BROWN UNIVERSITY
Department of Computer Science
Master's Project
CS-94-M8

"Executing Parallel Programs on a Network of User owned Workstations"

by

Elizabeth Jean Phalen

Executing Parallel Programs on a Network of User Owned Workstations

Elizabeth Jean Phalen

Department of Computer Science

Brown University

Submitted in partial fulfillment of the requirements for the

Degree of Master of Science in the Department of Computer Science

at Brown University

May 1994

This research project by Elizabeth J. Phalen is accepted in its present form

by the Department of Computer Science at Brown University

in partial fulfillment of the requirements for the degree of Master of Science.

[illegible]

Robert H. B. Netzer
Advisor

3-11-94

Date

Table of Contents

1.0 Introduction

A lack of sufficient CPU resources for compute-intensive parallel programs continues to impede efficiency in many areas of computing. Both in industry and academia, there is often a total lack of sufficient resources, or resources that do not provide satisfactory throughput. This impacts applications ranging from executions of regression tests, to extensive scientific experiments.

This project investigates networks of user-owned workstations as a potential platform for providing the needed resources. This platform is promising for two reasons. First, it is available in most technical work environments. Second, recent studies have shown that workstation utilization is low, especially at certain hours of the day.

This project focuses on the difficulties of sharing workstations between owners and remote parallel programs. The major difficulty in sharing workstations in this way, is that it must have minimal impact on workstation response time. The workstation owner's computing demands must be given priority. Addressing this difficulty requires scheduling software designed for dynamically changing CPU resources. It may also require unique approaches to parallel program design.

A solution must detect when a workstation becomes busy with user activity. It must react by reducing the parallel program's use of that workstation's resources. Possible approaches to this may be similar to those for other scheduling and load balancing problems. Examples of these approaches are migrating any of the parallel program's processes to another workstation or reducing their priority. Another category of approaches relies on greater flexibility in the parallel program's concurrency levels and order of execution of work. An example of this is to abandon and then reexecute work that is in progress when the workstation becomes busy. This may introduce requirements on the design of the parallel program.

As a secondary goal, a solution should aim at the greatest possible utilization of workstation resources. This suggests that execution of the parallel program not only adapts to decreases in CPU resources, but can also take advantage of increases.

Although there are existing techniques for executing remote jobs on user-owned workstations[3,4,9,13,14], most of them are not designed for parallel executions. Those techniques that support parallel executions, in general do not address sharing resources with the workstation owner[12]. Because of this lack of previous work, this project begins by developing a taxonomy of potential execution models. These execution models describe possible runtime scenarios. They are analyzed with respect to their relation to parallel programming paradigms and implementation techniques.

The second phase of the project investigates availability of workstation resources on a typical network of user-owned workstations. This was done by gathering empirical data on workstation usage within the Brown Computer Science department. The data was collected for a period of six months.

The final phase of the project directly investigates the feasibility of efficiently executing parallel programs on a network of user-owned workstations. This is done by simulating executions of a set of message-passing parallel programs in the environment described by the collected workstation data. The parallel programs are based on traces of four hypercube programs. The simulations are currently implemented to investigate one of the potential execution models. In the future they could be used to compare the relative performance of different execution models.

1.1 Functional Requirements

The software that is being investigated by this project can be described by the following functional requirements:

- Allocate workstations for all processes that will execute portions of the parallel programs. This includes initial allocations and any additional allocations required during execution. These additional allocations may be required because of program requests to spawn processes, or to react to changes in workstation availability.
- Detect increases and decreases in user demands on workstation resources.
- In response to increases, reduce use of CPU's in order to avoid impacting user response time.
- Optionally, in response to decreases, expand use of CPU's in order to optimize utilization.

The requirement that is most interesting to this project is how reacting to increases and decreases in workstation availability should be done for parallel executions. The other functionality has been covered by other research projects.

1.2 Assumptions

The following assumptions and restrictions have been made on the scope of the considered solutions.

- No operating system changes required.

- No dependency on particular parallel program structure or approach.
- Data accessible through shared file system.
- No requirements for accessing data or invoking system services at originating workstation.
- Homogenous network of workstations.
- Assume that availability of workstations is viewed from a digital perspective. That is, based on some criteria a workstation is either available or not for the parallel program's use. (Alternatively a workstation may be considered to be available to varying levels, this approach is taken by some related work - see section 4.0 .)

1.3 Summary of Results

The simulation results support the theory that a network of user-owned workstations can provide sufficient CPU resources for efficient parallel program executions. Specifically, the simulations support the effectiveness of a particular solution to reacting to user activity on a workstation. The solution is to migrate the parallel program's process at that workstation to a different workstation.

The simulations results show how the programs' execution times are impacted by the cost of reacting to user activity. The results show that if the cost of migrating a process is between one and 50 seconds, the program's execution time increases less than 100%. The increase is in comparison to an execution with zero cost for reacting to user activity. If the cost of migrating a process is between 50 and 100 seconds, the programs' execution time increases less than 350%. During some portions of the day, particularly the hours from Midnight to 6 a.m., the program execution times never increased more than 100%.

In other words, executions in this environment took 2-4 times longer than if the programs were executed on a dedicated pool of workstations. Considering the fact that a dedicated pool of 17 workstations (the number of subprogram's for the sample parallel programs) is often not an available resource, this seem to be a feasible solution. At a minimum, the results suggest that further investigation should be done into utilizing this environment for parallel executions.

2.0 Design Approaches

This section will discuss two categories of designs: adaptive and scheduling. Their runtime behavior will be described. Their compatibility with parallel programming paradigms and implementation techniques will be evaluated.

The two design approaches differ in whether they address the problem as:

- a scheduling problem where there are two classes of processes: those owned by the workstation user and those involved in the parallel program. The workstation user processes have priority over the other.
- an environment where CPU's are a dynamic resource, whose level of availability must be considered in the design of the parallel programs themselves.

Throughout this section, parallel programs will be described as consisting of a collection of subtasks. A subtask is any portion of the program that may execute autonomously without dependencies on other concurrent portions of the parallel program. For example, in the case of a message passing program, subtasks are delimited by successive receives.

2.1 Implementation Options

Before describing the design approaches, we will introduce potential implementation options so that they may be discussed with respect to their compatibility with the designs.

As stated in the introduction, it is implementation techniques for reacting to workstation availability changes that are of most interest to this project.

2.1.1 Techniques for Reacting to Decreases in Workstation Availability

The following techniques react to decreases in workstation availability by reducing the resource demands of a process executing as part of the parallel program:

- Migrate the current process state to another workstation.
- Rollback the process state to the beginning of the current subtask and then migrate the process. This assumes that the subtasks can be executed idempotently, since their execution may be at least partially repeated. The reason for this approach is the assumption that the process state at the beginning of a subtask will be simpler, and therefore less costly to migrate. The process state may be simpler as long as the subtask truly defines a unit of autonomous work. This would suggest that there would be less transient data at the beginning of the subtask.
- Abandon the current process state, but save any completed work and identify any uncompleted work. Uncompleted work may be returned to a pool of yet to be completed tasks, or sent to some other process executing on behalf of the parallel program.
- Reduce the priority of the process or suspend the process. The process may need to stay resident at the current workstation in a suspended state until another workstation becomes available.
- Other techniques could be considered by future work.

2.1.2 Techniques for Reacting to Increases in Workstation Availability

The following techniques react to increases in workstation availability:

- Spawn additional processes at workstations that have become available.

- Maintain daemon processes which can execute subtasks when a workstation becomes available.
- Restore the priority of a process if it had previously been reduced to react to a decrease in workstation availability.
- Send additional work to a process. This amounts to dynamically increasing the granularity of the subtask. An example is sending it additional data to process.
- Other techniques could be considered by future work.

2.2 Parallel Paradigm Options

Not only will a design choice impact implementation options, it may also place constraints on the types of parallel programs that can run in the environment. The following parallel programming paradigms are introduced so that their compatibility with potential designs may be evaluated.

- Message passing

  In message passing parallel programs, the work is divided among a set of subprograms that execute in different processes. Successive subtasks in the same subprogram communicate through the process state. Subtasks in different processes communicate through the send and receipt of messages. A message passing program may include spawning and killing of subprocesses as the level of concurrency changes.An example of a communication library that supports message passing is PVM (see 4.2.1).

- Task oriented parallelism

  In the task oriented approach, a parallel program is defined as a set of tasks, and dependencies between the tasks. Tasks are consumed by generic worker processes. The programming environment controls creating and destroying worker processes and assigning tasks to a particular process. Tasks communicate in a process independent manner, such as shared memory, since they are not bound to a particular process. The Piranha programming environment supports the task oriented parallelism (see 4.1.1).

- Other existing or yet to be defined parallel programming paradigms may be suitable to executing parallel programs in this environment. For simplicity, this project chose to focus on the two paradigms described here.

2.3 Adaptive Design Approaches

2.3.1 Description

The category of adaptive approaches is distinguished by parallel programs designed to adapt to changes in CPU resources. It is unique from traditional scheduling where the program is isolated from CPU allocations.

Traditionally, concurrency levels are set prior to execution, requiring a certain number of CPUs. This does not consider that the number of available CPUs may change during execution of the program, which is the situation on a network of user-owned workstations. Software that does recognize CPUs as a dynamic resource may benefit by being able to align its CPU requirements with CPU availability. The result being greater utilization of CPU resources, and more efficient responses to decreases in CPU resources.

The following are some of the ways a parallel program design may incorporate adaptivity. Since this is a relatively new approach, there are most likely additional techniques, and this is potentially an interesting area for future study.

An adaptive parallel program may allow flexibility in the order of execution of subtasks. The order of execution of is controlled by dependencies between the subtasks. (i.e., a message passing subprogram blocks until a receive is satisfied). However, beyond these dependencies there should be no limitation on the execution order. A design could take advantage of this by:

- Maintaining a pool of all subtasks whose dependencies have been met and allowing them to be executed as CPU resources become available.
- Allowing subtasks to be abandoned during execution in response to a workstation becoming busy and reexecuted at a later point.

Closely related to adapting the order of execution is adapting the level of concurrency. If subtasks can easily be farmed out to newly available workstations, the level of concurrency can potentially increase to the level of available CPUs. Likewise, abandoning subtasks in response to busy transitions may temporarily decrease levels of concurrency.

A slightly different approach is changing the granularity of subtasks. Rather than rely on pre-runtime definitions of the amount of work to be done by each subtask, work may be shifted during execution. This approach could be used to reassign work from an abandoned task. It could also be used to increase utilization when a workstation's level of availability increases.

Current levels of CPU availability could be used as a factor in program-driven changes in concurrency. Such changes include deciding on the number of subtasks to spawn at some point during an execution.

2.3.2 Relation to Implementation Options

One of the goals of adaptive approaches is that the increased flexibility in the program design allows for simpler techniques to respond to busy transitions. For example, if the current process can be abandoned, the overhead of process migration can be avoided. The program adapts by reexecuting the subtask at a later point, or changing the granularity of another subtask by sending it any uncompleted work.

With respect to reacting to increases in CPU resources, all of the options suggested in Section 2.1 could potentially be part of an adaptive solution. This includes increasing the amount of work to be executed by a process, starting additional processes as workstations become available or sending work to waiting daemon processes.

2.3.3 Relation to Parallel Paradigm Options

Introducing adaptivity into message passing programs is difficult for two reasons. First, message passing is based on process to process communication. This contradicts with allowing flexibility in what process a subtask is executed by. Second the structure of having a sequential ordering of subtasks limits the amount of flexibility in the execution order of subtasks.

However, since a large percentage of the existing parallel programs fall into this category, it seems important to consider if and how they could be made to take advantage of some of the adaptive techniques suggested here. One possibility would be if the program dynamically determines concurrency levels, in a manner that considers the current level of CPU availability.

The Piranha project suggests that defining parallel programs as a set of tasks is the most desirable way to take advantage of adaptivity. (footnote) This approach allows for greater flexibility in that the subtasks are defined independent of a particular process. They can be executed at generic worker processes as workstations become available. They can be abandoned and reexecuted as workstations become busy.

2.4 Load Balancing and Scheduling Design Approaches

2.4.1 Description

The alternative to adaptivity is to react to changes in CPU availability through scheduling and load balancing techniques. This is a more process oriented design approach. The subtasks execute within a particular process state and it is the process that is manipulated to react to CPU availability changes. As opposed to adaptive approaches, the program is isolated from CPU allocations.

Concurrency levels are defined prior to execution. This makes it difficult or impossible to take advantage of increases in CPU availability. It also means that the defined number of concurrent process must be maintained even when the available CPU resources decrease.

2.4.2 Relation to Implementation Options

Implementation options fall into two categories, those that rely on the operating system scheduling of the individual workstations, and those that schedule the processes among available workstations on the network. Keeping the process at the same workstation requires adjusting the process priority in response to changes in user demands on the system. Scheduling among available workstations suggests process migration from a workstation when it becomes busy with user activity.

Implementation options for utilizing increases in CPU availability are somewhat limited in a scheduling/load balancing design. Since the level of concurrency is defined prior to execution, its difficult to dynamically take advantage of increases in CPU availability.

2.4.3 Relation to Parallel Paradigm Options

The same reasons that message passing programs are somewhat unsuited for adaptive approaches make them a suitable match for load balancing and scheduling approaches. Since the subprograms are bound to a particular process state, scheduling at the process level is the logical approach. Maintaining message passing communication between processes despite changes in process locations should be achievable based on similar routing approaches.

A task oriented paradigm could also be scheduled through priority control, or migrating processes between workstations. However these techniques do not take advantage of the flexibility that is the key attribute of a task oriented paradigm.

3.0 Taxonomy of Execution Models

In order to define all combinations of design approaches and functionality options, this section will provide a taxonomy of execution models. The design options, as defined in the previous section are adaptivity or scheduling/load balancing. The functionality options are:

- changing CPU allocations in response to increases or decreases in CPU availability
- changing CPU allocations in response to program requests to spawn/kill processes.

The taxonomy specifically does not include implementation options nor parallel paradigm options. The intent is that these models could be used as a framework for investigating which implementation and parallel paradigm options are compatible with the different execution models.

On the next page, Table 1 shows the taxonomy of execution models. Each row identifies one model. The columns marked by an X in that row identify which attributes are included in that model. The first two columns identify the categories of design as defined in Section 2.0 . The taxonomy assumes that all models must react to decreases in CPU availability, otherwise user response time would be degraded. The following columns identify optional functionality.

TABLE 1. Taxonomy of Execution Models

Model	Adaptive Program Approach	Scheduling/ Load Balancing Approach	React to Decreases in CPU availability	React to Increases in CPU availability	React to Program Spawn/ kill commands
1	X		X		
2	X		X	X	
3	X		X		X
4	X		X	X	X
5		X	X		
6		X	X	X	
7		X	X		X
8		X	X	X	X

3.1 Runtime Walkthroughs

This section provides a set of walkthroughs to identify the major runtime components and events for the models described in TABLE 1. They are provided as a basis for discussion of what parallel paradigms and implementation techniques would work best with the execution models. Runtime components are identified by boxes. Runtime events are identified by arrows.

The runtime components include:

- Subtask Resource Manager

  Software that is responsible for identifying subtasks that are ready to be executed.

- CPU Resource Manager

  Software that is responsible for maintaining which CPUs are available for the parallel program's use.

- Reduce or expand CPU usage

  This could be implemented by any of the implementation options described in Section 2.1

- Busy or idle transitions

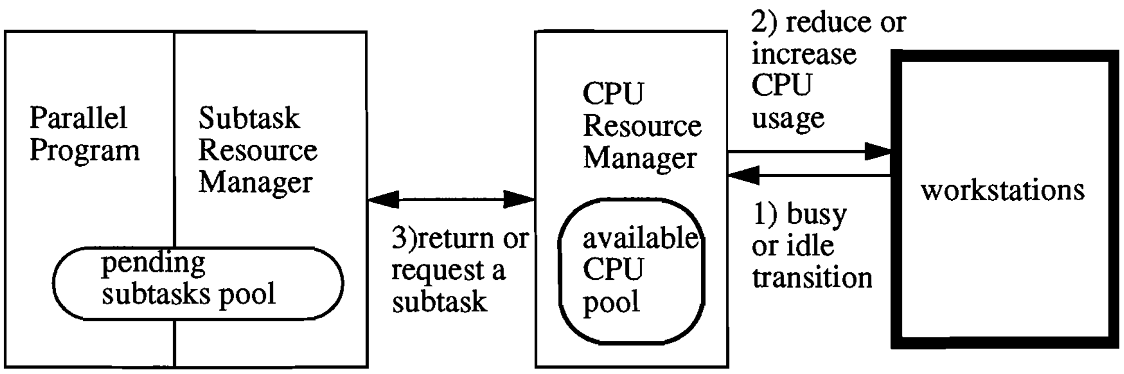
  Assumes a digital view of CPU resources. This event marks the workstation state changing from idle to busy or vice versa because of user activity.

To shorten the discussion, each submodel includes reacting to both increases and decreases in CPU resources. Although as defined in Table 1 a particular execution may potentially only include one of these features.

3.1.1 Adaptive approach to workstation driven changes

This walkthrough illustrates Models 1 and 2. It relies on adaptive interaction with the parallel program to react to changes in workstation availability. It is similar to the model implemented by the Piranha project (see 4.1.1).

FIGURE 1. Models 1 and 2 Runtime Walkthrough



1. CPU resource manager detects idle or busy transition. Pool of available CPU's is updated

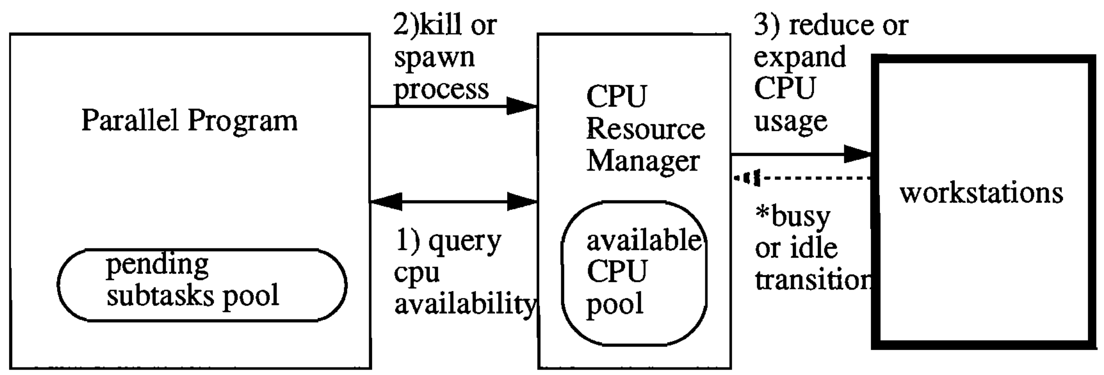
2. CPU resource manager reduces CPU usage at busy workstations or increases CPU usage at idle workstations.

3. Subtask resource manager is either returned work to be completed later, or requested for additional work.

3.1.2 Adaptive approach to program driven changes

This walkthrough illustrates Models 3 and 4. It incorporates data on current CPU availability in program driven changes in the level of concurrency. It was not found to be currently implemented by any research projects.

FIGURE 2. Models 3 and 4 Runtime Walkthrough.



1. Parallel program obtains information on CPU availability, and uses this in decision for amount of subtasks to spawn (or kill).

2. Parallel program issues kill or spawn commands.

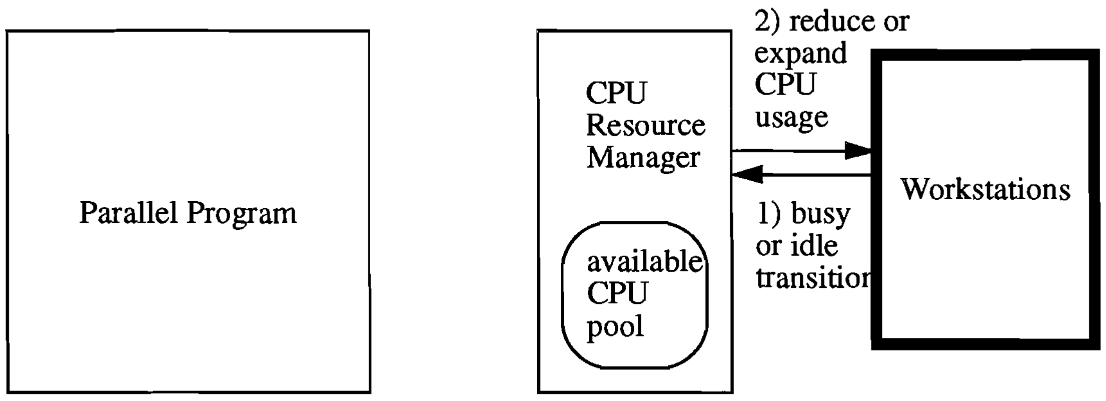
3. CPU resource manager allocates CPUs for new subtasks or deallocates CPUs for completed subtasks.

* Busy/idle transitions are being detected and the available CPU pool being maintained.

3.1.3 Load Balancing/ Scheduling approach to workstation driven changes

This walkthrough illustrates Models 5 and 6. It relies on scheduling processes among available workstations to react to changes in workstation availability. This is the model that is simulated as part of this project (See Section 6.0). It has not been addressed by other research projects. The closest research project were the co-scheduling papers (see 4.2.3). Their approach only differed in that resources were adjusted strictly at the workstation where the CPU availability changed, rather than among workstations.

FIGURE 3. Models 5 and 6 Runtime Walkthrough



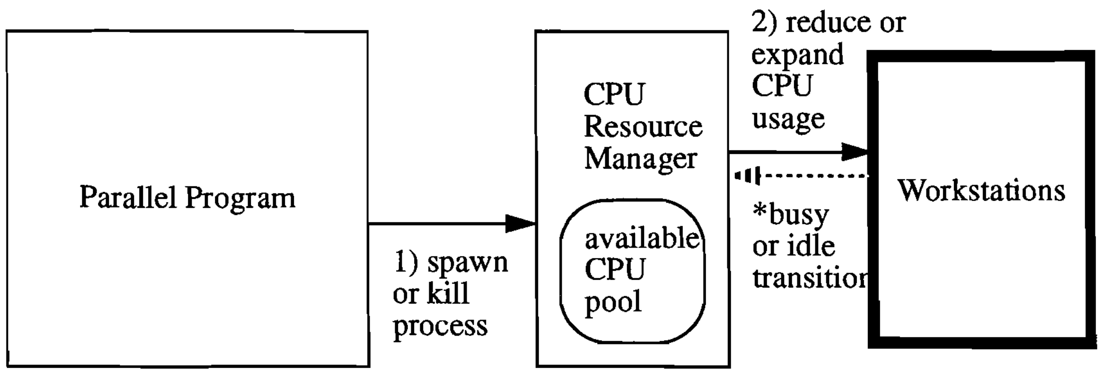
1. CPU resource manager detects idle or busy transition

2. CPU resource manager expands or reduces amount of CPU resources allocated and balances the currently executing subtasks accordingly.

3.1.4 Load Balancing/Scheduling Approach to program driven changes

This walkthrough illustrates Models 7 and 8. It shows a simple allocation environment, where the pool of CPUs available for allocations is maintained to reflect the current workstation availability.

FIGURE 4. Models 7 and 8 Runtime Walkthrough



1. Parallel program issues request to spawn or kill processes.

2. CPU resource manager expands or reduces the number of allocated CPUs based on programs request.

* Busy/idle transitions are being detected and the available CPU pool being maintained.

4.0 Related Work

In the introduction, we stated that most previous work either did not support execution of parallel programs[3,4,9,13.14], or supported the execution of parallel programs only on a dedicated network of workstations[12]. In this section, we will identify some of the related work and explain how it differs from this project.

The following taxonomies express the models of related work that support parallel and sequential executions. Each row identifies the attributes associated with the work named at that row.

TABLE 2. Taxonomy of Related Work that Supports Sequential Executions

Project	Adaptive	React to Decreases in CPU Availability	React to Increases in CPU Availability	Analog View of CPU Resources	Digital View of CPU Resources
Stealth	X	X	X	X	
Condor					X
Quahog	X	X			X

TABLE 3. Taxonomy of Related Work that Supports Parallel Executions

Project	Adaptive	Reacts to CPU Decreases	Reacts to CPU Increases	Analog View of CPU's	Digital View of CPU's	Support Task Oriented Design	Support message passing Design
Piranha	X	X	X		X	X	
Parform	X	X	X	X			
co-sched		X	X	X			
Marionette		X			X	X	
PVM					X		X

4.1 Adaptive Parallel

4.1.1 Piranha

Piranha is a current research project on "adaptive parallelism" being done at Yale University. The project defines software that "executes over a set of dynamically changing processors" as adaptive parallelism. They state that such software is "capable of taking advantage of new resources as they become available and of gracefully accommodating diminished resources without aborting."[7]

Piranha is designed to execute in the Linda distributed programming environment. It executes parallel programs that are defined as a set of tasks and do not depend on any particular set of processes. It is a strong example of a programming environment that supports the task oriented paradigm described in Section 2.2 . It relies on Linda's shared distributed tuple space for sharing task definitions and data.

The Piranha execution environment consists of two types of processes: a control process and piranha processes. The control process, called the feeder is responsible for distributing tasks to Piranha processes. Piranha processes are application dependent servers that accept and execute tasks from the feeder.

This model allows for new Piranha processes to be spawned when workstations become available. Since tasks are not tied to a particular process, they can be sent for execution at this new Piranha process.

When a workstation becomes unavailable, a retreat function defined for the current application is executed. The intent of retreating is to save any completed work, abandon any temporary state, and ensure that any uncompleted work is completed at some other point. Since Piranha relies on Linda's shared tuple space for identifying yet to be completed tasks, a simple retreat function need just return the uncompleted tasks to the tuple space.[6,7,11]

4.1.2 Parform

Parform is a software system for parallel programming on a network of workstations. It coordinates load balancing for the parallel program's processes with maintenance of workstation owner response time.

Load balancing is provided at the start of a parallel program's execution and during the execution. Workstation load values are factored into the initial division of work among subtasks. This differs from most parallel program environments which partition work at compile time. During execution, Parform monitors workstation loads and reacts by dynamically changing the division of work among subtasks.

To avoid interfering with the workstation owner, Parform reduces the parallel program's priority in response to user activity. The paper states that the execution time of tightly synchronized parallel programs can be significantly impacted by even minor changes in the priority of a particular process. However, the paper does not quantify this with data, or explain their interpretation of significant.

They counteract this impact, through the use of dynamic load balancing. During execution, Parform may change the amount of work assigned to subtasks. They suggest that this is easy for programs where the data is statically partitioned. In this case, subtasks can dynamically be directed to work on a changing amount of data. However, they recognize that a general solution would be difficult. They also state that in response to severe load changes it may be necessary to migrate an entire subtask to another workstation.[5]

4.2 Nonadaptive Parallel

4.2.1 PVM

PVM (Parallel Virtual Machine) provides a message passing system for execution of parallel programs on a network of heterogeneous computers. The main distinction of PVM from the work discussed here is that it is not designed to adapt to changes in workstation availability.[12]

4.2.2 Marionette

The Marionette system is another approach to distributed parallel programming. It relies on a master/slave execution environment, and shared data structures for communication between the slave processes. While it does not claim to adapt to workstation usage changes, it does allow for retreat and re-execution of work from a slave process in response to faults. [1]

4.2.3 Co-scheduling

Two papers produced at Ohio State University view the CPU resources from an analog perspective. Their approach, which they refer to as co-scheduling allows the parallel program subtasks to remain at a workstation when it becomes busy with user activity. To avoid degrading response time, the parallel process executes at a reduced priority. They claim that this avoids the overhead of migration and allows for a greater utilization of CPU resources. [2,10]

4.3 Sequential

4.3.1 Stealth

The stealth project allows for remote execution of sequential jobs. Their approach is to allow the remote job to remain at a workstation in conjunction with user activity, but at a reduced priority. They claim that generally 80-90% of CPU cycles on a workstation are unused. They also claim that techniques that migrate remote jobs to react to user activity still waste CPU cycles. However, the difficulty with their approach is that it requires operating system changes. Their results show that simply reducing the remote process's execution priority is not sufficient protection against degrading user response time. Therefore, they introduce operating system changes to provide priority controlled memory access and I/O. [13]

4.3.2 Quahog

Quahog is a system developed here at Brown for execution of remote sequential jobs on a network of user-owned workstations. It ranks workstations based on a method for interpreting system parameters to determine relative levels of user activity. Those workstations with the best ranking are chosen for execution of remote jobs. If a workstation becomes busy while a remote job is executing, the job will either be killed, or some simple migration techniques will be applied. [3]

4.4 Summary

Out of the above projects, only Piranha and Parform are designed for executing parallel programs on a network of user-owned workstations. The others are either designed for sequential programs or assume the workstations are dedicated resources.

Piranha provides the best example of an environment where the parallel programs easily adapt to workstation owner activity. However, it is limited in that it is designed specifically for the Linda environment. This leaves open the issue of incorporating adaptivity in message passing or other parallel paradigms.

Parform does allow for adaptivity in more of a message passing environment. However, it focuses on programs where it is relatively easy to load balance by changing the granularity of the subtasks. They recognize that this approach may not be generally applicable. They suggest that in some situations process migration may be necessary but have not yet investigated this approach.

Parform claims that reducing the priority of a parallel process significantly impacts the parallel program's performance. This contradicts with the results of our simulations, which show that reacting to busy transitions does not cause a major increase in the programs' execution times. One possible reason for the discrepancy is the approaches to reacting to a busy transition differ. Parform reduces the priority of the process, while our simulations add a discrete migration cost. It would be interesting to use the simulation tool to emulate a model where reduced priority is used. This would allow for a more direct comparison to Parform. It would also be interesting to get more information on the results they relied on to make the statement.

5.0 Collection of Workstation Usage Data

From September 1993 to February 1994, workstation usage data has been collected for 170 workstations in the Brown Computer Science Department. The data was collected on a twenty-four hour basis. The data records TTY activity and CPU load levels. CPU load level is the average number of jobs in the run queue.

The data was used to identify time segments when a parallel program could execute at a workstation without impacting user response time. These times were identified by no TTY activity and a load level below .25. When these criteria are met, the workstation is considered to be idle. A .25 load level was chosen based on an initial evaluation of the data. We found that during times periods with no active users processes, CPU load levels were often greater than zero. These load levels were due to system activities. An idle level of .25 safely filters out this type of activity.

A .25 load level was also considered low enough to not incorrectly label workstations as idle. A t this level there is a job in the run queue only 25% of the time. This suggests that even if some of the .25 load level is due to user processes, the CPU could service additional processes without delaying those that are currently active.

The data was collected by using the UNIX rwho facility. The rwho facility provides statistics on number of users, cpu loads and time since last TTY activity for each user. This data is updated every 3 minutes for all workstations on a local network. Parsing the rwho files in the department's file server's directory allowed access to data on all workstations in the department.

The collection code generates files that log all state transitions. A log file is created for each workstation. At three minute intervals, the collection code parses the rwho files to check for state transitions. When a transition occurs a transition record is written to the log file for that workstation The data included in the record is defined in Table 4.

TABLE 4. Transition Record Data

CPU load average for the last one, five and fifteen minute periods. Load average is the average number of jobs in the run queue.
Current number of users.
Idle time: The shortest time (in minutes) since any of the users have provided terminal input.
Timestamp of the transition.
Type of transition.

The categories of transitions are defined in Table 5. The categories were chosen to include

TABLE 5. Transition Categories

LOAD	ONE MINUTE CPU LOAD AVG. EXCEEDED IDLE CRITERION
TTY	TRANSITIONED FROM IDLE TO BUSY BECAUSE OF TTY ACTIVITY WITHIN ONE MINUTE PRIOR TO RWHO SNAPSHOT
BOTH	TRANSITIONED FROM IDLE TO BUSY BECAUSE OF BOTH LOAD AND TTY
CHANGE	LOAD AVERAGE FOR PREVIOUS MINUTE CHANGED MORE THAN TEN PERCENT AND LOAD IS GREATER THAN IDLE CRITERIA
IDLE	NO TTY ACTIVITY IN THE LAST MINUTE AND ONE MINUTE LOAD AVG. WAS LESS THAN IDLE CRITERIA
START	WORKSTATION BEGAN PROVIDING RWHO DATA
FINISH	WORKSTATION STOPPED PROVIDING RWHO DATA

as much information as possible in the logs. This is intended to allow the data to be used for further analysis of the workstation activity. For example, future work may decide on a different definition of idle. Future work may also be interested in evaluating the data with respect to only TTY activity or only load levels.

The idle criteria is, as explained above, a CPU load of less than .25. The CHANGE transition records CPU load changes of greater than ten percent. Ten percent was chosen to filter out minor changes in CPU load. Future data collection may decide to modify this interpretation of significant CPU load and/or significant load changes.

5.1 Interpretation of Data

After the data was collected, a portion of it was interpreted for use by the simulations.Which workstation's data was used was based on the order that their data files were read. Each simulation run defined a time segment for the workstation data. A workstation was not used if there had not been a transition prior to the start of the time segment, and a transition after the end of the time segment. This is required to determine the state of the workstation at the start of the time segment, and to ensure that data was being collected for the full duration of the time period.

For the purposes of the simulations, each workstation is modeled as either busy or idle at any particular point in time. The following processing is done to transform the raw data in the log to a sequence of idle and busy time periods.

The workstation data log categories: BOTH, TTY, LOAD, CHANGE are all interpreted as identifying busy time periods. The IDLE category is interpreted as identifying idle time periods.

Since the rwho data only reflects the state of the workstations at three minute intervals, it is necessary to do some calculations to more accurately define the transition times. Determination of transition times is complicated by the fact that rwho provides more information for TTY activity than for load. rwho records the time (in minutes) since the last TTY activity, so its possible to determine exactly when TTY activity caused a busy transition. In contrast, load data is only taken as snapshots at the three minute intervals. This means some estimation has to be done as to the exact time when a load change caused a busy transition.

There are four possible combinations for consecutive transitions. Although the rwho data is sampled every three minutes, a record is only logged to the workstation usage data file when a transition has occurred so its possible for the time between transition records to be greater than three minutes. The five transition sequences, how their timestamps are determined and any margin of error are described in.Table 6.

There is also a potential for inaccuracy if busy time periods occur entirely between the three minute rwho polling intervals. These would not be detected by rwho. This could lead to inaccurately long idle periods. However, since in order for this to occur the busy periods would have to be less than three minutes in length, the potential for a series of these to sufficiently affect the length of an idle period seems relatively small.

The timestamps are defined in order to determine the duration of idle and busy periods at individual workstations. The timestamps are not being used to determine the duration of time between events at different workstations. This removes the importance of another potential area for error; lack of synchronization among the workstation clocks.

TABLE 6. Determination of Transition Times*

Transitions as described in workstation usage log*	**Occurs when:**	**Transition times calculated for simulations***	**Margin of error***
Prev Trans: IDLE Curr. Trans: IDLE Idle duration is less than time between curr and prev. records	There were two transitions, first the node transitioned to busy, and then it transitioned back to idle. At time of current record, there has been no TTY activity for a time of idle duration and the current load is less than the idle criteria.	Transition 1 (idle ->busy): Estimated as: (prev time + (curr time - idle duration - prev time) / 2) Transition 2: (busy->idle) curr time -idle duration	Transition 1: Since this occurred within a three minute time period, and the idle duration was at least 1 minute, the margin of error is at most 1 minute. Transition 2: no margin of error
Prev. Trans: BUSY Curr. Trans: IDLE Idle duration is greater than time between curr and prev records	a user that caused a busy state at the time of previous rwho record is no longer on the system	(busy->idle) Estimated as: current time	3 minutes, since the transition could have occurred any time since the prev record.
Prev. Trans: BUSY Curr. Trans: IDLE Idle duration is less than time difference between curr and prev. records.	At time of current record, there has been no TTY activity for a time of idle duration and the current load is less than the idle criteria.	(busy->idle) curr time -idle duration	None
Prev Trans: IDLE Curr Trans: BUSY	Either: The CPU load has increased above the idle criteria or there has been TTY activity within the last minute.	(idle->busy) Estimated at: (curr time - prev time) / 2	Transition occurred sometime in 3 minute span, and is estimated as occurring at halfway point, the margin of error is 1 1/2 minutes.

*Definitions:

- Prev time: The time of the previous transition record in the workstation usage log.
- Current time: The time of the current transition record.

Idle duration: The length of time that there has been no TTY activity.

6.0 Simulations

6.1 Goals

To a large extent this project is about timing. How do delays introduced by reacting to changes in CPU availability impact the timing of a parallel program?

There are many related questions.

- How is the timing impacted by varying lengths of delays?

  by varying frequency of the delays?

  by the characteristics of the delay? for example a subprogram being slowed down by a priority reduction vs. a subprogram being totally suspended?

- How do does the structure of the parallel program alter the impact of the delays? For example, the level of concurrency, or the granularity of the subtasks.

A set of simulations were executed to investigate answers to some of these questions. Specifically, we used the simulations to experiment with using migration in conjunction with message passing parallel programs in the workstation environment.

We estimated a range of costs for process migration. We then ran a series of simulations that applied the different migration costs whenever a busy transition occurred. This exposed the relationship between the relative migration costs and percentage slowdown of the program's execution time. Migration was chosen because it has not been previously addressed with respect to parallel programs in this execution environment.

In addition, we used the simulations to gather information on how the workstation environment impacted the completion time of parallel programs. To accomplish this, the simulations were run against a range of the following two attributes:

- Number of workstations available for execution.
- Level of user workstation activity (frequency of busy transitions, duration of idle periods).

6.2 Extent of Simulations

A total of 400,000 simulated executions representing 400,000 combinations of the following variables were performed:

- Number of workstations on simulated network.
- Time period of workstation usage data. This defines the start and end hour of the workstation usage data that was used to emulate level of user activity.
- Range and increment of migration costs. The migrations costs were considered, in one second intervals, in the range from one to 100 seconds.

How the number of simulations broke down among these variables i s defined in Table 7.

TABLE 7. Breakdown of Variables for Simulated Executions of Migration Model

Size of Simulated Network	**Range of Migration Costs**	**Increment of Migration Costs**	**Number of different time periods of workstation usage data the simulations were run against.**	**Total Simulated Executions**
Thirty workstations	1-100 (secs.)	1 second	1200	120,000

TABLE 7. Breakdown of Variables for Simulated Executions of Migration Model

Size of Simulated Network	Range of Migration Costs	Increment of Migration Costs	Number of different time periods of workstation usage data the simulations were run against.	Total Simulated Executions
Fifty workstation	1-100 (secs.)	1 second	1600	160,000
One hundred workstations	1-100 (secs.)	1 second	1400	140,000

The fourth column defines the number of different time segments selected from the workstation usage data. For example, simulations were run against the workstation usage data that was collected between 10:00 and 11:00 a.m. on Friday Dec. 19. The time segments were selected to provide a cross section of the workstation usage data. They covered two different days from nine weeks spanning from October to January. The days were selected with the intent of providing equal coverage of the different days of the week.

The final column gives the total number of simulations.

6.3 Parallel Program Traces

The simulations were run against traces of four message passing parallel programs. The programs were originally executed on an Intel hypercube. The Intel hypercube supports fast and frequent message passing. This makes them a good choice for use as sample programs. Their design assumes message passing is cheaper than programs designed for a network environment. Therefore, if its possible to minimally impact their execution times, programs with more conservative message passing strategies should also be minimally impacted. The traces include the time, source and destination of each send and receive. Each of the programs required sixteen processors and a host processor.

TABLE 8. Hypercube Traces

Program	**Length of execution on Hypercube**
FFT Fast Fourier Transform	304032 (msecs.) 5.07 (min.)
TEST Circuit Test generator	148000 (msecs.) 2.46 (min.)
MSH Finite Differences	195663 (msecs.) 3.26 (min.)
DET Matrix Determinant	375507 (msecs.) 6.26 (min.)
Average across all four programs	255600 (msecs.) 4.26 (min.)

Within the simulations, the parallel subprograms are modeled as a series of subtasks. The duration of the subtasks is calculated as the time between successive receives. No blocking time is included in the calculation of the subtask duration.

6.4 Simulating Execution

The simulation tool models the parallel programs in the following manner. Each subprogram progresses by sequentially accruing the execution times of its subtasks. A signalling mechanism is used to simulate the send and receipt of messages. To simulate a send, a signal is sent to the destination subprogram. To simulate a receive a subprogram blocks until it is signalled by the source of the message. This use of signaling allows the simulations to synchronize the subprograms' progress. The simulation continues until the time for subtasks for all subprograms have been accrued.

At any point during the execution, each subprogram is allocated to a particular workstation. Allocations of workstations to a subprogram are done strictly in a LIFO manner. The last workstation that was added to the idle pool, will be the one allocated. If there are no idle workstations, the subprogram blocks until a workstation becomes idle.

When a subprogram's allocated workstation becomes busy, the subprogram is migrated. To model migration, a subprogram is allocated to a different workstation. The defined transition cost is added to the migrated subprogram's execution time. If after adding the migration cost, time has progressed to a point where the new workstation is busy with user activity, the subprogram is migrated again immediately.

Migration occurs whether the subprogram is executing a subtask or blocked at a receive. If the subprogram is blocked, any migration cost that occurs within the blocking time is hidden. In other words, since the subprogram can not currently proceed anyway, there is no additional cost for it to use the time to migrate. If the migration cost is greater than the blocking time, the subprogram is delayed by only the difference between them.

Throughout the simulated execution, the execution time for each parallel subprogram and the critical path are being calculated. These are calculated by accruing the computation time of each subtask and the blocking time for each receive. Blocking time is determined by comparing the time the message was sent to the time the receive was issued. The greater of the two is the new current time for the subprogram receiving the message.

6.5 Output

The following results are logged for each execution.

TABLE 9. Execution Data

Data	**Description**
Migration Cost	Performance overhead for each migration
Execution time	Program completion time (msecs.)
Percentage slowdown	Percentage slowdown between this execution and the same execution when the migration cost is zero.
Total Busy transitions encountered during execution of the parallel program	Number of times a node became busy with user activity while the parallel program was executing there.
Number of busy transitions encountered on critical path.	Number of busy transitions on that occur on workstations executing code that is on the critical path
Average node idle time during executions	Average duration of the time periods on potential nodes when node was not busy with user activity.
Number of nodes used	The number of nodes that were allocated at some point to the parallel program.

7.0 Simulation Results

The simulations results suggest that a network of user-owned workstations could provide a reasonable platform for message passing parallel program execution. The results show that if the cost of migrating a process is between one and 50 seconds, the program's completion time is impacted by a slowdown of less than 100%.

However, as migration costs increase linearly, the corresponding percentage slowdown tends to increase at a superlinear rate. This result suggests that a linear increase in the cost of reacting to a busy transition has a superlinear impact on the parallel program's performance.

The simulations also show some correlation between the percentage slowdown and the number of workstations on the simulated network. The number of workstations impact the percentage slowdown rate when it is less than twice the number of workstations required by the program. When the number of workstations is varied from three to five times the number required, there is little change in the rate of percentage slowdown.

The results are based on the average percentage slowdown for all executions under the defined conditions. (Results for the individual parallel programs are provided in Appendix 10.1 .) The percentage slowdown is in comparison to execution time when the migration cost is zero.

To determine the accuracy of the average as a measure of the percentage slowdown, the standard deviations were calculated. They are shown in Table 12, grouped by varying number of workstations. For 50 or 100 workstations, the standard deviation is very small.

TABLE 10. Standard deviations

Number of Workstations	Range of Standard Deviation from Average Percentage Slowdown
30	.04% to 33.00%
50	.01% to 4.20%
100	.01% to 3.84%

This suggests uniformity in the results. For 30 workstations the standard deviation was closer to ten percent of the slowdown values. This suggests that the results for 30 workstations should be considered as defining a range for the expected percentage slowdown.

This following sections identify major trends in the results. Theses sections include the averaged results for all four sample parallel programs. The plots for the individual programs are provided in Appendix 10.1 .

7.1 Statistical Results

An overview of the results is provided by the following statistics. On average, a migration cost of up to 20 seconds results in only a 10% slowdown. Even at migration costs of over 50 seconds, the percentage slowdown is only 100%.

TABLE 11. Transition Costs producing sample percent slowdowns

Number of Nodes	Migration cost that results in 5% slowdown	Migration Cost that results in 10% Slowdown	Migration cost that results in 50% slowdown	Migration cost that results in 100% slowdown
30	10 seconds	20 seconds	46 seconds	58 seconds
50	9 seconds	17 seconds	50 seconds	78 seconds
100	11 seconds	20 seconds	56 seconds	89 seconds

The smallest migration cost considered is one second, and in all cases this causes less than a 1% slowdown. At ten seconds, the percentage slowdown is still only 5%.

Only when the migration cost increases beyond 50 seconds does the number of workstations introduce a difference of more than 15% in the average percentage slowdown.

TABLE 12. Percent Slowdowns associated with sample transition costs

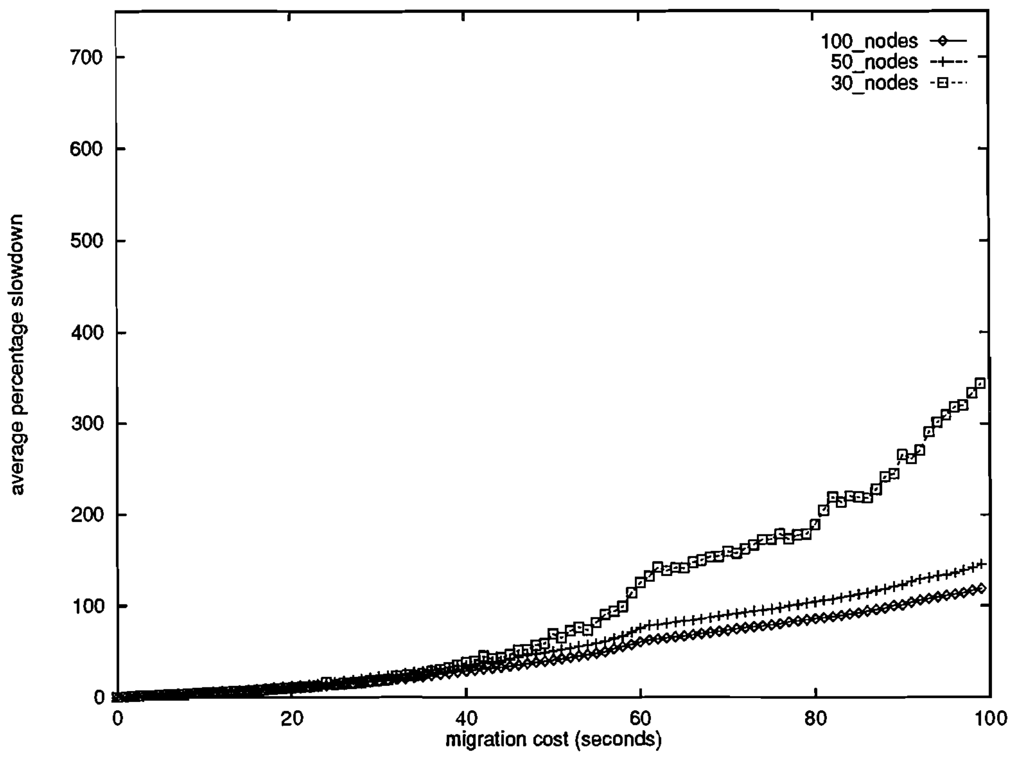
Number of workstations	**1 second**	**10 seconds**	**50 seconds**	**99 seconds**
30	.41%	4.83%	69.36%	343.85%
50	.4%	5.62%	50.68%	145.89%
100	.32%	4.62%	40.86%	119.14%

7.2 Results organized by varying number of workstations

Grouping the results by number of workstations on the simulated network identifies the following trends:

- Overall impact of migration costs on percentage slowdown.
- Rate of change of percentage slowdown with respect to increasing migration costs.
- Relation between number of workstations and percentage slowdown.

FIGURE 5. Average percentage slowdown for varying number of workstations



7.2.1 Impact of migration costs on percentage slowdown of execution time

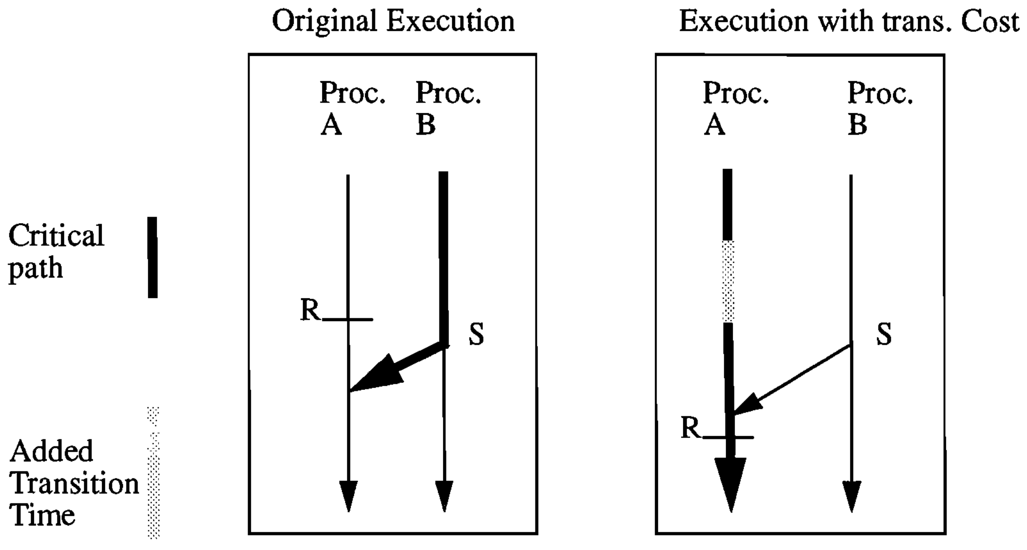
For 50 or 100 workstations, migration costs in the range of one to 100 seconds cause a percentage slowdown of less than 200%. A 100 second migration cost is from 27% to 67% of the original execution times (Defined in Table 8) of the parallel program emulated by these simulations. Thus, for these programs a migration cost of 25% of the initial execution time introduces a slowdown of no more than 200%. However, these programs all have execution times that are less than ten minutes. Future work could investigate how the relation between migration cost and percentage slowdown changes for longer programs.

For 30 workstations, the slowdown stays below 200% for up to 80 second migration costs. An 80 second migration time is from 21% to 54% of the original execution time (Defined in Table 8) of the parallel programs emulated by these simulations.This shows that for 30 workstations, a migration cost that is 20% of the initial execution results in an increase of less than 200%.

Possible reasons for this small impact are a low rate of busy transitions on the critical path or on the execution paths of all the parallel processes. Obviously, the addition of transition costs to the critical path increases the program's execution time. In addition, a process being delayed by a transition may alter the critical path. This can have additional, less predictable impacts on execution time.

An example is show in Figure 6. In the original execution, Process A blocks on a receive. In an execution with an added transition cost, Process A may be sufficiently slowed so that it no longer blocks. Thus the receiving process is now on the critical path, rather than the process sending the message.

FIGURE 6. Critical Path altered by added transition cost.



The following figures suggest that the size of the percentage slowdowns can be explained by a low rate of transitions on the critical path. It is important to note the short execution times of the simulated parallel programs. It is an open issue as to what the rate of transitions would be for programs with longer execution times. Figure 7 shows the average number of transitions during a parallel program's execution and Figure 3 shows the average number of transitions on the critical paths.

There are never more than 20 transitions, and there are 17 processes involved in the parallel programs. This suggests an average of only slightly more than one transition per process.

Using the simulation run with 30 workstations and a 100 second migration cost as an example, the number of transitions on the critical path seems to be the key contributor to the average percentage slowdown. The average initial execution time of the parallel programs is 255 seconds. For this simulation the average number of transitions is approximately seven and a half. Since the migration cost is 100 seconds, this would add 750 seconds to the total execution time, which is almost exactly a 300% increase. This provides a strong correlation between the average number of transitions on the critical paths for this particular simulation run.

FIGURE 7. Average total transitions for varying number of workstations

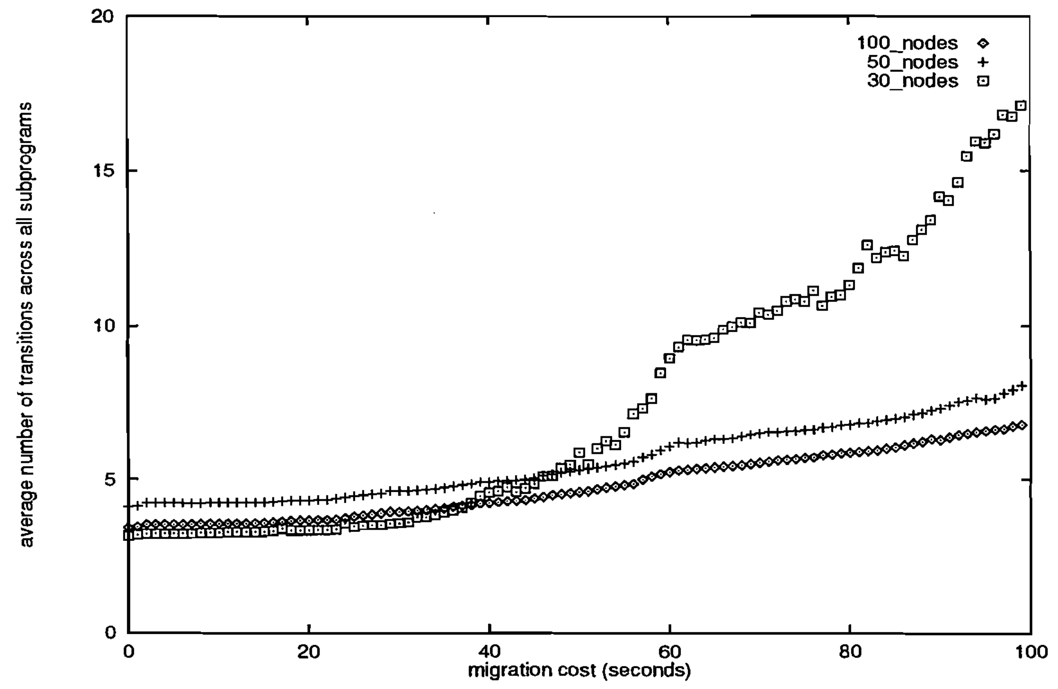


FIGURE 8. Average transitions on critical path for varying number of workstations

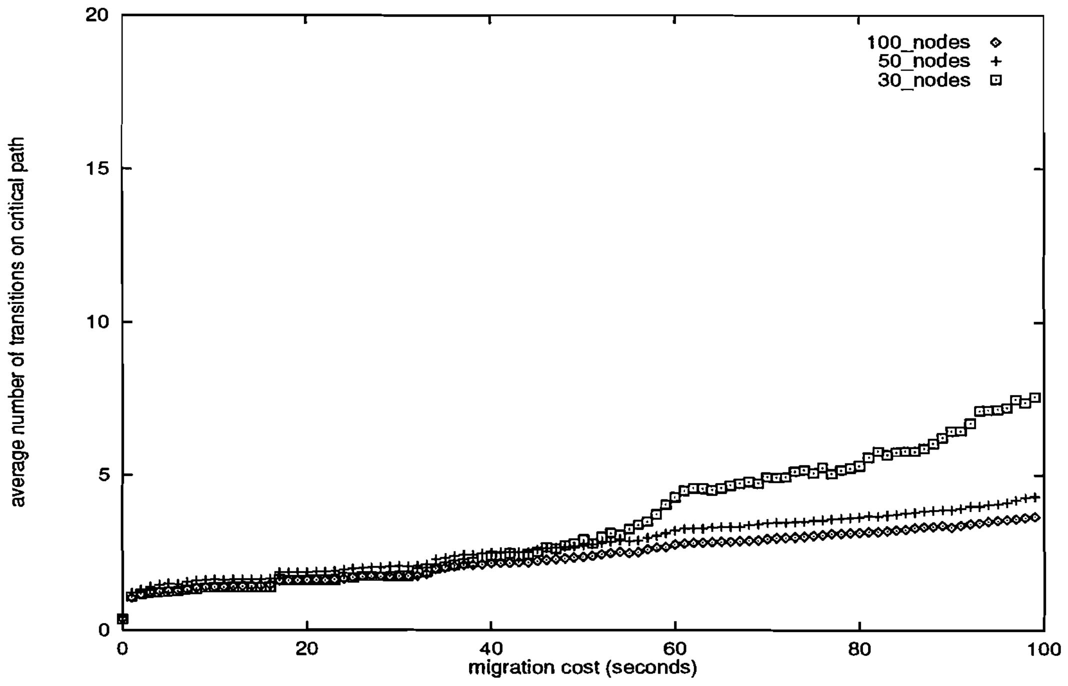
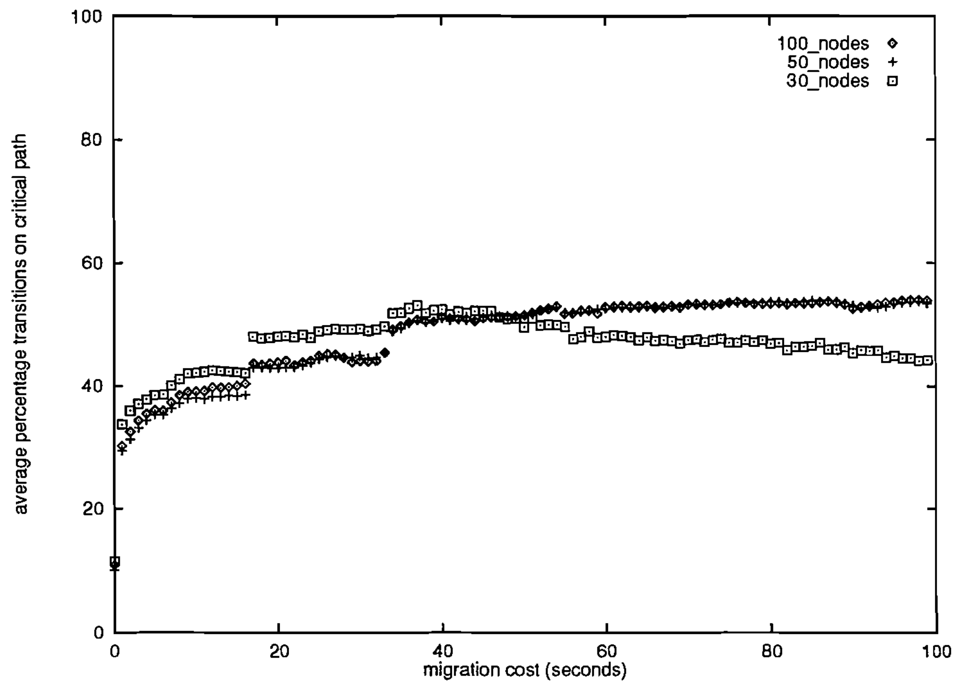


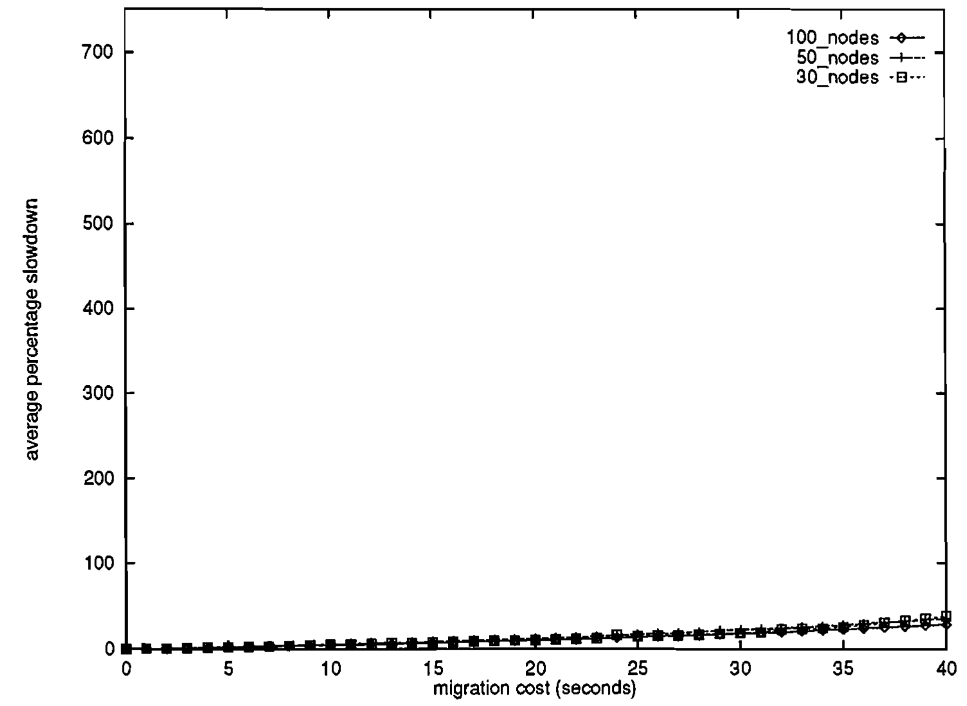
FIGURE 9. Average percentage of transitions on critical path for varying number of workstations



7.2.2 Rate of increase in percentage slowdown with respect to migration costs

Another important trend shown by Figure 5 is that the performance degradation occurs at a superlinear rate. Although, as Figure 10 shows, this is not apparent until migration costs are greater than 40 seconds. With respect to this particular simulation, this reduces the importance of the trend, since it seems likely process migration could be implemented to execute in less than 40 seconds.

FIGURE 10. Average percentage slowdown vs. migration cost of 1-40 seconds



However, it is important to attempt to explain the superlinear performance degradation, since it may suggest general trends in the relation between parallel program performance and the introduction of delays caused by reacting to busy transitions.

Potential explanations are

- The total number of migrations increases superlinearly - thus impacting the critical path in a superlinear manner.
- The total number of migrations increases linearly, but the percentage of migrations that occur on the critical path increases super-linearly.

- The number of migrations does not increase super-linearly, but the impact is a super-linear increase of the blocking time on the critical path. A possible, though as yet unsupported, reason for this is that multiple dependencies between the subprograms magnifies the impact of the added migration cost.
- The increase in migration cost alters the critical path in some other manner that results in the superlinear increase.

Figure 2, Figure 3 and Figure 4 rule out the hypotheses relating to the number of transitions. They show that from all perspectives the number of transitions is either constant or increasing at a less than linear rate.

To evaluate the other hypotheses relating to total blocking time and changes in the critical path, it would be necessary to do additional simulations that breakdown the time on the critical path into the following categories:

Computation time + blocking time + total migration cost.

By understanding the rate of increase of the 3 areas, and their relative percentage of the entire critical path, it should be possible to describe how the critical path is being affected superlinearly by the migration costs. This is left to future work.

7.2.3 Impact of number of workstations on percentage slowdown of execution time

Figure 5 also show the impact of the number of workstations in the pool of potential CPU's for the parallel programs. The major trend here is in the difference between 30 workstations and 50 or 100 workstations. With only 30 workstations, for the higher migration costs, the rate of performance degradation is much greater. The logical reason for this would seem to be that process's have to wait longer for idle workstations. To confirm this, additional simulations would have to be run to record the blocking time. An additional reason is explained by Figure 7 and Figure 3, which show the number of migrations increases at a greater rate for 30 workstations. This may be caused by the current design of the simulations, which does not enforce that the same subset of workstations are used for the varying number of workstations. To more accurately reflect the impact of the number of workstations, the simulations should allocate idle workstations on a FIFO basis. This would ensure that the first workstations chosen out of a sampling of 100, would be the same as those chosen out of a sampling of 30.

It is also interesting to note that when the migration cost is less than 40 seconds, the rates are almost the same. In fact the 30 workstation executions show a slightly smaller slowdown. Again, this should be reexamined after modifying to a FIFO allocation strategy.

7.3 Results grouped by time period of execution

To show the impact on the time of period execution on performance, the results were categorized as occurring within one of the following four time periods:

• night	Midnight - six a.m.
• morning	6:00 a.m. - Noon
• afternoon	Noon - 6:00 p.m.
• evening	6:00 p.m. to Midnight

FIGURE 11. Average percentage slowdown for varying time periods on 30 workstations

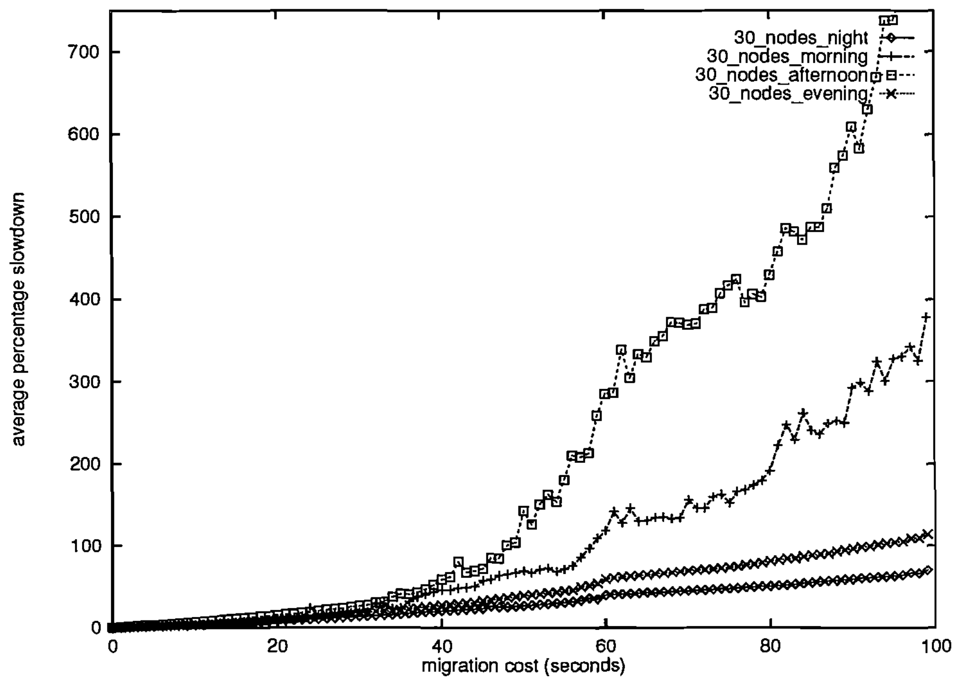


FIGURE 12. Average percentage slowdown for varying time periods on 50 workstations

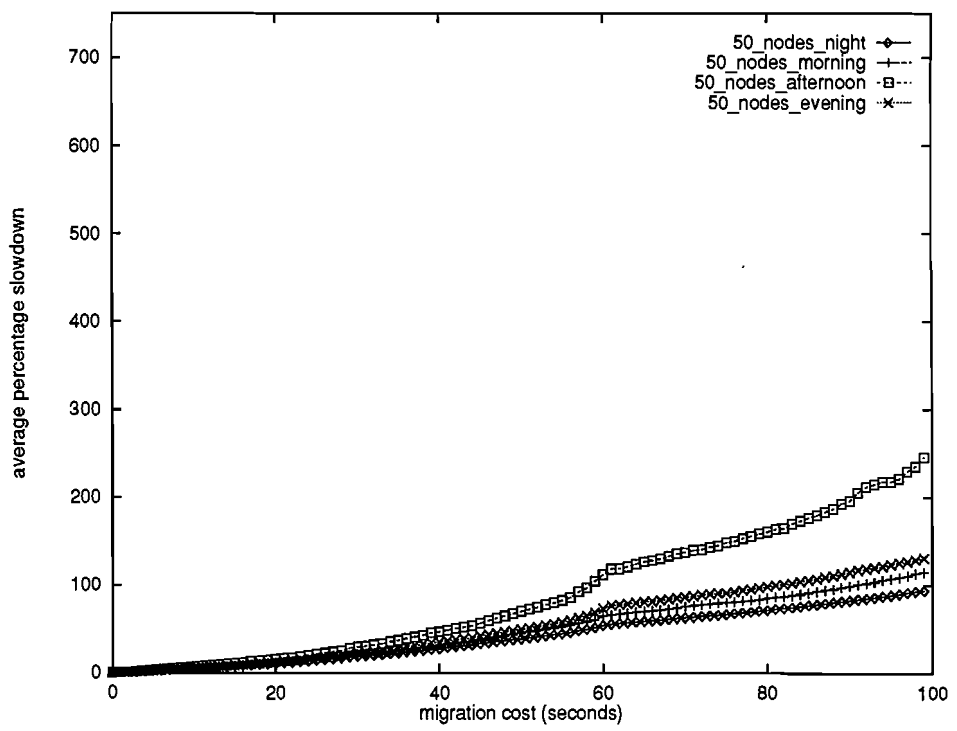
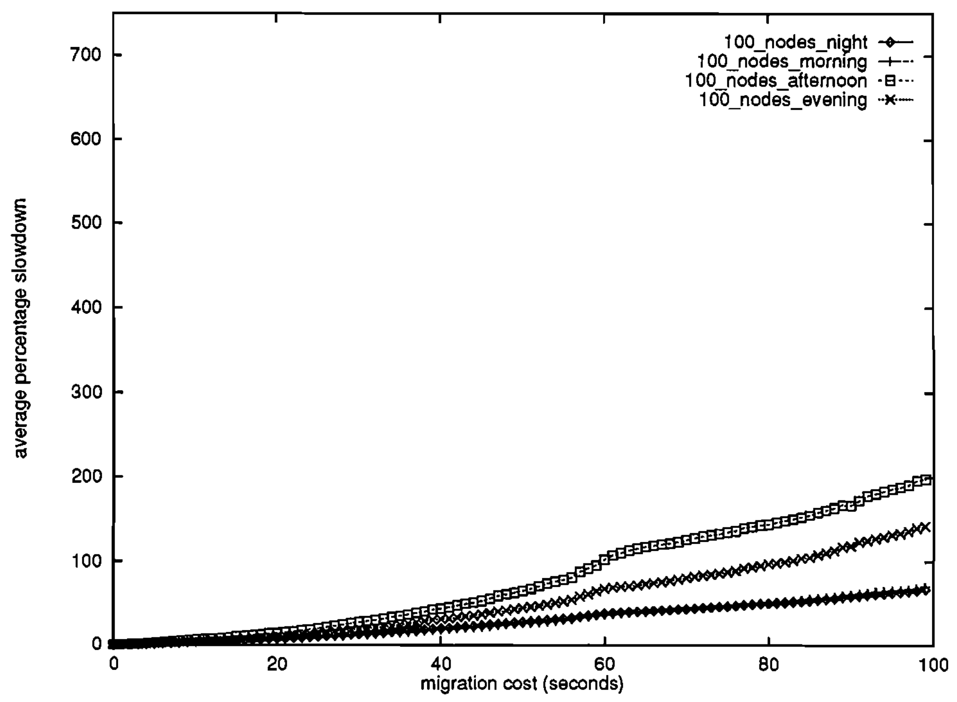


FIGURE 13. Average percentage slowdown for varying time periods on 100 workstations



Grouping the executions by time of day provides the results that one would expect. The performance rates align with the expected levels of user activity during the day. The plots confirm the expected result that the night time hours would provide the best potential for efficient executions. The expected reason for these results is that the number of busy transitions reduces during the time periods with less user activity. This reasoning is supported by the Figures 14-16.

FIGURE 14. Average transition count for 30 workstations for different time periods

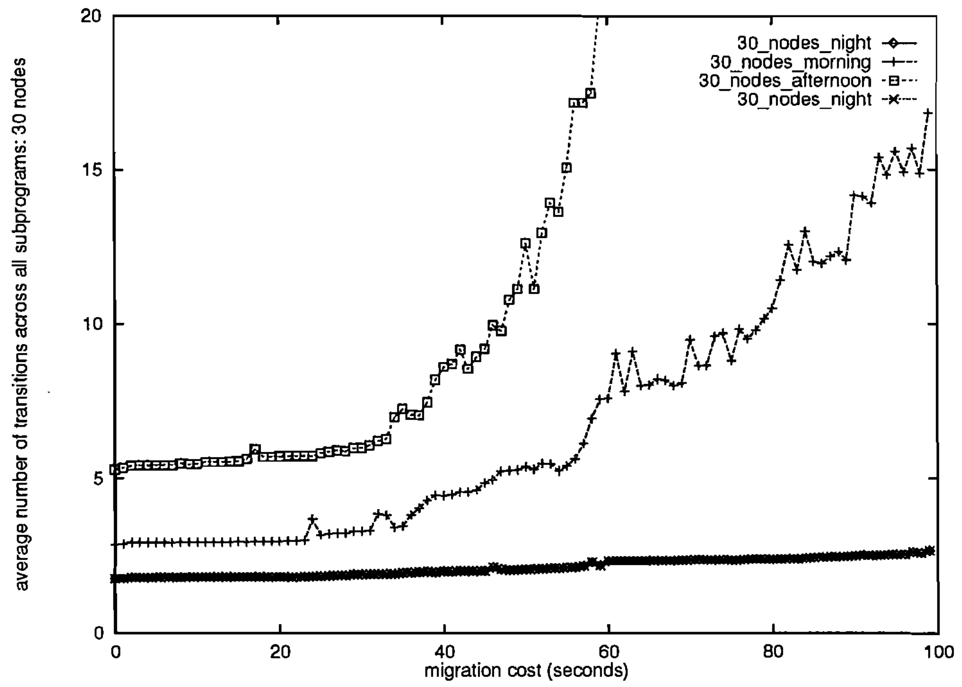


FIGURE 15. Average transition count for 50 workstations for different time periods

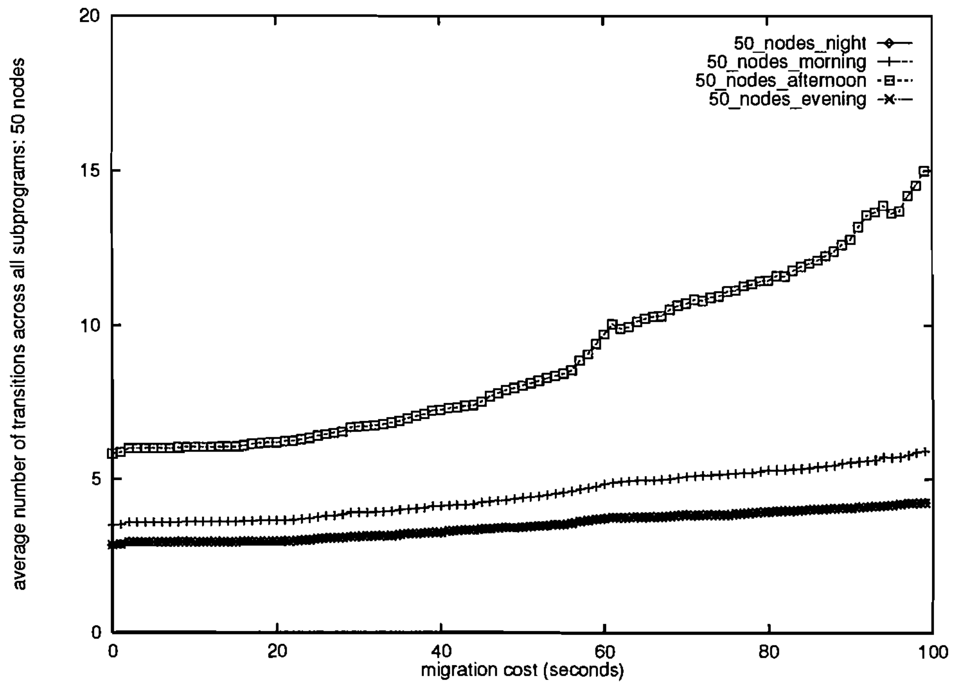
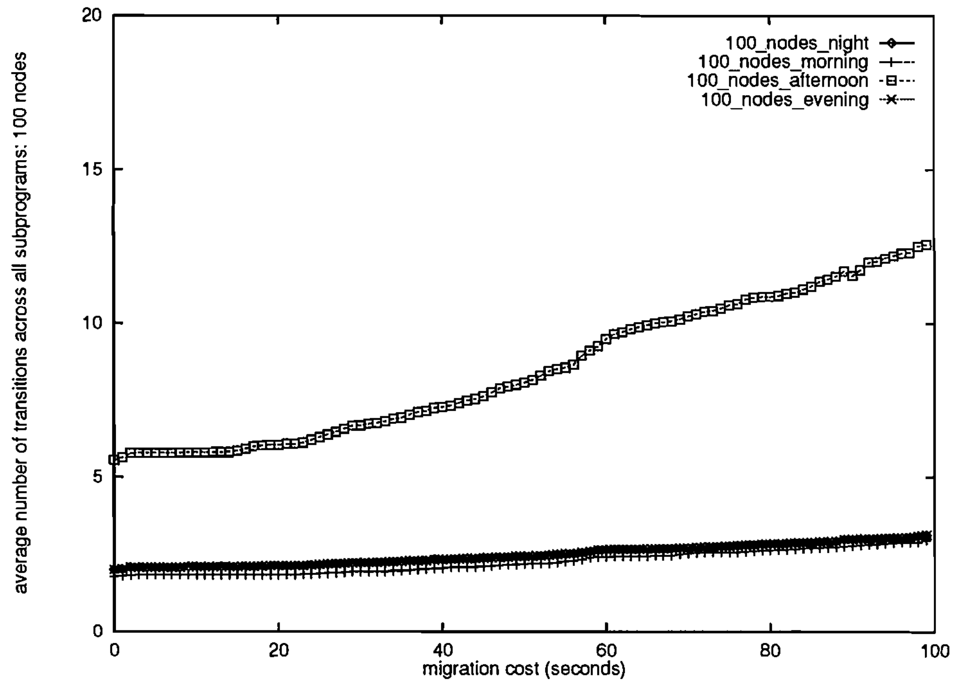


FIGURE 16. Average transition count for 100 workstations for different time periods



7.4 Results by varying workstation activity levels

To show the relation between workstation availability and performance, the results were grouped by average duration of workstation idle time segments. The idle times were divided into the following categories:

- 0 - 20 minutes
- 20 - 40 minutes
- 40-60 minutes
- Greater than 60 minutes

FIGURE 17. Average percentage slowdown for varying workstation activity levels on 30 workstations.

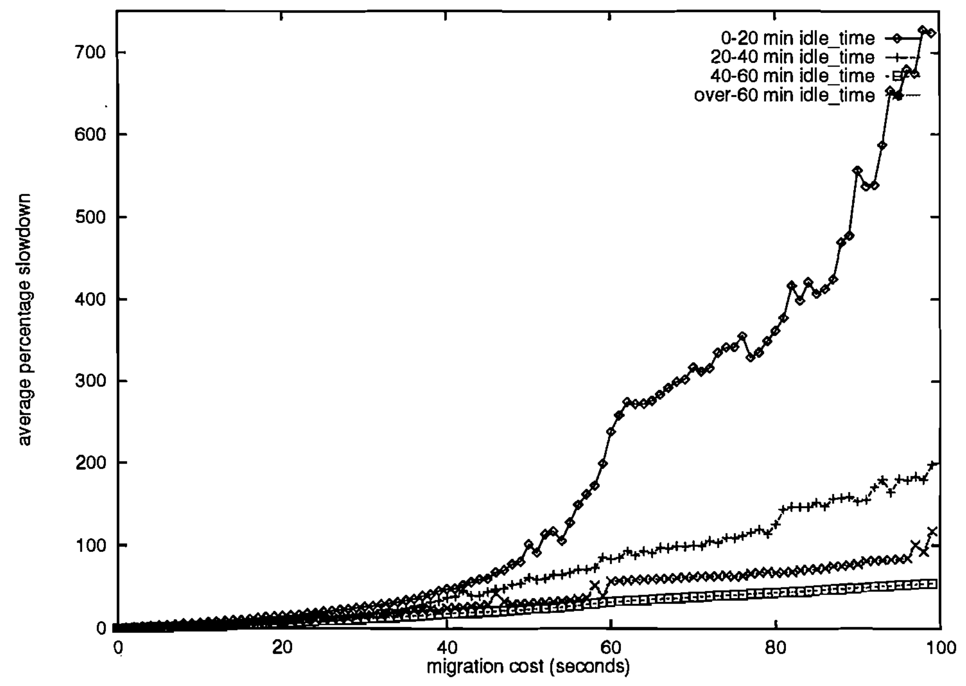


FIGURE 18. Average percentage slowdown for varying workstation activity levels on 50 workstations

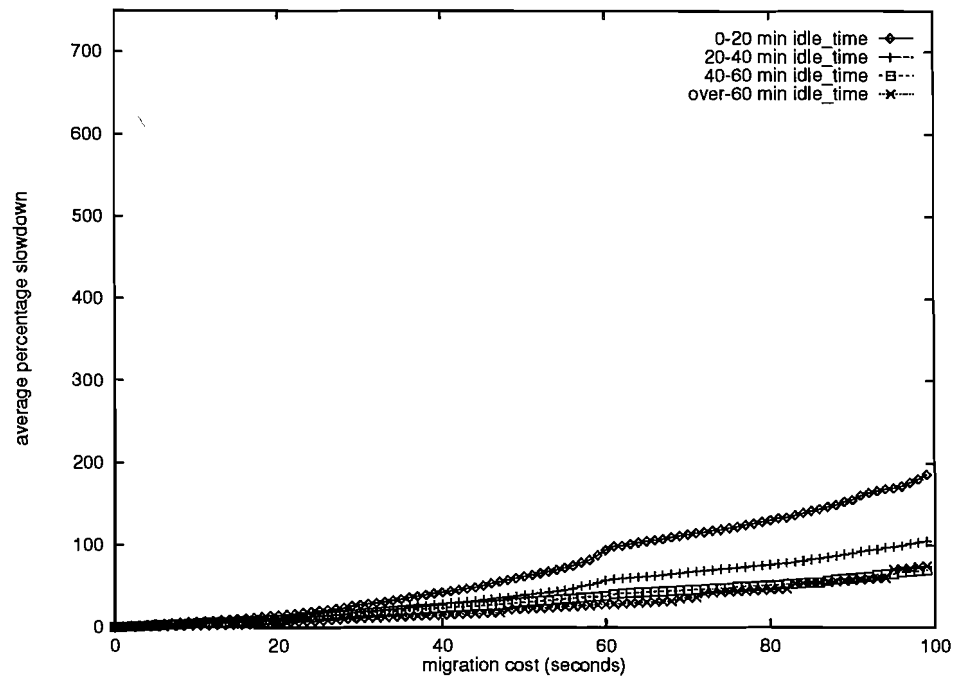
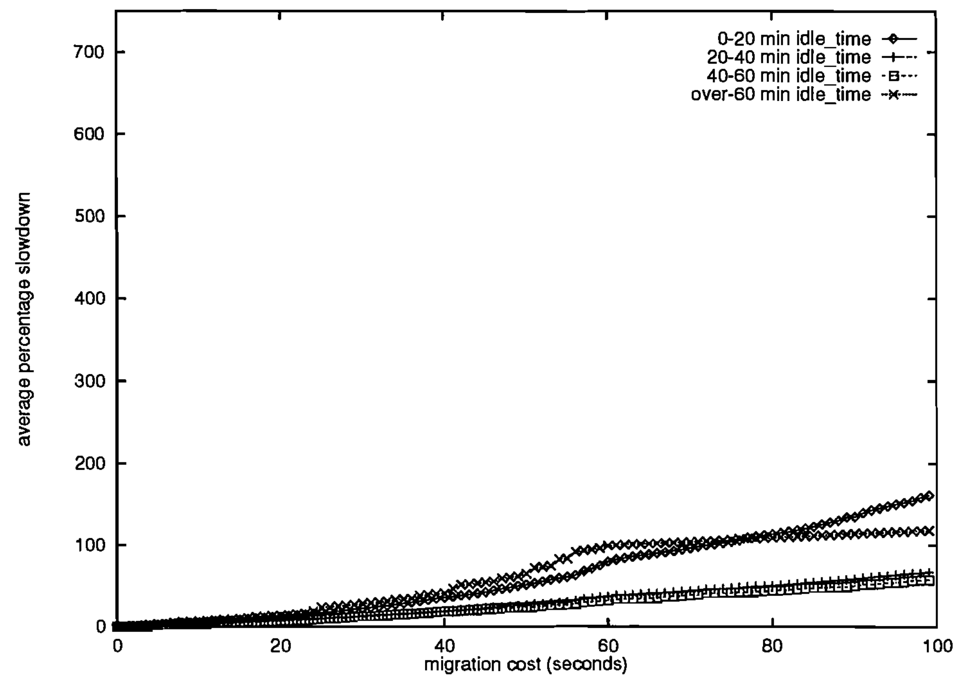


FIGURE 19. Average percentage slowdown for varying workstation activity levels on 100 workstations.



Grouping the results by average duration of idle times on the set of potential workstations show, again as would be expected, the shorter idle times produce a greater performance degradation. In particular with 30 and 50 workstations, idle times of less than 20 minutes cause the performance to degrade at a greater rate. With 100 workstations there is less of a correlation, in fact 20 to 40 minute idles times produce less degradation than idle the idle times over 60 minutes. This can be explained by Figures 16-18 which show the number of transitions on the critical path for 30 and 100 workstations, averaged by duration of idle times. With 100 workstations, the number of migrations does not increase dramatically with respect to the duration of the idle times.

FIGURE 20. 30 Workstations. Trans on Critical path, averaged by idle time

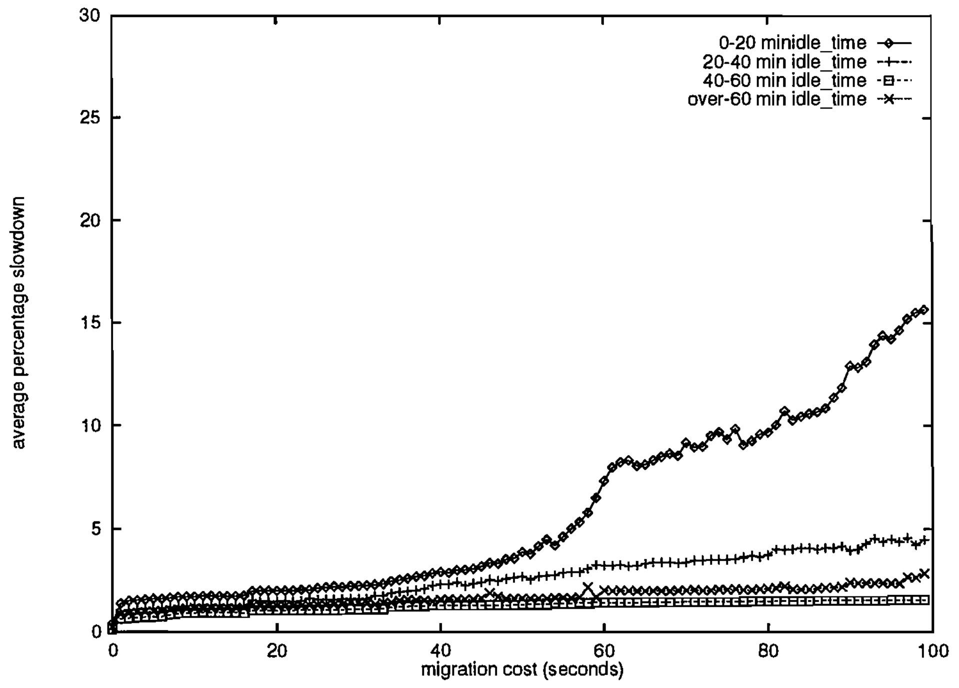


FIGURE 21. 50 Workstations. Trans. on Critical Path, averaged by idle time

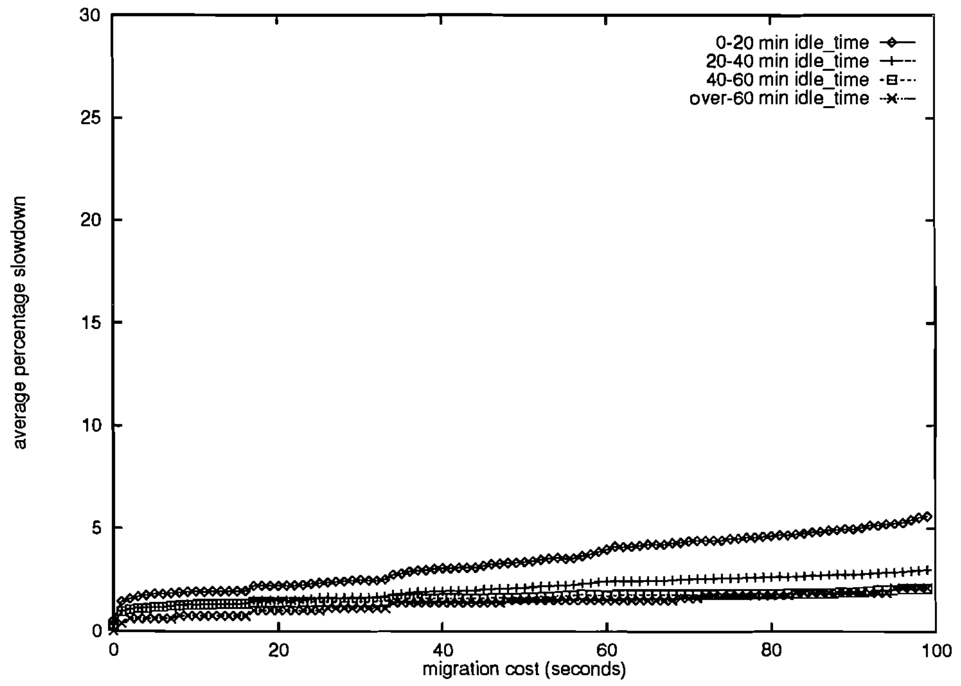
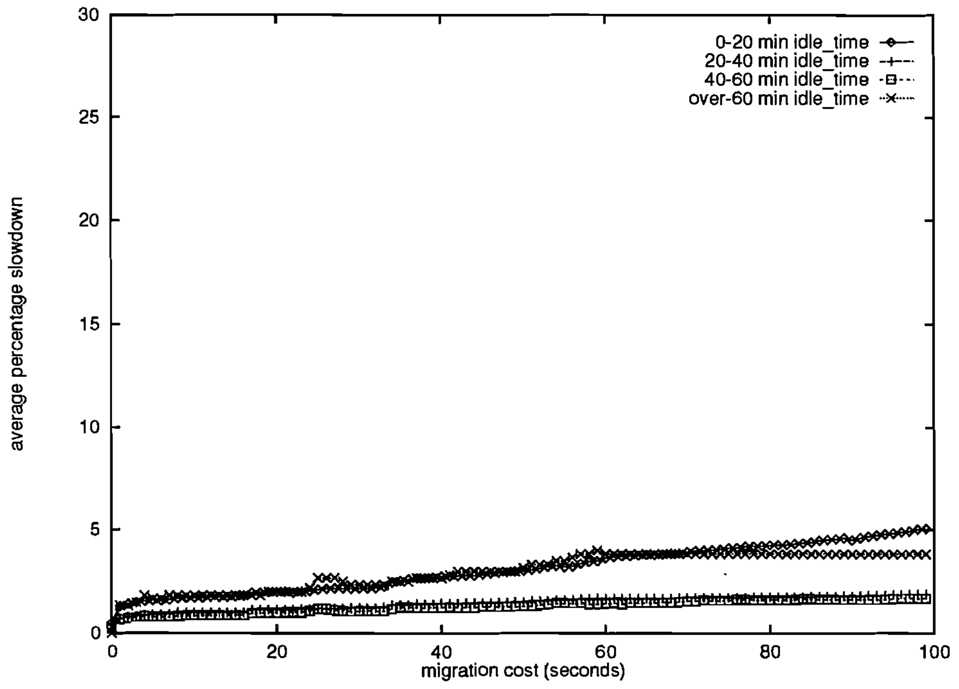


FIGURE 22. 100 Workstations. Trans on Critical Path, averaged by idle time



8.0 Future Work

All phases of this project suggested a wealth of interesting areas to be further researched. This section documents some of these areas.

- Investigation into adaptivity and parallel programming paradigms

  This project has only begun to consider approaches and implications of incorporating adaptivity into the design of a parallel programs. Further research should be done in analyzing the relationship between adaptivity and parallel programming paradigms. Several ideas were discussed but, due to time constraints not fully developed, during this project. One area of investigation is the compatibility of logic programming languages with this environment. Another possibility is researching the relationship between the size of a program's subtasks, and the duration of idle periods on the workstations. The idea is that a parallel program that is designed as small subtasks could minimizes the number of times that the parallel program must react to a busy transition. This is because the smaller the subtask, the more likely they could execute completely within the duration of an idle period on a workstation.

- Investigation into implementation techniques

  In discussing techniques for executing on a network of user-owned workstations, similarities with other problems were suggested. Specifically, there are similarities with fault-tolerance and with load balancing. This environment is similar to requirements for fault tolerance if each busy transition is viewed as a fault. This could be seen as an environment where this particular type of fault is expected and frequent.

The network of user-owned workstation environment also has many similarities with load-balancing. Additional work could investigate using load balancing techniques to this environment.

- Research the behavior of combinations of: execution models, implementation techniques and parallel programming paradigms

  The execution models illustrated in Section 3.0 provide a base for considering what implementation techniques and parallel programming paradigms may fit will with the various execution models.

  The simulations for this project looked at one combination: a scheduling approach using migration for a set of message passing programs. It would be interesting to use the simulation tool to compare the performance impact of other combinations.

  In particular the simulations include code to react to busy transitions by reducing the priority of the parallel process. The simulations could also quickly be extended to implement retreating to the beginning of the subtask before migrating. This could answer additional questions such as: how great of an improvement in migration costs is needed to justify retreating.

- Further compare the merits of the two design approaches:

  To what extent can the performance goals be satisfied with scheduling designs?

  Do the benefits of adaptivity outweigh the disadvantage of increased constraints on the programming paradigm?

  How far can the parallel paradigm constraints be relaxed and still reap some of the benefits of adaptivity?

  How far can the two design approach be converged to obtain the advantages of both?

- Extend the simulations to collect additional statistics.

  In the Introduction, it was suggested that any solution should balance user response time, performance of parallel programs and to a lesser degree, CPU utilization. This project has only considered program performance. It should be extended to consider the other two critieria. CPU utilization could be incorporated into the simulations statistics. Evaluating response time would require a more complicated simulation environment, or some other form of experiment.

- Analyze the workstation data.

  A large volume of data has been collected, and some relationship between idle times and performance impact has been exposed by the simulations. The next step would be to collect statistics on the characteristics of the collected data. These could include duration of idle and busy periods, and statistics on the actual CPU load levels.

- Further Analyze the simulation results

  As suggested in Figure 7.2.2, collecting additional data on how the critical path time breaks down may help explain the superlinear increase in performance degradation. The critical path could be broken into computation time, blocking time and transition costs. By understanding the rate of increase of the 3 areas, and their relative percentage of the critical path, it should be possible to describe how the critical path is being affected superlinearly by the migration costs.

- Definition of idle system state

  For this project we chose a CPU load of.25 as the boundary identifying idle workstations. In fact, a variety of definitions for idleness have been suggested by related work. Further work should be done in researching how to accurately define an idle system.

- Implementation

  This project was not intended to address the issue of implementing techniques for adapting to changes in workstation availability. The implementation details of any of the suggested techniques, applied in the network of user-owned workstations environment would certainly entail much additional thought and investigation.

9.0 Conclusions

By analysis and simulations, this project investigated executing parallel programs on a network of user-owned workstations. It analyzed two design approaches and their relation to parallel programming paradigms and implementation options. It investigated performance by simulations based on empirical data describing a workstation environment and sample parallel program traces.

This project was concerned with two major performance goals: efficient execution of parallel programs and maintenance of workstation user response time. With this in mind, the analysis of the two design approaches can be summarized by the following question. Can the performance goals be better met by extensions to existing scheduling techniques or by a paradigm shift where the parallel program actively adapts to dynamic CPU resources?

The main advantages of the scheduling approach are that it relies on previously exercised scheduling techniques and puts few restraints on the structure of the parallel program. The main disadvantages are potentially high implementation overhead and difficulty in taking advantage of increases in CPU availability.

The main advantage of the adaptive programming approach is that increased flexibility allows for more efficient response to changes in CPU availability. This should improve performance by minimizing overhead and allowing greater utilization of CPUs. The main disadvantage is that it is more likely to introduce constraints on the parallel program's structure.

Based on the simulation results it is possible to conclude that it is feasible to efficiently execute parallel programs in this environment. The results showed that over thousands of sample executions, percentage slowdown of the parallel program's execution time was under 100%. This was supported by data showing that relatively low rate of encountering (less than 20) busy transitions. This suggests CPU availability was high enough to support the parallel programs along with user activity. As would be expected, the simulations also showed that the fastest execution times occurred with a nighttime level of user activity. The simulations also showed that the number of workstations impact the percentage slowdown rate when it is less than twice the number of workstations required by the program. When the number of workstations is varied from three to five times the number required, there is little change in the rate of percentage slowdown.

These results are promising and suggest many areas of future work that should be considered. This environment is potentially a significant source of needed CPU resources for parallel program executions. Taking full advantage of its potential will require creative, fully researched solutions. This project has intended to contribute to the development of such solutions.

10.0 Appendices

10.1 Supporting Plots

The following key is for supporting plots that show executions based on varying time period of workstation.

• night	Midnight - six a.m.
• morning	6:00 a.m. - Noon
• afternoon	Noon - 6:00 p.m.
• evening	6:00 p.m. to Midnight

10.1.1 FFT program

FIGURE 23. Average percentage slowdown for varying number of workstations

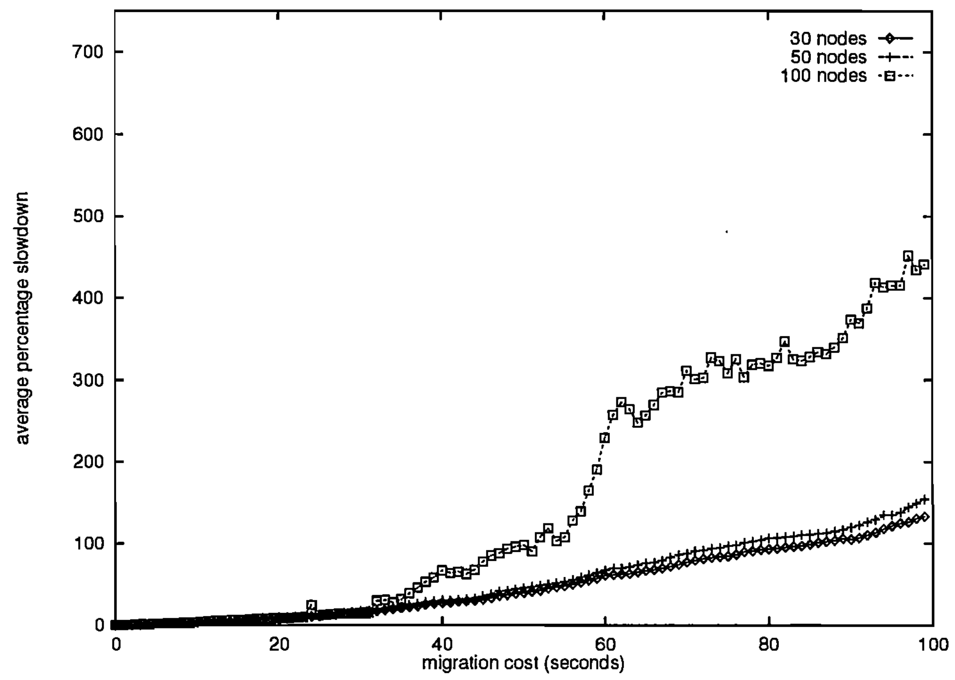


FIGURE 24. Average percentage slowdown with varying time periods on 30 workstations

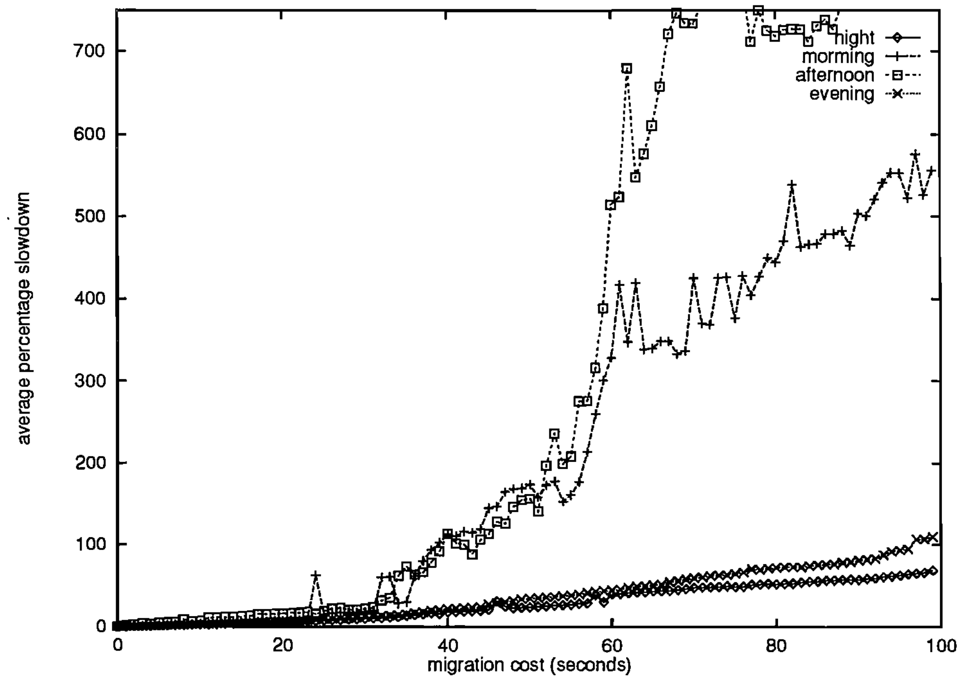


FIGURE 25. Average percentage slowdown with varying workstation activity levels on 30 workstations

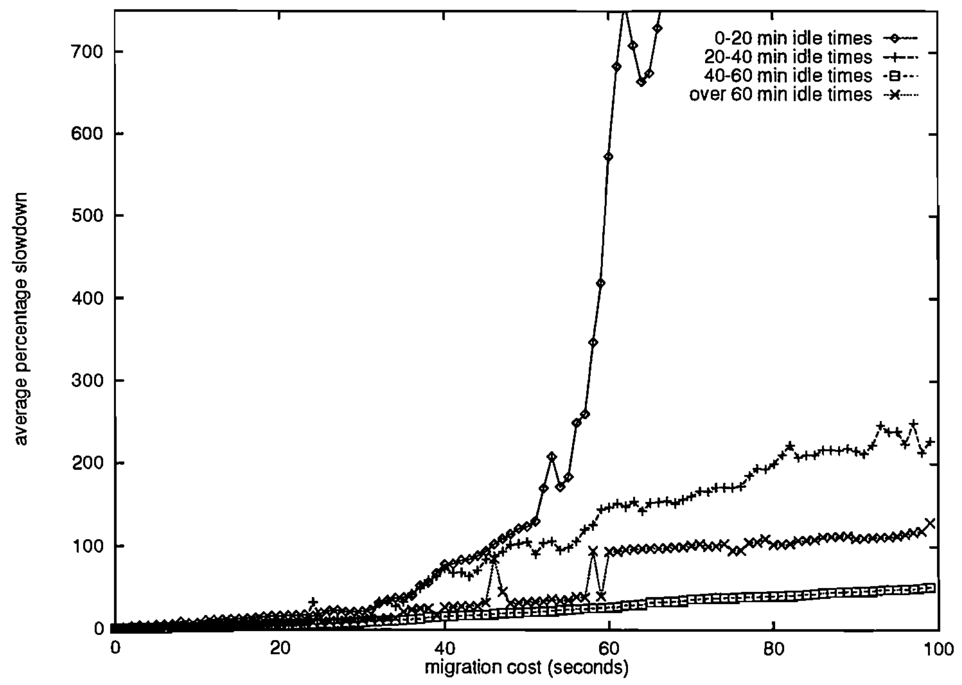


FIGURE 26. Average percentage slowdown with varying time periods on 50 workstations

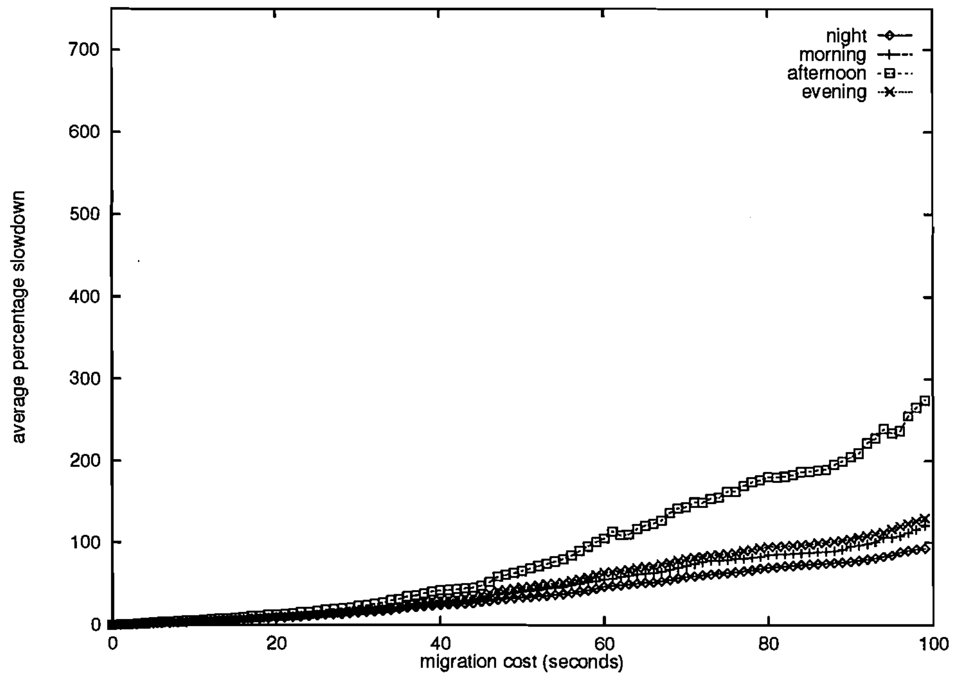


FIGURE 27. Average percentage slowdown with varying workstation activity levels on 50 workstations

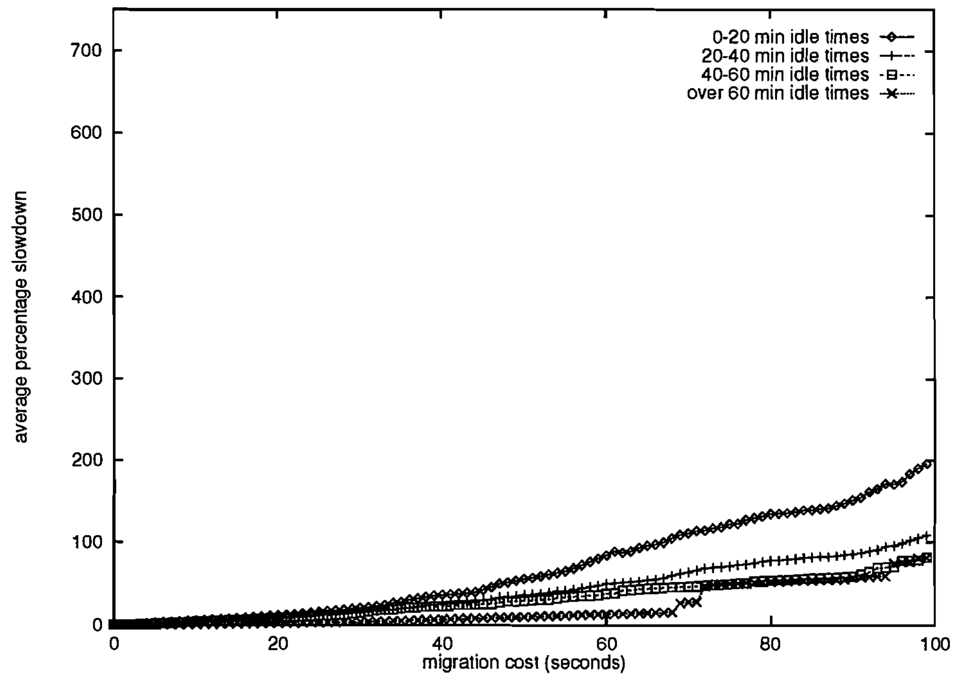


FIGURE 28. Average percentage slowdown with varying time periods on 100 workstations

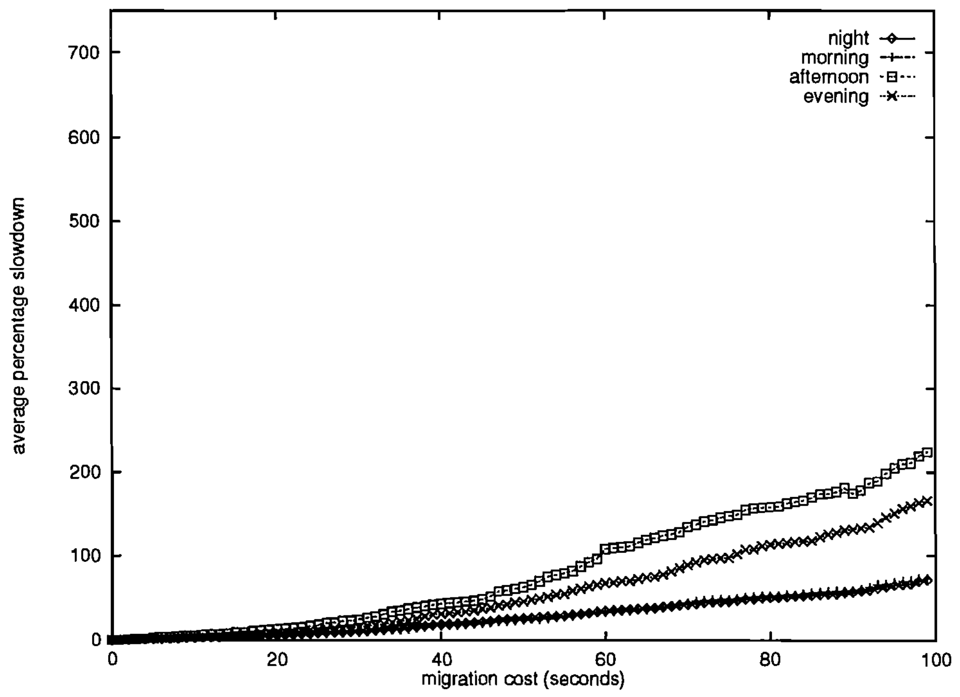
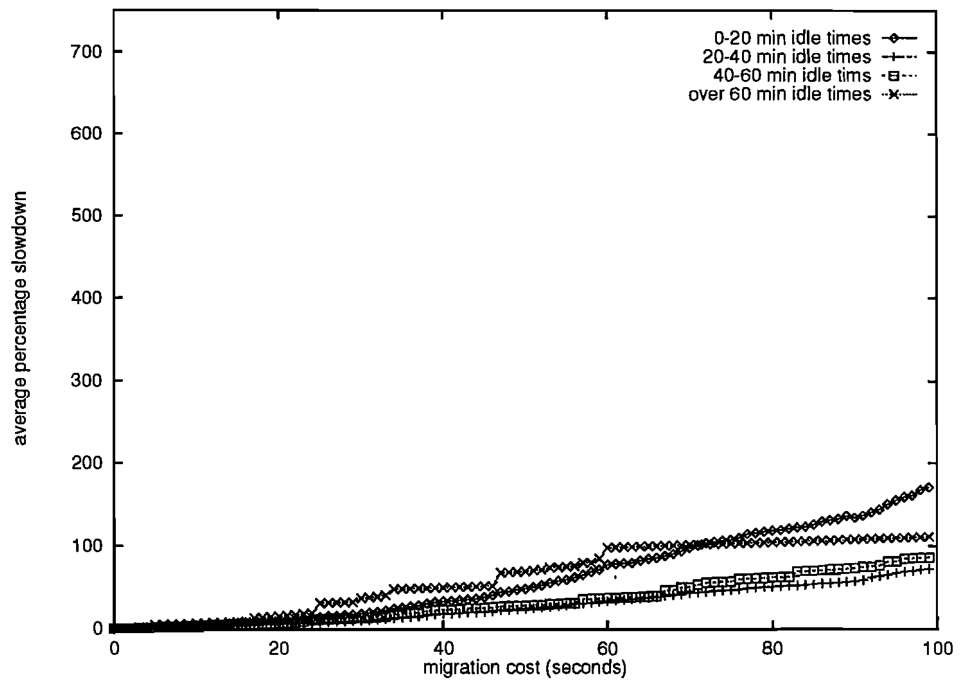


FIGURE 29. Average percentage slowdown with varying workstation activity levels on 100 workstations



10.1.2 MSH program

FIGURE 30. Average percentage slowdown for varying number of workstations

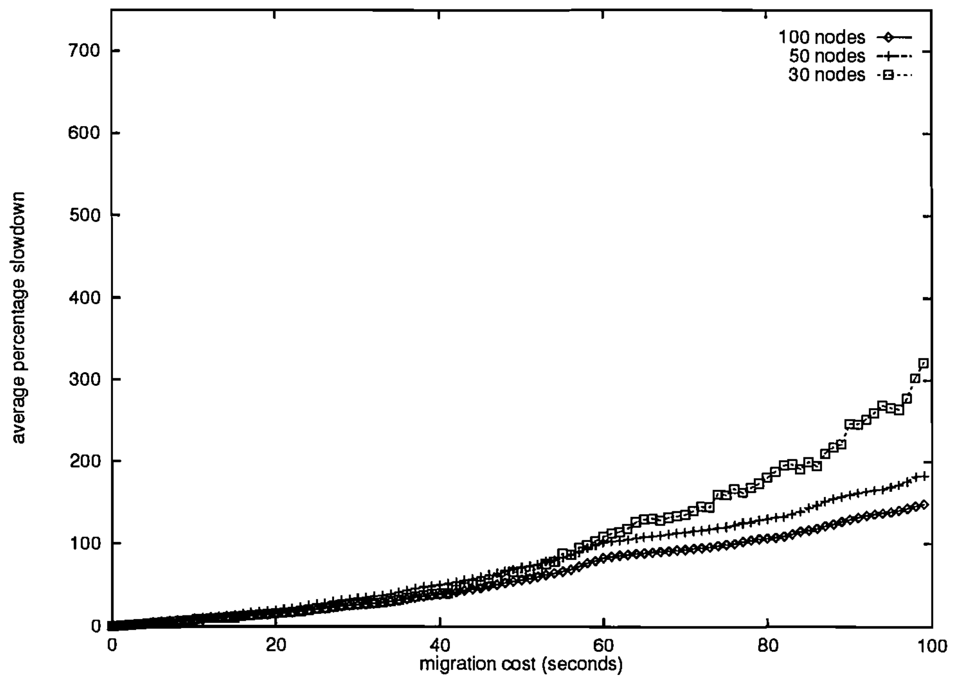


FIGURE 31. Average percentage slowdown with varying time periods on 30 workstations

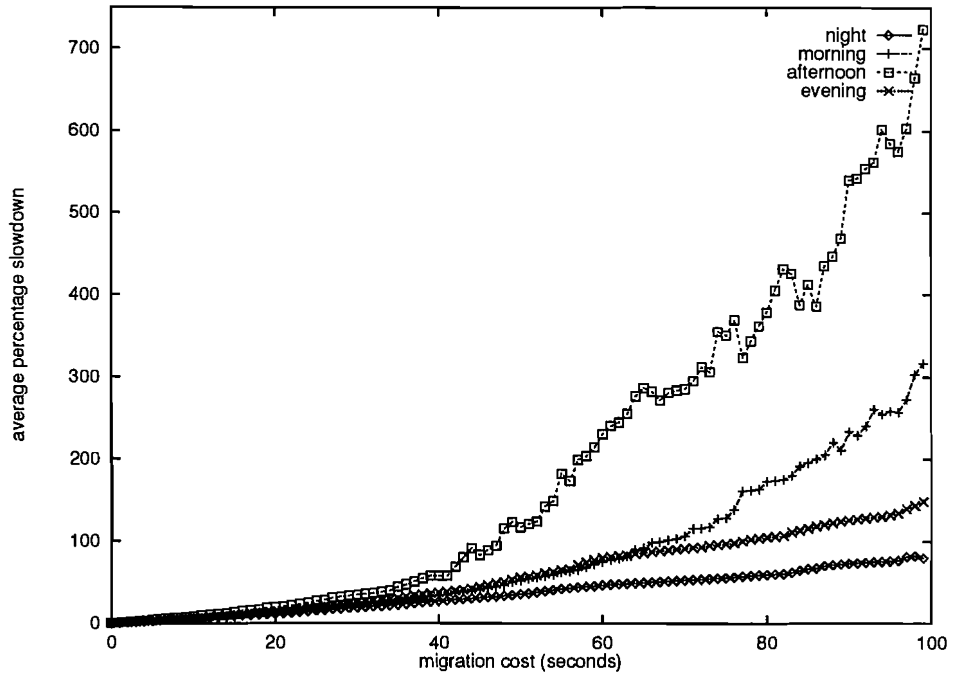


FIGURE 32. Average percentage slowdown with varying workstation activity levels on 30 workstations

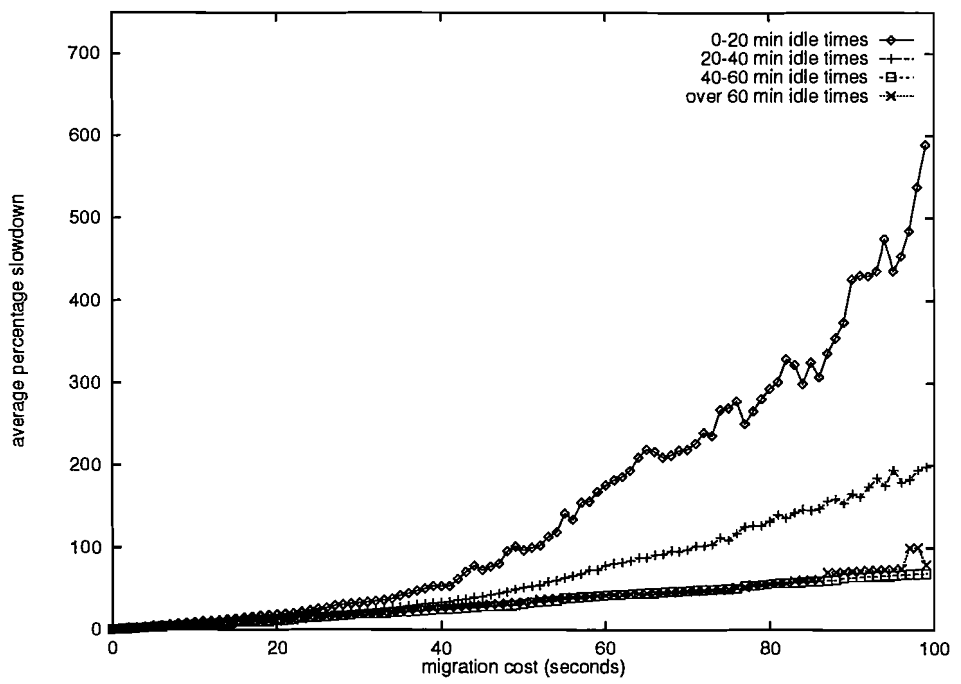


FIGURE 33. Average percentage slowdown with varying time periods on 50 workstations

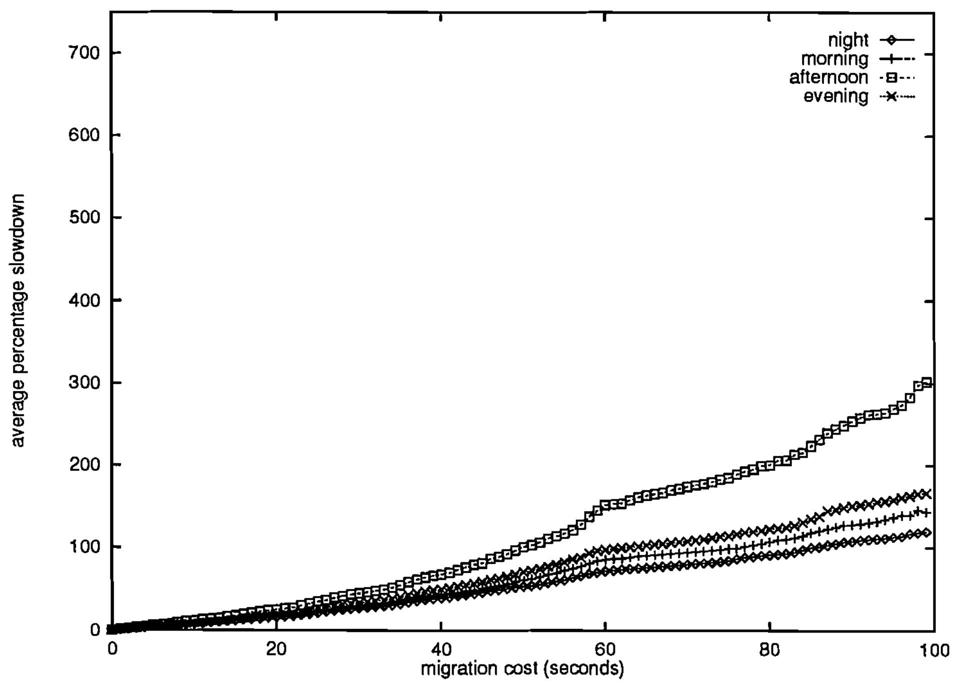


FIGURE 34. Average percentage slowdown with varying workstation activity levels on 50 workstations

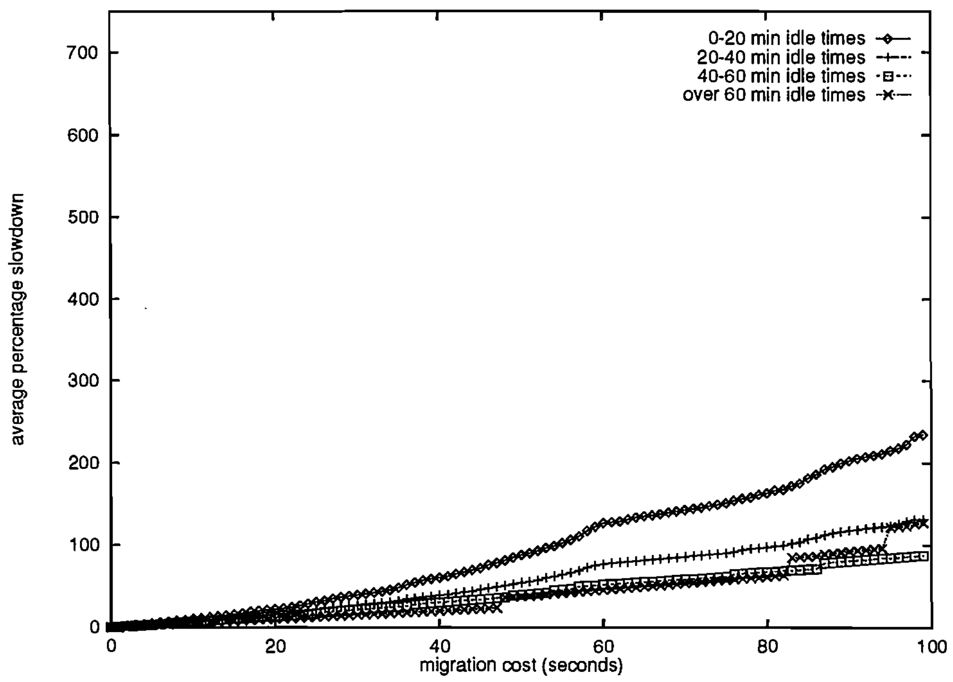


FIGURE 35. Average percentage slowdown with varying time periods on 100 workstations

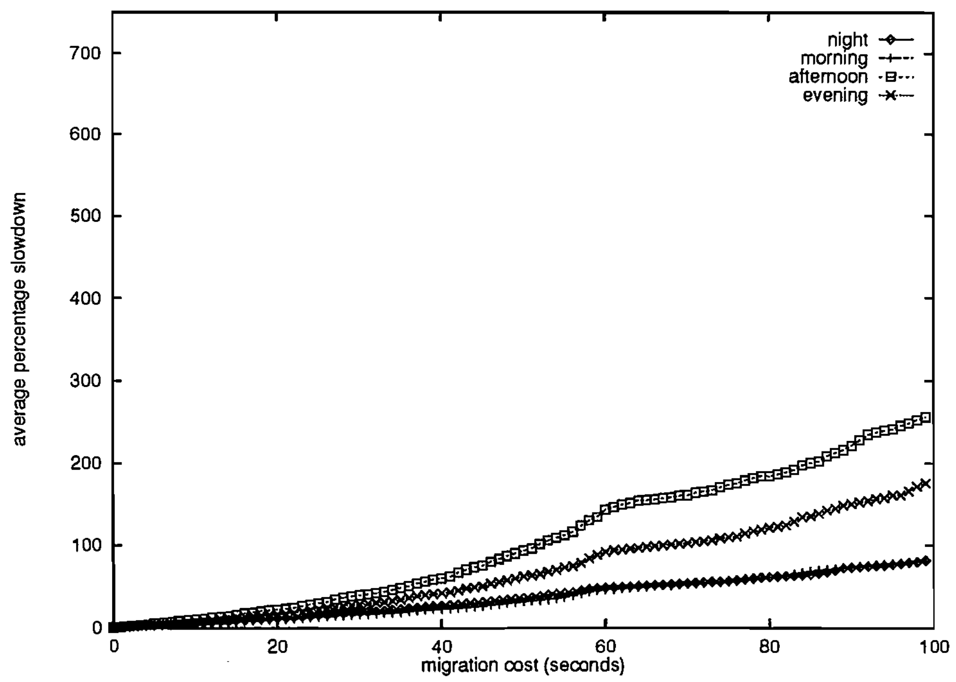
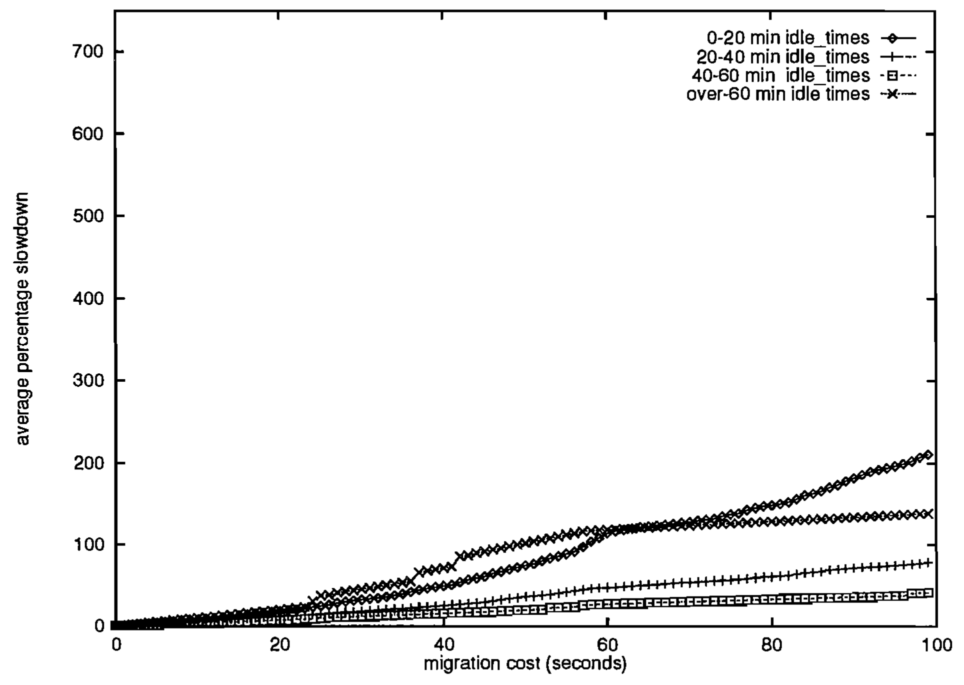


FIGURE 36. Average percentage slowdown with varying workstation activity levels on 100 workstations



10.1.3 DET

FIGURE 37. Average percentage slowdown for varying number of workstations

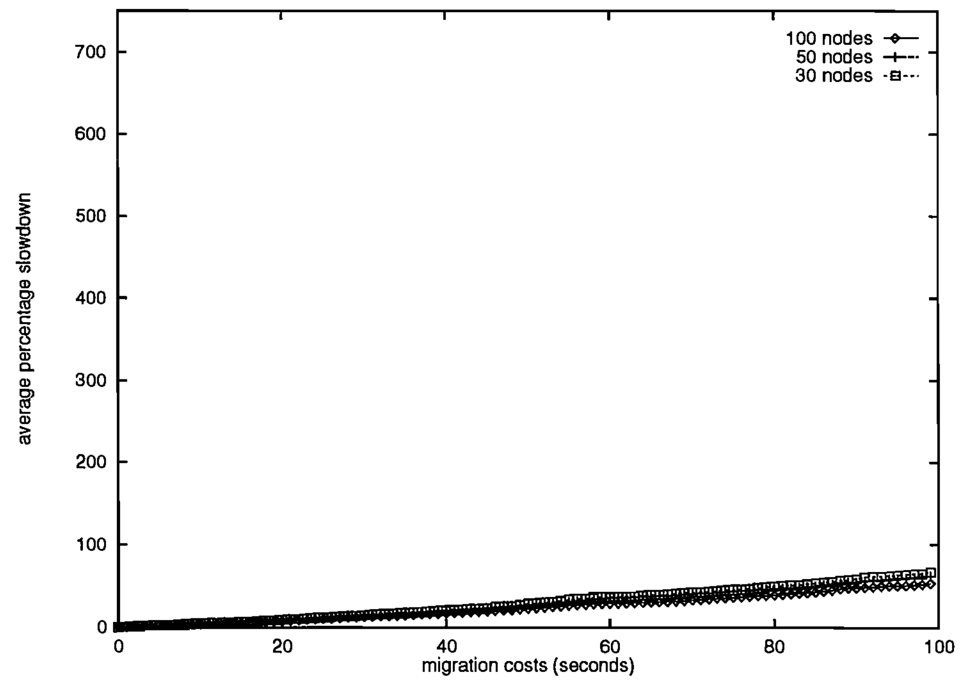


FIGURE 38. Average percentage slowdown with varying time periods on 30 workstations

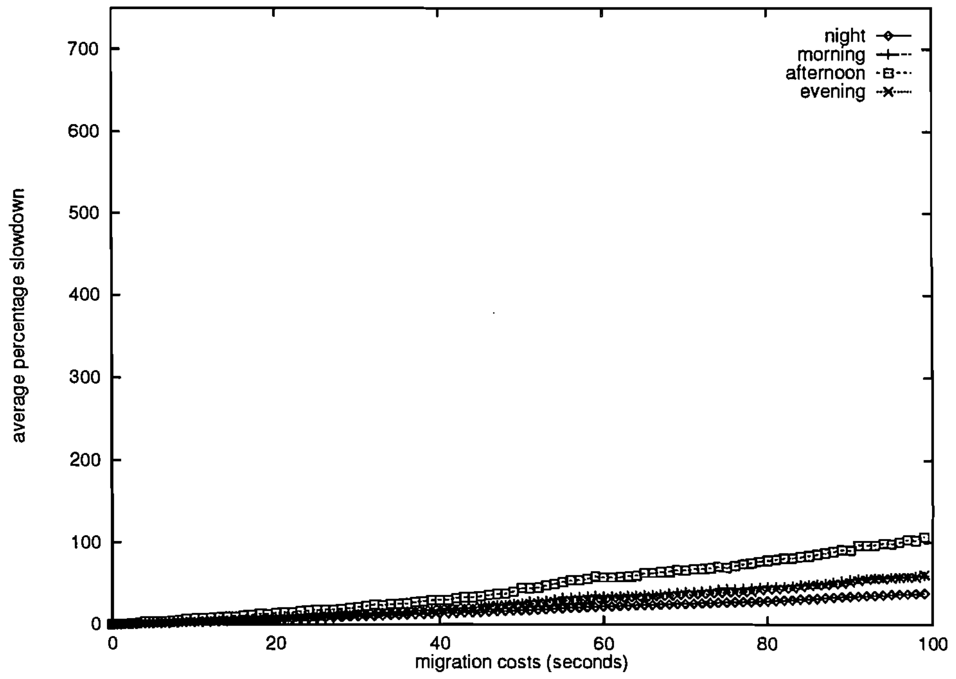


FIGURE 39. Average percentage slowdown with varying workstation activity levels on 30 workstations

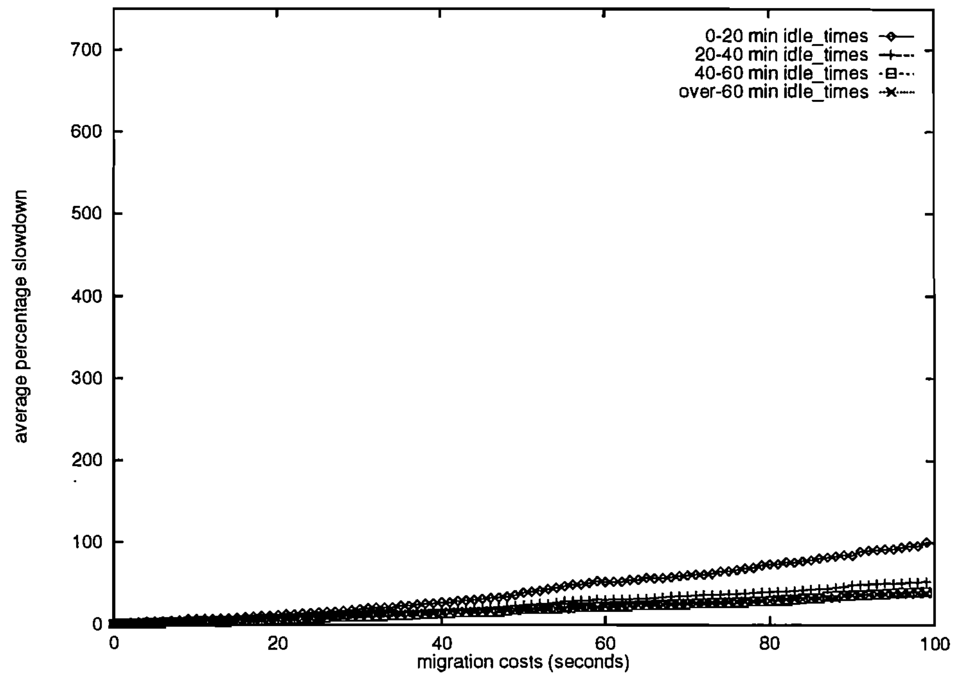


FIGURE 40. Average percentage slowdown with varying time periods on 50 workstations

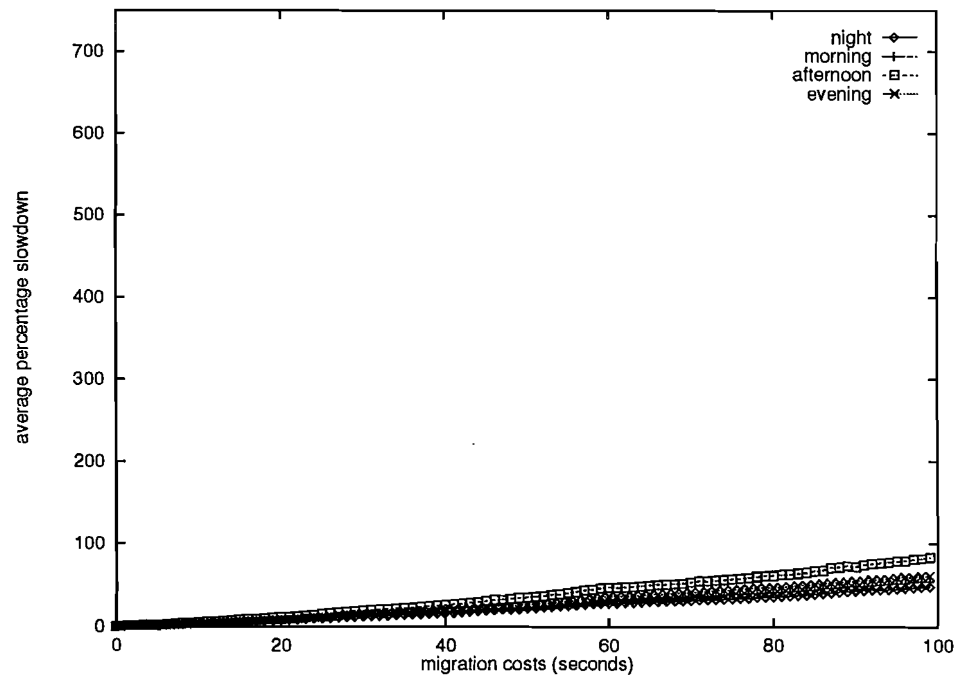


FIGURE 41. Average percentage slowdown with varying workstation activity levels on 50 workstations

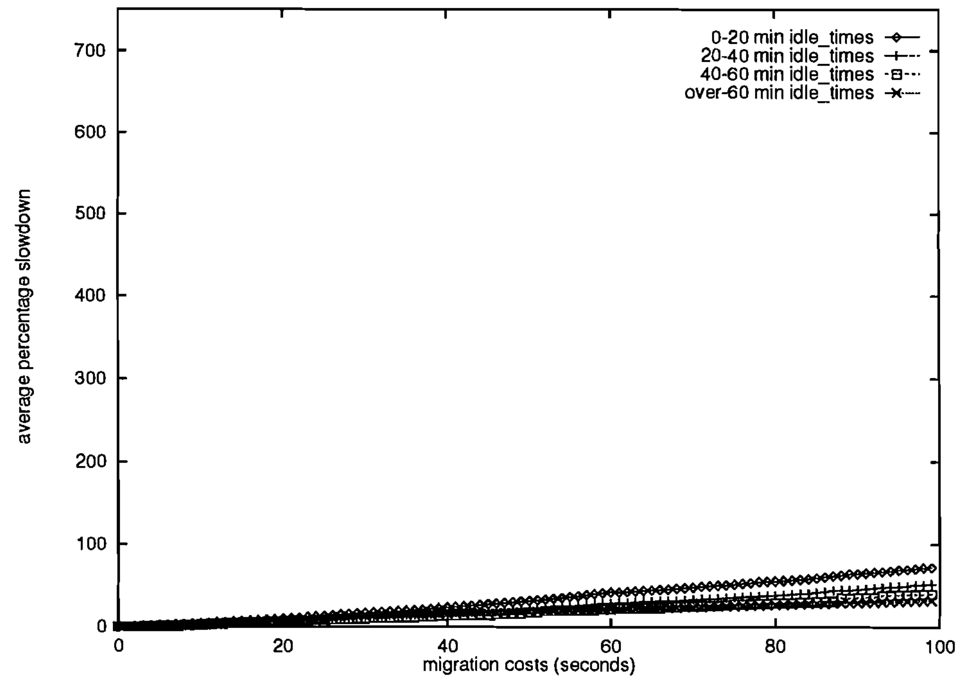


FIGURE 42. Average percentage slowdown with varying time periods on 100 workstations

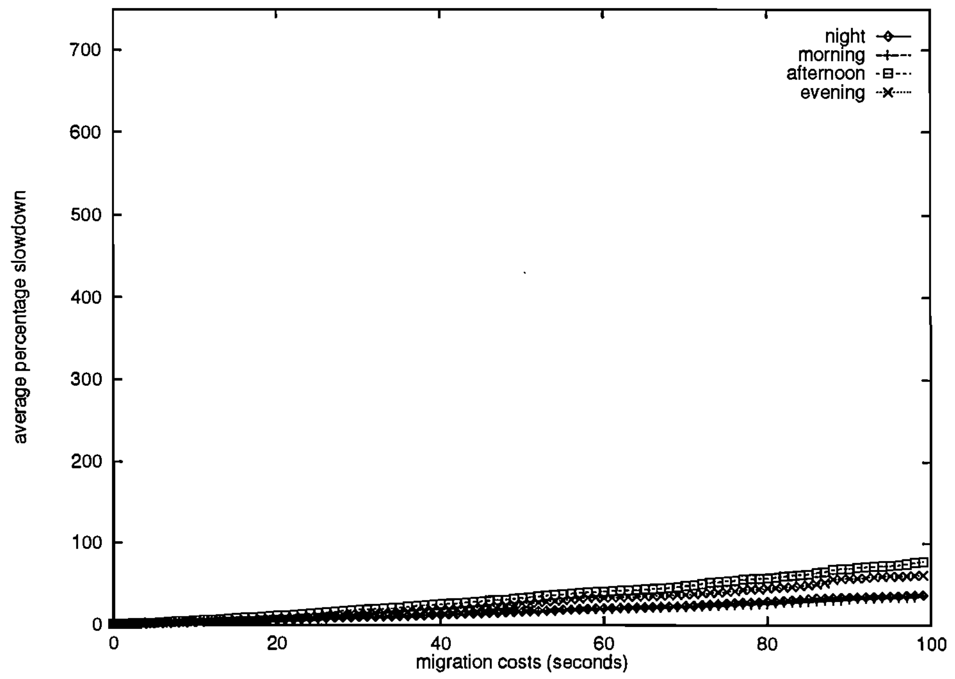
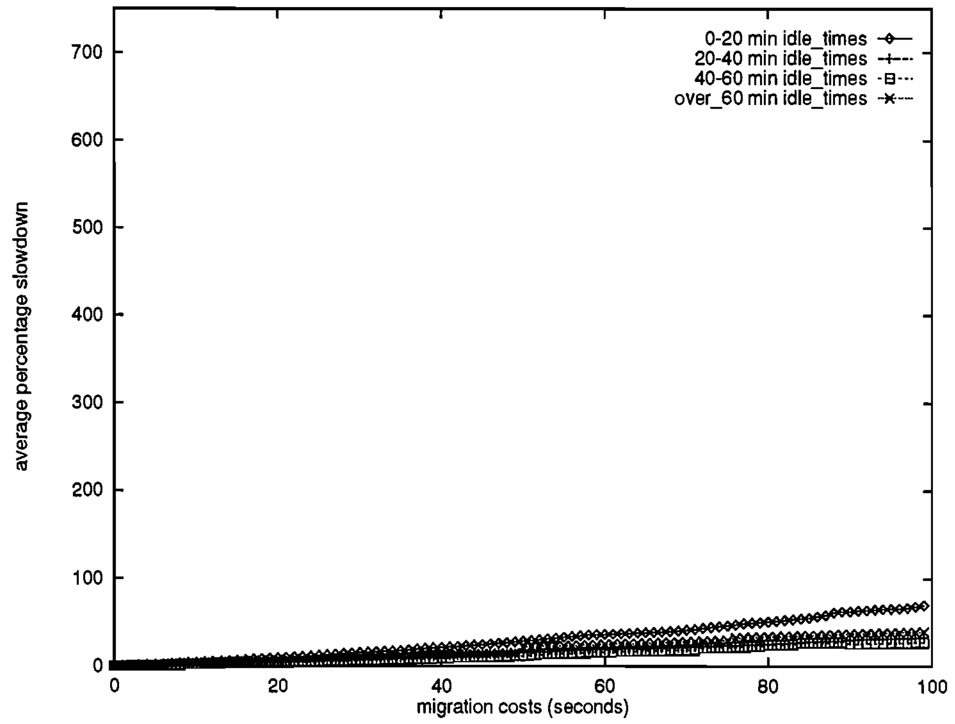


FIGURE 43. Average percentage slowdown with varying workstation activity levels on 100 workstations



10.1.4 TEST program

FIGURE 44. Average percentage slowdown for varying number of workstations

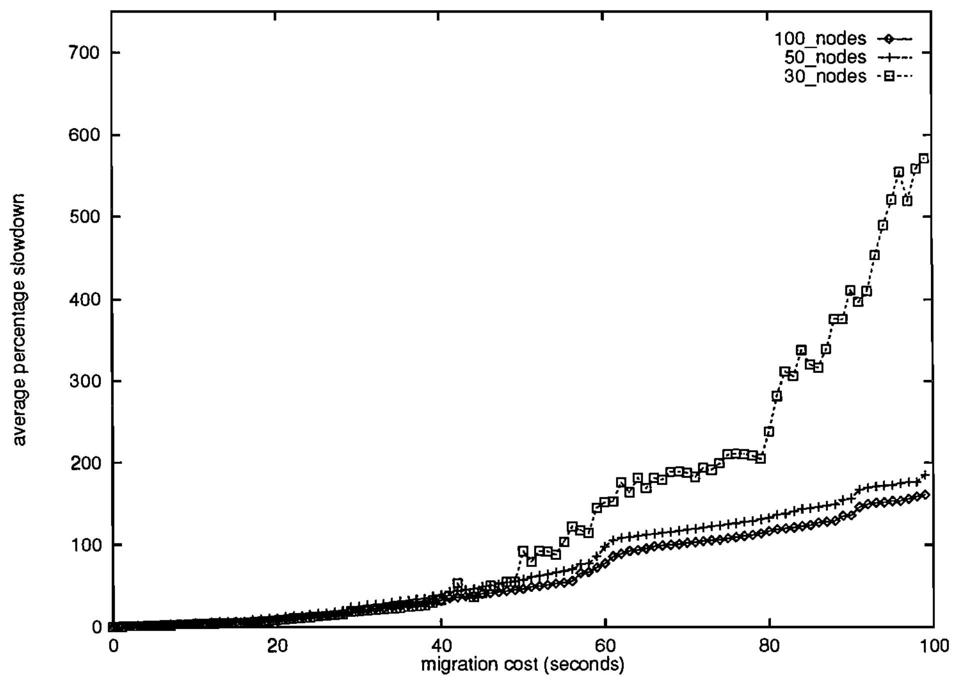


FIGURE 45. Average percentage slowdown with varying time periods on 30 workstations

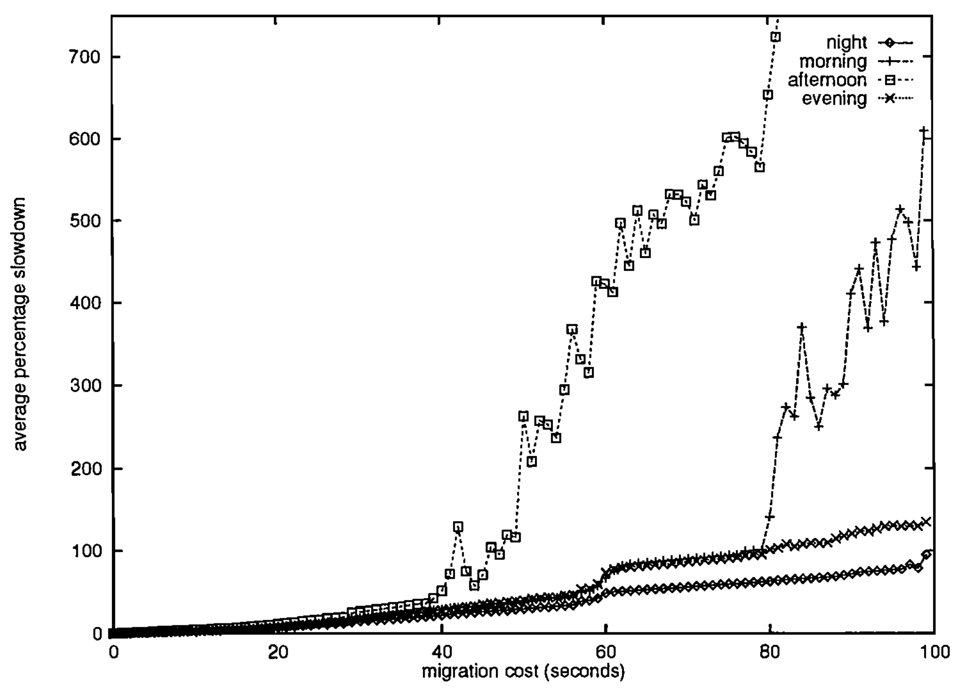


FIGURE 46. Average percentage slowdown with varying workstation activity levels on 30 workstations

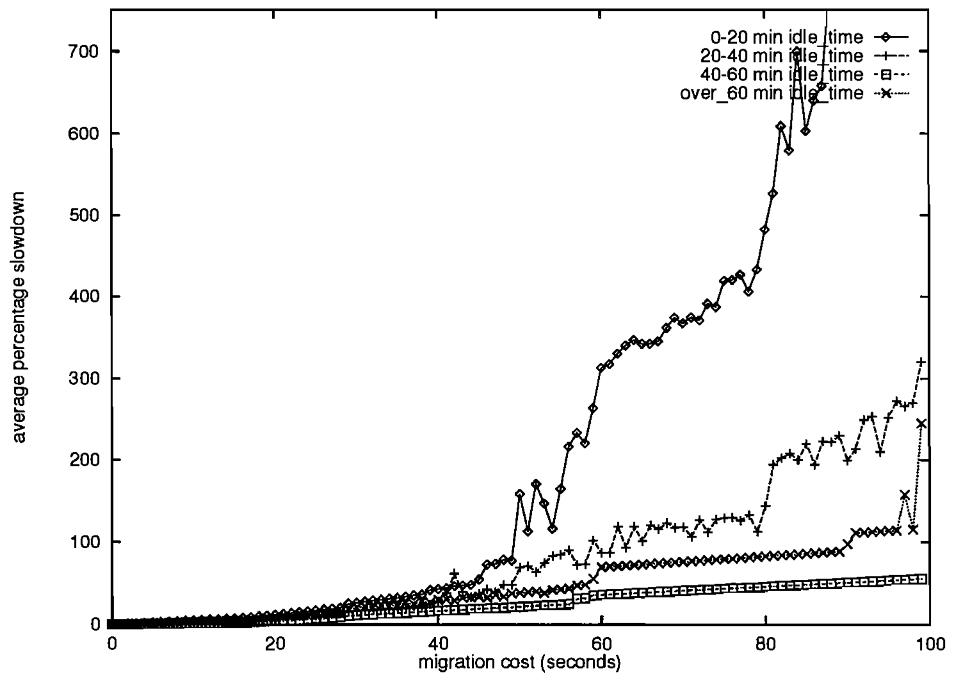


FIGURE 47. Average percentage slowdown with varying time periods on 50 workstations

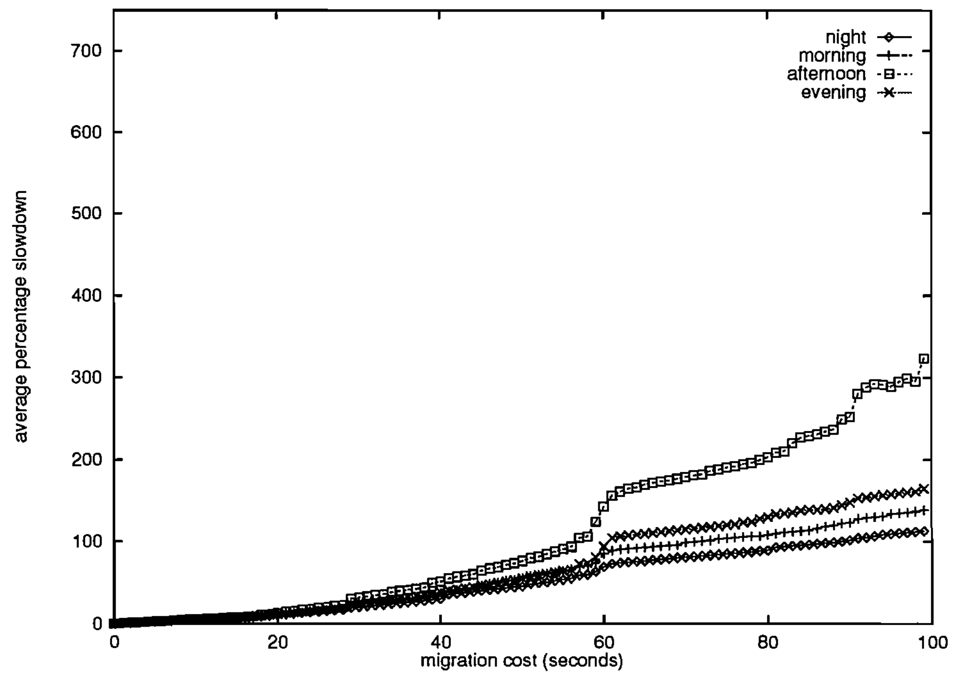


FIGURE 48. Average percentage slowdown with varying workstation activity levels on 50 workstations

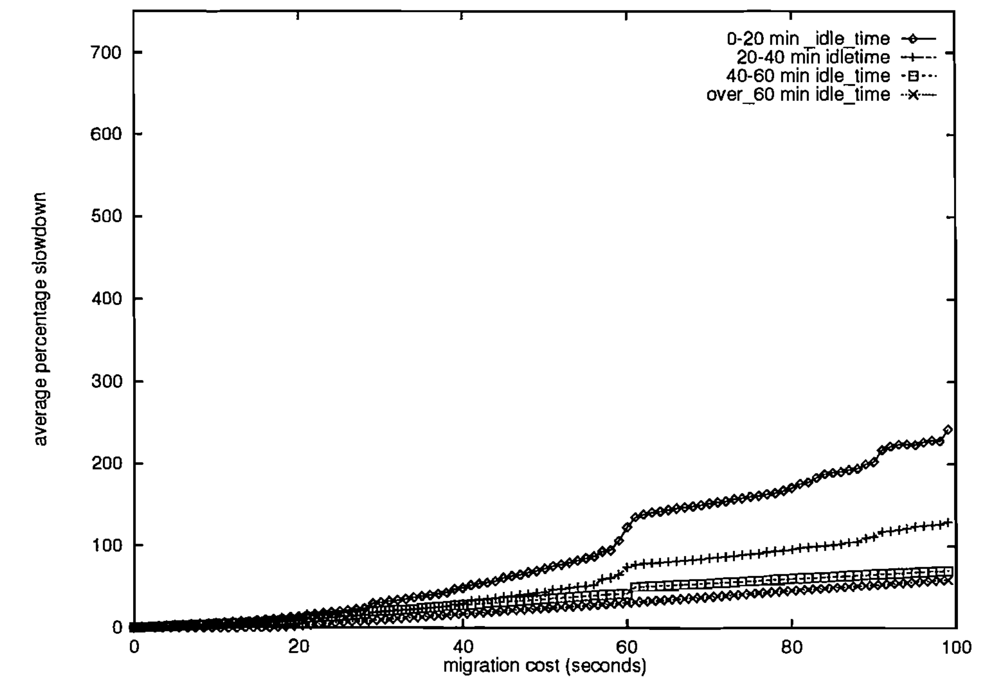


FIGURE 49. Average percentage slowdown with varying time periods on 100 workstations

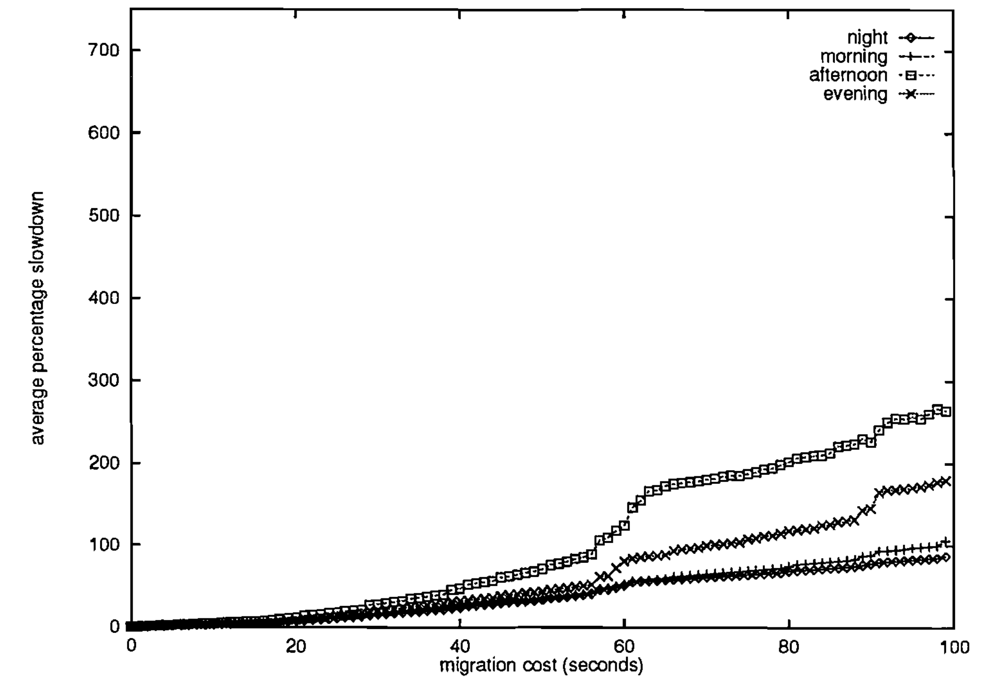
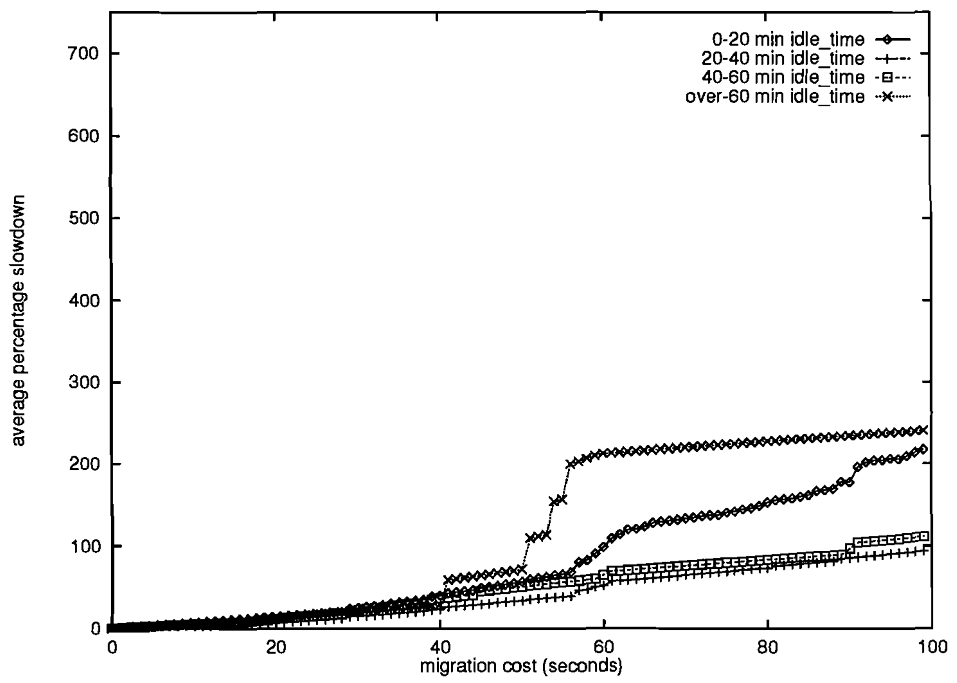


FIGURE 50. Average percentage slowdown with varying workstation activity levels on 100 workstations



10.2 Location of Project Files

All source, data and executable files for this project are located off of the following directory: /u/rn/public/rtraces.

A description of all subdirectories and pointers to additional README files are provided in : /u/rn/public/rtraces/README

Bibliography

[1] David Anderson and Mark Sullivan. Marionette: a System for Parallel Distributed Programming Using a Master/Slave Model. Proc. 9th Intl. Conf. on Distributed Computing Systems, pp. 181-188, June 1989.

[2] M.J. Atallah, C. Lock, D.C. Marinescu, H.J. Siegel, and T.L. Casavant, "Co-Scheduling Compute-Intensive Tasks on a Network of Workstations: Models and Algorithms," Proc. 11th International Conference on Distributed Computing Systems, May 1991, pp. 344-352.

[3] John Basik. Quahog: Polite Remote Processing. Brown University, June 1992.

[4]Allan Bricker and Michael J. Litzkow. Condor Technical Summary. University of Wisconsin, 1989.

[5]Clemens H. Cap, Volker Strumpen. "The PARFORM - A High Performance Platform for Parallel Computing in a Distributed Workstation Environment", Zurich Univesity, June 23, 1992.

[6] Nicholas Carriero, Eric Freeman, and David Gelernter. "Adaptive Parallelism on Multiprocessors: Preliminary Experience with Piranha on the CM-5", YALEU/DCS/RR-969,May 1993.

[7] Nicholas Carriero, David Gelernter, David Kaminsky, and Jeffrey Westbrook. "Adaptive Parallelism with Piranha", YALEU/DCS/RR-954, February 1993.

[8]Chase, J.S., Amador, F.G., Lazowska, E.D., Levy, H.M., Littlefield, R.J. "The Amber System: Parallel Programming on a Network Of Multiprocessors," Proceedings of the 12 ACM Symposium on Operating systems Principles, pp 147-158, Dec 1989.

[9]Henry Lcark and Bruce Mcmillin, "DAWGS- A Distributed Compute Server Utilizing Idle Workstations", Journal of Parallel and Distribruted Computing 14, 175-186, 1992.

[10]Kemal Efe and Margarete A. Schaar. Performance of Co-Scheduling on a Network of Workstations. Proc 13th International Conference on Distributed Computing Systems, 1993, pp. 525-531.

[11]David Gelernter and David Kaminsky. "Supercomputing out of Recycled Garbage: Preliminary Experience with Piranha", Proceedings of the ACM, International Conference on Supercomputing, July 19-23, 1992.

[12]G.A. Geist and V.S. Sunderam. Network Based Concurrent Computing on the PVM System. Concurrency: Practice and Experience, 4(4)293-311,June 1992.

[13]P. Krueger and R. Chawla. The Stealth Distributed Scheduler. In Proceedings of the 11th International Conference on Distributed Computing Systems, pages 336-343, May 1991.

[14]Louis H. Turcotte. A Survey of Software Environments for Exploiting Networked Computing Resources. Engineering Research Center for Computational Field Simulation. June 1993.