender;
<19> }
```

Now we will explain how a competition mechanism works based on Figure 4.3. Assume that a clone p_1 requests to send a message clone m of a message M to a process Q earlier than clones p_2 or p_3. When p_1 requests to send m, a competition mechanism first checks whether p_1's peer clones have already sent m in the line 4 of procedure SEND. If so, it just ignores that request, and it is said that message sending is *suppressed*. Otherwise, it unicasts m to one (say q_1) of Q's clones in

the line 9 of procedure SEND and multicasts m to the other peer clones of p_1 in the line 12 of procedure SEND.

When m arrives at q_1, the competition mechanism enqueues m into the input message queue in the line 6 of procedure MESSAGE_ARRIVAL (assuming that m's peer message clone has never arrived at q_1) and multicasts m to the other peer clones of q_1 in the line 10 of procedure MESSAGE_ARRIVAL because m is a unicast message clone. When a multicast message clone of m arrives at p_1's peer clones, m is enqueued into the output message queue if they have not sent m yet. This is done in lines 15 through 18 of procedure MESSAGE_ARRIVAL.

When a peer clone (say p_3) of p_1 requests to send m *concurrently*, that is, p_3 requests to send m before m that p_1 did broadcast is enqueued in its output message queue, the competition mechanism will deliver m to one of Q's clones. Thus multiple peer message clones may be delivered to receiving process clones, but they must be detected so that one and only one message clone should be received and processed by each receiving process clone to preserve *exactly once* semantics of distributed programs.

In order to detect the arrival of redundant peer message clones at a receiving clone, a message clone should not be removed from its input message queue immediately even after a process clone received and processed it. In order to detect sending redundant peer message clones from sending process clones, copies of message clones are saved in the output queues of sending process clones whenever one message clone is sent by one of sending process clones. The line 4 of procedure SEND checks whether a message clone has been sent or not. When all process clones have produced a message clone, it can be removed from the input message queues of its receiving process clones and from the output message

queues of its sending process clones.

4.2.2 Message Counters

In order to remove message clones from their input message queue immediately when they are received, we can maintain two counters for each pair of a sending process (say P) and a receiving processes (say Q); one (called $sent$) at each clone of P to record how many message clones have been sent and the other (called $arrived$) at each clone of Q to record how many message clones have arrived. A message clone that has arrived can be received by its receiving clone at any time later. Initially $sent = 0$ and $arrived = 0$. Values of $sent$ at clones of P and values of $arrived$ at clones of Q can differ at any given time.

When a clone of P sends a message clone m to a clone of Q earlier than the other clones of P, $sent$ at the clone is increased by one. Then the message clone m carries its sending sequence number so that receiving clones can detect whether the message clone m has arrived or not; if $arrived$ of a receiving clone is smaller than m's sending sequence number, m has never arrived in the past and thus m is enqueued in the input message queue. Otherwise, it will be ignored.

The difference of $sent$ at a sending clone of P and $arrived$ at a receiving clone of Q at any given time shows the number of message clones that have been sent from the point of view of the sending clone but never arrived from the point of view of the receiving clone. When a message clone arrives, its corresponding counter $arrived$ at the receiving clone has to be increased by one if the message clone has never arrived before.

Now we do not need to maintain a real output message queue at all. We just need to maintain one more counter $local_sent$ in addition to $sent$ for each

sending clone where *sent* represents how many message clones have been sent by all process clones and *local_sent* represents how many message clone has been produced by the local clone. The following relation is always satisfied; $sent \geq local_sent$. If $sent > local_sent$ at a process clone, the clone is behind some of its peer clone.

We need to modify the pseudocode described above in order to include the concept of message counters. We add asterisks after line numbers in the pseudocode below if the lines are new or modified.

```
Content of a message clone at the competition layer
    msg         /* message at the application layer */
    seq         /* message sending sequence number */
    sender      /* sender of message */
    receiver    /* receiver of message */
    type        /* unicast or multicast */
    direction   /* input or output message queue */

Procedure SEND(msg, receiver)
<1*> local_sent = local_sent + 1;
<2>  Msg.msg = msg;
<3>  Msg.sender = sender;   /* Sender is known */
<4>  Msg.receiver = receiver;
<5*> If (local_sent <= sent)
<6>     do nothing;   /* message sending is suppressed */
<7>  else {
<8*>    sent = local_sent;
<9>     Msg.type = unicast;
<10>    Msg.direction = input;
<11*>   Msg.seq = local_sent;
<12>    UNICAST Msg to one clone of receiver;
<13>    Msg.type = multicast;
<14>    Msg.direction = output;
<15>    MULTICAST Msg to all peer clones of sender;
<16> }

Procedure MESSAGE_ARRIVAL(Msg)   /* When a message arrives */
```

```
<1>  msg = Msg.msg;
<2*> seq = Msg.seq;
<3>  If (Msg.direction == input) {
<4*>    if (seq <= arrived)
<5>       do nothing;   /* its peer msg clone already arrived */
<6>    else {
<7*>       arrived = arrived + 1;
<8>       ENQUEUE msg into the input queue of Msg.receiver;
<9>       if (Msg.type == unicast) {
<10>         Msg.type = multicast;
<11>         Msg.direction = input;
<12>         MULTICAST Msg to peer clones of Msg.receiver;
<13>      }
<14>   }
<15  }
<16> else {   /* output queue */
<17*>   if (seq <= sent)
<18>      do nothing;   /* its peer message clone already sent */
<19>   else
<20*>     sent = seq;
<21> }
```

4.2.3 State Propagation

We assume that the set of message counters, one for each communicating processes, is a part of a state, and that the sending sequence number of each message clone in the input message queue is accessible. If we propagate a state of a process clone that is most ahead, we may have to remove message clones in the input message queue at the destination whose sending sequence number of a message clone is equal to or smaller than corresponding *arrived* in the propagated state. However, if we propagate a state of a process clone that is behind, we may have to recover some of messages clones because they may have already been received by the clone that is ahead.

We can consider different competition mechanisms depending on how to deliver a message clone to receiving process clones and on whether state propagation is allowed or not. The mechanism in Figure 4.3 delivers one message clone to a single receiving clone. However, another mechanism may deliver one message clone to all receiving clones, and message multicasting among receiving clones is not necessary. However, a sending clone may have to multicast a message clone to its peer clones for suppressing sending redundant message clones.

Peer clones of a process in a SNOW environment are distinguished from any other because it may not be known that which clone will be ahead at any given time. This is the reason why a message clone must be delivered to all clones of its receiving process. In a SNOW environment with multicasting capability, only one multicast is sufficient to deliver a message clone to every process clone in both of its receiver and its sender.

If a clone of a process is most of time behind the other peer clones, competition protocols should kill the clone behind and perhaps create another clone on another processor. In this case competition protocols may degenerate into a form of migration because we can consider that the killed clone is incarnated on another processor.

4.3 Communication between Process Clones

4.3.1 Group Communication

A number of protocols related to group communication has been proposed by researchers. One of major issues is how to order messages among receiving clones for maintaining their consistency. Some protocols provide total ordering [CM84,

BSS91], and some protocols do causal ordering [BSS91].

Most of protocols use a coordinator to sequence messages among receiving clones [CM84, KT91, EZ92, CC94]. However, competition protocols do not use the concept of a coordinator when each communicating process names each other as in the model of our distributed programs in Chapter 3. We can easily employ the concept of a coordinator in competition protocols to sequence messages in the other models of distributed programs.

Assume that the number of clones of a sending process and that of a receiving process are M and N respectively. Let us consider the following four methods for message communication;

- one-to-many [Coo85]

  Each clone of a sending process sends a message clone to all receiving clones as shown in Figure 4.4. The Circus system used this method for improving fault tolerance [Coo85]. But this method always requires the quadratic $(M \cdot N)$ number of messages in a point-to-point network and the linear (M) number of messages in a broadcasting network.

- one-to-many with a coordinator [CC94]

  Before a clone of a sending process sends a message clone to all receiving clones, it has to get permission for sending from a coordinator. Only the process clone that makes a request for permission faster than the other peer clones, is allowed to send the message clone.

  This method requires $N + 1$ in the best case and $M + N$ messages in the worst case in a point-to-point network, and requires 2 messages in the best case and $M + 1$ messages in the worst case in a broadcasting network.

- one-to-one and multicasting among peer clones

  A clone of a sending process sends (unicasts) a message clone to one of receiving clones if none of its peer process clone has sent the message clone. If so, the clone also multicasts the message clone among its peer process clones. Otherwise, message sending is suppressed. When a unicast message clone arrive at a receiving clone, the receiving clone multicasts the message clone among its peer process clones if the message clone has never arrived.

  In a point-to-point network, this method requires $M + N - 1$ messages in the best case but $(M - 1)M + M + (N - 1)N$ in the worst case. This is the method we have proposed for competition protocols, and is shown in Figure 4.5. This method requires 3 messages in a broadcasting network when no concurrent sending occurs.

- one-to-all

  Each clone of a sending process multicasts a message clone to all receiving process clones as well as its peer process clones. In a point-to-point network, this method requires $M-1+N$ messages in the best case but $M(M-1+N)$ in the worst case. In a broadcasting network, this method requires only one message when no concurrent sending occurs, but M in the worst case. This is shown in Figure 4.6.

  When a message clone arrives at a process clone, we have to distinguish whether the process clone is a sending process clone or a receiving process clone, and then it is straight forward to use the pseudocode implementation of a competition mechanism.

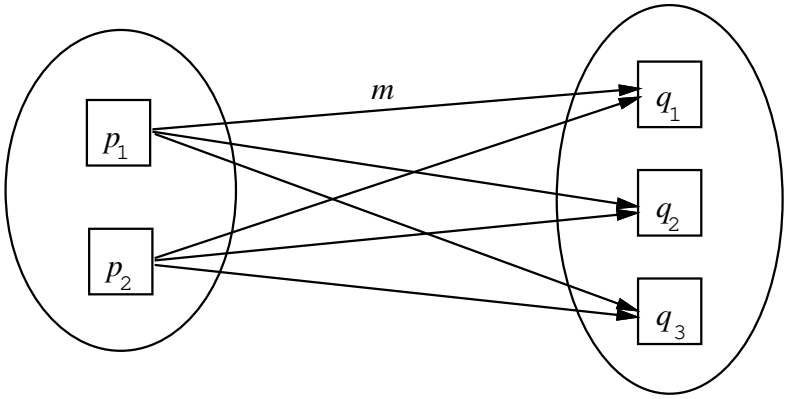


Figure 4.4: One-to-many Communication

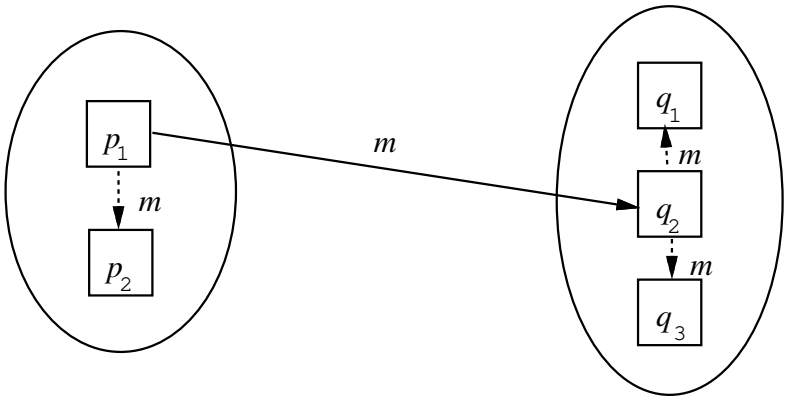


Figure 4.5: One-to-one and Multicasting among Peer Clones

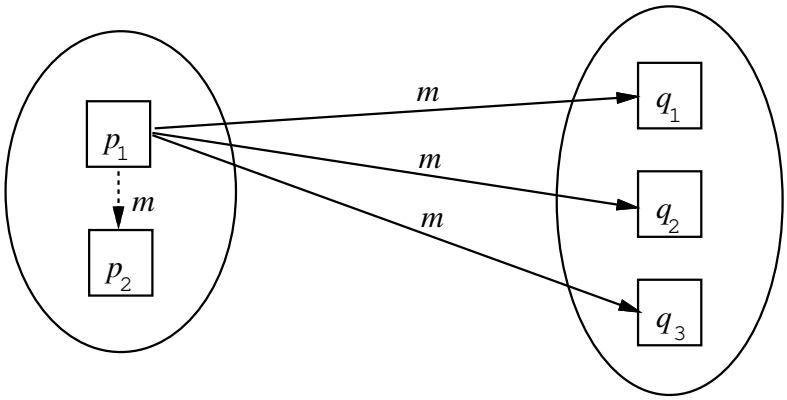


Figure 4.6: One-to-all Communication

4.4 Correctness of Competition Protocols

A replication algorithm is correct if it ensures the following; For each process, all clones of the process receive and execute the same sequence of messages [Sch90]. Our competition mechanism guarantees that process clones of a process receive message clones in the same sequence without omission or duplication because of the sending sequence number of message clones and the FIFO assumption between communicating processors. Thus nonpropagating competition protocols are correct.

Now let us consider propagating competition protocols. When a state of the process clone that is ahead is propagated to a process clone that is behind, message clones can be received more than once because the input message queue at the destination may still have message clones that have already been received by the process clone that is ahead. However, our competition mechanism can easily detect this case by checking whether the sending sequence number of a message clone is greater than its corresponding *received*.

When a state of the process clone that is behind is propagated to a process clone that is ahead, message clones can be omitted because the process clone at the destination has already received message clones that must be received by the process clone that is behind. Our competition mechanism can detect this case by checking whether the sending sequence number of a message clone is greater than its corresponding *received* by more than one or not. In this case some message clones must be recovered from other process clones,

CHAPTER 5

Analytic Study of Competition for a Sequential Process

5.1 Introduction

In this chapter we consider only a *sequential* process called *sp*, which is a special case of a distributed program. We study two extreme competition protocols; a nonpropagating competition protocol and a propagating competition protocol that propagates states instantaneously at optimal times. The latter is called an *ideal* competition protocol.

The performance of competition protocols will fall somewhere in between that of the nonpropagating competition protocol and that of the ideal competition protocol. The goal of competition protocols is to execute *sp* as quickly as possible on a SNOW environment without interfering with foreground processes.

We are going to derive the closed form solution for the performance of the ideal competition protocol. This is the best performance we can achieve from competition under ideal conditions, e.g., state propagation decision without any overhead and instantaneous state propagation.

5.1.1 Assumptions and Terminology

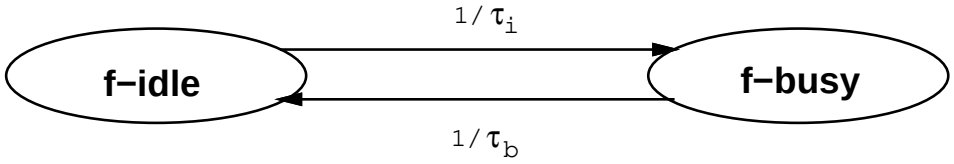


Figure 5.1: Two State Model of a Processor

When a processor executes a foreground process, we say that it is *f-busy*. Otherwise, it is *f-idle*. We assume that processors alternate between f-busy and f-idle states at constant rates as in Figure 5.1. Each processor is f-busy a fraction $p_b = \tau_b/(\tau_i + \tau_b)$ of time, and f-idle the remaining fraction $p_i = \tau_i/(\tau_i + \tau_b)$ of time. The probability that k out of M processors are f-idle at any given time is

$$P[N = k] = \begin{pmatrix} M \\ \\ k \end{pmatrix} (p_i)^k (p_b)^{M-k}$$

which is a binomial density.

The clones of *sp* make their progress depending on the arrival of foreground processes at their hosting processors. We assume that the lengths of f-busy periods at all processors are independent and identically distributed (i.i.d.) random variables from an exponential distribution $B(t)$, with mean τ_b, and corresponding density $b(t)$. Then $B(t)$ and $b(t)$ are given respectively by:

$$B(t) = 1 - e^{-t/\tau_b}, \quad t \geq 0 \tag{5.1}$$

$$b(t) = \frac{1}{\tau_b} e^{-t/\tau_b}, \quad t \geq 0 \tag{5.2}$$

Likewise, the lengths of f-idle periods at all processors are i.i.d. random variables from an exponential distribution $I(t)$, with mean τ_i, and density $i(t)$. We assume

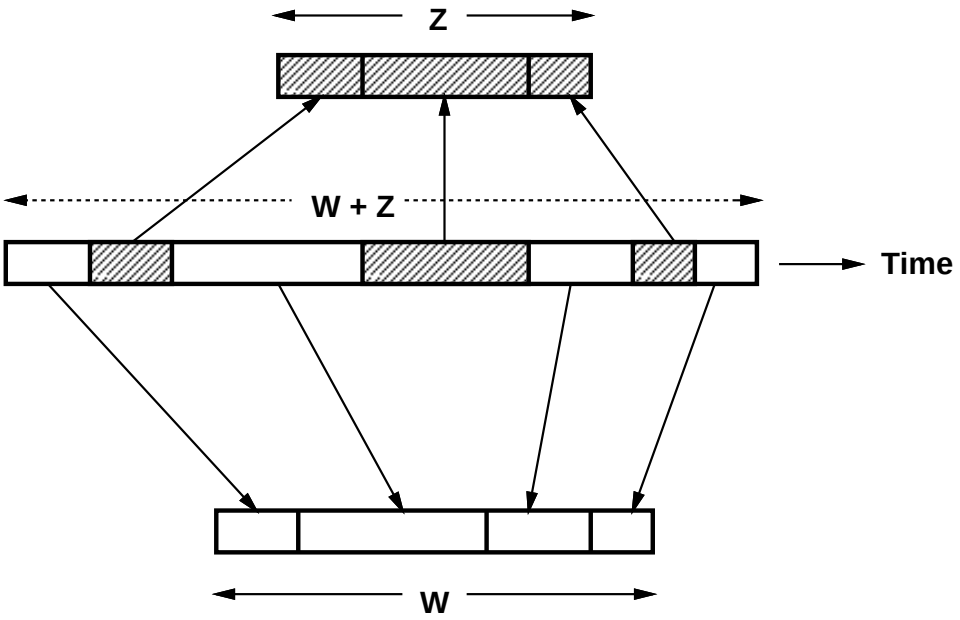


Figure 5.2: Execution Time on a Variable-speed Processor

that all processors are physically identical. Let W_Q^{sp} denote the constant amount of time to finish sp on a dedicated processor Q.

5.2 Stretching Ratio

A time line of execution of sp on a variable-speed processor Q is shown in Figure 5.2. Shadowed regions indicate f-busy periods and white regions indicate f-idle periods when sp can be executed. If sp starts to run during a f-idle period, as shown in the middle of Figure 5.2, sp will finish at time $W_Q^{sp} + Z_Q^{sp}$, where Z_Q^{sp} is the additional time sp spends in the processor because of preempting f-busy periods. That is, Z_Q^{sp} is the sum of f-busy periods that occur while sp runs on Q. When there is no ambiguity, W_Q^{sp} and Z_Q^{sp} will be denoted by W and Z respectively for simplicity.

We can analyze the total length of the f-busy periods during the time $W + Z$ by examining the arrival process of these periods because *sp* finishes in a f-idle period and none of those f-busy periods are truncated. The arrival process for f-busy periods stops during these f-busy periods in the two state model of a processor. The lengths of such f-busy periods does not affect the rate at which they arrive, and thus we may take f-busy periods to have zero length. Therefore the number of real f-busy periods arriving in $W + Z$ is the same as the number of zero-length f-busy periods arriving in W [Kor89].

We let $p(k|W)$ be the probability density that k zero-length f-busy periods arrive in W. Since we assume that the length of f-idle periods is exponentially distributed, f-busy periods arrive according to a Poisson process with rate $1/\tau_i$. Thus

$$p(k|W) = \frac{(W/\tau_i)^k}{k!} e^{-(W/\tau_i)}, k = 0, 1, 2, \cdots \tag{5.3}$$

which has mean and variance both equal to W/τ_i.

The completion time of *sp* in Figure 5.2 is stretched from W to $W + Z$ because of preempting foreground processes. Now we define a *stretching ratio* on a variable-speed processor as follows:

Definition 5.1 *Let W be a finishing time of a sequential process on a dedicated processor and let T denote a finishing time of the process on a variable-speed processor. The* **stretching ratio** *SR of the variable-speed processor is defined as $SR = T/W$.*

The stretching ratio SR shows how many times the completion time of *sp* on a variable-speed processor is stretched due to foreground processes. SR may depend on W or the time when *sp* starts to execute, but we will often ignore these

factors when we consider long running applications. We assume that the mean stretching ratio $\overline{SR}$ does not change for the entire computation of applications.

The stretching ratio SR is a non-negative real number greater than one because $T \geq W$. Since $T = W + Z$ as shown in Figure 5.2,

$$SR = T/W = (W + Z)/W = 1 + Z/W.$$

Z/W is the ratio of CPU cycles used for foreground processes to those used for a background process. We call the ratio Z/W an *busy ratio*, which is denoted by η. Then $SR = 1 + \eta$. Thus a processor with a smaller busy ratio will complete a background process faster than another with a larger busy ratio. The operating system can measure η at runtime based on how much portion of CPU cycles are used for foreground processes and how much portion of CPU cycles are available for background processes.

When $W \gg \tau_i$, many f-busy periods will arrive while a variable-speed processor Q executes a background process sp. Assume that $S_1, S_2, \cdots, S_n$ are i.i.d. random variables with a common mean μ and finite variance σ^2. Let $X = S_1 + S_2 + \cdots + S_n$. Then it is well known that $\overline{X} = n\mu$ and $\sigma_X^2 = n\sigma^2$. The coefficient of variation of X, $C_X = \sigma_X/\overline{X} = \sigma/\mu\sqrt{n}$, and thus $\lim_{n\to\infty} C_X = 0$, which means that the probability density of the sum of many i.i.d. random variables will be tight about its mean regardless of their distribution. This means that SR will be almost independent of W when $W \gg \tau_i$.

In this study we are mostly interested in the case where $W \gg \tau_i$, and we will omit sp and Q in SR_Q^{sp} because there is no ambiguity. When SR is known, the completion time of a background process can be estimated based on $T = SR \cdot W$. When the foreground processes arrive more frequently or demand more resources, SR increases and in turn T also increases since $T = SR \cdot W$. In our mathematical

model, the stretching ratio of the near future is independent of the recent past. In a practical SNOW, however, we may predict the stretching ratio of the near future based on the stretching ratio of the recent past.

It should be clear that no competition protocols can execute a program faster than W. When SR is small, there may not be much room for competition protocols to improve the performance of a program because $T \approx W$. However, when SR is sufficiently large, competition protocols may have a good potential to improve the performance of applications.

5.2.1 Performance of a Variable-speed Processor

The mean stretching ratio of Q, $\overline{SR}_Q$, can be expressed in terms of τ_i and τ_b as follows;

Proposition 5.1 *Let the mean length of f-busy periods and that of f-idle periods on a processor Q be τ_b and τ_i respectively. Then $\overline{SR}_Q = 1 + \tau_b/\tau_i$.*

Proof The processor Q is f-busy a fraction $\tau_b/(\tau_i + \tau_b)$ of time, and f-idle the remaining fraction $\tau_i/(\tau_i + \tau_b)$ of time. Thus the average amount of time Q can execute a background process per second is equal to the fraction of time Q is f-idle, that is, $\tau_i/(\tau_i + \tau_b)$. Assuming a background process that runs in W on a dedicated processor Q, $\overline{T}_Q = W(\tau_i + \tau_b)/\tau_i$, and $\overline{T}_Q/W = 1 + \tau_b/\tau_i$, which is $\overline{SR}_Q$. □

The mean stretching ratio $\overline{SR}_Q$ characterizes the performance of a variable-processor Q for a background process sp because $\overline{T}_Q = W \cdot \overline{SR}_Q$. Since τ_i and τ_b are measurable at runtime, $\overline{SR}_Q$ can be calculated. Thus we can predict the mean finishing time on a variable-speed processor if τ_i and τ_b are known.

Let $\overline{p}$ and σ_p^2 be the mean and variance of $p(k|W)$ given in Equation 5.3. Since Z is a random sum of f-busy period durations, the mean time to finish sp on Q, $\overline{T}_Q$, is the sum of a constant W, plus i.i.d. random variables representing the lengths of f-busy periods [Kle75, Tri82]. Therefore

$$\overline{T}_Q = W + \overline{p}\tau_b = W\frac{\tau_b + \tau_i}{\tau_i}, \tag{5.4}$$

where $\overline{p} = W/\tau_i$ is the average number of zero-length f-busy periods (with an average interarrival time of τ_i seconds) that arrive in W. The variance is

$$\sigma_{T_Q}^2 = \overline{p}\sigma_b^2 + \tau_b^2\sigma_p^2. \tag{5.5}$$

Equation 5.4 shows that $\overline{T}_Q/W = 1 + \tau_b/\tau_i$, which is the mean stretching ratio $\overline{SR}_Q$ in Proposition 5.1. The goal of competition protocols in a SNOW is to minimize the mean stretching factor of applications by executing multiple clones on different variable-speed processors, and thus improve the performance of applications.

5.2.2 A Nonpropagating Competition Protocol

Let random variables $F_1, F_2, \cdots, F_k$ denote the finishing times of the k clones. Let $T_k = \min_{i=1}^{k} F_i$. That is, T_k is the minimum finishing time among k clones. The speedup of the nonpropagating competition protocol can be defined as

$$SP(1,k) = \overline{T}_1/\overline{T}_k. \tag{5.6}$$

When f-busy periods arrive according to a Poisson process with rate $1/\tau_i$, Equation 5.5 reduces to

$$\sigma_{T_Q}^2 = 2\tau_b^2 W/\tau_i.$$

Note that the mean and variance of the Poisson process are identical. Now the coefficient of variation of T_Q becomes:

$$C_{T_Q} = \frac{\sigma_{T_Q}}{\overline{T}_Q} = \sqrt{\frac{2\tau_b}{W}} \frac{\sqrt{\tau_b/\tau_i}}{1 + \tau_b/\tau_i}. \tag{5.7}$$

Let us consider when C_{T_Q} approaches to 0, that is, the nonpropagating competition protocol cannot speed *sp* up.

- $\lim_{W \to \infty} C_{T_Q} = 0$. For sufficiently large W, the finishing time distribution tends to be an impulse at the mean finishing time, and $SP(1, k)$ goes to 1.

- With τ_b/W fixed, $\lim_{\tau_i \to 0} C_{T_Q} = 0$. For small τ_i, many f-busy periods occur before *sp* finishes. The law of large numbers ensures that the finishing time density will be tight about its mean because of many f-busy periods. $SP(1, k)$ goes to 1.

- $\lim_{\tau_i \to \infty} C_{T_Q} = 0$. When $\tau_i \gg \tau_b$, the f-busy periods become negligible because variable-speed processors are f-idle most of the time. The finishing time density becomes tight about its mean because f-busy periods add little variability to the finishing time. In this case, competition protocols can offer no advantage.

In summary, when the finishing time on a variable-speed processor has a relatively small variance, $SP(k)$ goes to 1 and the nonpropagating competition protocol cannot speed *sp* up. In the first and second cases, the finishing time has a small variance because many f-busy periods are likely to occur before *sp* finishes. On the other hand, the finishing time in the last case has a small variance because f-busy periods are not likely to occur before *sp* finishes.

The nonpropagating competition protocol can be useful only when W is not significantly larger than τ_i, because finishing times may have a sufficient variation in this case. When $W \gg \tau_i$, this situation is not likely to occur on processors with the same underlying statistics.

5.3 An Ideal Competition Protocol

We showed that the nonpropagating competition protocol has a limited performance benefit. But this limitation does not apply to a family of propagating competition protocols. In this section we consider the performance benefit of an ideal competition protocol. It works without any overhead as follows; a processor executes its clone only when it is f-idle, and it preempts the clone immediately when foreground processes arrive. If it is ahead of other clones, the ideal competition protocol propagates its state to other clones instantaneously. Note that an ideal competition protocol belongs to the family of propagating competition protocols.

5.3.1 Speedup of an Ideal Competition Protocol

Assume that the ideal competition protocol replicates *sp*'s clones k times, and allocates them to different processors, and that the arrival of foreground processes forms a Poisson process with rate $1/\tau_i$ at all processors, and that the service time for each foreground process is exponentially distributed with parameter $1/\tau_b$. Then processors can be modeled as independent M/M/1 queues, and their utilization for foreground processes is $\rho = \tau_b/(\tau_i + \tau_b)$. The foreground processes create certain pattern of activities into which background processes have to fit.

All we have to do in our queueing analysis is simply to figure out what is the probability that at least one of processors is f-idle.

The probability that a processor is f-idle is $p_0 = 1 - \rho$. Let W is the finishing time of *sp* to finish on a dedicated processor. Then it will take $W/p_0 = W/(1-\rho)$ on a single processor, and thus the mean stretching ratio of each processor is $1/(1-\rho)$. The mean stretching ratio increases or decreases when ρ increases or decreases. We will analyze the best speedup of the ideal competition protocol by considering independent M/M/1 queues.

Let us consider the speedup of the ideal competition protocol on two processors whose foreground activities are modeled as two independent M/M/1 queues. As long as one of two processors is f-idle, the ideal competition protocol allows *sp* to make progress because of instantaneous state propagation. The probability that one of two processors is f-idle is $2p_0 - p_0^2 = 1 - \rho^2$, and thus the mean stretching ratio of the ideal competition protocol on two processors is $1/(1-\rho^2)$. Therefore the speedup of the ideal competition protocol on two M/M/1 queues over a single M/M/1 queue is $\{1/(1-\rho)\}/\{1/(1-\rho^2)\} = 1 + \rho$.

In fact, the result above is independent of the underlying exponential distributions. We will generalize the result in the following theorem:

Theorem 5.1 *Assume that all processors alternate f-idle periods and f-busy periods. Let the mean length of f-idle periods and that of f-busy periods be t_i and t_b respectively. Then the foreground utilization of all processors is $\rho = t_b/(t_i + t_b)$. The speedup of the ideal competition protocol on k independent processors over one processor is $1 + \rho + \cdots + \rho^{k-1} = (1-\rho^k)/(1-\rho)$.*

Proof The ideal competition protocol allows *sp* to make progress as long as one

of k processors is f-idle. The probability that one of k independent processors is f-idle is $1-\rho^k$. Thus the stretching ratio of k processors with the ideal competition protocol is $1/(1-\rho^k)$, and the stretching ratio on one processor is $1/(1-\rho)$. Thus the speedup of the ideal competition protocol over noncompetition is given by

$$\frac{1/(1-\rho)}{1/(1-\rho^k)} = 1 + \rho + \cdots + \rho^{k-1}.$$

□

This theorem is true for *independent* processors with any underlying distributions as long as they have the same ρ. In fact, this is the best possible performance of any dynamic methods including migration in executing a sequential process without any parallelism in the background. This theorem shows that the ideal competition protocol has a diminishing return in adding another processor, and how it returns quantitatively. When $\rho < 0.5$, there may be not too much point in adding another processor after four because $\rho^4 < .0625$. This theorem also shows that if there are k processors, the maximum possible speedup is k and the minimum is one. So the ideal competition protocol can't get a slow down because the minimum speedup is one and can't get more than k-fold speedup because the maximum is k.

The best speedup from competition in the background occurs when ρ is very high. But that's exactly the same time when it takes the longest. So we have the tradeoff; we can finish soon if ρ is low because there are lots of spare cycles, and competition probably won't help you much. Or if ρ is high, there is high degree of competition efficiency and thus we can finish soon by cloning more.

Assume that the ideal competition protocol increases the number of clones until the diminishing return is less than 5%, that is, it clones more clones until an additional clone offer less than 5% speedup. In order to find when the per-

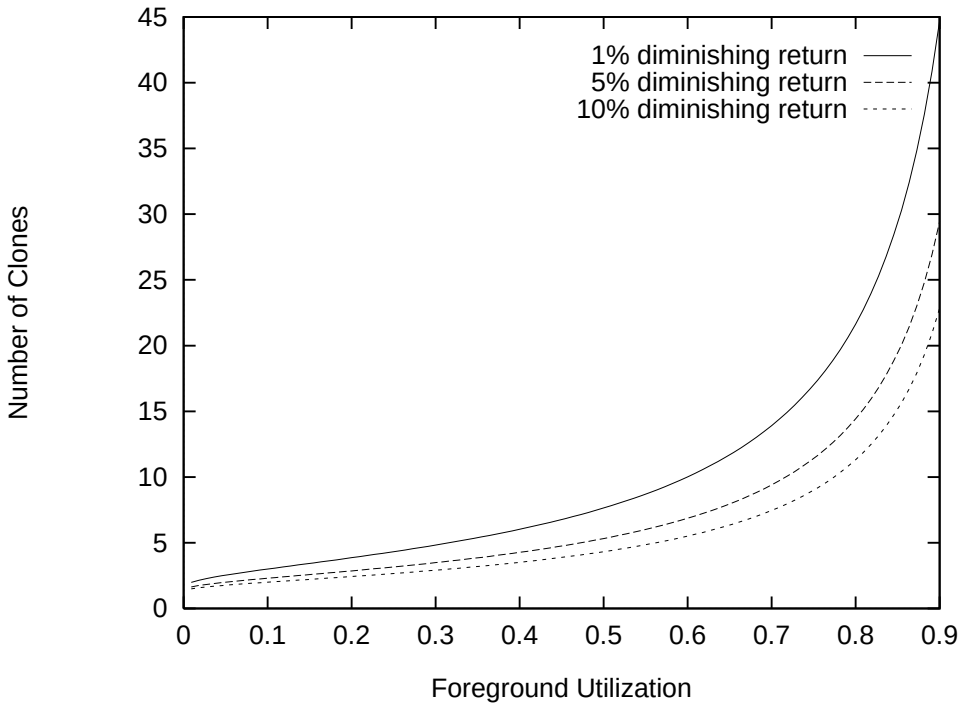


Figure 5.3: Number of Clones to Have a 5% Diminishing Return

formance benefit of adding one more clone is less than 5%, we have to solve the inequality $\rho^{k-1} < 0.05$. Thus $k > 1 + log(0.05)/log(\rho)$. The curve in the middle of Figure 5.3 shows how many clones we need as a function of ρ to have a 5% diminishing return. When $\rho < 0.5$, 5 clones are enough to have a 5% diminishing return. When ρ increases, the number of clones sharply increases to have the 5% diminishing return. Figure 5.3 also shows cases of 1% and 10%, and we need more clones to have a 1% diminishing return than to have a 10% diminishing return.

The theorem shows that when the foreground load on processors gets high, we should clone more, and that when the foreground load gets low, it is not worth having all clones. In the ideal case this does not hurt the system performance. In the non-ideal case, this degrades system performance because the ideal competition protocol has to coordinate them all. The ideal competition protocol

can measure how busy processors are at runtime, and it can clone or unclone a process *sp* based on the theorem to improve the performance of *sp*. In general, competition becomes more valuable in the worst case, namely, when ρ increases, and this is counter intuitive.

If the arrival rate of the foreground processes and their service times are not the same on all processors, this theorem won't be applied exactly. If there is non-zero propagation delay, the theorem won't be applied, either. However, the theorem shows the limit of what can be achieved by competition protocols. So no competition protocols can do better than that.

If statistics of foreground processes on processors are different, there will be a formula with a similar shape that the ideal competition protocol cannot do better. The benefit you get for each additional processor exponentially trails off. But the shape of that exponential will be slightly different.

The upper bound for the speedup of the ideal competition protocol on k processors over a single processor approaches k when ρ approaches 1, and 1 when ρ approaches 0. It was shown that the speedup of a nonpropagating competition protocol approaches 1 when W goes to infinity. Thus the speedup of the ideal competition protocol over the nonpropagating competition protocol is also approaching k when ρ approaches 1. The absolute execution time without competition approaches infinity when ρ approaches 1. That is, we achieve the best speedup when the finishing time is worst.

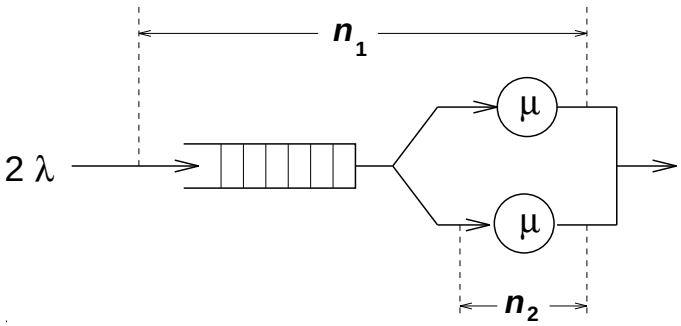


Figure 5.4: A M/M/2 Queue

5.4 Effect of Load Balancing

In order to see the load balancing effect on the ideal competition protocol, let us consider a M/M/2 queue in Figure 5.4 where foreground processes arrive at rate 2λ and service rate for each foreground process at two servers is μ. Foreground processes wait in the queue in the order of their arrival.

We are interested in the probability that one of servers is f-idle. The state of the system is defined to be the tuple (n_1, n_2) where $n_1 \geq 0$ denotes the number of foreground processes in the queue including any at one specific server, and $n_2 \in 0, 1$ denotes the number of foreground processes at the other server. If a foreground process arrives when both servers are idle, it is scheduled for service at one of the servers randomly. Obviously the probability that a server is idle while foreground processes are waiting on the other server is zero in this M/M/2 queue. Clearly the M/M/2 queue is more load balanced than two independent M/M/1 queues.

We consider the speedup of the ideal competition protocol on two processors whose foreground activity is characterized by a M/M/2 queue. The state diagram of the system is given in Figure 5.5. Balance equations in the steady state, can

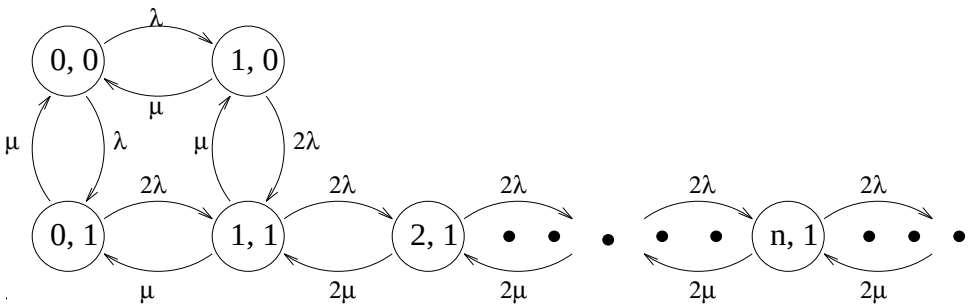


Figure 5.5: The State Diagram of the M/M/2 Queue

be written by equating the rate of flow into a state to the rate of flow out of that state:

$$2\lambda p(0,0) = \mu p(0,1) + \mu p(1,0), \tag{5.8}$$

$$\{2\lambda + \mu\}p(1,0) = \lambda p(0,0) + \mu p(1,1), \tag{5.9}$$

$$\{2\lambda + \mu\}p(0,1) = \lambda p(0,0) + \mu p(1,1), \tag{5.10}$$

$$\{2\lambda + 2\mu\}p(1,1) = 2\lambda\{p(1,0) + p(0,1)\} + 2\mu p(2,1), \tag{5.11}$$

$$\{2\lambda + 2\mu\}p(n,1) = 2\mu p(n+1,1) + 2\lambda p(n-1,1), n > 1 \tag{5.12}$$

The traffic intensity for this system is $\rho = \lambda/\mu$. By applying the concept of local balance [Kle75], we have the following balance equation:

$$2\mu p(n,1) = 2\lambda p(n-1,1), n > 1 \tag{5.13}$$

Equation 5.13 can easily be seen to satisfy equation 5.12. By repeated use of equation 5.13, we have:

$$p(n,1) = \rho p(n-1,1) = \rho^{n-1} p(1,1), n > 1. \tag{5.14}$$

From equations 5.8 to 5.10 we can obtain:

$$p(0,1) = \rho p(0,0), \tag{5.15}$$

$$p(1,0) = \rho p(0,0), \tag{5.16}$$

$$p(1,1) = \rho^2 p(0,0). \tag{5.17}$$

Now, observing that:

$$p(0,0) + p(1,0) + p(0,1) + \sum_{n \geq 1} p(n,1) = 1,$$

we have:

$$p(0,0)[1 + \rho + \rho] + p(1,1) \sum_{n \geq 1} \rho^{n-1} = 1$$

or

$$p(0,0)[1 + 2\rho + \rho^2(\frac{1}{1-\rho})] = 1,$$

from which we get:

$$p(0,0) = \frac{1-\rho}{1+\rho-\rho^2}.$$

Thus

$$p(1,0) = p(0,1) = \frac{\rho(1-\rho)}{1+\rho-\rho^2}.$$

When we execute *sp* on only one processor of the two processors in the background, the probability that one processor is f-idle is $p(0,0) + p(1,0) = [(1-\rho)(1+\rho)]/(1+\rho-\rho^2)$. Thus the mean stretching ratio on one processor of the two processors is $(1+\rho-\rho^2)/[(1-\rho)(1+\rho)]$. Note that the mean stretching ratio on a single M/M/1 queue is $1/(1-\rho)$.

Since $1/(1-\rho) > (1+\rho-\rho^2)/[(1-\rho)(1+\rho)]$, we can finish *sp* faster on one processor of the M/M/2 queue than on the processor of the M/M/1 queue. Because the M/M/2 queue is more balanced, there is more fraction of idle times on one processor of the M/M/2 queue than on one processor of the M/M/1 queue with the same statistics. Consequently it should take shorter to finish on one processor of the M/M/2 queue than on the processor of the M/M/1 queue.

The ideal competition protocol on two processors allows sp to make progress as long as one of them is f-idle, and that probability is $p(0,0)+p(1,0)+p(0,1) = [(1-\rho)(1+2\rho)]/(1+\rho-\rho^2)$. The mean stretching ratio is $(1+\rho-\rho^2)/[(1-\rho)(1+2\rho)]$. Therefore the speedup of the ideal competition protocol with two processors over one processor in a M/M/2 queue is:

$$\frac{(1+\rho-\rho^2)/[(1-\rho)(1+\rho)]}{(1+\rho-\rho^2)/[(1-\rho)(1+2\rho)]} = \frac{1+2\rho}{1+\rho}.$$

Thus the best speedup of the ideal competition protocol approaches 1.5 on two processors when ρ goes to 1. Note that the best speedup of the ideal competition protocol on 2 independent M/M/1 queues goes to 2 when ρ goes to 1. Thus the speedup on a M/M/2 queue is smaller than that on two independent M/M/1 queues. The reason is that a M/M/2 queue is more balanced than two independent M/M/1 queues; when the load among processors is more balanced, the f-idle periods on processors are more likely to overlap and thus the effectiveness of competition decreases. We can conclude that the speedup of the ideal competition protocol decreases when the load among processors is more balanced.

It has been widely observed that usage of computing resources in a distributed environment is usually bursty over time and uneven among processors [Zho93]. Thus we believe that the ideal competition protocol may get more performance benefit than a M/M/k queue in a real SNOW with k processors because of burstiness and uneven load.

CHAPTER 6

Competition and Migration for a Sequential Process

6.1 Introduction

We can use migration or competition to speed up distributed programs on a SNOW. In order to compare the performance of competition and migration, we will consider a sequential process *sp* without any parallelism in it. We assume that all processors alternate f-busy periods and f-idle periods, and that the lengths of f-busy periods at all processors are i.i.d. random variables from an exponential distribution with mean τ_b, and that the lengths of f-idle periods are also i.i.d. random variables from an exponential distribution with mean τ_i. Thus the mean stretching ratio of all processors is $1 + \tau_b/\tau_i$.

When migration delay or state propagation delay is larger than the mean length of f-busy periods, migration protocols or competition protocols generally cannot improve the performance of *sp*. Thus we will focus on the case where the mean delay of migration or state propagation is smaller than the mean length of f-busy periods.

In this chapter we will compare the performance of competition with that of migration. It is not our goal to develop efficient migration protocols and competition protocols, but to develop migration protocols and competition protocols

that are fairly comparable. We will show that when migration is beneficial, competition may be even more beneficial than migration under our comparable assumptions.

We will consider two migration protocols and two competition protocols, which have different performance characteristics. We assume that all competition protocols replicate the sequential process sp, and statically put k clones $sp_1, sp_2, \cdots, sp_k$ on k different variable-speed processors $P_1, P_2, \cdots, P_k$ out of M processors where $M \geq k$. Thus the number of clones is fixed and never changes.

We assume that migration protocols migrate sp among k variable-speed processors out of M processors. At any given time sp is executed by one of them, which is called a *hosting* processor. We compare the performance of competition with that of migration assuming that the processors participating in competition are the same as those participating in migration.

For simplicity, we assume that migration delay or state propagation delay is constant. This assumption could be unrealistic in the following ways; some states are large, some states are small, and there may be congestion on a network. But we are going to ignore those, and assume that migration delay or state propagation delay is independent of other factors in the system.

6.1.1 A Migration Policy and a Competition Policy

There are many choices for migration protocols in deciding when to migrate, what to migrate, and where to migrate. In this chapter we focus on a sequential process sp, and thus we do not need to consider what to migrate. We will consider a specific migration policy; when sp is preempted, migration protocols migrate sp randomly to one f-idle processor out of $k-1$ processors immediately if at least

one f-idle processor is available. Otherwise, migration protocols migrate sp to an f-idle processor later when one processor goes f-idle out of k processors. If the current hosting processor goes f-idle earlier than the other processors, migration protocols do not need to migrate sp.

There are many choices for competition protocols in deciding when to clone or unclone to what degree (e.g, two folds or three folds), what to clone or unclone, and where to clone or unclone. Since we consider a sequential process sp and assume that the number of clones of sp is fixed, we just need to consider when to propagate states from which clone to which clones.

Competition protocols ideally propagate states of the clone that is most ahead to other clones that are behind, but not vice versa. They can determine which clone is ahead based on the number of instructions they have executed, or CPU cycles they have used. We consider a specific competition policy; when a clone that is most ahead is preempted, competition protocols propagate its state to other clones. Otherwise, competition protocols do not propagate the state of the clone that is preempted.

In summary, when foreground processes arrive, sp is preempted and it is migrated to one f-idle processor of $k-1$ processors immediately if there is a f-idle processor, or later when one processor goes f-idle. When a clone of sp is preempted, competition protocols propagate its state to the other clones only if it is ahead of the others.

In practice, there will be some delay in choosing a migration target and determining whether a clone is ahead of the others, but we assume that migration decision and state propagation decision are made instantaneously in all migration and competition protocols in this study.

The amount of information to transfer in competition protocols can be larger than that in migration protocols in a general network topology when there are more than two clones of *sp*. However, there will be no difference on a network with multicasting capability such as Ethernets or token rings.

Competition protocols have to deliver at least one message clone of a message to each process clone of its receiver, and thus communication requirement in competition can be higher than that in migration. Again this difference disappears on a network with multicasting capability. In this chapter we assume that overheads of migration and competition are identical independent of the number of clones because we consider SNOW environments with multicasting capability.

6.2 Migration Protocols and Competition Protocols

In this section we will describe how migration protocols and competition protocols work, and explain what the differences among these protocols are. We assume that the decision making part of protocols in migration or competition is done with high priority, but that process migration or state propagation itself is done with low priority in order to minimize the interference with foreground processes. Thus migration delay or state propagation delay can vary in practice, but we will assume that those delays are constant for the sake of analysis. Decision making may interfere with foreground processes, but its delay won't be as long as that of migration or state propagation itself.

6.2.1 Migration Protocols

MIG_0, MIG_D	*COMP_0, COMP_D*

migration decision;
/* A process consists of n packets */
for (i=0; i<n; i++) {
  transmit packet i to the target;
}
	state propagation decision;
	/* A state consists of n packets */
	for (i=0; i<n; i++) {
	  multicast packet i to the targets;
	}

Figure 6.1: Skeletons of Protocols

Migration protocols considered in this chapter are based on the specific migration policy described in the subsection 6.1.1 above. We show skeletons of *MIG_0* and *MIG_D* protocols in Figure 6.1. When *sp* is preempted, the migration decision is made instantaneously. We assume that n packets have to be transferred from the source to the target. The difference between *MIG_0* and *MIG_D* is that migration delay in *MIG_0* protocol is 0 while that in *MIG_D* protocol is D, which is constant.

Thus migration in *MIG_0* protocol is always successful because the target must be still f-idle when migration is completed. On the other hand, migration in *MIG_D* protocol may not be successful due to the migration delay, that is, the target may become f-busy while *sp* is migrating. Note that migration decision itself is always right because we assume that migration decision is instantaneous.

The probability of migration success decreases as the migration delay increases. When migration is unsuccessful, we assume that the target treats *sp* in the same way as it preempts *sp*. Thus it is possible for *sp* to migrate more than once without having any chance to run. We will focus on *MIG_0* and *MIG_D* migration protocols in this chapter.

6.2.2 Competition Protocols

Competition protocols considered in this chapter are based on the specific competition policy described in the subsection 6.1.1 above. When competition protocols propagate a state of a clone that is most ahead, they *freeze* execution of all the other clones instantaneously. Then they start to propagate its state to the other clones, which reset the state of their clones to the state propagated. We assume that state propagation decision and freezing all other clones are instantaneous.

We consider *COMP_0* protocol where the state propagation is instantaneous similarly to the *MIG_0* protocol. The skeleton of the *COMP_0* protocol is shown in Figure 6.1. We consider another competition protocol, called *COMP_D* protocol, where the delay for state propagation is a constant D for fair comparison between *MIG_D* protocol and *COMP_D* protocol. The *COMP_0* and the *COMP_D* protocols are identical except state propagation delay.

None of competition protocols replaces the clone that is ahead with a clone that is behind because of freezing. If clones are not frozen, their states may be partly modified while states are propagating. In practice, it is not necessary to freeze all clones during the same period of time. A clone that is ahead needs to be frozen while its state is locally copied, and the other clones need to be frozen while switching from a state to another by maintaining two states.

6.2.3 Inherent Differences between Competition and Migration

When a critical process in a distributed program is executing on a heavily loaded processor, we may migrate it from the heavily loaded processor to a lightly load processor. However, other processes in the same program may not do anything

but wait for the critical process due to synchronization or message communication during its migration. The critical process must have been already slowest in the program, and thus migration delay can degrade the completion time of the entire program. Furthermore, migration delay on a SNOW can be large because migration for background process should have lower priority than the other foreground message traffic.

The fundamental limitation of migration comes from the fact that there is only one instance of the process executing at any given time. Migration protocols have to choose one of targets, and then start migration. The target may become f-busy while the process is migrating, even though the other targets are still f-idle. When migration delay is large or the workload of foreground processes on processors changes fast or drastically, migration may be often unsuccessful.

Even if migration is successful, the f-idle period of the target may be smaller than those of other targets. Migration protocols may choose any of f-idle periods, including the *minimum* and the *maximum*. On the average, the f-idle period chosen will be the *mean* length of f-idle periods independently of the number of participating processors.

However, the f-idle period in competition protocols is not a typical f-idle period but the *maximum* of f-idle periods; when more than one clone start to execute at the same time after state propagation, all of them must have made the same progress when one of them is preempted earliest. We assume that only the clone that is preempted last can propagate its state in all competition protocols.

If all processors are f-busy when *sp* is preempted, *sp* is dormant at the hosting processor in migration protocols. When a non-hosting processor becomes f-idle

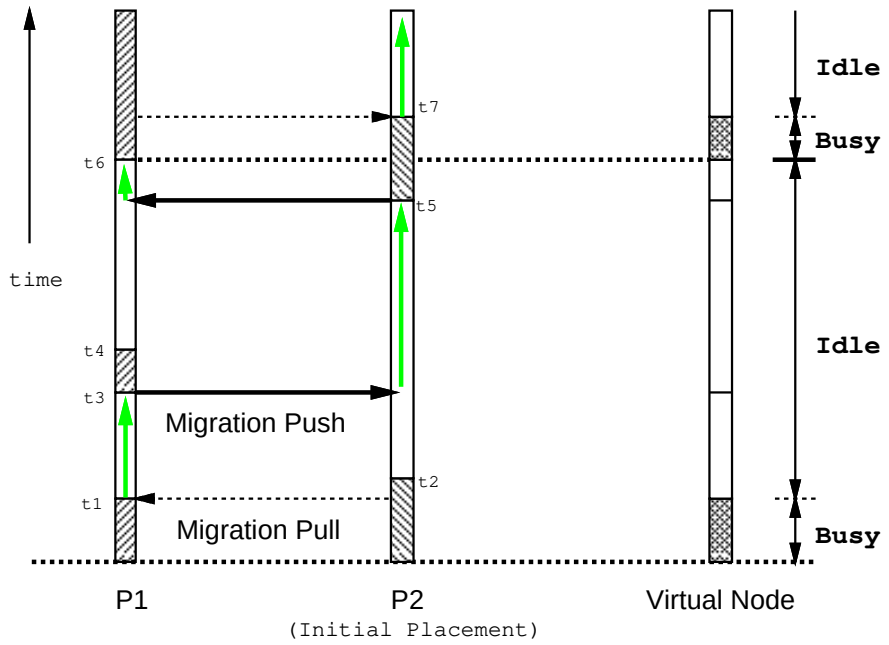


Figure 6.2: The *MIG_0* Protocol

earlier than the hosting processor, the non-hosting processor has to migrate *sp* from its current hosting processor in order to execute *sp* in migration protocols. This migration is called a *migration pull*. However, competition guarantees that the clone ahead of other clones is always on the fastest processor among processors because competition protocols always propagate its state to the other clones whenever the clone that is most ahead is preempted.

6.3 Performance Comparison of *MIG_0* and *COMP_0*

The *MIG_0* protocol and the *COMP_0* protocol have similar skeletons as in Figure 6.1 except for unicasting in migration and multicasting in competition.

6.3.1 *MIG_0* Protocol

Figure 6.2 shows the case of two processors, $P1$ and $P2$, and that the process *sp* initially starts on $P2$. Shaded periods indicate that processors are f-busy, and white periods indicate that processors are f-idle. A solid horizontal arrow indicates a process migration (called *migration push*) at the end of a f-idle period, while a dotted horizontal arrow indicate a migration pull at the beginning of a f-idle period. We need to distinguish a migration push and a migration pull later when migration delay is not zero.

In the *MIG_0* protocol, *sp* can be migrated to another processor instantaneously, and thus it does not matter where *sp* starts. $P1$ migrates *sp* from $P2$ at `t1`. Then *sp* is migrated to $P2$ at `t3`, and to $P1$ at `t5`. When $P1$ preempts *sp* at `t6`, it cannot migrate *sp* to $P2$ because $P2$ is f-busy. Thus *sp* stays dormant on $P1$ until `t7`, when $P2$ becomes f-idle and thus *sp* is migrated to $P2$. The execution trace of *sp* is shown by thick vertical arrows in Figure 6.2.

In order to analyze the performance of the *MIG_0* protocol, we project f-busy and f-idle periods on $P1$ and $P2$ horizontally onto an imaginary processor (called a *virtual node*) as in Figure 6.2. If both $P1$ and $P2$ are f-busy at a given time, the virtual node is called *busy* because it is not possible to execute *sp* in any processor. Otherwise, the virtual node is called *idle* at the moment. A period that is continuously busy is called a busy period, and a period which is continuously idle is called an idle period.

Figure 6.2 shows that *sp* starts when $P1$ and $P2$ are f-busy. The initial f-busy periods on two processors are in fact the residual f-busy periods when *sp* starts. But the distribution of time remaining for an exponentially distributed random variable is independent of the acquired "age" of that random variable [Kle75]. Thus the length of the first busy period on the virtual node is simply the minimum

of the lengths of two f-busy periods on $P1$ and $P2$. The second one is also the minimum of the lengths of two residual f-busy periods on $P1$ and $P2$ at `t6`. The f-busy period on $P2$ starts at `t5`, but the distribution of the residual f-busy period at `t6` is still the same because of the memoryless property of exponential distributions.

In general the length of a busy period is the minimum of lengths of two f-busy periods on $P1$ and $P2$. When the lengths of two f-busy periods on $P1$ and $P2$ are exponentially distributed with parameters λ_1 and λ_2 respectively, the minimum of two such lengths is distributed with a parameter $\lambda_1 + \lambda_2$ [Kle75, Tri82].

An idle period on the virtual node may consist of more than one segment as shown in Figure 6.2, where each segment indicates that *sp* executes at a processor during that period. The *MIG_0* protocol checks whether there are any f-idle processors at the end of each segment when a preemption occurs. Note that a segment may be a part of a f-idle period on a processor. The first idle period consists of three segments, and they are from `t1` to `t3` on $P1$, from `t3` to `t5` on $P2$, and from `t5` to `t6` on $P1$. The f-idle period from `t3` to `t5` on $P2$ is the residual f-idle period from `t2` to `t5`, but the residual f-idle period has the same exponential distribution of the f-idle period.

Migration stops when both $P1$ and $P2$ are f-busy at `t6` when an idle period ends and a busy period starts. Busy periods and idle periods on the virtual node alternate. We can form a renewal process from the alternating states of the virtual node by letting a renewal period be a busy period followed by an idle period. We will analyze the performance of the *MIG_0* protocol on the virtual node based on the renewal period [Cox62]. If we know the means of busy periods and idle periods, we can easily find the stretching ratio on the virtual node, that

is, the performance of the *MIG_0* protocol. The renewal period is indicated by a thick horizontal dotted line in Figure 6.2.

We observe that the lengths of busy periods on the virtual node are shortened and the lengths of idle periods are lengthened, and this is the reason why the virtual node, that is, the *MIG_0* protocol on two processors renders the better performance than a single processor. Let B_k and I_k denote a busy period and an idle period respectively on the virtual node when there are k participating processors. Then B_k will be exponentially distributed with a mean τ_b/k because B_k is the minimum of k exponentially distributed random variables with the same mean τ_b. Thus $\overline{B_k}$, the mean of B_k, is

$$\overline{B_k} = \tau_b/k.$$

Now let us consider the mean idle period. The probability that a processor $P_i(i = 1, 2, \cdots, k)$ is f-busy at any given time, is $\rho = \tau_b/(\tau_i + \tau_b)$. In Figure 6.2, if *sp* starts at $P1$ initially, $P1$ can start to execute *sp* without a migration pull, In this case $P1$ is called a *proper* processor, and the number of migration is 2 in the idle period. Since *sp* is positioned at $P2$ initially, however, $P1$ has to migrate *sp* from $P2$ first.

Note that a migration pull can occur only at the beginning of an idle period on a virtual node. A migration pull is different from a migration push because the migration pull occurs at the beginning of a f-idle period while the other migrations occur at the end of a f-idle period. Furthermore, the migration pull may or may not be necessary depending on whether *sp* is positioned at the proper processor or not. The probability that *sp* is positioned at a proper processor at the beginning of an idle period will be $1/k$ assuming k processors.

Now let us focus on only two processors. At the end of each segment in an

idle period, the hosting processor becomes f-busy, and checks the other processor. The probability that a processor is f-idle at a random point of time is $1-\rho$, but the probability that the other processor is f-idle at the end of each segment, that is, at the moment of preemption, is not $1-\rho$, and this will be validated by our simulation and the theoretic upper bound derived in Chapter 5.

Let X denote the number of migration excluding the migration pull in an idle period. In *MIG_0* protocol the number of segments in an idle period is $X+1$ because migration stops at the end of the last segment. The probability mass function of X is given by a modified geometric distribution [Tri82]:

$$P_X(i) = fs^i, i = 0, 1, 2, \cdots \tag{6.1}$$

where s is the probability that the other processor is f-idle when a preemption occurs at one processor, and $f = 1-s$. Then $\overline{X} = s/f = s/(1-s)$.

An idle period consists of the random sum of segments, and each segment is exponentially distributed as a f-idle period due to the memoryless property of exponential distributions. Let I_2 denote the length of an idle period when there are two processors participating in migration. $\overline{I_2}$, the mean of I_2, is given by [Tri82]

$$\overline{I_2} = \tau_i(\overline{X}+1) = \tau_i/(1-s).$$

Thus the mean stretching ratio of the *MIG_0* protocol on two processors is given by

$$SR_{MIG_0}(2) = 1 + \frac{\overline{B_2}}{\overline{I_2}} = 1 + \frac{\tau_b}{\tau_i} \cdot \frac{1-s}{2}.$$

The speedup of the *MIG_0* protocol with two processors over a single processor is

$$SP_{MIG_0}(2) = \frac{1+(\tau_b/\tau_i)}{SR_{MIG_0}(2)} = \frac{1+(\tau_b/\tau_i)}{1+(\tau_b/\tau_i)((1-s)/2)}. \tag{6.2}$$

Theorem 6.1 *Let us assume that there are two processors participating in migration. When a preemption occurs on a processor, the probability that the other processor is f-idle in the MIG_0 protocol is not equal to but less than* $1 - \rho = \tau_i/(\tau_i + \tau_b)$.

Proof Let us assume that the probability that the other processor is f-idle, is equal to $1 - \rho$. We will show that $SP_{MIG_0}(2)$ in Equation 6.2 is even larger than the optimal speedup we derived in Theorem 5.1. By replacing s in Equation 6.2 with $1 - \rho$, we get

$$SP_{MIG_0}(2) = \frac{2\tau_i^2 + 4\tau_i\tau_b + 2\tau_b^2}{(\tau_i + \tau_b)^2 + \tau_i^2}. \tag{6.3}$$

The optimal speedup with two processors is $1 + \rho$ according to Theorem 5.1.

$$1 + \rho = 1 + \frac{\tau_b}{\tau_i + \tau_i} = \frac{\tau_i + 2\tau_b}{\tau_i + \tau_b}.$$

By multiplying $\tau_i + \tau_b$ to both the numerator and the denominator, we get

$$1 + \rho = \frac{\tau_i^2 + 3\tau_i\tau_b + 2\tau_b^2}{(\tau_i + \tau_b)^2}. \tag{6.4}$$

The numerator of Equation 6.3 is $\tau_i^2 + \tau_i\tau_b$ larger than that of Equation 6.4, but the denominator of Equation 6.3 is τ_i^2 larger than that of Equation 6.4. Therefore, Equation 6.3 is always larger than Equation 6.4. This means that $SP_{MIG_0}(2)$ is larger than the optimal speedup, which is contradictory. This implies that the probability should be smaller than $1 - \rho = \tau_i/(\tau_i + \tau_b)$. □

Equation 6.2 shows the performance of an ideal case where migration decision and migration itself have no overheads. Thus this result should be the same as the optimal speedup, which is $1 + \rho$. By setting $SP_{MIG_0}(2) = 1 + \rho$, the probability of f-idleness when a preemption occurs is given by

$$s = \frac{\tau_i}{\tau_i + 2\tau_b}. \tag{6.5}$$

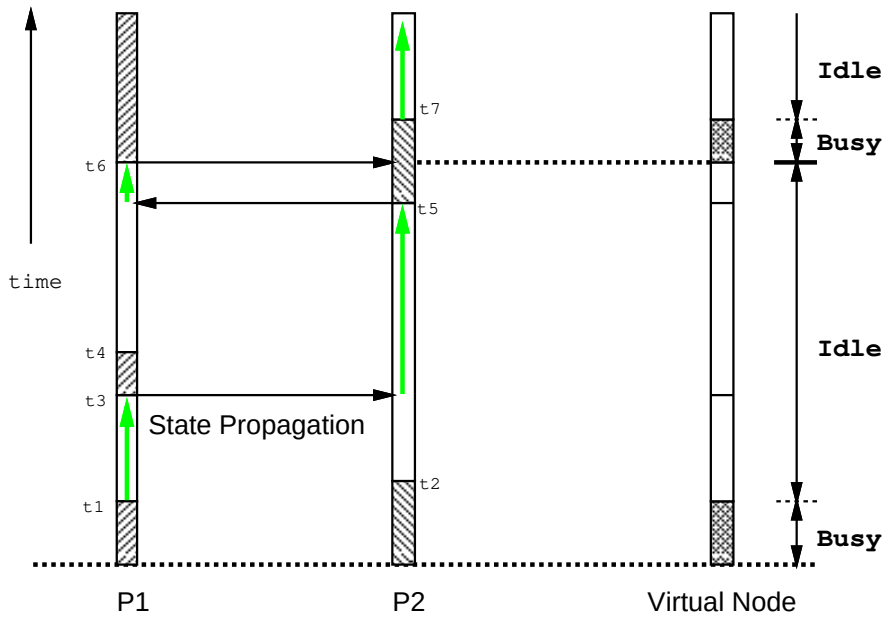


Figure 6.3: The *COMP_0* Protocol

It is easily seen that s is smaller than $1 - \rho$. We are going to use this probability as an approximate probability of f-idleness when the migration delay is not instantaneous in the next section.

6.3.2 *COMP_0* Protocol

Figure 6.3 shows that two clones sp_1 and sp_2 start on $P1$ and $P2$ respectively, and activities of foreground processes on $P1$ and $P2$ are the same as those in Figure 6.2. A solid arrow indicates a state propagation. The clone sp_1 starts to execute at `t1` and sp_2 starts to execute at `t2`. Since $D = 0$, sp_2 may start with sp_1's state at `t2`. But this is not necessary because $P1$ is going to propagate sp_1's state at `t3` anyway.

Similarly the clone sp_1 may resume its execution at `t4` based on sp_2's state at the moment. Here again this is not necessary because $P2$ preempts sp_2 and

propagates $b2$'s state at `t5`. $P1$ replaces sp_1's state with sp_2's state at `t5`. $P1$ preempts sp_1 at `t6` and propagates its state even though $P2$ is f-busy, and $P2$ replaces sp_2's state with sp_1's state. This differs from migration protocols where migration stops if $P2$ is f-busy. $P2$ resumes to execute sp_2 at `t7`. In competition protocols we do not need to worry about the initial placement at the beginning of an idle period on a virtual node because the clone that is most ahead is guaranteed to be on a proper processor.

In order to analyze the performance of the *COMP_0* protocol, we project f-busy and f-idle periods on $P1$ and $P2$ on a virtual node horizontally as shown in Figure 6.3, which is the same as Figure 6.2 except the migration pull. Similarly to the *MIG_0* protocol, we can form a renewal process from the alternating states of the virtual node by letting a renewal period be a busy period followed by an idle period. Thus we analyze the performance of the *COMP_0* protocol by inspecting the mean lengths of idle periods and busy periods.

The length of a busy period is the minimum of the lengths of the two f-busy periods on $P1$ and $P2$ as in the *MIG_0* protocol. An idle period may consist of more than one segment as shown in Figure 6.3, and each segment always traces the performance profile of the clone that is ahead at any time. The first idle period consists of three segments where the first traces sp_1's performance from `t1` to `t3`, the second traces sp_2's performance from `t3` to `t5`, and the third traces sp_1's performance from `t5` to `t6`.

Now let us focus on two processors. A processor propagates a state when it becomes f-busy. An idle period on a virtual node ends when the other processor is f-busy. Thus we need to know what is the probability that the other processor is f-busy or f-idle at the end of a state propagation. Since state propagation

is instantaneous, this probability is the same as the probability that the other processor is f-idle at the moment of preemption at one processor. Thus the probability will be the same as the one in Equation 6.5.

Let Y denote the number of state propagations in an idle period. Then in the *COMP_0* protocol the number of segments in an idle period is Y because state propagations occur at the end of each segment including the last one.

The probability mass function of Y is given by a geometric distribution [Tri82]:

$$P_Y(i) = fs^i, i = 1, 2, \cdots \tag{6.6}$$

where $s = \tau_i/(\tau_i + 2\tau_b)$ and $f = 1 - s$. And $\overline{Y} = 1/f = 1 + \tau_i/2\tau_b$.

When there are two clones only, they cannot start execution at the same time in the first segment in an idle period in the *COMP_0* protocol because the state propagation is instantaneous. Thus one of them starts earlier in the first segment, and thus that clone will have made more progress than the other when it is preempted.

As in the *MIG_0* protocol, $\overline{B_2} = \tau_b/2$. An idle period consists of the random sum of segments, and each segment is exponentially distributed as a f-idle period due to the memoryless property. Let I_2 denote the length of an idle period when there are two processors participating in competition. $\overline{I_2}$, the mean of I_2, is

$$\overline{I_2} = \tau_i(\overline{Y}) = \tau_i(1 + \tau_i/2\tau_b).$$

Thus the mean stretching ratio of the *COMP_0* protocol on two processors is given by

$$SR_{COMP_0}(2) = 1 + \frac{\overline{B_2}}{\overline{I_2}} = \frac{(\tau_i + \tau_b)^2}{\tau_i(\tau_i + 2\tau_b)}.$$

The speedup of the *COMP_0* protocol with two processors over a single processor is

$$SP_{COMP_0}(2) = \frac{1 + (\tau_b/\tau_i)}{SR_{COMP_0}(2)} = 1 + \frac{\tau_b}{\tau_i + \tau_b} = 1 + \rho. \tag{6.7}$$

Thus the performance of the *COMP_0* protocol is the same as the optimal speedup.

When there are two participating processors and $D = 0$, the processor that executes the clone which is ahead in the *COMP_0* protocol would be the same as the hosting processor for *sp* in the *MIG_0* protocol if migration and competition starts at two independent and identical SNOW environments. Thus the execution trace of *sp* in the *MIG_0* protocol will always be the same as the execution trace of the faster clone in the *COMP_0* protocol. However, this is not true when there are more than two participating processors or $D > 0$.

When $D = 0$, the migration pull in the *MIG_0* protocol is instantaneous, and thus it appears that *sp* always stays on the proper processor at the beginning of an idle period. Furthermore, the *COMP_0* protocol and the *MIG_0* protocol can completely exploit f-idle periods on two processors when $D = 0$. They never waste any f-idle periods on processors, and thus both of them must achieve the optimal speedup.

6.3.3 Simulation Results

We simulated *MIG_0* and *COMP_0* protocols to validate our analysis. In all of our simulations τ_i was set to 1.0, and we simulated 20 times for a period of $1000\tau_i$ and our simulation results are the mean of 20 simulation results. The x axis (mean length of busy periods) in Figure 6.4 is shown in multiples of τ_i. Figure 6.4 shows three simulation results; the curves labeled as 'Competition' and 'Migration' are

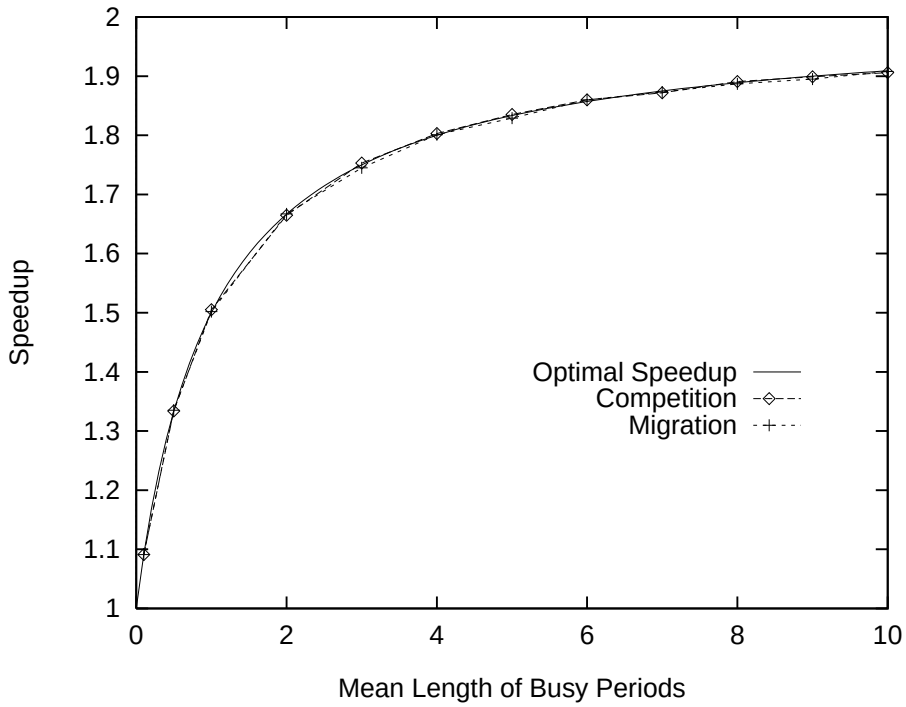


Figure 6.4: Simulation Results with Two Processors

simulation results of *MIG_0* protocol and *COMP_0* protocol respectively. The curve labeled as 'Optimal Speedup' is $1 + \rho$, which is an optimal speedup with two processors derived in Chapter 5.

Our analytic results here must agree with the optimal speedup because we find the probability of f-idleness by setting our analytic speedup equal to the optimal speedup. Figure 6.4 shows that our analytic results and simulation results are perfectly matched to validate our simulation code, which will be used when there is the delay for migration or state propagation in the next section. We are going to use $\tau_i/(\tau_i + 2\tau_b)$ as an approximate probability of f-idleness when the delay for migration or state propagation is not zero in the next section.

Figure 6.5 shows the optimal speedup and simulation results of *MIG_0* and *COMP_0* protocols on three processors. Here the x axis (mean length of busy

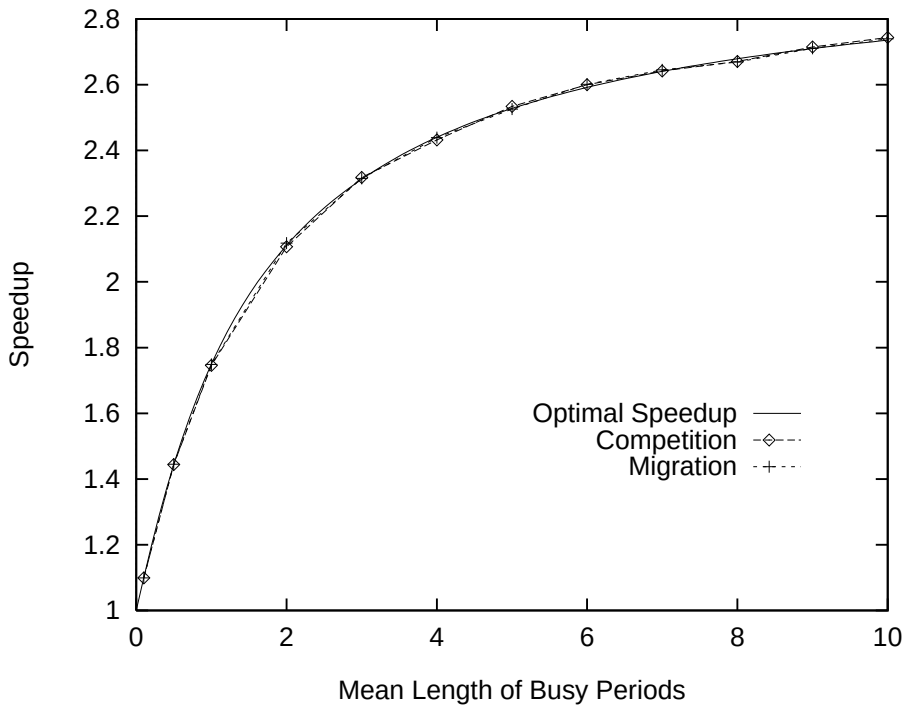


Figure 6.5: Simulation Results with Three Processors

periods) in Figure 6.4 is also shown in multiples of τ_i. We can find the probability of f-idleness as in the two processor case, but we omit this because our analytic result will be the same as the optimal speedup. The correctness of our simulation codes are validated again.

6.4 Performance Comparison of *MIG_D* and *COMP_D*

6.4.1 *MIG_D* Protocol

Figure 6.6 shows the same activities of foreground processes on $P1$ and $P2$ as in Figure 6.2, but shows that an idle period consists of black periods and white periods. A solid arrow indicates a migration pull and a dotted arrow indicates a migration pull. Shaded periods and white periods denote the same as in Figure 6.2 while a black period denotes the migration delay D, which is also shown by a

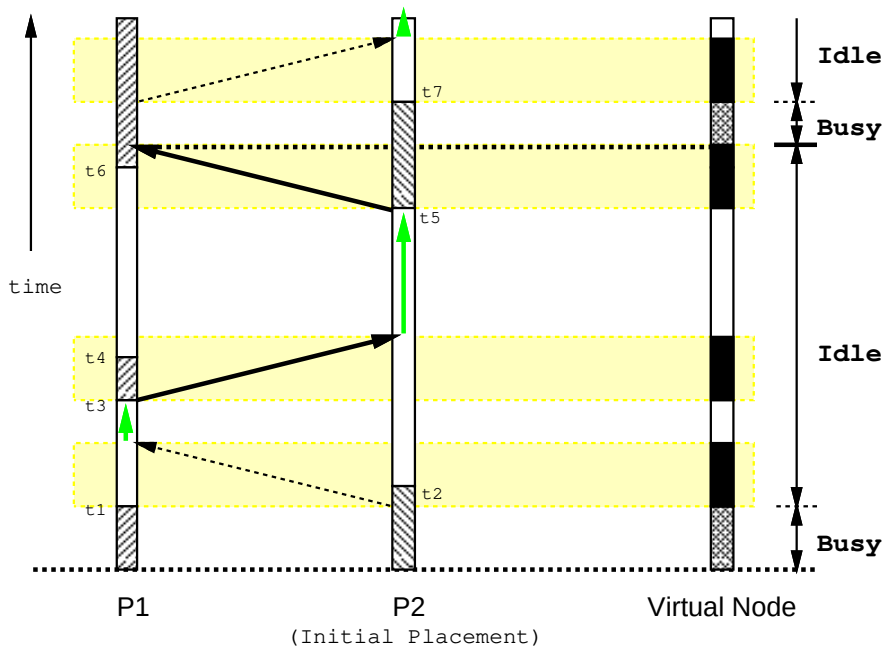


Figure 6.6: The *MIG_D* Protocol

rectangular band. Only white periods are effectively available to execute *sp* because black periods represent migration delay. Let A_k and O_k denote the sum of white periods and that of black periods in an idle period on a virtual node respectively when the number of participating processors is k. A_k is called an *available* time.

An idle period can start with a white or a black period depending on whether *sp* is positioned at the proper processor at the beginning of an idle period. Let X denote the number of migration except the migration pull in an idle period. Then a random variable X will be distributed as in Equation 6.1.

In this section, we are going to use $\tau_i/(\tau_i+2\tau_b)$ as an approximate probability of f-idleness when a preemption occurs. We will show that this approximation is good so that our analytic results and simulations are very close. Busy periods and idle periods alternate on a virtual node, and a busy period and the following

idle period form a renewal period. Since the available time in a renewal period can be used to execute sp, the stretching ratio will be $1 + (B_k + O_k)/A_k$. Note that $I_k = A_k + O_k$. When $D = 0$, $I_k = A_k$ because $O_k = 0$.

Migration may not be successful because of the migration delay. When migration is not successful, sp cannot run after migration but may have to migrate more than once without having any chance to run. Thus we need to know what is the probability of migration success. We are mostly interested in the case where $D < \tau_b$ because a f-idle period is highly likely to be available locally if $D > \tau_b$. Even if $D < \tau_b$, the migration target may become f-busy before a migration completes.

When the hosting processor preempts sp at time `r`, it checks whether the other processor is f-idle or not. If so, the f-idle period must have started before `r`. Let I be a random variable which represents a residual f-idle period at time `r`. Because of the memoryless property of an exponential distribution, the probability of migration success is $P[I > D] = e^{-D/\tau_i}$.

The mean of available time in an idle period, $\overline{A_2}$, is given by:

$$\overline{A_2} = \frac{\tau_i}{2} + \frac{\tau_i}{2} P[I > D] + \tau_i \overline{X} P[I > D].$$

The first term accounts for the case that sp is positioned at the proper processor in the beginning of an idle period and this probability is 1/2 in the two processor case. The second accounts for the case that sp is positioned at the improper processor. The mean number of migration push (not including a migration pull) is $\overline{X} = s/(1-s) = \tau_i/2\tau_b$, and the last term accounts for the accumulated f-idle periods on $P1$ and $P2$ in an idle period. The second and third terms are multiplied by $P[I > D]$ because sp has the chance to run only when migrations

are successful. Thus

$$\overline{A_2} = \frac{\tau_i}{2}[1 + (1 + \frac{\tau_i}{\tau_b})e^{-D/\tau_i}].$$

Here $\overline{B_2} = \tau_b/2$ as in the *MIG_0* protocol, and the mean of the sum of migration delays $\overline{O_2}$ in an idle period is

$$\overline{O_2} = \frac{D}{2} + \overline{X}D = \frac{D}{2}(1 + \frac{\tau_i}{\tau_b}).$$

The first term accounts for the migration pull, and the second term accounts for the migration delay for each migration push regardless of its success or failure.

Thus the stretching ratio of the *MIG_D* protocol on two processors is

$$SR_{MIG_D}(2) = 1 + \frac{\overline{B_2} + \overline{O_2}}{\overline{A_2}} = 1 + \frac{\tau_b + \omega D}{\tau_i(1 + \omega e^{-D/\tau_i})}$$

where $\omega = 1 + \tau_i/\tau_b$.

$$\lim_{D \to 0} SR_{MIG_D}(2) = \frac{(\tau_i + \tau_b)^2}{\tau_i(\tau_i + 2\tau_b)} = SR_{MIG_0}(2).$$

When D approaches 0, the performance of the *MIG_D* approaches that of the *MIG_0* as we expected.

The speedup of the *MIG_D* protocol on two processors over a single processor is

$$SP_{MIG_D}(2) = \frac{1 + \tau_b/\tau_i}{SR_{MIG_D}(2)} = \frac{1 + \tau_b/\tau_i}{1 + (\tau_b + \omega D)/[\tau_i(1 + \omega e^{-D/\tau_i})]}. \qquad (6.8)$$

6.4.2 *COMP_D* Protocol

We assumed that all clones are frozen during a state propagation, and this freezing is shown by a rectangular band in Figure 6.7. Thus $P1$ cannot start to execute sp_1 at `t4`. In fact the first state propagation harms the performance of the process

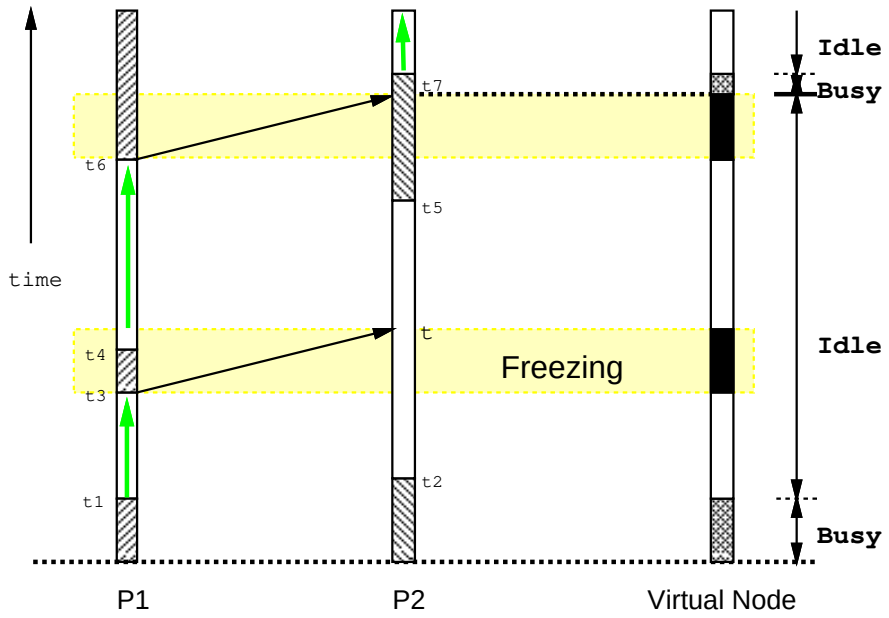


Figure 6.7: The *COMP_D* Protocol

sp. Both sp_1 and sp_2 resume execution at the same time `t`. Thus they will make the same progress at `t5` when $P2$ preempts sp_2. However, we assume that competition protocols break the tie for sp_1 because sp_1 is still making progress while sp_2 stops making progress. If competition protocols allow $P2$ to propagate sp_2's state at `t5`, both sp_1 and sp_2 cannot make any progress from `t5` to `t6` because of freezing.

In competition protocols, an idle period always starts with a white period as long as state propagation is finished before any f-idle periods on participating processors start. The reason is that a clone is always positioned at the proper processor at the beginning of an idle period. Thus competition protocols do not have the delay similar to migration pull in migration protocols.

We say that a state propagation is successful when a state propagation is completed in a f-idle period on one of processors and thus *sp* can make its progress

without any interruption other than propagation delay. This occurs when the f-busy period at the propagating processor is smaller than D or the other processor is f-idle when a state propagation is completed. We are going to use $s = \tau_i/(\tau_i + 2\tau_b)$ as the probability that the other processor is f-idle when a state propagation is completed. Then the probability that a state propagation succeeds is given by

$$s_1 = 1 - e^{-D/\tau_b} + e^{-D/\tau_b}\frac{\tau_i}{\tau_i + 2\tau_b},$$

and the probability that a state propagation fails is $f_1 = 1 - s_1$.

Let Y be the number of state propagation in an idle period, and then the number of white periods in an idle period is the same as that of state propagation. Thus the probability mass function of Y is given by the geometric distribution in Equation 6.6 where $s = s_1$ and $f = f_1$. Then

$$\overline{Y} = 1/f_1 = (1 + \frac{\tau_i}{2\tau_b})e^{D/\tau_b}.$$

Now let us focus on two processors. When the state propagation is completed, two clones may start execution at the same time, for example, at `t` in Figure 6.7. But only the clone with the longer f-idle period can propagate its state. Thus the corresponding segment here is not the typical f-idle period, but the maximum of two f-idle periods. When one clone starts earlier than the other, the corresponding segment is distributed as the typical f-idle period.

Two clones start at the same time when the f-busy period on the state propagator is smaller than D and the other processor is f-idle when a state propagation is completed. This probability is

$$b = (1 - e^{-D/\tau_b})\frac{\tau_i}{\tau_i + 2\tau_b}.$$

The probability that one clone starts earlier than the other is

$$c = e^{-D/\tau_b}\frac{\tau_i}{\tau_i + 2\tau_b} + (1 - e^{-D/\tau_b})\frac{2\tau_b}{\tau_i + 2\tau_b}.$$

In order to find the mean of the sum of available time in an idle period, $\overline{A_2}$, we need to know what is the probability that a segment in an idle period is the maximum of two f-idle periods and what is the probability that a segment is simply a typical f-idle period. Since

$$b + c = \frac{1}{\tau_i + 2\tau_b}[\tau_i + 2\tau_b(1 - e^{-D/\tau_b})] < 1,$$

we need to normalize b and c. Let $b' = b/(b + c)$ and $c' = c/(b + c)$. Then

$$b' = \frac{\tau_i(1 - e^{-D/\tau_b})}{\tau_i + 2\tau_b(1 - e^{-D/\tau_b})}.$$

$$c' = \frac{\tau_i e^{-D/\tau_b} + 2\tau_b(1 - e^{-D/\tau_b})}{\tau_i + 2\tau_b(1 - e^{-D/\tau_b})}.$$

Note that one clone always starts earlier than the other in the beginning segment in an idle period. The mean of the maximum of lengths of two exponentially distributed f-idle periods is $3\tau_i/2$ [ABN92]. Thus the mean of the sum of available time in an idle period is

$$\overline{A_2} = \tau_i + (\overline{Y} - 1)(\frac{3\tau_i}{2}b' + \tau_i c').$$

By replacing b', c', and $\overline{Y}$ in the equation above, we get

$$\overline{A_2} = \tau_i + \tau_i\{(1 + \frac{\tau_i}{2\tau_b})e^{D/\tau_b} - 1\}[\frac{(1 - e^{-D/\tau_b})(3\tau_i/2 + 2\tau_b) + \tau_i e^{-D/\tau_b}}{\tau_i + 2\tau_b(1 - e^{-D/\tau_b})}].$$

As in *COMP_0* protocol, $\overline{B_2} = \tau_b/2$, and

$$\overline{O_2} = D\overline{Y} = D(1 + \frac{\tau_i}{2\tau_b})e^{D/\tau_b}.$$

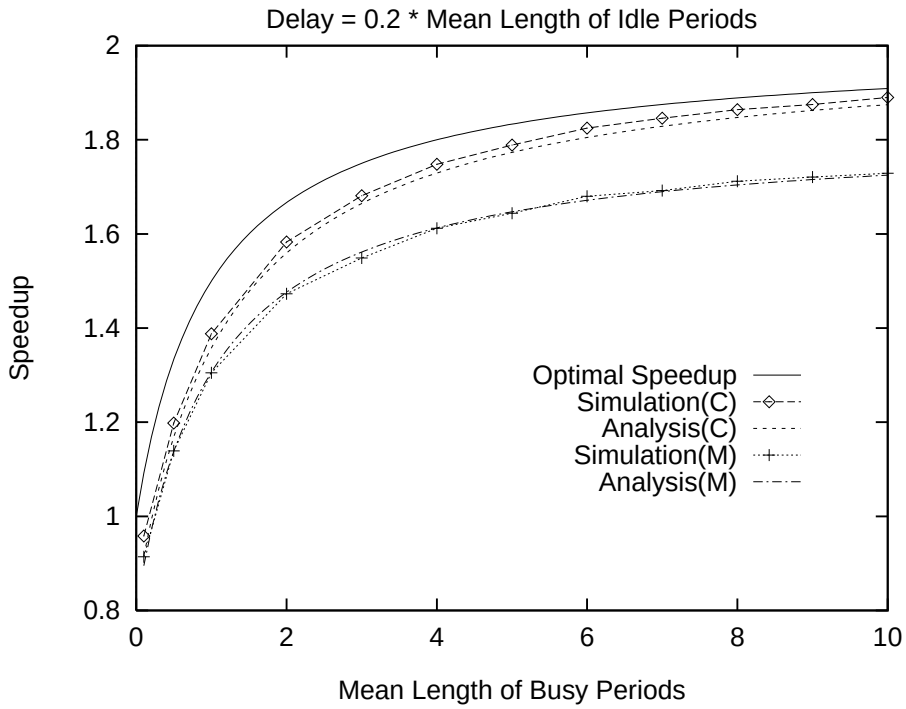


Figure 6.8: Speedup with Two Processors When $D = 0.2\tau_i$

Thus

$$SR_{COMP_D}(2) = 1 + \frac{\overline{B_2} + \overline{O_2}}{\overline{A_2}}.$$

$$\lim_{D \to 0} SR_{COMP_D}(2) = \frac{(\tau_i + \tau_b)^2}{\tau_i(\tau_i + 2\tau_b)} = SR_{COMP_0}(2).$$

When D approaches 0, the performance of the *COMP_D* approaches that of the *COMP_0* as we expected.

$$SP_{COMP_D}(2) = \frac{1 + \tau_b/\tau_i}{SR_{COMP_D}(2)}. \tag{6.9}$$

6.4.3 Simulation Results

We simulated *MIG_D* and *COMP_D* protocols to validate our analysis. In all of our simulations τ_i was set to 1.0, and τ_b (mean of f-busy periods) and the

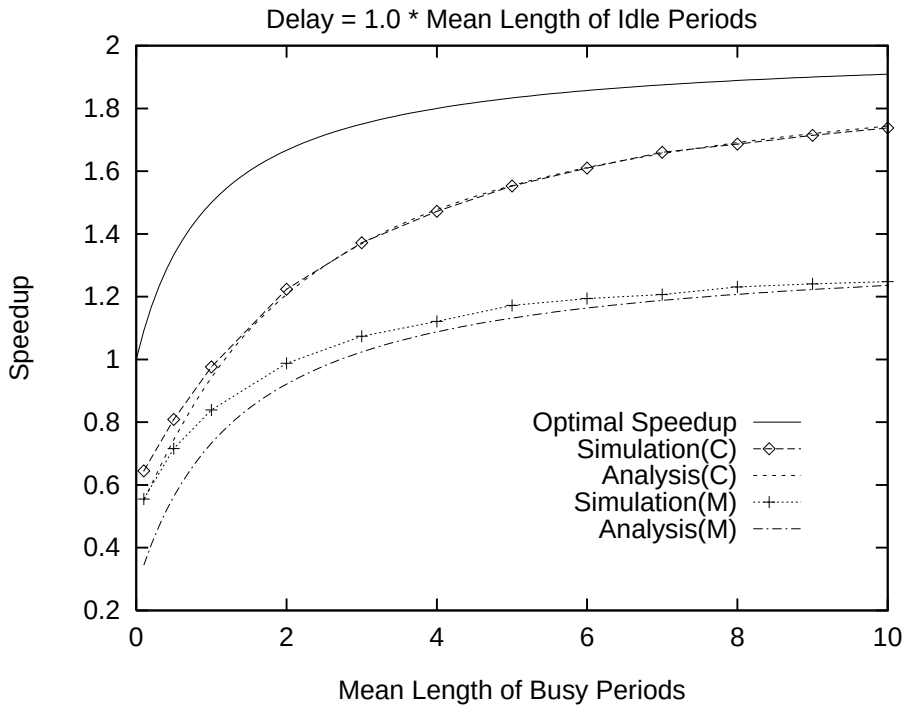


Figure 6.9: Speedup with Two Processors When $D = \tau_i$

delay for migration or state propagation are represented in multiples of τ_i. We simulated 20 times for a period of $1000\tau_i$ and our simulation results are means of 20 simulation results. Figure 6.8 and Figure 6.9 show our simulation results on two processors when the delay for migration or state propagation is $0.2\tau_i$ and $1.0\tau_i$ respectively.

In both figures, the curves labeled as 'Simulation(M)' and 'Simulation(C)' are simulation results of *MIG_D* and *COMP_D* protocols respectively. The curves labeled as 'Analysis(M)' and 'Analysis(C)' are our analytic results assuming the probability of migration or the probability of the success of state propagation is $\tau_i/(\tau_i + 2\tau_b)$. The curve labeled as 'Optimal Speedup' is $1 + \rho$ on two processors.

Because of the delay for migration or state propagation, migration and state propagation can deteriorate the completion time of applications for the small

mean length of busy periods; f-idle periods may be locally available before migration or state propagation is completed. When $D = 0.2$, Figure 6.8 shows that *MIG_D* and *COMP_D* protocols improve the performance for the most of the region. However, when $D = 1.0$, Figure 6.9 shows that speedups of *MIG_D* and *COMP_D* protocols are larger than one when $\tau_b > 2.8$ and $\tau_b > 1.0$ respectively. We are interested in the cases $\tau_b > 0.2 * \tau_i$ in Figure 6.8 and $\tau_b > 1.0 * \tau_i$ in Figure 6.9 where the delay for migration or state propagation is smaller than the mean of f-busy periods. In those regions our simulation results and analytic results are close.

However, there is significant discrepancy between our simulation results and analytic results for the region where the mean of f-busy periods is smaller than the delay for migration or state propagation, especially for *MIG_D* protocol; this occurs because we use $\tau_i/(\tau_i + 2\tau_b)$ as an approximate probability of migration. In Equation 6.8, $\omega = 1 + \tau_i/\tau_b$. When τ_b approaches 0, the error in the approximate probability of migration will be magnified for a given τ_i. The discrepancy for *COMP_D* protocol is smaller than that for *MIG_D* protocol because the term $(1 + \tau_i/2\tau_b)$ in $\overline{A_2}$ and $\overline{O_2}$ in *COMP_D* protocol is less sensitive in errors than $\omega = 1 + \tau_i/\tau_b$ in *MIG_D* protocol, and e^{-D/τ_b} in $\overline{A_2}$ approaches 0 when τ_b approaches 0.

When τ_b increases for a given D as shown in both Figure 6.8 and Figure 6.9, the performance difference between *MIG_D* protocol and *COMP_D* protocol also increases; the reason is that the frequency of migration pulls in *MIG_D* protocol increases when τ_b increases and the overhead of migration pulls become dominating, whereas *COMP_D* protocol does not have such an overhead.

When D increases, the overhead of migration pulls also increases and the

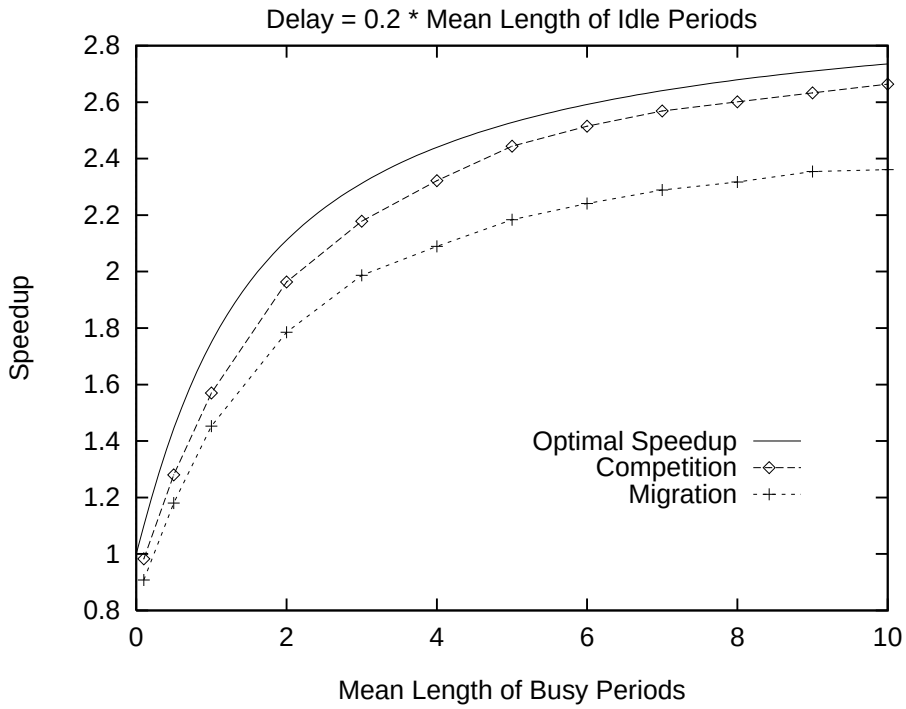


Figure 6.10: Speedup with Three Processors When $D = 0.2\tau_i$

probability of migration success decreases. When a process migration fails, a process may have to migrate more than once without the chance of running in *MIG_D* protocol. The probability of state propagation success also decreases, but each state propagation is guaranteed to improve the performance of applications because *COMP_D* protocol propagates the state of a clone that is ahead to other clones that is behind. Thus the performance difference between *MIG_D* protocol and *COMP_D* protocol in Figure 6.9 is larger than that in Figure 6.8.

Figure 6.10 and Figure 6.11 show our simulation results on three processors when the delay for migration or state propagation is $0.2\tau_i$ and $1.0\tau_i$ respectively. The shape of curves in both of the figures are similar to that in Figure 6.8 and Figure 6.9 respectively.

Figure 6.12 shows our simulation results when $\tau_b = \tau_i$, that is, $\rho = 0.5$.

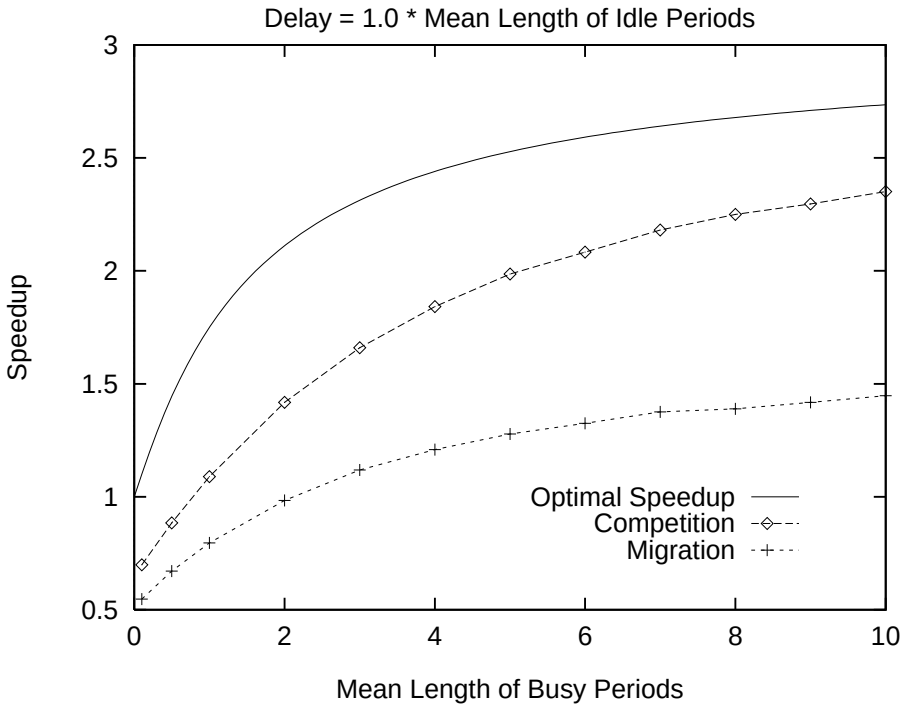


Figure 6.11: Speedup with Three Processors When $D = \tau_i$

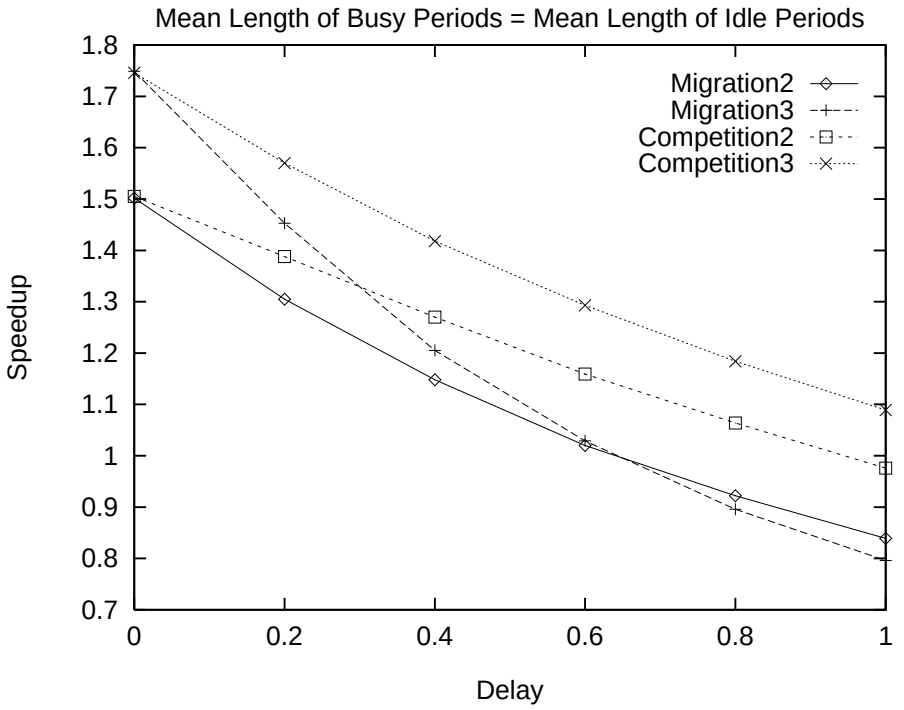


Figure 6.12: Speedup with Two and Three processors When $\tau_b = \tau_i$

When D (delay for migration or state propagation) increases, the performance of *MIG_D* protocol deteriorates faster than that of *COMP_D* protocol because the overhead of migration pull dominates and the probability that migration success decreases. When D is sufficiently large, Figure 6.12 shows that competition on two processors may perform better than migration among three processors, and that migration among two processors may perform better than migration among three processors because of the large delay for migration.

CHAPTER 7

Competition and Migration for Distributed Programs

7.1 Introduction

In this chapter we consider the performance of competition protocols for distributed programs. First, we consider the question "What is the maximum speedup competition protocols can achieve?" by simulating the completion time of three distributed programs. Our simulation results show that competition protocols can achieve more speedups for distributed programs than for sequential programs. Then we compare the performance of competition protocols and the performance of migration protocols for distributed programs. Finally, we consider the effect of load imbalance and granularity on the performance of competition protocols.

A simulator was built in order to evaluate the performance of competition protocols and migration protocols. It simulates a SNOW environment with competition protocols or migration protocols on a sequential processor. The simulator was written in the language Maisie [BL93].

The simulator is composed of three layers as described in Chapter 4; the application layer, competition layer, and kernel layer. The application layer consists of a set of processes that communicate each other via messages. In the application

layer, the fact that processes may be cloned over a SNOW is hidden.

The competition layer implements competition protocols including detecting which clone of each process is ahead and making decision about state propagation. The kernel layer simulates variable-speed processors in a SNOW environment and message communication between processors.

7.1.1 The Application Layer

There are so many different kinds of distributed programs, but we consider only a limited set of distributed programs with full connectivity, 2D mesh connectivity, and ring connectivity. Assume that a distributed program consists of k processes. Then the numbers of neighbors in those topologies are $k-1$, 4, and 2 respectively.

We assume that each process in a distributed program alternates computation phases and communication phases when processes exchange messages with their neighbors, and that each process starts with a computation phase. After each process sends messages to all of its neighbors, it waits for messages from all of its neighbors. A process resumes execution only after it receive messages from all of its neighbors. We assume that message communication is bidirectional except one case; we consider a unidirectional ring to see the effect of the number of neighbors on the performance of competition protocols.

The time a process needs to complete a computation phase on a dedicated processor is called the *granularity* of the process, which is denoted by G. We assume that the granularity of all processes in a given distributed program is identical except one case; we consider a case where the granularity of a process is larger than that of the other processes in order to see the effect of load imbalance among processes on the performance of competition protocols.

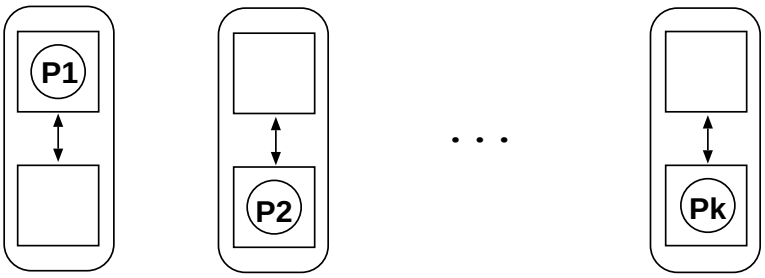


Figure 7.1: Static Boundary

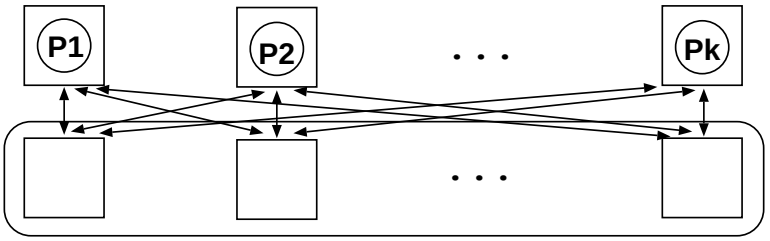


Figure 7.2: Dynamic Boundary

We assume that the latency for message communication is zero because that latency will be relatively small compared with the latency for state propagation.

7.2 Some Issues in Our Simulation

Our simulation model for distributed programs is similar to the simulation model for sequential programs except message communication. We consider two cases; the delay for migration or state propagation is zero and the delay is a positive constant.

7.2.1 Static and Dynamic Boundary for Migration

Processes in a distributed program may be allocated to different processors for parallel execution, and any such allocation is called a *configuration* of the pro-

gram. Assume that a distributed program consists of k processes and there are $2k$ variable-speed processors available to execute the program.

We can consider two ways to execute the program on $2k$ processors. The first is to allocate two processors to each process so that a process is allowed to migrate between only those two processors in migration protocols. Figure 7.1 shows the static boundary for migration. Squares represent processors, circles represent processes, and rectangles with round corners represent boundaries processes can migrate.

The second is to allocate one processor to each process in migration protocols, and the remaining k processors are used as a pool of processors where each process can migrate to. Figure 7.2 shows the dynamic boundary for migration. Whenever a process migrates, the configuration of the program and the pool of processors change.

Competition protocols can also change the configuration of each process dynamically by uncloning some clones and cloning them on other processors, especially when peer clones of a process do not offer any competition advantage, e.g., one processor is always faster than the other processors or their execution speeds are identical.

In general, a dynamic strategy offers the better performance than a static strategy in both migration and competition when the communication delay between any pair of processors is identical. However, a dynamic strategy requires complicated policies to make process migration decision in migration protocols and to make uncloning and cloning decision in competition. This is beyond the scope of our study, and we consider only a static strategy in this chapter.

7.2.2 A Migration Policy and a Competition Policy

Sequential processes can have only two states; running or preempted. In addition to those two states, however, processes in a distributed program can have one more state, that is, waiting for messages. If a message clone arrives at a f-busy processor while a process clone is in a 'waiting' state, our migration policy migrates the process clone with the message clone to another f-idle processor if one is available. Otherwise, a processor that becomes f-idle earlier than the other processor migrates the process to execute on it.

Our competition policy propagates a state of a process clone only when the process clone is ahead of the other. When there is a positive latency for state propagation, however, it should propagate states only when a process clone is sufficiently ahead of the other peer process clones. We can optimize our competition protocols in this direction, but we do not optimize any of competition protocols because this kind of optimization is not available in migration protocols.

7.2.3 Assumptions in Our Simulation

Let I and B denote the mean length of f-idle periods and the mean length of f-busy periods respectively. In this chapter, I and τ_i are used interchangeably, and B and τ_b are also used interchangeably.

In all of our simulations, τ_i was set to 1.0, and all the other delays (e.g., granularity, the mean length of f-busy periods, and the delay for migration or state propagation) are represented in multiples of τ_i. We simulated 10 times each until processes in a distributed program exchange messages with their neighbors 1000 times, and our simulation results are the means of 10 simulation results.

We assumed that the latency for message communication is zero, and thus the configuration of a program does not have any performance effect in this study. We also assumed that each clone runs on its own processor.

7.3 Speedup of Competition Protocols

The performance measures would ideally be obtained by observing an implementation of competition protocols running on an actual SNOW, but building such a system was beyond the scope of this study. We consider simple distributed programs with the following three connectivities; full connectivity, 2D mesh connectivity, and ring connectivity.

We simulated the execution of each of these programs on a SNOW. Our performance measure is the speedup achieved by competition protocols for each program. The speedup is the ratio of each program's completion time without competition to the program's completion time with competition on a SNOW. We assume that there are enough processors that are available to execute each clone on its own processor.

7.3.1 Programs with Full Connectivity

We consider a distributed program consisting of 10 fully connected processes first. We assume that all processes are uniformly cloned two, three, or four times, and that the granularity (the mean execution time between communication phases) of processes is $G = 2\tau_i$, that is, $G = 2$ since $\tau_i = 1.0$.

Let the following parameters describe the system.

B – the mean length of busy periods on processors

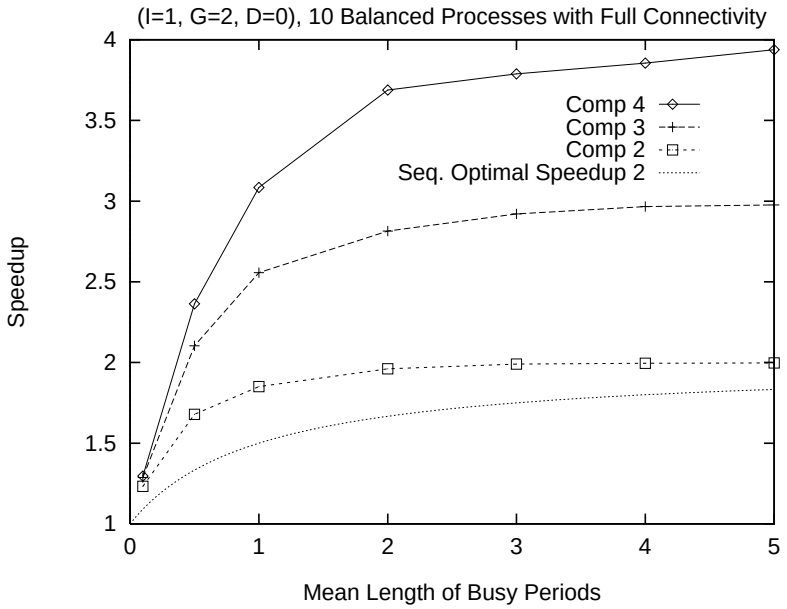


Figure 7.3: 10 Processes with Full Connectivity When D = 0

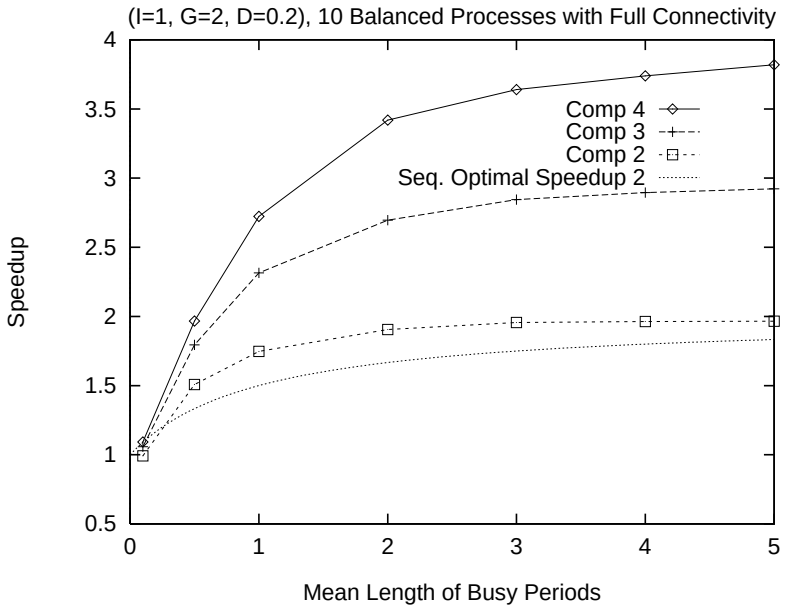


Figure 7.4: 10 Processes with Full Connectivity When D = $0.2\tau_i$

C – the number of clones for processes

D – the delay for process migration or state propagation

First we present the performance of competition protocols when $D = 0$ in Figure 7.3. The curve labeled as 'Comp i' represents the speedup of competition when all the processes are cloned i times, and the curve labeled as 'Seq. Optimal Speedup 2' represents the optimal speedup for sequential programs on two processors, that is, $1 + \rho$. The speedup curves increases monotonically when the mean length of f-busy periods increases as we have shown for sequential processes in Chapter 6. Figure 7.3 shows that we can achieve more speedups for the distributed program with 10 fully connected processes than for sequential programs because the optimal speedup for sequential programs is the lower bound for the distributed program.

Now we present speedup curves when $D = 0.2\tau_i$ in Figure 7.4. We can observe that the speedup when $D = 0.2\tau_i$ is smaller than the speedup when $D = 0$ because of the delay for state propagation. For the most region of the x axis, the speedup of competition for the distributed program with 10 fully connected processes on two processors when $D = 0.2\tau_i$ is even larger than the optimal speedup for sequential programs on two processors when $D = 0$.

Figure 7.5 and Figure 7.6 shows the speedups of competition protocols and migration protocols on three processors respectively. When $D = 0$, it is shown that competition protocols and migration protocols achieve the same speedup as we expected. However, when D increases, the speedups of migration protocols deteriorate quicker than those of competition protocols. These phenomena have been observed for sequential processes, too.

Figure 7.7 shows the speedups of competition protocols and migration pro-

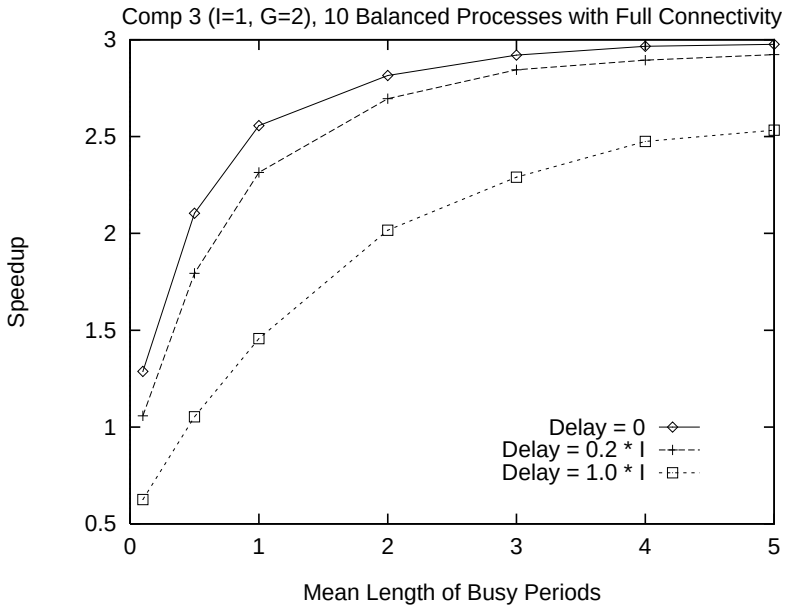


Figure 7.5: Speedup of Competition on Three Processors

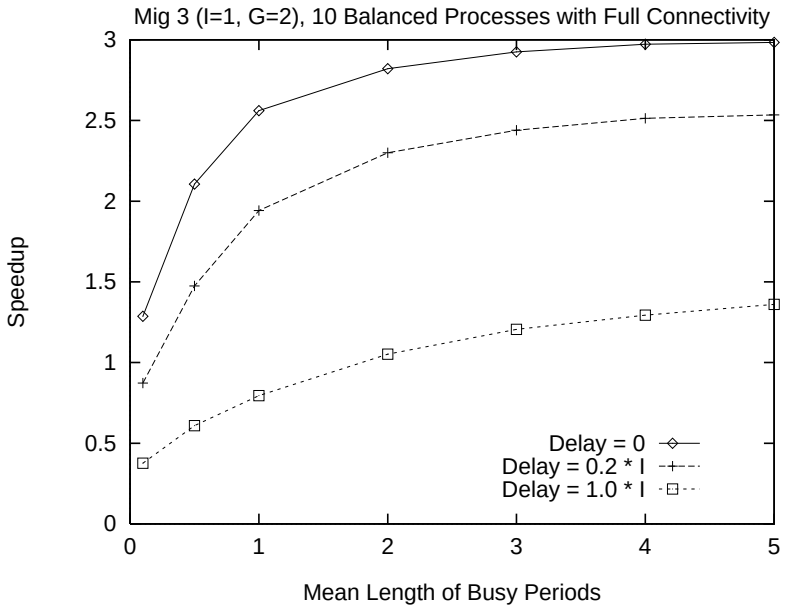


Figure 7.6: Speedup of Migration on Three Processors

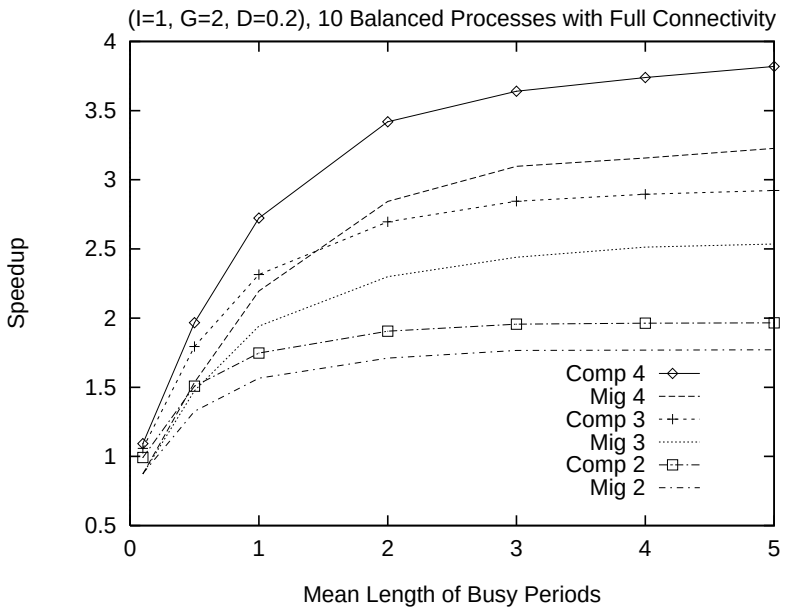


Figure 7.7: Comparison of Competition and Migration When D = $0.2\tau_i$

tocols with different number of processors when $D = 0.2\tau_i$. The speedups of competition protocols are consistently greater than those of migration protocols with the same number of participating processors.

Figure 7.8 shows the effect of the number of processes with full connectivity on the speedup competition protocols can achieve. When the number of processes increases, competition protocols can achieve more speedups because the probability that one of them executes on a slow processor without competition increases.

Now let us consider the effect of granularity on the performance of competition protocols. Figure 7.9 shows that the speedup of competition protocols decreases when the granularity increases. When the granularity is sufficiently large, the speedup curves approaches the optimal speedup of sequential processes. The reason is that all processes tend to be fully parallel processes without communication, and that the execution time of the whole program approaches that of any

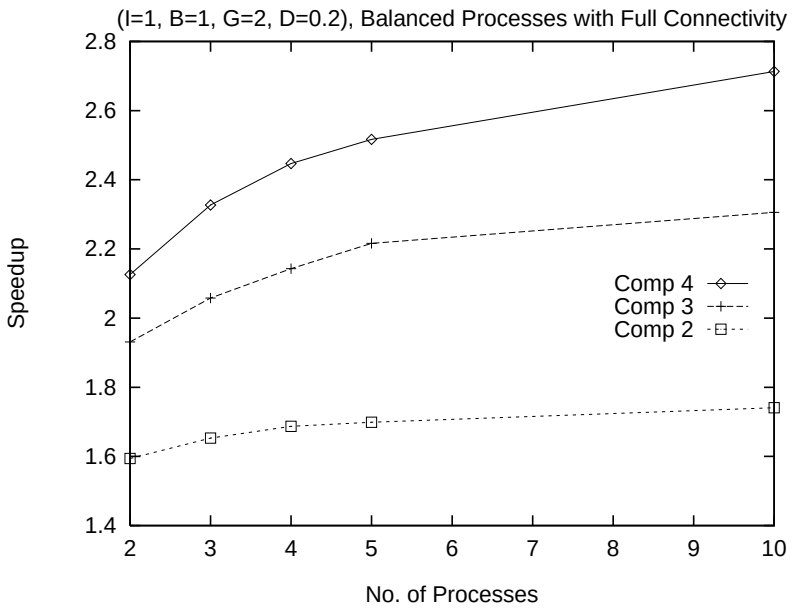


Figure 7.8: The Effect of the Number of Processes on the Speedup

single process.

We will consider the effect of load imbalance among processes in Figure 7.10 where one process has $G = 10\tau_i$ and all the others have $G = 2\tau_i$. The curve labeled as 'Uniform Comp' shows the performance of competition protocols where the degree of competition of all processes are identical. The curve labeled as 'Selective Comp' shows the performance of competition protocols where only the process whose granularity is 10 is cloned. This figure shows that we may not need to clone all processes in a distributed program but only critical processes.

7.3.2 Programs with Mesh Connectivity

Now we consider a distributed program consisting of 9 processes with 2D mesh connectivity. Figure 7.11 shows the performance of competition protocols when $D = 0.2\tau_i$. As in 10 fully connected processes, for the most region of the x axis, the speedup of competition for 9 processes with 2D mesh connectivity when

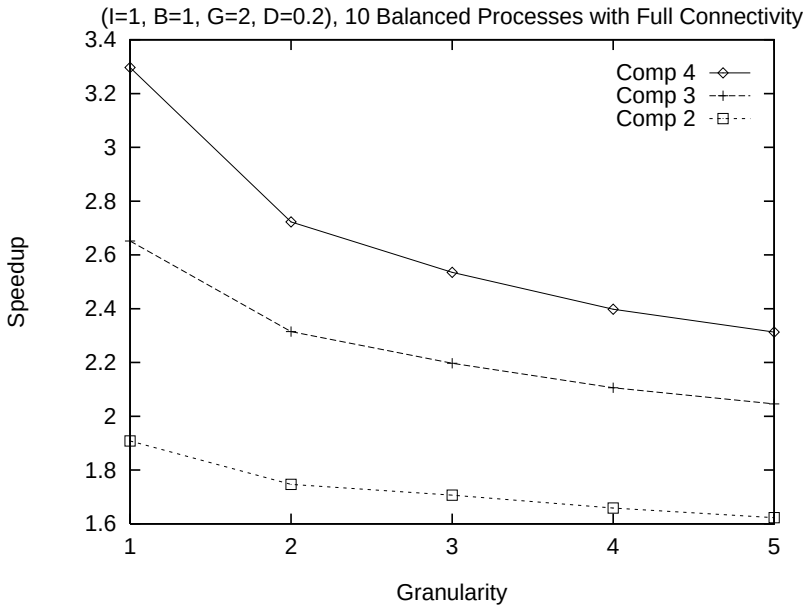


Figure 7.9: The Effect of Granularity on the Speedup

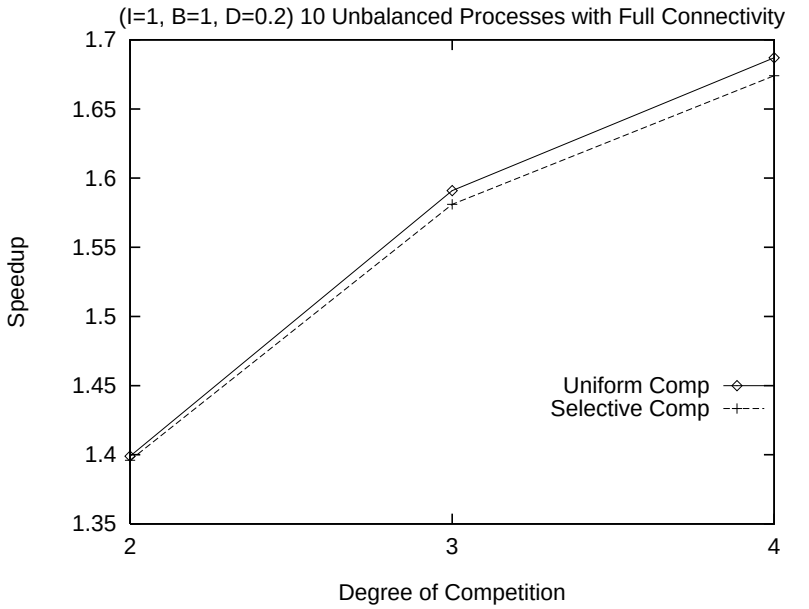


Figure 7.10: The Effect of Load Imbalance on the Speedup

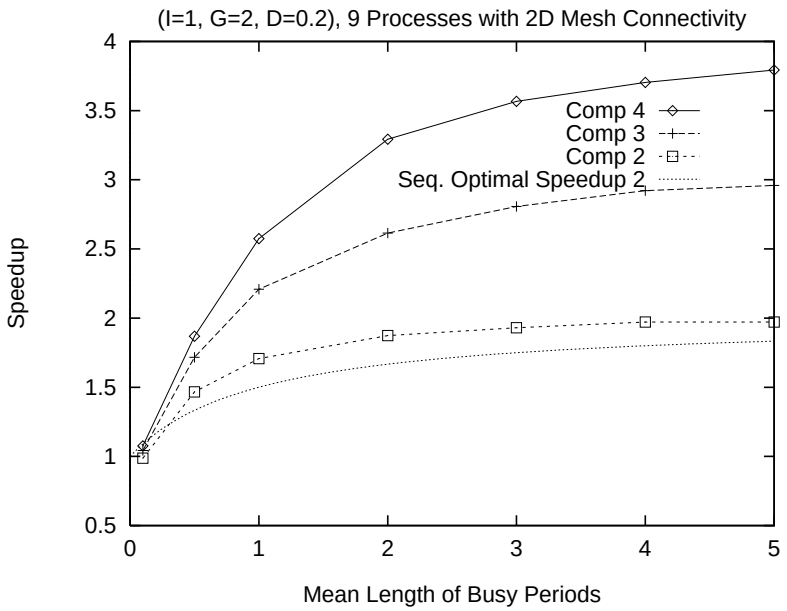


Figure 7.11: 9 Processes with 2D Mesh Connectivity When D = $0.2\tau_i$

$D = 0.2\tau_i$ is even larger than the optimal speedup for sequential programs on two processors when $D = 0$.

The shapes of speedup curves are similar to those for 10 fully connected processes as shown in Figure 7.4. However, we can observe that the speedups for 10 fully connected processes are greater than those for 9 processes with 2D mesh connectivity. The reason is that processes in the former program has stronger dependency among processes because they have more neighbors than processes in the latter program.

All of our simulation results for 9 processes with 2D mesh connectivity consistently show that the shapes of their speedup curves are similar to those of 10 fully connected processes. Thus we will not present our simulation results of 9 processes with 2D mesh connectivity any more.

However, we consider two more distributed programs consisting of 4 processes and 16 processes with 2D mesh connectivity respectively. Their speedups are

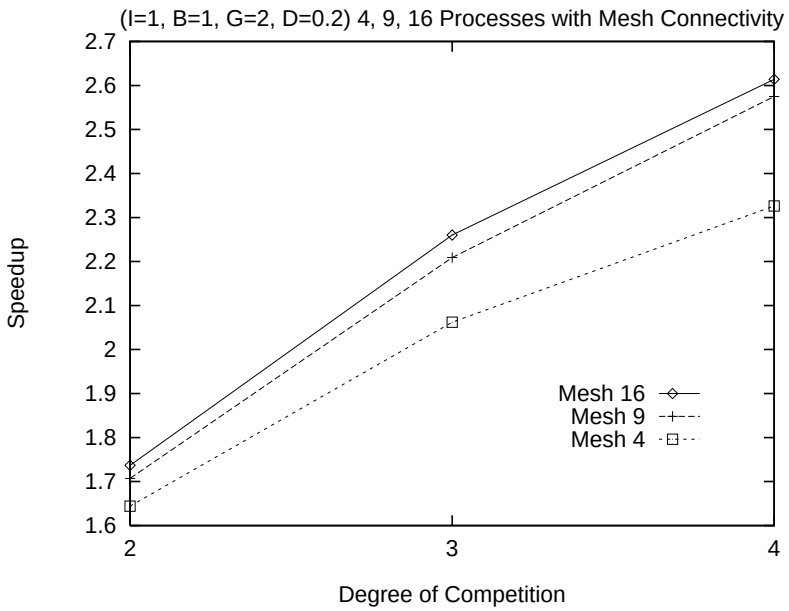


Figure 7.12: 4, 9, and 16 Processes with 2D Mesh Connectivity When D = $0.2\tau_i$

shown in Figure 7.12. All processes have the same number of neighbors, but processes in a program with 16 processes are indirectly dependent upon more number of other processes. Thus competition protocols can achieve more speedup for the program with 16 processes than the program with 4 or 9 processes with the same 2D mesh connectivity.

7.3.3 Programs with Ring Connectivity

We also consider a distributed program with 10 processes with ring connectivity. Figure 7.13 shows the performance of competition protocols with different degrees of competition when $D = 0.2\tau_i$. The number of neighbors of processes in a undirected ring is two while that in a directed ring is one. Thus competition can speed up the program with a undirected ring connectivity more than that with a directed ring connectivity.

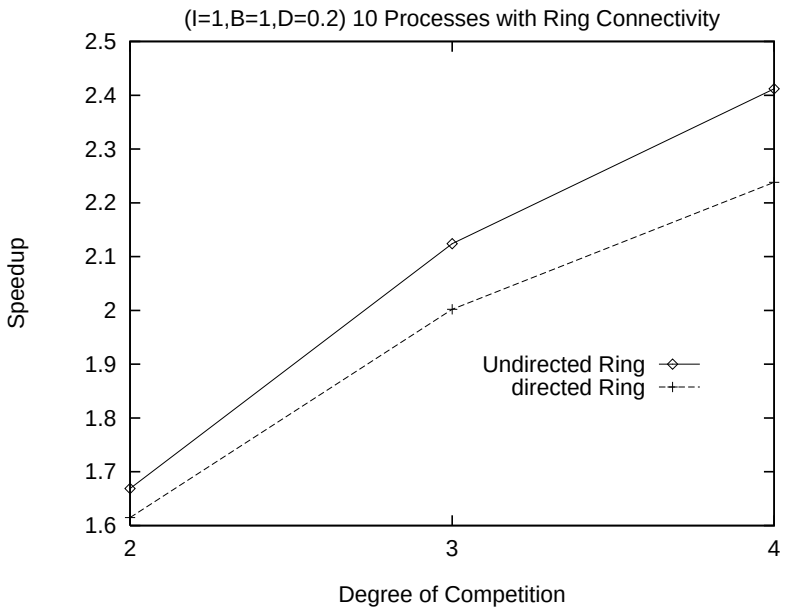


Figure 7.13: 10 Processes with Ring Connectivity When D = $0.2\tau_i$

7.4 Conclusion

In order to understand more about speedups competition protocols can obtain for distributed programs with different topologies and different number of processes, we present simulation results of the following cases in Figure 7.14; 10 processes with full connectivity, 16 processes with 2D mesh connectivity, 9 processes with 2D mesh connectivity, 10 processes with undirected ring connectivity, 10 processes with directed ring connectivity, and 2 processes with full connectivity.

Processes in a 10 fully connected program has the strongest dependency among processes out of six cases. 10 processes with directed ring connectivity has the same number of direct neighbors as 2 fully connected processes, but processes in the former depends on others indirectly. Thus competition protocols can achieve more speedup for the former than for the latter.

In conclusion, when processes in a distributed program has stronger depen-

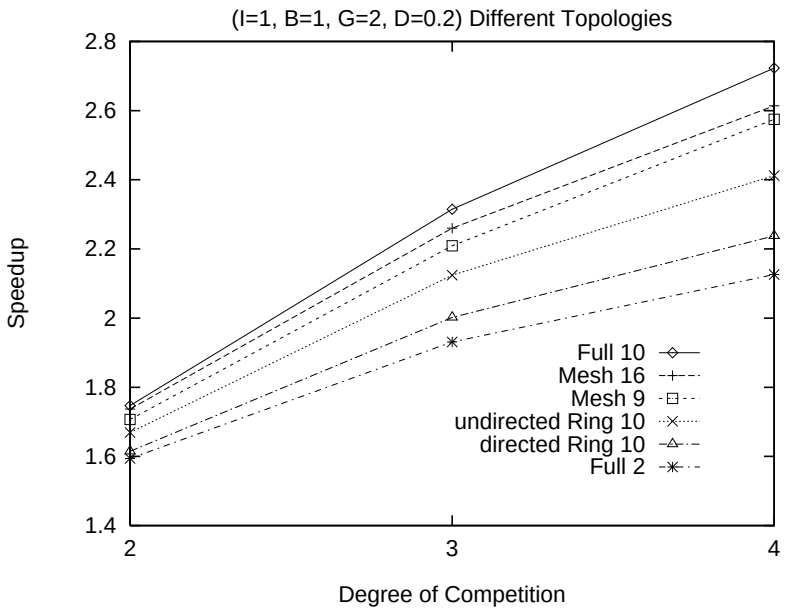


Figure 7.14: The Speedup with Different Topologies When D = $0.2\tau_i$

dency on others directly or indirectly, their progress are more likely to be hindered by other processes without competition. Competition protocols can achieve more speedup for a distributed program that has stronger dependency on others.

CHAPTER 8

Conclusions and Future Work

8.1 Conclusions

In this research, we have studied competitive execution in a distributed environment such as a network of workstations. Our analytic results and simulation results illustrate the ability of competitive execution to speed up the execution of distributed programs.

We showed that migration cannot achieve the optimal performance without the complete knowledge of the future foreground activities. However, competition does not need to know the future foreground activities to achieve the optimal performance. Competition easily obtains this benefit at the cost of redundant execution.

We presented competition protocols with state propagation and without state propagation, and showed that nonpropagating competition protocols have a limited performance benefit.

We also presented competition mechanisms which deals with message multicasting and state propagation among peer process clones in addition to message communication between clones of communicating processes. We showed that competitive execution can finish distributed programs significantly faster than noncompetitive execution, especially when foreground load on each processor is

sufficiently high.

We derived the closed form solution for the performance of an ideal competition protocol, and extensively studied the performance of competition protocols for a sequential process based on mathematical analysis and simulation.

It was shown that competition protocols offer performance benefits that are as good as or better than migration protocols for sequential or distributed programs under comparable assumptions. This conclusion is supported by our analytic results and simulation results for sequential programs.

Finally we presented the performance of competition protocols for distributed programs based on simulation. Our simulation results showed that competition protocols offer even more speedups for distributed programs than for sequential programs.

8.2 Future Work

We have by no means finished the subject of competition in the distributed context. In order to better understand the performance of competition for distributed programs, we intend to implement and/or simulate distributed programs that have different synchronization and communication requirement.

We did not study competition policy issues. However, they should be understood before real competitive systems are designed. We also need to study competitive mechanisms and policies for dynamic cloning and uncloning.

We also have to study more on implementation issues, e.g., when we consider that a processor is idle, how to minimize interference from background programs, etc. Preemptive scheduling for foreground processes may not be enough to insu-

late foreground processes from background programs. Memory management, file access, and network access must be redesigned to support competition.

References

[ABN92] Barry C. Arnold, N. Balakrishnan, and H. N. Nagaraja. *A First Course in Order Statistics.* John Wiley and Sons, Inc., 1992.

[ACP95] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW team. "A Case for NOW (Networks of Workstations)." *IEEE Micro*, pp. 54–64, February 1995.

[AE87] R. Agrawal and A. K. Ezzat. "Location Independent Remote Execution in NEST." *IEEE Transactions on Software Engineering*, **13**(8):905–912, 1987.

[AF89] Y. Artsy and R. Finkel. "Designing a Process Migration Facility: The Charlotte Experience." *Computer*, **22**(9):47–56, September 1989.

[AJ87] A.P.Goldberg and D. Jefferson. "Transparent Process Cloning: A Tool for Load Management of Distributed Programs." In *1987 Int'l Conf. on Parallel Processing*, pp. 728–734, 1987.

[AK84] Algirdas Avižienis and John Kelly. "Fault Tolerance by Design Diversity: Concepts and Experiments." *Computer*, **17**(8):67–80, August 1984.

[Amd67] G. Amdahl. "Validity of the Single-Processor Approach to Achieving Large-Scale Computer Capacities." In *AFIPS Conference Proceedings*, volume 30, pp. 483–485, 1967.

[Bir85] K. P. Birman. "Replication and fault tolerance in the ISIS system." In *Proc. 10th ACM Symp. on Operating Systems Principles*, pp. 79–86, 1985.

[BKT92] Henri E. Bal, M. Frans Kaashoek, Andrew Tanenbaum, and Jack Jansen. "Replication techniques for speeding up parallel applications on distributed systems." *Concurrency: Practice and Experience*, **4**(5):337–355, August 1992.

[BL93] Rajive Bagrodia and Wen-Toh Liao. *Maisie User Manual.* 1993.

[BM89] Kenneth Birman and Keith Marzullo. "Isis and the META project." *Sun Technology*, pp. 90–104, Summer 1989.

[Bok87] Sahid H. Bokhari. *Assignment Problems in Parallel and Distributed Computing*. Kluwer Academic Publishers, 1987.

[Boy87] J. Boyle et al. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.

[BSS91] K. P. Birman, A. Schiper, and P. Stephenson. "Lightweight casual and atomic group multicast." *ACM Transactions on Computer Systems*, **9**(3), August 1991.

[CC94] Jane-Ferng Chiu and Ge-Ming Chiu. "Process-Replication Technique for Fault-Tolerance and Performance Improvement in Distributed Computing Systems." In *Proceedings of the Third International Symp. on High Performance Distributed Computing*, pp. 236–243, San Francisco, California, August 2-5 1994.

[CG89] Nicholas Carriero and David Gelernter. "How to write parallel programs: A guide to the perplexed." *ACM Computing Surveys*, pp. 323–357, September 1989.

[Cha94] Chran-Ham Chang et al. "High-Performance TCP/IP and UDP/IP Networking in DEC OSF/1 for Alpha AXP." In *High Performance Distributed Computing*, pp. 35–42, San Francisco, California, August 2-5 1994.

[Che91] Li-Wen Chen. *A Model for Distributed Dynamic Progress Management*. PhD thesis, University of California, Los Angeles, 1991.

[CM84] Jo-Mei Chang and N. F. Maxemchuk. "Reliable Broadcast Protocols." *ACM Transactions on Computer Systems*, **2**(3), August 1984.

[Coo85] E. C. Cooper. "Replicated distributed programs." In *Proc. of the 10th ACM Symposium on Operating Systems Principles*, pp. 63–78, December 1985.

[Cox62] D. R. Cox. *Renewal Theory*. Methuen and Co., Ltd., London, 1962.

[DO91] Fred Douglis and John Ousterhout. "Transparent Process Migration: Design Alternatives and the Sprite Implementation." *Software - Practice and Experience*, **21**(8):757–785, August 1991.

[EBB95] Thorsten von Eicken, Anindya Basu, and Vineet Buch. "Low-Latency Communication Over ATM Networks Using Active Messages." *IEEE Micro*, pp. 46–53, February 1995.

[ELA94] H. El-Rewini, T. G. Lewis, and H. H. Ali. *Task Scheduling in Parallel and Distributed Systems.* Prentice Hall, 1994.

[ELZ86a] D. Eager, E. Lazowska, and J. Zahorjan. "Adaptive load sharing in homogeneous distributed systems." *IEEE transactions on Software Engineering*, **12**(5), May 1986.

[ELZ86b] D. Eager, E. Lazowska, and J. Zahorjan. "A comparison of receiver initiated and sender initiated dynamic load sharing." *Performance Evaluation*, **6**(1), April 1986.

[ELZ88] Derek L. Eager, Edward D. Lazowaska, and John Zahorjan. "The Limited Performance Benefits of Migrating Active Processes for Load Sharing." In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 63–72, Santa Fe, New Mexico, May 24-27 1988.

[EZ92] E. N. Elnozahy and Willy Zwaenepoel. "Replicated distributed processes in manetho." In *Proc. Conf. on Fault-Tolerant Computing Systems*, pp. 18–27, July 1992.

[Fat95] Rod Fatoohi. "Performance Evaluation of Communication Networks for Distributed Computing." Technical Report NAS-95-009, NASA Ames Research Center, March 1995.

[FKB91] J. Flower, A. Kolawa, and S. Bharadwaj. "The express way to distributed processing." *Supercomputing Review*, pp. 54–55, May 1991.

[FSK89] Robert E. Felderman, Eve M. Schooler, and Leonard Kleinrock. "The Benevolent Bandit Laboratory: A Testbed for Distributed Algorithms." *IEEE Journal on Selected Areas in Communications*, **7**(2):303–311, February 1989.

[Gei94] Al Geist et al. *PVM: Parallel Virtual Machine; A Users' Guide and Tutorial for Networked Parallel Computing.* The MIT Press, Cambridge, Massachusetts, 1994.

[GLS94] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI.* MIT Press, 1994.

[Gus88] John L. Gustafson. "Reevaluating Amdahl's Law." *Communications of the ACM*, **31**(5):532–533, May 1988.

[Hag86] Robert Hagmann. "Process Server: Sharing Processing Power in a Workstation Environment." In *6th International Conf. on Distributed Computing Systems*, pp. 260–267, Cambridge, Massachusetts, May 19-23 1986.

[Jef85] David Jefferson. "Virtual Time." *ACM Trans. on Programming Languages and Systems*, **7**(3):404–425, July 1985.

[JXT93] Jiubin Ju, Gaochao Xu, and Jie Tao. "Parallel Computing Using Idle Workstations." *Operating Systems Review*, **27**(3):87–96, July 1993.

[KC91] Phillip Krueger and Rohit Chawla. "The Stealth Distributed Scheduler." In *11th Int'l Conf. on Distributed Computing Systems*, pp. 336–343, Arlington, Texas, May 20-24 1991.

[KK89] Leonard Kleinrock and Willard Korfhage. "Collecting Unused Processing Capacity: An Analysis of Transient Distributed Systems." In *9th Int'l Conf. on Distributed Computing Systems*, pp. 482–489, 1989.

[KK92] Orly Kremien and Jeff Kramer. "Methodical Analysis of Adaptive Load Sharing Algorithms." *IEEE Transactions on Parallel and Distributed Systems*, **3**(6):747–760, November 1992.

[KL88] Phillip Krueger and Miron Linvy. "A Comparison of Preemptive and Non-Preemptive Load Distributing." In *8th Int'l Conf. on Distributed Computing Systems*, pp. 123–130, San Jose, California, June 13-17 1988.

[Kle75] Leonard Kleinrock. *Queueing Systems: Theory*. John Wiley and Sons, Inc., 1975.

[Kor89] Willard Robert Korflage. *Distributed Systems and Transient Processors*. PhD thesis, University of California, Los Angeles, 1989.

[KT91] M. F. Kaashoek and A. S. Tanenbaum. "Group communication in the amoeba distributed operating system." In *Int'l Conf. on Distributed Computing Systems*, pp. 222–230, 1991.

[Lam78] L. Lamport. "Time. Clock, and the Ordering of Events in a Distributed System." *Commun. ACM*, **21**:558–564, July 1978.

[LHD94] Mengjou Lin, Jenwei Hsieh, David H.C. Du, and Joseph P. Thomas. "Distributed Network Computing over Local ATM Networks." In

Proc. Supercomputing '94, pp. 154–163, Washington, D.C., November 14-18 1994.

[LLM88] M. J. Litzkow, M. Linvy, and M. W. Mutka. "Condor - A Hunter of Idle Workstations." In *8th Int'l Conf. on Distributed Computing Systems*, pp. 104–111, San Jose, California, June 13-17 1988.

[LM82] M. Linvy and M. Melman. "Load Balancing in Homogeneous Broadcast Distributed Systems." In *Proc. ACM Computer Network Performance Symposium*, pp. 47–55, April 1982.

[LS93] Scott T. Leutenegger and Xian-He Sun. "Distributed Computing Feasibility in a Non-Dedicated Homogeneous Distributed System." In *Proc. Supercomputing '93*, pp. 143–151, Portland, Oregon, November 15-19 1993.

[ML91] Matt W. Mutka and Miron Linvy. "The availability capacity of a privately owned workstation environment." *Performance Evaluation*, pp. 269–284, 1991.

[Nic87] David A. Nicols. "Using Idle Workstations in a Shared Computing Environment." *Operating Systems Review*, **21**(5):5–12, November 8-11 1987.

[Nut94] Mark Nuttall. "A brief survey of systems providing process or object migration facilities." *Operating System Review*, **28**(4):64–80, October 1994.

[NXG85] L.M. Ni, C.W. Xu, and T.B. Gendreau. "Drafting algorithm - A dynamic process migration protocol for distributed system." In *Proc. 5th Int'l Conf. on Distributed Computing Systems*, pp. 539–546, May 1985.

[Phi89] Chrétienne Philippe. *Task Scheduling over Distributed Memory Machines*, pp. 165–176. Elsevier Science Publishers B. V., North Holland, 1989.

[PW85] G. J. Popek and B. J. Walker. *The LOCUS Distributed System*. MIT Press, 1985.

[Sch90] Fred B. Schneider. "Implementing Fault-Tolerant Services Using the State Machine Approaches: A Tutorial." *ACM Computing Surveys*, **22**(4):299–319, December 1990.

[SG89] Jonathan M. Smith and Jr. Gerald Q. Maquire. "Exploring Multiple Worlds in Parallel." In *1989 Int'l Conf. on Parallel Processing*, pp. II–239–245, 1989.

[Sho91] G. C. Shoja. "A Distributed Facility for Load Sharing and Parallel Processing Among Workstations." *Journal of Systems and Software*, **14**:163–172, 1991.

[Smi88] Jonathan M. Smith. "A Survey of Process Migration Mechanisms." *Operating System Review*, **22**(3):28–40, July 1988.

[SN93] Xian-He Sun and Lionel M. Ni. "Scalable Problems and Memory-Bounded Speedup." *Journal of Parallel and Distributed Computing*, **19**:27–37, 1993.

[Sun90] V. S. Sunderam. "PVM: A Framework for Parallel Distributed Computing." *Concurrency: Practice and Experience*, **2**(4), December 1990.

[SW95] Wei Shu and Min-You Wu. "Runtime Incremental Parallel Scheduling (RIPS) for Large-Scale Parallel Computers." In *Proc. Frontiers'95*, pp. 456–463, McLean, Virginia, February 6-9 1995.

[SZM94] C. Steketee, W. P. Zhu, and P. Moseley. "Implementation of Process Migration in Amoeba." In *Proc. 14th Int. Conf. on Distributed Computing Systems*, June 1994.

[TLC86] M. Theimer, K. Lantz, and D. Cheriton. "Preemptable Remote Execution Facilities for V-System." *Proc. 10th ACM Symp. on Operating Systems Principles*, pp. 2–12, 1986.

[Tri82] Kishor Shridharbhai Trivedi. *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*. Prentice-Hall, Inc., 1982.

[Wex89] John Wexler. *Concurrent Programming in Occam 2*. John Wiley and Sons, Inc., 1989.

[Zho93] Songnian Zhou et al. "Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems." *Software: Practice and Experience*, **23**(12):1305–1336, December 1993.