Competitive Execution of Sequential Programs on a Network of Shared Processors

(To appear in ICA^3PP, Singapole, June 11-13, 1996)

Sung Hyun Cho

Department of Computer Science
University of California, Los Angeles, CA 90024-1596, USA
E-mail: cho@cs.ucla.edu

Abstract

We propose an alternative to process migration, called *competition*, to speed up distributed programs in the background on a network of shared 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 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.

In this paper, we focus on sequential programs, but our results for sequential programs can be extended to distributed programs because we can speed up a distributed program by speeding up communicating processes in the program.

We show that competition protocols offer performance benefits that are as good as or better than migration protocols for *sequential* programs under comparable assumptions. This claim is supported by analytical results and simulation results.

1 Introduction

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) [2] 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 [12], NEST [1], Butler [10], Condor [9], Benevolent Bandit [6], REM [11], and Stealth [8], and Utopia [14]. The goal of those systems is to execute processes on remote idle workstations, maximizing the utilization of workstations. However, the goal of competitive execution is to execute programs on idle processors on a NOW without interfering with foreground processes, minimizing the completion time of programs.

We may dedicate a subset of workstations in a NOW to execute 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 sharing. We focus on methods which are able to execute sequential or distributed programs on a SNOW in the background as quickly as possible.

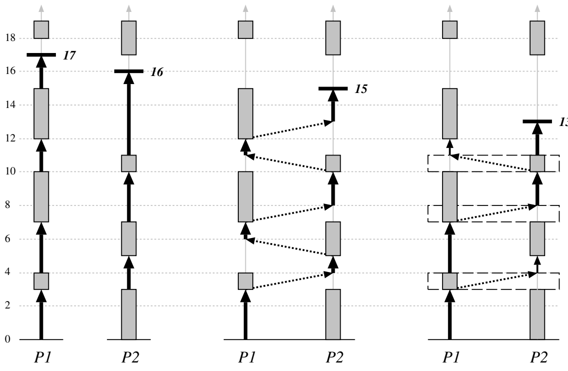
In a SNOW the available computing capacity to background processes changes dynamically, probably fast or drastically. This unpredictable variability may drastically degrade the performance of background programs; an entire program can perform poorly when one of its processes is executing on a heavily loaded workstation. Competition protocols overcome this problem by maintaining mulitple clones of processes at the cost of redundant execution of clones. However, redundant execution may not cost anything in a SNOW because competition exploits processors that are otherwise idle.

One of key issues in a SNOW is how intrusive a background program is to foreground processes and vice versa. We assume that workstation operating systems manage resources so that background programs do not interfere with foreground processes, but foreground processes will interfere with the performance of background programs. Thus CPU scheduling, memory management, and file access must be prioritized to insulate foreground processes from background programs [8]. We refer to workstations scheduled this way as *variable-speed* processors for background programs. We use the terms "workstations" and "processors" interchangeably in this paper.

1.1 An Example of Competition

Suppose that we have a non-interactive sequential process sp that we want to run in the background. In addition, we have two 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



(a) Static Choices (b) Migration (c) Competition

Fig. 1. Execution on Two Variable_speed Processors

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. 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.

In this paper, we assume a simple policy for migration and competition respectively: 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 are made instantly.

Here is an example showing that a competition protocol can execute *sp* quicker than a migration protocol. Consider two variable-speed processors $P1$ and $P2$ in Figure 1. Shaded rectangles represent foreground activities. Foreground activities on $P1$ and $P2$ are identical in Figure 1 (a), (b), and (c). Thick vertical arrows represent the fastest execution trace of *sp*, and dotted arrows represent process migration in Figure 1 (b) or state propagation in Figure 1 (c). Numbers in the left side of Figure 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 *sp* when a processor is dedicated. 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 (a). When we migrate *sp* back and forth as shown in Figure 1 (b), the *MIG* protocol completes *sp* in 15 units of time in which process migration occurs five times.

We assume that both of *sp*'s clones are not scheduled during the period of state propagation. Figure 1 (c) shows those periods by dashed rectangles. 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 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 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.

Competition speeds up *sp* by executing two clones of *sp* on two processors, that is, making use of more CPU cycles. However, competition protocols use only otherwise idle CPU cycles. We present other reasons why competition protocols are preferable to migration protocols on a SNOW in section 4.2.

1.2 Progress of Clones

In order to detect which clone of a sequential program is ahead, we can use instruction counter or CPU time clock. In a distributed program, we do not need to clone all the processes in a distributed program, but we just need to clone some of performance critical processes. The number of clones of a process can differ from that of another process depending on their performance criticality. However, this issus is beyond the scope of this paper.

In this paper 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?

2 Assumptions and Terminologies

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 paper, we make only one assumption about execution speeds of processors for background processes: they all have the same average execution speed. However, background execution speeds may fluctuate dynamically on processors over time.

2.1 Assumptions

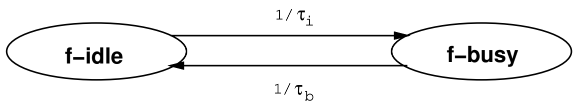


Fig. 2. Two State Model of a Processor

When a processor executes a foreground process, we say that it is *f-busy*. Otherwise, it is *f-idle*. We assume that all processors are physically identical, and 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:

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

$$b(t) = \frac{1}{\tau_b} e^{-t/\tau_b}, \qquad t \geq 0 \tag{2}$$

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 $i(t)$.

Processors alternate between f-busy and f-idle states at constant rates as in Figure 2, and 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.

For competition, 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. We assume that 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.

2.2 Stretching Ratio

The completion time of a clone can be stretched according to the preemption of foreground processes.

Definition 2.1 *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 ignore these factors because we are interested in long running applications. We assume that the mean stretching ratio $\overline{SR}$ does not change for the entire computation of applications.

The mean stretching ratio of a variable-speed processor, $\overline{SR}$, can be expressed in terms of τ_i and τ_b on the processor as follows [4];

$$\overline{SR} = \frac{\tau_i + \tau_b}{\tau_i} = 1 + \tau_b/\tau_i. \tag{3}$$

2.3 Speedup of Competition

We assume that the mean foreground processor utilization ρ is statistically identical at all processors. Thus the mean background execution speed at all processors is also statistically identical. Now 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 $\overline{T_{nc}(K)}$ and $\overline{T_c(N)}$ 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

$$SP(K, N, \rho) = \frac{\overline{T_{nc}(K, \rho)}}{\overline{T_c(N, \rho)}}. \tag{4}$$

3 Optimal Speedup of Competition

We consider an *ideal* competition protocol, which makes state propagation decision without any overhead and propagates states instantaneously at optimal times.

Assume that the ideal competition protocol replicates *sp*'s clones k times, and allocates them to different processors. The foreground processes create certain pattern of activities into which *sp*'s clones have to fit.

Theorem 3.1 *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 τ_i and τ_b respectively. Then the foreground utilization of all processors is $\rho = \tau_b/(\tau_i + \tau_b)$. The optimal speedup of the ideal competition protocol on k independent processors over one processor is $1 + \rho + \cdots + \rho^{k-1}$.*

Proof The ideal competition protocol allows a process *sp* to make progress as long as one of k processors is f-idle. The probability that one of independent k processors is f-idle is $1 - \rho^k$. Let W is the finishing time of *sp* on a dedicated processor. Then it will take $W/(1-\rho)$ on a single processor, and thus the mean stretching ratio of each processor is $1/(1-\rho)$. The stretching ratio of k processors with the ideal competition protocol is $1/(1-\rho^k)$. Thus the optimal speedup of the ideal competition protocol 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, and 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 is the reason we just consider the small number of clones in this paper.

The best speedup from competition 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, but competition probably won't help much. Or if ρ is high, there is high degree of competition efficiency and thus we can finish soon by cloning more.

4 Competition and Migration

4.1 Assumptions

We assume that the competition protocol replicates the sequential process *sp* and statically assign k clones $sp_1, sp_2, \cdots, sp_k$ to k different variable-speed processors $P_1, P_2, \cdots, P_k$ out of M processors where $M \geq k$. We also assume that the migration protocol migrates *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.

We considered *MIG* protocol and *COMP* protocol in the section 1.1, and their migration delay or state propagation delay is zero. However, we will also consider *MIG* protocol and *COMP* protocol where their migration delay or state propagation delay is non-zero but constant. They are called *MIG_D* protocol and *COMP_D* protocol respectively.

Migration in *MIG* protocol is always successful, but migration in *MIG_D* protocol may not be successful due to the migration delay. When migration is unsuccessful, we assume that the target treats *sp* in the same way as it preempts *sp*.

When a migration protocol propagates a process, we assume that it cannot execute until migration is finished. Similarly 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 in this paper.

4.2 Inherent Differences

Migration protocols have to choose one of targets and 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*. 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. But only the clone that is preempted last can propagate its state in competition protocols.

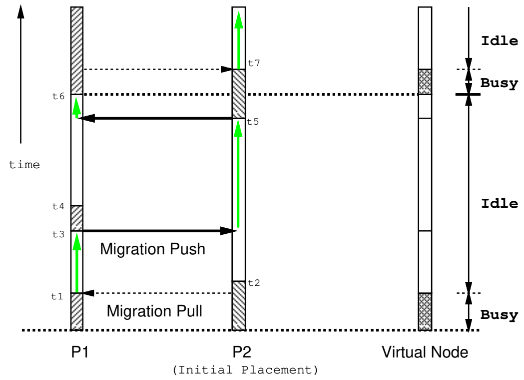


Fig. 3. The *MIG* Protocol

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 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.

5 *MIG* Protocol and *COMP* Protocol

We derive the closed form solutions for the speedup of *MIG* Protocol and *COMP* Protocol.

5.1 *MIG* Protocol

Figure 3 shows two processors $P1$ and $P2$ where *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 when migration delay is not zero. The execution trace of *sp* is shown by thick vertical arrows in Figure 3.

In order to analyze the performance of the *MIG* 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 3. If both $P1$ and $P2$ are f-busy at a given time, the virtual node is called *busy.* 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.

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 [7]. The length of each busy period on the virtual node is simply the minimum of the lengths of two f-busy periods on $P1$ and $P2$.

An idle period on the virtual node may consist of more than one segment where each segment indicates that *sp* executes at a processor during that period. Busy periods and idle periods on the virtual node alternate. We will analyze the performance of the *MIG* protocol on the virtual node based on the renewal period [5]. 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* protocol.

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 is the minimum of k exponentially distributed random variables with the same mean τ_b. $\overline{B_k}$, 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 3, if sp starts at $P1$ initially, $P1$ can start to execute sp without a migration pull, and $P1$ is called a *proper* processor. Otherwise, $P1$ is called an *improper* processor.

Let X denote the number of migration excluding the migration pull in an idle period. Then the number of segments in an idle period is $X+1$. The probability mass function of X is given by a modified geometric distribution [13]:

$$P_X(i) = fs^i, i = 0, 1, 2, \cdots \tag{5}$$

where s is the probability that another processor is f-idle when sp is preempted 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. $\overline{I_2}$, the mean of I_2, is given by [13]

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

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

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

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

$$SP_{MIG}(2) = \frac{1 + (\tau_b/\tau_i)}{SR_{MIG}(2)} = \frac{1 + (\tau_b/\tau_i)}{1 + (\tau_b/\tau_i)((1-s)/2)}. \tag{6}$$

Since the *MIG* protocol does not have any overhead, Equation 6 should be the same as the optimal speedup derived in Theorem 3.1, which is $1 + \rho$. When sp is preempted at a processor, the probability that the other processor is f-idle is given by

$$s = \frac{\tau_i}{\tau_i + 2\tau_b} \tag{7}$$

by setting $SP_{MIG}(2) = 1 + \rho$. The probability is referred to as the probability of f-idleness.

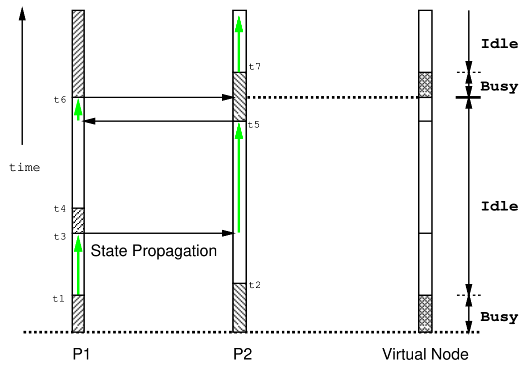


Fig. 4. The *COMP* Protocol

5.2 *COMP* Protocol

Figure 4 shows that clones sp_1 and sp_2 start to execute at `t1` on $P1$ and at `t2` on $P2$. A solid arrow indicates a state propagation. Note that $P1$ preempts sp_1 at `t6` and propagates its state even though $P2$ is f-busy. This differs from migration protocols where migration stops if $P2$ is f-busy.

An idle period may consist of more than one segments, and each segment always traces the performance profile of the clone that is ahead at any time.

Now we consider a two processor case. An idle period on a virtual node ends when both of processors 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 7.

Let Y denote the number of state propagations in an idle period. Then in the *COMP* 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 [13]:

$$P_Y(i) = fs^i, i = 1, 2, \cdots \tag{8}$$

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

As in the *MIG* protocol, $\overline{B_2} = \tau_b/2$. 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* protocol on two processors is given by

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

The speedup of the *COMP* protocol with two pro-

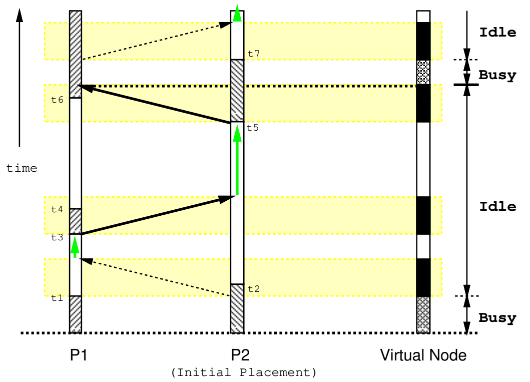


Fig. 5. *MIG_D* Protocol

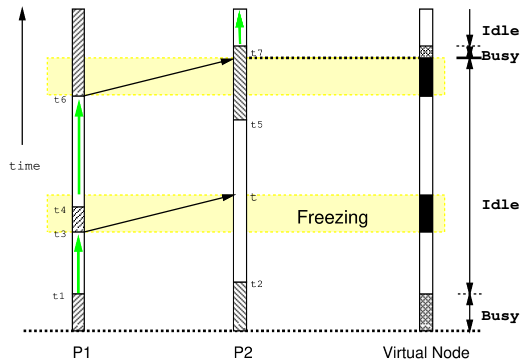


Fig. 6. *COMP_D* Protocol

cessors over a single processor is

$$SP_{COMP}(2) = \frac{1 + (\tau_b/\tau_i)}{SR_{COMP}(2)} = 1 + \frac{\tau_b}{\tau_i + \tau_b} = 1 + \rho. \tag{9}$$

Thus the performance of the *COMP* protocol is the same as the optimal speedup given in the theorem 3.1. 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.

6 *MIG_D* Protocol and *COMP_D* Protocol

We derive the speedup solutions of *MIG_D* Protocol and *COMP_D* Protocol.

6.1 *MIG_D* Protocol

Figure 5 shows the same activities of foreground processes on $P1$ and $P2$ as in Figure 3, 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 3 while a black period denotes the migration delay D, which is also shown by a shaded rectangle. Only white periods will be effectively available to execute sp. 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. Since the available time in a renewal period can be used to execute sp, the stretching ratio will be $1 + (B_k + O_k)/A_k$.

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 5.

We are mostly interested in the case where $D < \tau_b$ because a f-idle period is highly likely to be available locally if $D > \tau_b$. Let I be a random variable which represents a residual f-idle period. 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 and the second terms account for the case that sp is positioned at the proper processor and at the improper processor in the beginning of an idle period respectively. 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. Since the mean number of migration push (not including a migration pull) is $\overline{X} = s/(1-s) = \tau_i/2\tau_b$,

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

Here $\overline{B_2} = \tau_b/2$, 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

$$SR_{MIG\_D}(2) = 1 + \frac{\tau_b + \omega D}{\tau_i(1 + \omega e^{-D/\tau_i})} \tag{10}$$

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

6.2 *COMP_D* Protocol

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 $s = \tau_i/(\tau_i + 2\tau_b)$ in Equation 7 as the probability that the other processor is f-idle when a state propagation is completed. 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 probability mass function of Y is given by the geometric distribution in Equation 8 where $s = s_1$ and $f = f_1$.

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

Now let us focus on two processors. When the state propagation is completed, two clones may start execution at the same time, 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

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

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

The mean of the maximum of lengths of two exponentially distributed f-idle periods is $3\tau_i/2$ [3]. Thus the mean of the sum of available time in an idle period is

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

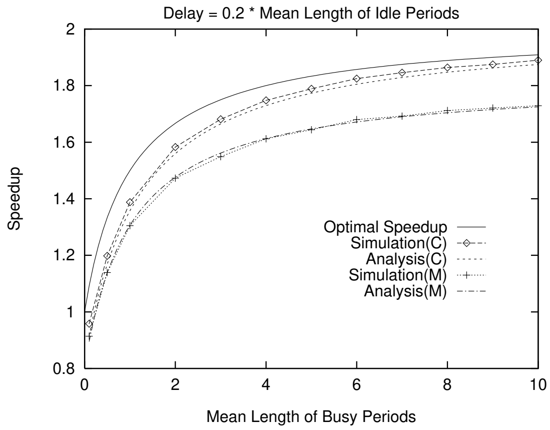


Fig. 7. Speedup with Two Processors When $D = 0.2\tau_i$

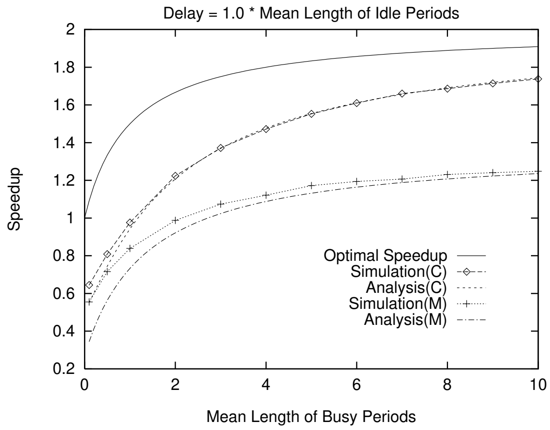


Fig. 8. Speedup with Two Processors When $D = \tau_i$

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

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

$\overline{B_2} = \tau_b/2$, and

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

Thus

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

6.3 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 and D are represented in multiples of τ_i. We simulated for a period of $1000\tau_i$ and our results are means of 20 simulations. Figures 7 and 8 show our results when D 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 $1 + \rho$.

The delay for 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. In Figure 8 we see that *MIG_D* and *COMP_D* protocols improve the performance when $\tau_b > 2.8\tau_i$ while in Figure 7 we get an improved performance for the most of the region.

When $\tau_b < D$, there is significant discrepancy between our simulation results and analytic results (especially in Figure 8). This occurs due to the fact that we approximate the probability of migration as $\tau_i/(\tau_i + 2\tau_b)$. The error in ω is magnified for a given τ_i when τ_b approaches 0 because we have used $\omega = 1 + \tau_i/\tau_b$ in Equation 10. However, this is not the case we are interested because f-idle periods are likely to be available locally.

As is evident from figures 7 and 8, the performance difference between *MIG_D* protocol and *COMP_D* protocol increases as τ_b increases for a given D. The reason is that in *MIG_D* protocol as τ_b increases, the average number of migration pushes in an idle period decreases and therefore the overhead of migration pulls become dominating. On the other hand, the *COMP_D* protocol does not suffer from any such overhead.

Similarly, as D increases, there is an increase in the overhead of migration pulls and a corresponding decrease in the probability of migration success. 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 of other clones.

7 Conclusions

When a migration delay or a state propagation delay is not instantaneous, we demonstrated that competition protocols for sequential programs offer performance benefits that may be better than migration protocols under comparable overhead assumptions at the cost of redundant execution. Competition protocols use only otherwise idle CPU cycles, so redundant execution does not really cost at all. Of course, competitive execution cannot be employed when there is no enough idle CPU cycles. However, many researchers have observed that significant portion of CPU cycles on SNOWs has gone unused.

We derived the closed form solution for the optimal speedup of the ideal competition protocol, and we showed that competitive execution can finish sequential programs significantly faster than noncompetitive execution, especially when foreground load on each processor is sufficiently high.

Competitive execution may allow a person without his own workstation to execute his long running (sequential or distributed) programs on a SNOW significantly faster than on a private or shared workstations without competition. We conjecture that competition protocols offer even more speedups for distributed programs than for sequential programs.

References

[1] R. Agrawal and A. K. Ezzat. Location independent remote execution in nest. *IEEE Transactions on Software Engineering*, 13(8):905–912, 1987.

[2] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW team. A case for now (networks of workstations). *IEEE Micro*, pages 54–64, February 1995.

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

[4] Sung Hyun Cho. *Competitive Execution in a Distributed Environment.* PhD thesis, University of California, Los Angeles, 1995.

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

[6] 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.

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

[8] Phillip Krueger and Rohit Chawla. The stealth distributed scheduler. In *11th Int'l Conf. on Distributed Computing Systems*, pages 336–343, Arlington, Texas, May 20-24 1991.

[9] M. J. Litzkow, M. Linvy, and M. W. Mutka. Condor - a hunter of idle workstations. In *8th Int'l Conf. on Distributed Computing Systems*, pages 104–111, San Jose, California, June 13-17 1988.

[10] David A. Nicols. Using idle workstations in a shared computing environment. *Operating Systems Review*, 21(5):5–12, November 8-11 1987.

[11] G. C. Shoja. A distributed facility for load sharing and parallel processing among workstations. *Journal of Systems and Software*, 14:163–172, 1991.

[12] M. Theimer, K. Lantz, and D. Cheriton. Preemptable remote execution facilities for v-system. *Proc. 10th ACM Symp. on Operating Systems Principles*, pages 2–12, 1986.

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

[14] 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.