Speedup of Distributed Programs on a Network of Shared Processors

Sung Hyun Cho
Department of Computer Science
University of California, Los Angeles CA 90024
Email: cho@cs.ucla.edu, Phone: (310) 208-7459, Fax: (310) 825-2273

Abstract

This paper presents an alternative to process migration, called *competition*, to speed up distributed programs in the background on a network of 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.

We show that competition protocols offer performance benefits that are as good as or better than migration protocols for *distributed* programs under comparable assumptions. We also demonstrate that competition protocols offer even more speedup for distributed programs than for sequential programs.

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. Their goal is to execute processes on remote idle workstations, maximizing the utilization of workstations.

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 sharing. Our goal in this paper is to execute *distributed* programs on a SNOW in the background[1] 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.

We assume that workstation operating systems (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 paper.

1.1 What is 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 *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

[1] We consider that all processes of workstation users are foreground processes regardless of their priority, and that competing processes are background processes. We assume that background processes on a workstation are scheduled only when no foreground processes are active.

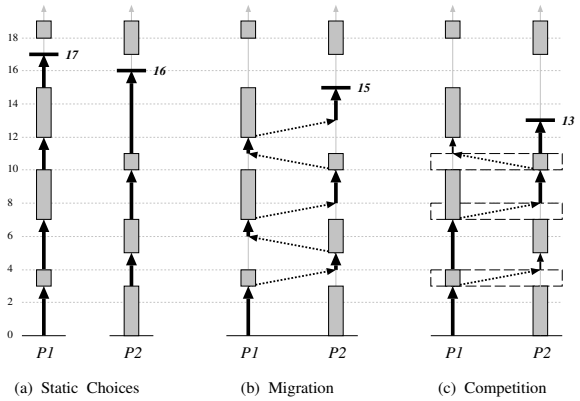

Figure 1: Execution on Two Variable_speed Processors

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*.

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 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 is made instantly in this paper.

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 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 (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 to prevent state modification. 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 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 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. However, competition does not increase the *parallelism* of *sp*, because *sp* is a sequential process and has no parallelism in it.

1.2 A 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 communicate 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.

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. Thus all clones of a process execute the same instructions in the same order, and send the same messages in the same sequence, and go through the same sequence of states when all clones have the same initial state and they receive the same messages in the same sequence.

1.3 Definition of Speedup for Competition

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 stochastic process with the same rate, and their execution times have a distribution with the same parameter. Then the mean foreground processor utilization ρ is statistically identical at all processors, and thus the mean background execution speed at all processors is also statistically 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

$$SP(K, N, \rho) = \frac{T_{nc}(K, \rho)}{T_c(N, \rho)}. \quad (1)$$

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.

2 Related Work

Many researchers have tried to harness idle or underutilized workstations by using remote execution facilities; these include the V system [24], Butler [19], Condor [16], Stealth [13], and Utopia [25].

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 [24, 19], availability is defined absolutely; an idle workstation is the one not used at all. Refer to [13, 17] 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 [24, 1, 16]; 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 [13, 12]. 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 in the background, the other processes may have to wait for the process because of communication or synchronization. Therefore a priority-driven local scheduler alone is not sufficient to execute distributed programs in a SNOW. Process migration appears to be 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 sharing [15, 18, 9, 14, 8, 22]. Refer to the survey on process or object migration in [21, 20].

Several software tools have been developed to support SNOW environments; PVM [23, 10], P4 [4], Linda [5], and MPI [11]. These tools allow programmers to treat a heterogeneous collection of processors in a SNOW as one large virtual machine. However, these tools do not help to execute programs in the background due to the interference of foreground processes.

3 A Competition Mechanism

There are two issues in the implementation of competition protocols: competition policy and competition mechanism. In this paper, we do not investigate 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 two processes P and Q at the application level as shown in Figure 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 level as in Figure 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 competition mechanism roughly works as follows: assume that p_1 is ahead of its peer clones, and that p_1 produces a message M earlier than its peer clones.

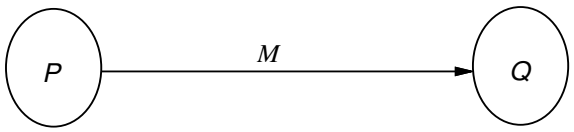

Figure 2: A Message M from a Process P to another Process Q

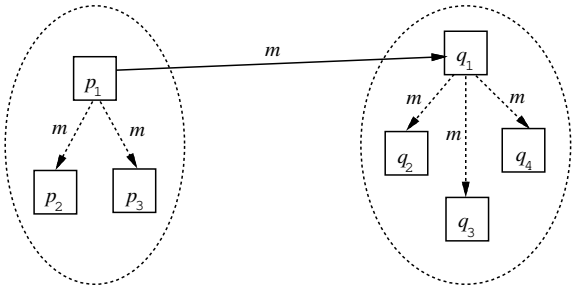

Figure 3: Message Communication in a Competition Mechanism

When p_1 requests to send a message M to a process Q, the mechanism does two things. First, it delivers a message clone m of M to a clone (q_1 in Figure 3) of Q, and multicasts m to the other peer clones of p_1 to inform that p_1 has already sent m. Second, the mechanism multicasts m to the other peer clones of q_1 when m arrives at a clone q_1. If p_2 or p_3 requests to send a message M after m that is multicast by p_1 arrives, those requests are suppressed. Refer to details of the competition mechanism in [6] where it is shown that one message multicasting on a broadcasting network is sufficient to achieve a message delivery between communicating clones and message multicasting among sending process clones and receiving process clones respectively.

Competition mechanisms have to make the collective behavior of clones of a process be 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 An Ideal Competition Protocol

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

4.1 Assumptions

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:

$$B(t) = 1 - e^{-t/\tau_b}, \quad t \geq 0 \tag{2}$$

$$b(t) = \frac{1}{\tau_b} e^{-t/\tau_b}, \quad t \geq 0 \tag{3}$$

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 [6]. Thus 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.

4.2 Stretching Ratio

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

Definition 4.1 *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;

Proposition 4.1 *Let the mean length of f-busy periods and that of f-idle periods on a variable-speed 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, $\tau_i/(\tau_i + \tau_b)$. Thus $\overline{SR}_Q = 1 + \tau_b/\tau_i$. □

4.3 Optimal Speedup of the Ideal Competition Protocol

Assume that the ideal competition protocol replicates *sp*'s clones k times, and allocates them to differ-

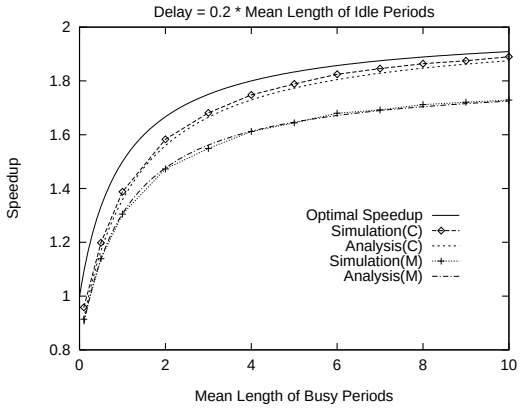

Figure 4: Speedup with Two Processors when $D = 0.2\tau_i$

ent processors. The foreground processes create certain pattern of activities into which background processes have to fit.

Theorem 4.1 *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 optimal speedup of the ideal competition protocol for a sequential program on k independent processors over one processor is $1 + \rho + \cdots + \rho^{k-1}$.*

The proof of this theorem is omitted because it was proved in [7]. 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 in the background.

5 Sequential Programs

We reported the performance of *MIG* and *COMP* protocols in [6, 7] based on analytic results and simulation results for a sequential program. We will show one of typical results here for the reference purpose.

Figure 4 shows our simulation results and analytic results on two processors when the delay for migration or state propagation is $0.2\tau_i$. The curves labeled as 'Simulation(M)' and 'Simulation(C)' are simulation results of *MIG* and *COMP* protocols respectively. The curves labeled as 'Analysis(M)' and 'Analysis(C)' are our analytic results. The curve labeled as 'Optimal Speedup' is $1 + \rho$ on two processors according to the Theorem 4.1.

In our simulations τ_i was set to 1.0, and τ_b (mean of f-busy periods) and D (the 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 simulations. You can find more details in [6, 7].

6 Distributed Programs

We study the performance of competition protocols for distributed programs, and we also consider the effect of granularity on the performance of competition protocols.

6.1 Introduction

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 a process. 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. However, there will be no difference on a network with multicasting capability. In this paper we assume a network with multicasting capability. Thus overheads of migration and competition are identical independently of the number of participating processors.

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 [3].

The simulator is composed of three layers; 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.

6.1.1 The Application Layer

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

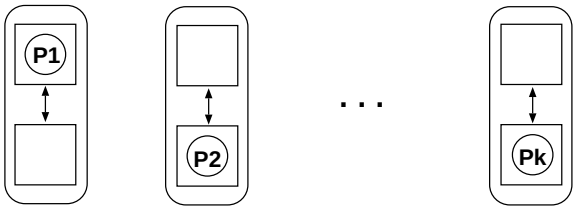

Figure 5: Static Boundary

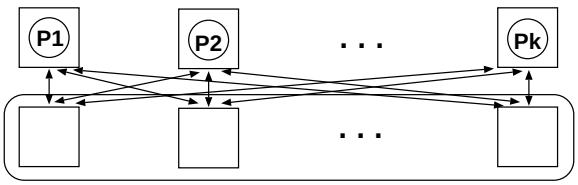

Figure 6: Dynamic Boundary

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.

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 in this paper.

6.1.2 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 program. 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 5 shows the static boundary for migration. Squares represent processors, circles represent processes, and rectangles with round corners represent boundaries a process 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 processes can migrate to. Figure 6 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 this paper, and we consider only a static strategy in this paper.

6.1.3 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 for fair comparison because migration protocols are not optimized.

A clone is ahead of others if the clone executes more messages than others. We assume that the number of messages a clone has executed is a good measure for the progress of the clone.

6.1.4 Assumptions in Our Simulation

We assume that message communication delay is zero because we are interested in the case that the size of messages is much smaller than that of states. Thus the configuration of a program does not have any performance effect in this paper. We assume that there are enough processors that are available to execute each

clone on its own processor.

Let I and B denote the mean length of f-idle periods and the mean length of f-busy periods respectively. I and τ_i are used interchangeably in this section, and so are B and τ_b.

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 exchange messages with their neighbors 1000 times, and our results are the means of 10 simulations.

6.2 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 paper. We consider simple distributed programs with the following three connectivities; full connectivity, 2D mesh connectivity, and ring connectivity.

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 as defined in Equation 1.

6.2.1 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 $G = 2\tau_i$.

The following describes the simulation parameters for a fixed $\tau_i (= 1.0)$.

B – the mean length of busy periods on processors

C – the number of clones for processes

D – the delay for process migration or state propagation

Figure 7 shows the performance of competition protocols when $D = 0$. 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. Figure 7 shows that we can achieve more speedups for the distributed program with 10 fully connected pro-

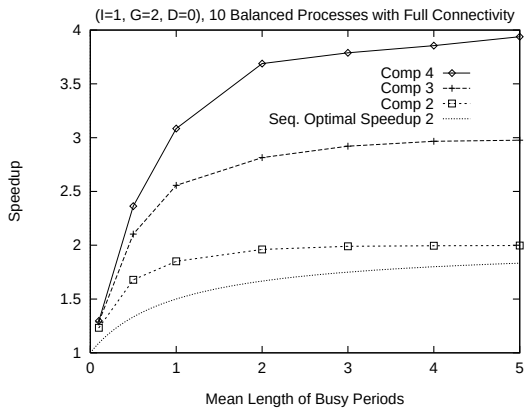

Figure 7: 10 Processes with Full Connectivity when D = 0

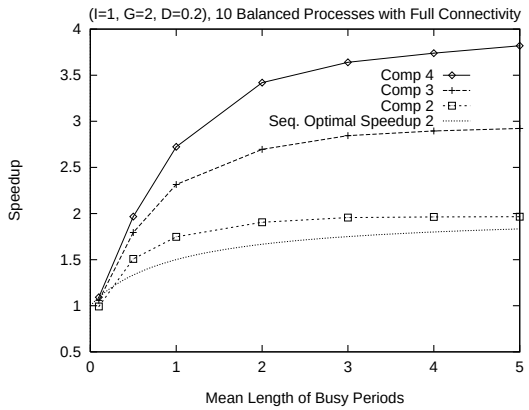

Figure 8: 10 Processes with Full Connectivity when D = $0.2\tau_i$

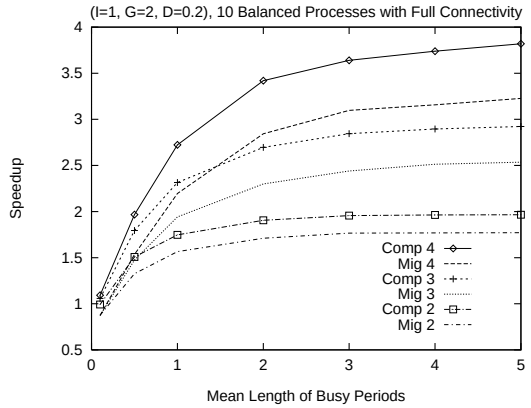

Figure 9: Comparison of Competition and Migration when D = $0.2\tau_i$

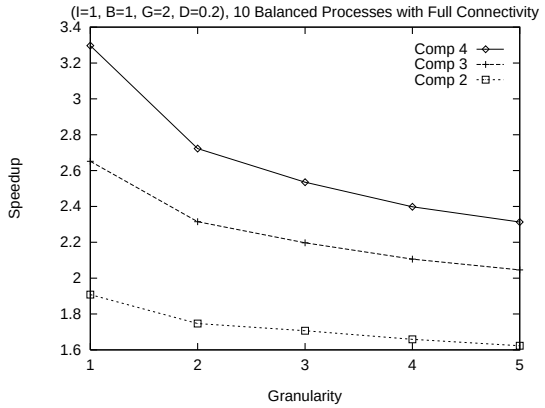

Figure 10: The Effect of Granularity on the Speedup

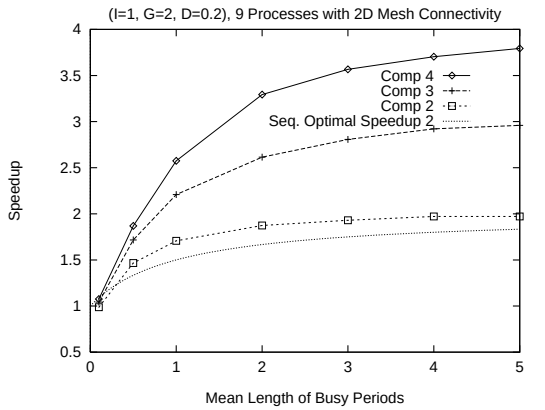

Figure 11: 9 Processes with 2D Mesh Connectivity when D = $0.2\tau_i$

cesses than for sequential programs because the optimal speedup for sequential programs on two processors is the lower bound for the distributed program.

Now we present speedup curves when $D = 0.2\tau_i$ in Figure 8. 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. When $B > 0.3\tau_i$, 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 9 shows the speedups of competition protocols and migration protocols 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.

Now let us consider the effect of granularity on the performance of competition protocols. Figure 10 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 single process.

6.2.2 Programs with Mesh Connectivity

Now we consider a distributed program consisting of 9 processes with 2D mesh connectivity. Figure 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 $D = 0.2\tau_i$ 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 8. 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.

6.2.3 Programs with Ring Connectivity

We also consider a distributed program with 10 processes with ring connectivity. Figure 12 shows the performance of competition protocols with different degrees of competition when $D = 0.2\tau_i$. At each communication phase, each process in a undirected ring

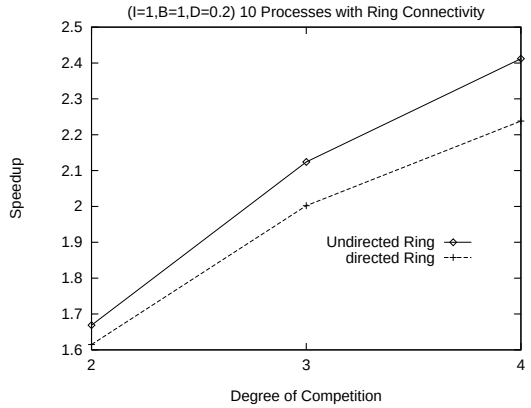

Figure 12: Ten Processes with Ring Connectivity when D = $0.2\tau_i$

sends two messages and wait for two messages while each process in a directed ring sends one message and wait for one message. Thus competition can speed up the program with a undirected ring connectivity more than that with a directed ring connectivity.

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

7 Conclusions

In this paper, we demonstrated that competition protocols offer performance benefits that are as good as or better than migration protocols for sequential and distributed programs under comparable assumptions at the cost of redundant execution. However, competition protocols use only otherwise idle CPU cycles, so they do not really have a higher "cost" at all.

We showed that competitive execution could finish sequential programs significantly faster than noncompetitive execution in [7], especially when foreground load on each processor is sufficiently high. In this paper, we showed that competition protocols offer even more speedups for distributed programs than for sequential programs. We believe that competition protocols are the better alternative than migration protocols to speed up distributed programs in the background.

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

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

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

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

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

[7] Sung Hyun Cho. Competitive execution of sequential programs on a network of shared processors. In *IEEE 2nd International Conference on Algorithms and Architectures for Parallel Processing*, pages 116–123, June 1996.

[8] Fred Douglis and John Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software - Practice and Experience*, 21(8):757–785, August 1991.

[9] 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 Model-*

ing of Computer Systems, pages 63–72, Santa Fe, New Mexico, May 24-27 1988.

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

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

[12] Jiubin Ju, Gaochao Xu, and Jie Tao. Parallel computing using idle workstations. *Operating Systems Review*, 27(3):87–96, July 1993.

[13] 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.

[14] Phillip Krueger and Miron Linvy. A comparison of preemptive and non-preemptive load distributing. In *8th Int'l Conf. on Distributed Computing Systems*, pages 123–130, San Jose, California, June 13-17 1988.

[15] M. Linvy and M. Melman. Load balancing in homogeneous broadcast distributed systems. In *Proc. ACM Computer Network Performance Symposium*, pages 47–55, April 1982.

[16] 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.

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

[18] 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*, pages 539–546, May 1985.

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

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

[21] Jonathan M. Smith. A survey of process migration mechanisms. *Operating System Review*, 22(3):28–40, July 1988.

[22] 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.

[23] V. S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4), December 1990.

[24] 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.

[25] 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.