PScheD − Political Scheduling for Unicos/mk

Richard Lagerstrom and Stephan Gipp
Cray Research
655F Lone Oak Drive, Eagan MN 55121 USA
rnl@cray.com skg@cray.com

ABSTRACT:

Large parallel processing environments present serious administrative challenges if high utilization of the available resources is a goal. In many cases there is also the need to support critical or time−dependent applications at the same time as development and routine production work is going on.

This paper describes the components that help realize the Political Scheduling goals of the CRAY T3E system. The meaning of Political Scheduling is defined, we present a general overview of the Cray T3E hardware and operating system and describe the current implementation of the Political Scheduling feature of Unicos/mk.

KEYWORDS:

Job scheduling
Process scheduling
Share
Gang scheduling

Table of Contents

1 Introduction
  1.1 The Complete Picture
  1.2 The Global Resource Manager

2 An Introducation to Political Scheduling
  2.1 Scaling and Feature Design
  2.2 Scheduling Domains
  2.2.1 So why name them Domains?

3 The Feature Manager
4 The Gang Scheduler
  4.1 Controlling a Gang Scheduling Domain
  4.2 Prime Jobs

5 The Load Balancer
  5.1 Load Balancing a Command Domain
  5.1.1 Filtering Evaluation Candidates
  5.2 Load Balancing an Application Domain

6 MUSE
  6.1 Muse Entitlement
  6.2 An Outline of MUSE
  6.3 MUSE Domain Configuration
  6.4 The MUSE Factor

7 Does Migration Improve Performance?
8 Summary

1 Introduction

What do we mean by the term ***Political Scheduling***? In a presentation one of us stated that it was ‘‘irrational’’ scheduling as opposed to ‘‘technical’’ scheduling. What we mean is that there are scheduling goals not easily described in terms of machine utilization or performance, but rather by organizational or economic requirements. This sort of requirement often cannot be well handled by classical scheduling mechanisms, especially if they try to support a very wide class of users and a complex environment at the same time.

A brief overview of the global environment will be followed by a description of the scheduler.

1.1 The Complete Picture

All parts of Unicos/mk including the microkernel cooperate to achieve the desired scheduling results. Figure 1 is a schematic view of the scheduling daemon (inside the dashed−line rectangle labeled *PScheD*) and how it fits into the system. Although this diagram is complex, we intend to show the variety of relationships among the components of Unicos/mk which must be coordinated for successful scheduling behavior.

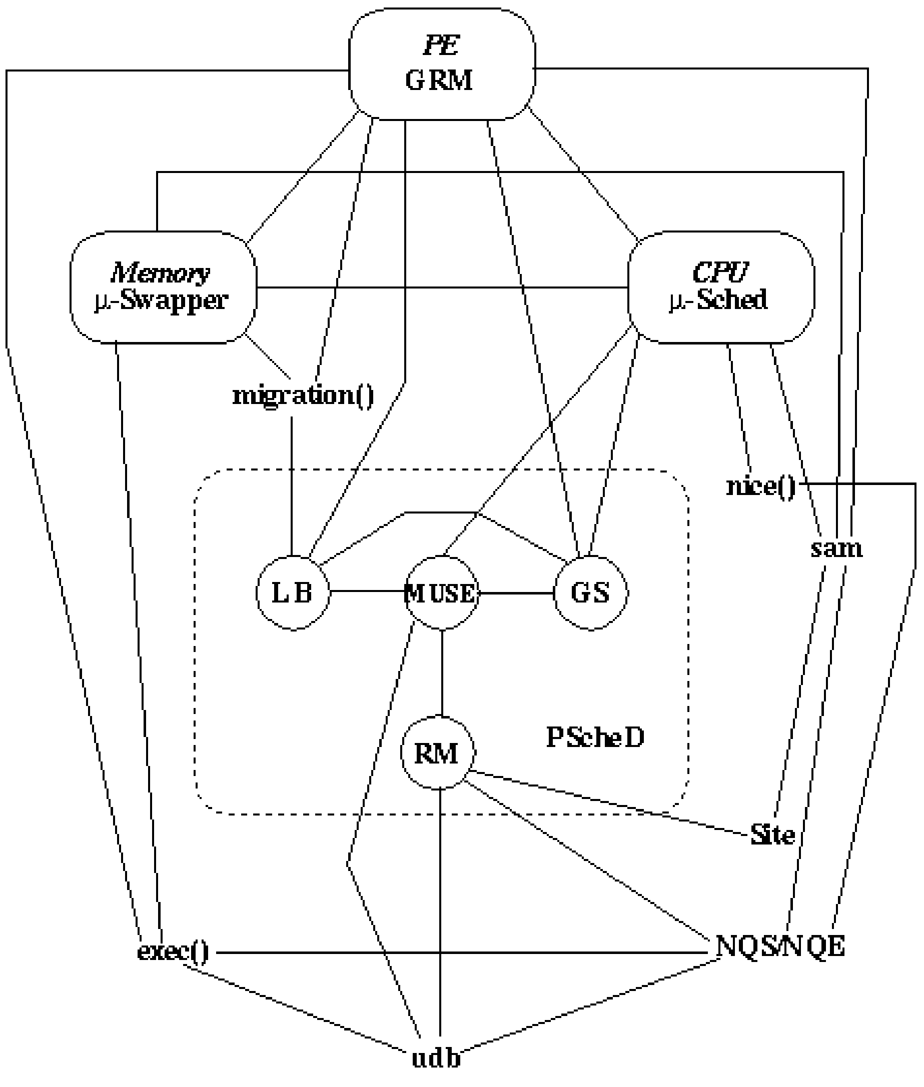


Figure 1

The objects labeled *Memory μ−Sched* and *CPU μ−Sched* reside in each PE and manage local memory, swapping and CPU scheduling. *GRM* (Global Resource Manager) is the PE selector so all `exec()` and `migrate()` system calls make GRM PE allocation requests. The `nice()` system calls alters priority, which must be made known to the micro−kernel of the PE running the thread. The figure also shows how the System Activity Monitor (*sam*), *NQE* and the User Database (*udb*) fit into the scheduling facility.

The scheduling features of PScheD shown in Figure 1 are Load Balancing, *LB*; Multilayered User−fair Scheduling Environment, *MUSE*; Gang Scheduling, *GS* and Resource Management, *RM.* These features will be described later.

1.2 The Global Resource Manager

All user[1] PEs have the capability of running single−PE processes, named *commands*, or multiple−PE entities, named *applications*. Command PEs run shells, daemons and other familiar Unix processes. All systems must have some number of Command and Operating

System PEs[2] configured as the *Command* and Support regions while the remaining PEs are configured into one or more *application region(s)* in which applications execute.

Figure 2 shows a configuration with large and small application regions, a command region and some Support PEs. (Typically a recommend maximum of one or two application regions will be configured although special circumstances could make more regions useful.) Regions are made up of a number of PEs with consecutive *logical PE number*s. These numbers (integers in the range *0...machinesize − 1*) are assigned when the machine is booted and are mapped to PE torus coordinates in a way to provide good physical proximity within the machine. Not every PE can be ‘‘next’’ to every other, so mapping is a compromise between the physical relationship of the PEs and their logical numbering. Each application *must* be assigned to a range of PEs having consecutive logical PE numbers.

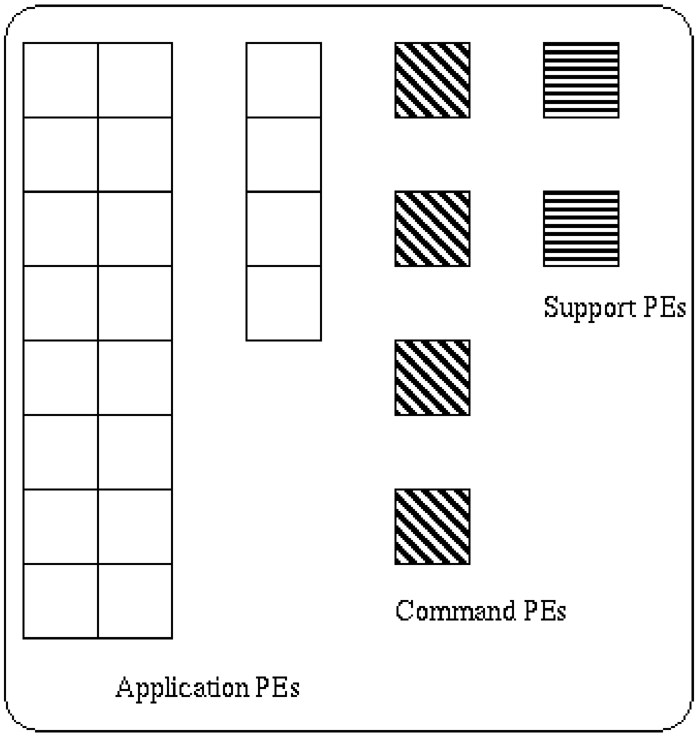


Figure 2

In the command region GRM assigns each process to a PE having attributes compatible with those of the user[3] while at the same time attempting a degree of load balancing. A command will execute to completion in the same PE unless it is moved through a process known as *migration*}[4].

Application regions may be configured to accept applications with only certain attributes. Some of the region attributes are User ID, Group ID, Account ID, Service Provider Type[5], size of the application, and some others.

It is the responsibility of GRM to match the attributes of an application requesting service with regions which will both allow it to run and have free resources with which to run it. GRM is not capable of very sophisticated scheduling since it is aware only of the running load and the immediate launch request backlog. Such information as batch queue backlog and the relative priorities of jobs waiting in the backlog are invisible to it. The Political Scheduler, however, does have access to that information and will direct GRM to do the "right" thing or take action to "fix" PE allocation problems as they arise.

The final major task of GRM is to manage the Barrier Context Registers, construct barrier routing trees and initialize the barrier routing registers when applications are started and manage the Global Memory Descriptors each application uses.

2 An Introduction to Political Scheduling

The term *feature* is used in this paper to generically include all of the different decision making components. Most features will be described separately.

High−level scheduling as defined in this paper is based on the concept of *scheduling domains*. Each scheduling domain represents a portion of the CRAY T3E that is managed by a common set of scheduling rules. Scheduling domains will be more fully described later in Scheduling Domains.

The Political Scheduler (PS) is implemented as a daemon which runs on one of the command PEs. There are a few special low−level system ‘‘hooks’’ to control such things as time slice width and to send special commands to GRM, but the remainder of the operating system interfaces are normal to Unicos/mk. An *information server* exists in the kernel for general use and this capability is heavily used by the various features of PS to collect system−wide information. As seen in Figure 3, PS is organized into the following major modules:

Object Manager − Provides an information repository for configuration objects and other data. Communication among the components and with the outside world is centered here. Data objects consist of fundamental types such as integers and strings as well as more complex objects defined as needed. A hierarchical naming convention similar to names of directories and files in a file system is used. For example, an object used to specify the name of the global log file could be named `/PScheD/logFile`. This is a string object containing the name of the log file.
Command Interface − This component implements an RPC interface used by administrative commands through which configuration, viewing and manipulation of the data controlled by the Object Manager. Other uses by various service daemons is also supported.
Feature Manager − Each component registers itself so its *bind*, *verify*, *action* and *exception* functions are known to the feature manager. The meaning of these functions will be discussed below.

The remaining items are the features of PS that implement Political Scheduling.

Gang Scheduler − Application CPU and memory residency control is provided by this feature.
Load Balancer − Measurements of how well processes and applications are being serviced in each scheduling domain are made and acted upon by this feature. Moving commands and applications among eligible PEs in each domain is managed here.
MUSE − A fair−share like capability is implemented by the Multilayered User−fair Scheduling Environment.
Resource Manager − This is somewhat misnamed for historical reasons, but is the place where information about resource usage within the machine is collected, analyzed and formed for both internal and external uses. The Object Manager is used to make this information available in a uniform way to service providers such as NQS or NQE. Unfortunately, the deadline makes a detailed description of this feature impossible.
Site Supplied Scheduling Features − Each feature has an RPC interface allowing connection to a site−written program that can change the decisions made by the standard feature. To connect a feature to an external assistant, the RPC address of the assistant is made known to the feature through the configuration interface.

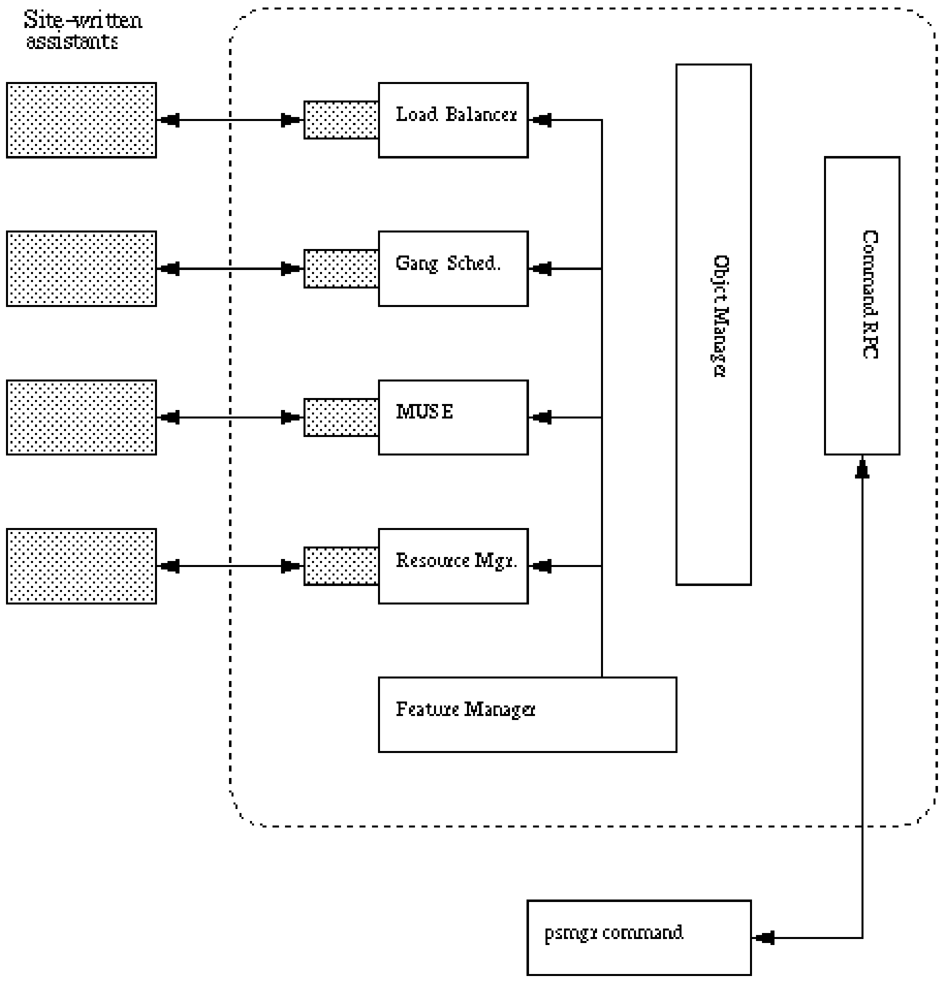


Figure 3

2.1 Scaling and Feature Design

The design of almost every feature of PScheD must deal with the scaling issue in some way. The same software is expected to run on machines of all sizes since special software configurations based on machine size will become a testing, maintenance and development nightmare if they are allowed to proliferate unchecked.

Another painfully discovered truth is that it is difficult to precisely control this class of machine with global controlling software. All of the features of PScheD are designed to guide the micro kernel toward delivering a desired global machine utilization goal. Since each micro kernel has a unique environment, the global managers must expect neither immediate nor full compliance with their requests in every case. This means that all management software must constantly analyze system information and adjust controlling parameters accordingly. Another issue arises since events at the PE level happen at much faster rates than the global controllers can monitor[6]. Often by the time information has traveled to a global manager and it takes some action, events have moved on and conditions are different.

All of these issues taught us that traditional kernel designs which expect to control every aspect of every event in a central place will not generally succeed. A different way of approaching these control requirements is needed, and some time must be spent simply to understand the environment and become comfortable with the range of control that it is reasonable to be able to maintain. A fairly strict expectation that the controllers will not consume a significant amount of network bandwidth and CPU resources is implicit.

2.2 Scheduling Domains

In earlier CRAY Parallel−Vector Processor (PVP) systems a high−level scheduler named the Unified Resource Manager (URM) analyzed system load information and resource usage. An interface to major service providers such as the Network Queuing System (NQS) existed to make the work backlog visible. Knowing the work backlog and with information about current machine activity acquired from the system, URM would compile recommendation lists from the backlog to suggest the order in which jobs should be initiated and send these lists to the registered service providers. The service providers perform the task of job initiation. Early design approaches to Unicos/mk recommended simply moving URM to Unicos/mk.

Deeper consideration of the implications of this recommendation led to the conclusion that the scheduling issues raised by the nature of the CRAY T3E were not similar enough to those of the PVP machines for URM to be useful. The fundamental flaw in the design of URM if simply made to work on the CRAY T3E is the idea that the machine is a uniform provider of computing resources. The CRAY T3E intrinsically divides into two very different domains. The command region can be looked upon as a number of separate single−CPU machines which must be managed so their workloads are fairly equal. The greater part[7] of the CRAY T3E is used to run multi−PE applications. The scheduling issues in this region involve making sure applications reside in memory and are given CPU resources at the same time, especially if they have fine−grain synchronization. It turns out that the URM on UNICOS can be considered a special, simplified case of Political Scheduling.

From a machine utilization point of view, the goals are to minimize fragmentation of PE allocation while reducing swapping and migration to a minimum. Even rough estimates result in very discouraging projected utilization levels if hundreds of large pieces of application memory must be transferred to and from swap space on a context switch.

Two regions[8] are present by default but user requirements often cause the administrator to divide the application region into two parts (see Figure 2), splitting it into a work region and a smaller region intended for development and testing. In the development region, test applications need few PEs and normally execute for short periods of time. Developers also may be using debugging tools so they want their applications to execute often, even if they are being gang scheduled. This behavior is different from that desired when running production work where long time slices improve system utilization.

To make these different scheduling approaches possible, the Political Scheduler is configured to have an instantiation of its scheduling features for each region. Each instantiation is independent of the others so time slices and PE loading can be tailored to the demands of each region. From the point of view of the administrator, the Political Scheduler behaves as though a number of separate schedulers were present. Appropriate scheduling rules are created, each with a separate *domain* name. The domains are bound to the scheduling features with a `bind` directive. Figure 3 shows a single instance of each feature, but imagine that there is a ‘‘depth’’ dimension to each feature where different instantiations can exist. Of course, some features may need no more than a global view of the entire machine. In these cases the depth is one.

2.2.1 So why name them Domains?

Each feature has some sanity−checking capability to help assure a reasonable relationship between the scheduling domains and the regions known to GRM. Early releases will not automatically keep the Political Scheduler and GRM synchronized, but future configuration tools are planned to integrate the configuration of both subsystems. A PS domain and a GRM region must now, and probably always will, agree in size and location. During the design of PS it was thought important to recognize the difference between the GRM configuration and that of PS. In retrospect, it seems that having the two names results in more confusion than clarity. Save us from our cleverness!

3 The Feature Manager

The Feature Manager implements the internal execution control functions of the daemon. The daemon is single−threaded since at the time it was developed, multi−threading support for user−level processes was not available in Unicos/mk.

When execution begins, each feature registers its *bind* function with the Feature Manager. This function is called when a `bind` directive is received at the Command Interface. Binding associates a *node* in the Object Tree with the feature also named on the `bind` directive. The portion of the Object Tree below the named node typically contains the configuration parameters for this instance of the feature. The range of PEs making up the domain is generally a part of configuring a feature.

The specific binding function in the named feature instantiates an instance of the feature for this domain and will register an *action* function. The Feature Manager saves the pointer to the action function and an associated parameter pointer in a list of registered actions for the feature. An optional *exception* function may also be registered at this time. The same parameter pointer as that for the associated action function is assumed.

A *verify* function may also be registered. Verify is called by the feature manager when a `verify` directive is received. Verify is usually used by the administrator to make sure a changed or new configuration instance is acceptable to the feature.

On each cycle of the Feature Manager each of the registered action functions for each feature will be called with the indicated parameter pointer. The parameter is typically a *this* pointer to an instance of the feature *class* and establishes the environment of the feature for this specific domain. The cycle of calls to the action functions continues while the daemon is active.

Some features must perform cleanup or other transition activity when the daemon is terminated. The *exception* functions will be called when the daemon receives a shutdown directive or catches one of a set of registered signals. The daemon executes all of the exception functions before it terminates.

4 The Gang Scheduler

On the CRAY T3E Gang Scheduling is used to assure that in each PE assigned to an application, the execution thread of that application runs at the same time. Applications with fine−grain synchronization using the hardware barrier network require this service if they are to have reasonable performance. The Gang Scheduling feature of the Political Scheduler is designed to deliver the required scheduling behavior without imposing a high synchronization overhead cost. Achieving low overhead meant that methods requiring the CPU schedulers in each PE to have knowledge of each other were unacceptable.

Gang scheduling works in the CRAY T3E with a small amount of kernel support while the major part of the feature resides in the PScheD daemon. Kernel support consists of setting aside a range of priorities named *gang priorities*, making the thread scheduler and memory manager in each kernel aware of these priorities and enhancing an existing system call[9] to allow the Gang Scheduling feature of PScheD to communicate with the kernel. Briefly, the daemon picks an application and consequently the thread which will become the *gang thread* in each PE of its domain and broadcasts that information to the appropriate kernels. The kernels adjust their thread priorities as directed and schedule the threads as those priorities dictate. Since gang priorities are higher than any other user priority, the selected application executes as though it were dedicated.

The memory manager also knows the gang priorities so it takes the necessary action to make sure the memory segments belonging to that application remain resident in memory. When it becomes necessary to swap out memory belonging to an application, all of the PEs on which that application resides are informed to stop remote memory accesses from being issued. All remote memory accesses that are in progress at the time a memory swap begins will be completed.

A practical side−effect of this design is that, if the application which has gang priority for some reason gives up the CPU, the kernel will allow another thread to execute providing it can find one to run. When multiple applications are competing for the same CPU, the Gang Scheduler rotates them through the gang priorities on a configured time slice. Applications which have less than maximum gang priority still enjoy an enhanced priority so they will execute in priority order if the CPU becomes free.

In Figure 4 four applications share a domain of 10 PEs. The slots are intervals of time specified by the configuration of the domain. The application assigned priority 84 will run. In Slot 0 Application A occupying PEs 0−4 and Application B occupying PEs 5−9 run, in Slot 1 Application C occupying PEs 2−6 runs and in Slot 2 Application A and Application D occupying PEs 5−9 run. Slot 3 begins the next cycle. Application A runs twice in three slot periods while the other applications run once.

Application placement greatly influences how often each application runs, so the Gang Scheduler and the Application Load Balancer (see Load Balancing an Application Domain and Figure 7) cooperate to reduce the *depth* of the gangs. A smaller depth means each application runs more often.

Each domain of Gang Scheduling is configured separately making it possible to provide long slot periods for domains running large batch applications and short slot periods for domains running smaller interactive applications. Thus, the amount of overhead needed to manage the gangs and their time slices can be controlled by the administrator to suite the needs of the users. Domains configured to allow only one application to be assigned to a PE at once have no need for supervision by the Gang Scheduler unless the *prime job* feature is used. In these domains the Gang Scheduler feature is not bound and does not function.

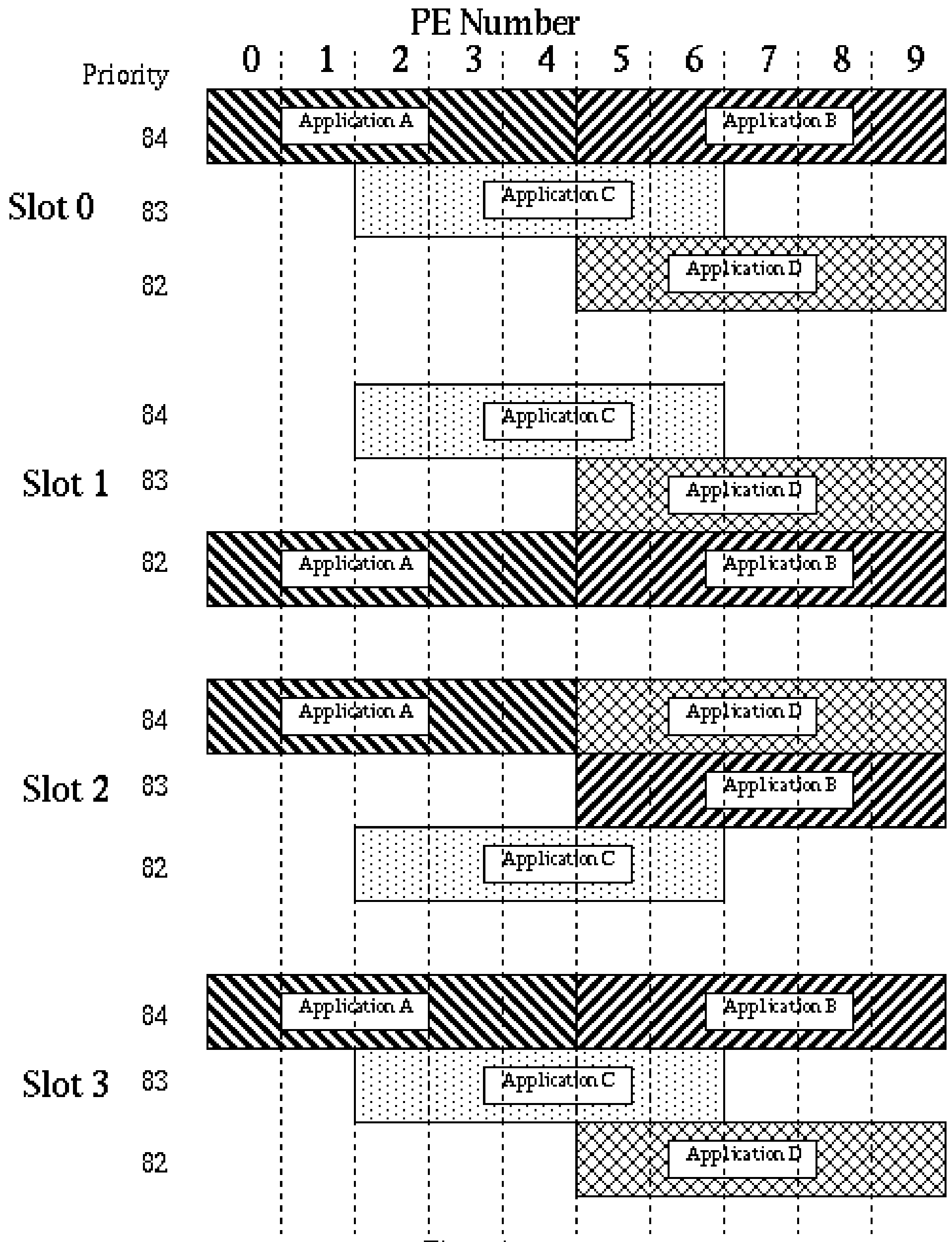


Figure 4

4.1 Controlling a Gang Scheduling Domain

Each domain has these configuration attributes.

`Heartbeat`: The gang time slice center point in seconds.
`Partial`: If true, applications not currently assigned to the prime gang will execute; if false, non−prime gangs will not consume CPU time even if the prime gang is idle.
`Variation`: A floating point number *n*, *n >= 1.0*, that `Heartbeat` is multiplied or divided by to effect the MUSE factor when MUSE is active in the domain.

4.2 Prime Jobs

A prime job is specified by an NQE operator commandor an administrator. When a job is made prime its execution steps will be scheduled as quickly as possible by overriding certain GRM restrictions and bypassing other requests in the queue. The Gang Scheduler and Load Balancer must be bound to the domain in order to support this feature. An assumption is made that only one prime job competes for allocation to a domain. If more prime requests are queued than can be exclusively allocated, they compete with each other and prevent any other execution steps from being allocated to those domains. Placing prime job status on a job is a drastic measure to force an important job to be run. Abusing this feature will destroy the efficiencies and expectations offered by PScheD.

5 The Load Balancer

Load balancing is done in order to maximize overall system utilization. The three steps to load balancing are:

- Filter the processes into eligible and ineligible groups,
- classify the eligible processes, and
- balance the processes by migrating them.

The balancing process is identical for both application and command domains, but the details of how candidates are picked and the evaluation of the cost of migration are different. Command balancing will be described first to lay the basis for the additional work needed to properly balance the application domain.

5.1 Load Balancing a Command Domain

The classification stage involves comparing the candidates[10] in the domain. This is done by generating a *Classification Score*, **C**, of each candidate, **p**. In order to properly compare resource consumption levels among the candidates, each is assigned a normalized entitlement[11], (**E**), memory, (**M**), and CPU, (**U**), classification score component. The administrator will have configured each domain with the desired evaluation weights for these factors. The factors are entitlement weight, **We**, memory weight, **Wm**, and usage weight, **Wu**. The **W** factors are assigned[12] by the administrator through the configuration interface and the values are assumed to range between zero and one. The classification score of each candidate, **Cp**, is determined by evaluating

$$\mathbf{Cp = WeEp + WmMp + WuMp}$$

The list of candidates is ordered by decreasing numerical value of **Cp** as shown in the top portion of Figure 5, *Commands A−E*. This list is then used to create an ideal balance given the weights and the number of PEs available. This is shown in the same Figure labeled *Best balance*. If the cost of migrating the candidates was not a consideration, this would be the end of the evaluation process. In reality, though, the ideal balance is usually a poor choice since many of the candidates would have to be moved and the overhead to do this could be unacceptably high. As a compromise to lower migration cost, only candidates which would most effectively improve overall load balance will be moved. This is shown in Figure 5 next to the label *Lowest cost*.

In this example, candidate *D* was the only one migrated while the ideal balance would have migrated both *C* and *D*. Of course, in actual systems, the number of candidates would be much greater and the number of choices increase dramatically. Poor choices can result in high overhead cost, perhaps without much improvement in utilization.

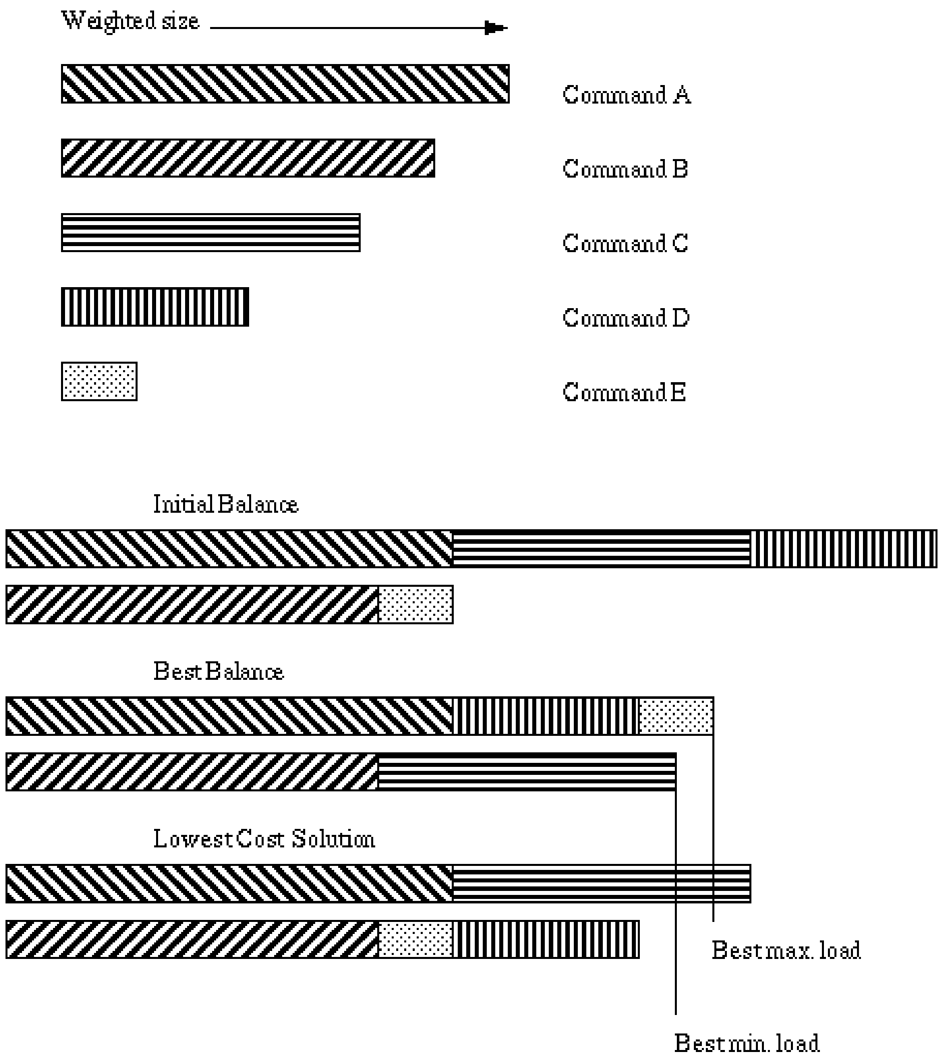


Figure 5

There are many other considerations with important consequences to an effective solution to domain load balancing. Constraints must be established to prevent trying continually to "fine tune" the load. Undesired fine tuning occurs when an evaluation cycle decides the results of a prior cycle were not "best" and so proceeds to rearrange candidates. A way to deal with this potential instability is to keep track of the time a candidate was migrated and leave it alone until a configured time period elapses. In some cases this will allow the candidate to terminate and so remove itself and its load from the system. Another strategy is to set the evaluation frequency with the *Heartbeat* rate to a value suitable to the type of work being done in the domain. It is not productive to deal with short−lived processes. It is more efficient to allow them to finish where they are. Filtering undesired candidates from consideration is described in Filtering Evaluation Candidates.

If many migration actions were initiated in a short time period, a nasty problem involving the order in which candidates are migrated could arise. It is possible to induce a cascade of ultimately useless swap activity as PEs try to accommodate what to them is temporarily *increased* memory usage when a process is migrated to a PE while some candidate, present but destined to be migrated elsewhere, still consumes local resources[13]. There is no way to completely eliminate this side−effect of migration but care in choosing migration order could mitigate it. Such analysis would be complex, constantly controversial and a configuration headache so the load balancer avoids it by migrating no more than one candidate per cycle.

The command domain load balancer has the configuration controls listed below. Recall that an instance of the load balancer sees only its own domain.

- Minimum candidate CPU usage
- Minimum time before a migrated candidate will be reconsidered
- Frequency of evaluation

- Minimum candidate memory size
- Lower bound of candidate User IDs (This can be used to exempt system or maintenance processes from consideration.)
- Entitlement weight
- CPU usage weight
- Memory usage weight

It is possible that other controls will be necessary as experience with the actual environments in which the evaluator must be effective grows.

5.1.1 Filtering Evaluation Candidates

The measurement of consumption rates, especially in the command region, can be very noisy since short−lived processes and the uneven resource consumption of many processes can lead to misleading evaluation scores. The filters are configured with the minimum CPU usage and minimum memory size to help moderate the effects of short−term process behavior on the evaluation of migration candidates. The minimum UID factor is intended to exempt system processes and other special users from consideration.

5.2 Load Balancing an Application Domain

Balancing an application domain is a somewhat more complex issue than that of a command domain. The objectives of application load balancing are to

- minimize swapping,
- minimize migration cost,
- do expensive migrations only when needed,
- minimize the number of *gangs*, and
- maximize contiguously allocated PEs per *gang*

Unicos/mk imposes the requirement that all PEs allocated to an application have contiguous logical PE numbers. The location of an application is specified by its base PE number and its size (in number of PEs). This can lead to situations where occupied PEs are scattered throughout a region in such a way that no application waiting to be allocated can be fit into any contiguous span of available PEs. Fragmentation of this kind lowers the utilization of the machine by leaving portions of it effectively unavailable.

In Figure 6 a fragmented domain of PEs has developed. The load balancer will have recognized this but will take no action unless the fragmentation is causing some application to wait for initiation. Further, the load balancer must be able to make space available in sufficient quantity to allow at least one of the waiting applications to be accommodated before it will initiate migration.

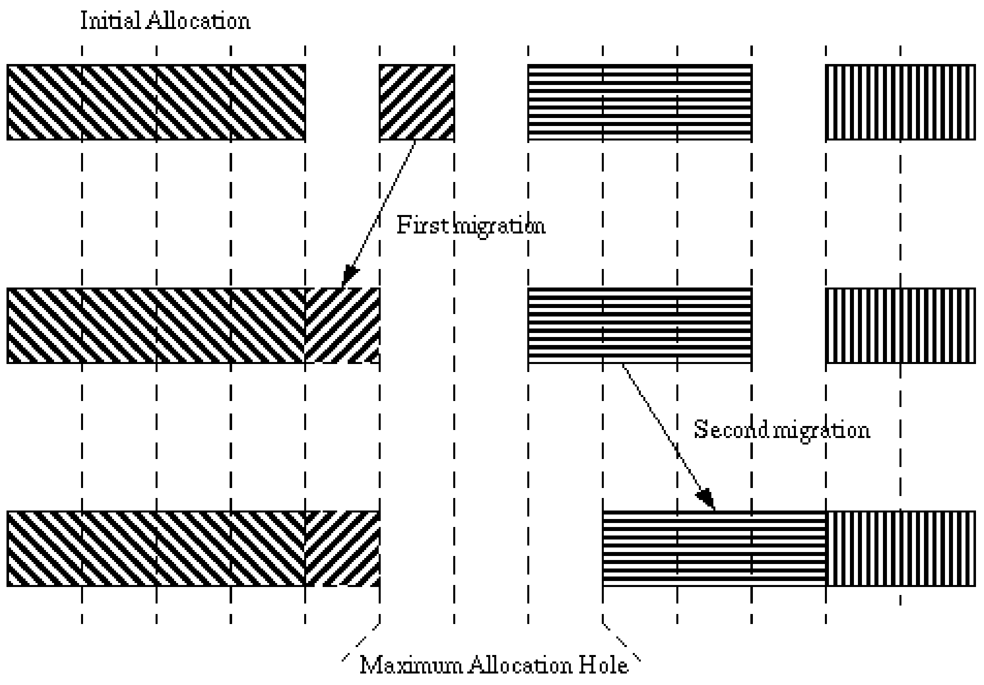


Figure 6

Assuming that the cost considerations have been satisfied, migration will increase the size of contiguous free space by pushing applications right or left in the domain to squeeze out allocation holes, starting with the lowest cost migration that increases the span of available PEs. As with command balancing, this is done one application at a time. Figure 6 shows the migration steps (*First migration* and *Second migration*) as the applications are moved into contiguous space. At the same time this is going on, GRM will be reevaluating its waiting applications. As soon as space becomes available, GRM will initiate whatever it can. Because of this competition between GRM and the Load Balancer, it is necessary to completely reevaluate the domain on every cycle.

Another consideration in managing an application domain is *gang balancing*. Gang balancing goals apply only when a domain is configured to allow more than one application to be assigned at once to the PEs. This decision is made by the administrator based on typical application sizes and behavior and the performance demands of the computing environment. From the standpoint of throughput of an application, sharing PEs simply means it will take longer for each application to complete. Gangs are more fully described in The Gang Scheduler.

Two of the load balancing goals deal with the number of gangs in a domain. This refers to how many applications share the same PEs in a domain. Reducing this number improves the performance of each application and makes better use of the resources of each PE as is shown in Figure 4. The steps needed to reduce the number of gangs in a domain and improve utilization by keeping more of the PEs busy is shown in Figure 7.

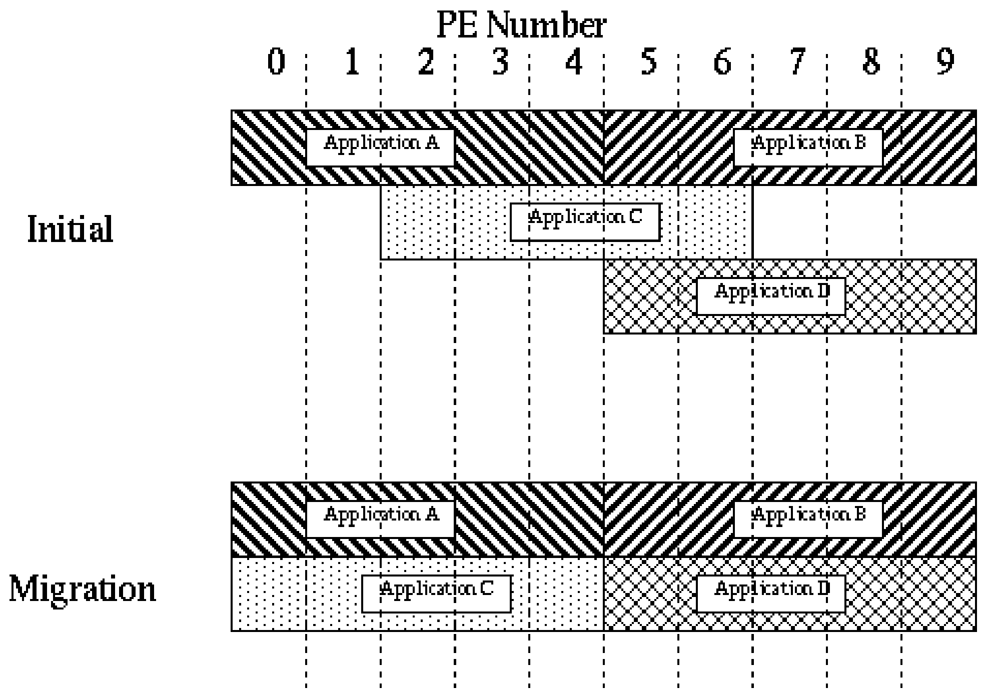


Figure 7

6 MUSE

The Multi layered User−fair Scheduling Environment (MUSE) feature implements a scheduling strategy similar to the well known Fair−Share Scheduler as implemented on systems such as UNICOS. In contrast to those implementations, MUSE and its integration with the concept of scheduling domains allows for better scaling to large CRAY T3E environments. The CRAY T3E presents many challenges to a useful implementation of fair−share scheduling strategies. The first challenge is of scaling, the second of determining exactly what an entitlement means under these circumstances.

6.1 MUSE Entitlement

Every user or account is assigned an *entitlement* by the administrator. A user's entitlement is the proportion of the machine resource (in this implementation, CPU time) that user should be given in competition with other active users. MUSE determines, based on the load and usage history, what priority each user should be given in order to reach the entitlement goal.

6.2 An Outline of MUSE

As in UNICOS, a global representation of the entitlement tree is maintained in the User Database (UDB). Because of this, a global instantiation of MUSE exists to collect resource consumption information delivered by each of the MUSE domains into the global domain, the UDB. The administrator's view is that of a single−system image even though there may be many domains controlling resources being consumed at vastly differing rates.

When the MUSE UDB domain is bound, a global entitlement representation is created as the source of the information needed by the controlling MUSE domains. The UDB domain neither adjusts PE parameters nor collects usage information from the PEs. The controlling domains do that work. Usage information filters up to the global UDB domain from the controlling domains and from there is generally distributed.

In the typical minimum case two controlling domains are bound. The first domain is that of the command PEs which run ordinary Unicos/mk processes, while the second domain is responsible for multi−PE applications. Each controlling domain maintains a virtual entitlement tree private to the domain, having the same structure as that of the global tree. The virtual tree, however, includes only the resource consumers that populate that domain. Domain usage is propagated to the global domain at a configured rate, and the global domain distributes new consumption information to interested controlling domains.

Every PE has a Process Manager (PM) responsible for handling work assigned to the PE. When a process is assigned to a PE, the responsible controlling domain creates a controlling node within the PM for the user[14]. PM uses the nodes to adjust local priorities based on effective entitlement and collects usage information in the nodes for harvesting by the controlling domain for global dissemination.

A PM has no global view of a user’s activity, working only with resources locally consumed. The controlling domain periodically assesses both global usage and overall domain usage and, when necessary, adjusts PE entitlement.

6.3 MUSE Domain Configuration

The configuration parameters provide a good view of the MUSE feature. Each domain (including the global domain) has its own set of configuration parameters. A certain degree of configuration consistency among the domains must be assumed since each domain should be working toward more or less uniform goals. Certain domains may intentionally be excluded from some of the general rules to provide for dedicated applications or other special needs. Sanity checks examine individual domain configurations and report unexpected or contradictory rules. Such mistakes as configuring more than one domain to include the same PE are strictly prohibited, but many seemingly inconsistent rules may be intentional and necessary. The administrator must act, based on this analysis and the established performance goals, to make any needed configuration adjustments.

`Heartbeat`: How often (in seconds) the MUSE scheduling feature is executed for the domain.
`Decay` : The decay rate (in seconds) of accumulated usage. This discards historical usage over time.
`IdleThreshold`: The percentage of entitled usage below which a resource consumer is considered idle. Being considered idle means

- the priority of the associated process is set to the non−MUSE priority of 100 and
- the entitlement and usage of the resource consumer are no longer considered in any global calculation. Thus, the scheduler behaves as if the idle resource consumer is not active in this domain. This effectively controls redistribution of usage.

`ShareByACID`: If true, the domain is controlled by the user’s account ID; if false, the domain is controlled by the user’s UID.
`NodeDecay`: How long (in seconds) resource consumer usage information is kept on the same levels of the system as information about active resource consumers.
`UdbHeartbeat`: How often (in seconds) the domain synchronizes its usage information with the global UDB domain. This makes usage information visible to other domains and controls how often usage accumulated in other domains becomes visible to this domain.
`OsHeartbeat`: The frequency (in seconds) that the operating system should accumulate usage and adjust the priorities of running processes or applications.
`OsActive`: If true, the system actively enforces the assigned entitlement by adjusting priorities; if false, the system accumulates usage but does not adjust priorities.

6.4 The MUSE Factor

Service providers such as NQS/NQE need information from a system to help order the backlog and initiate jobs which are most likely to run. It is harmful to system performance when many jobs belonging to users with little effective entitlement are brought into the system and compete for resources with users of high entitlement. These jobs generally end up swapped after some small initial activity and stay inactive until system workload drops to a point where the job is eligible for system resources. Users and administrators would also like to have a simple way to determine how well a given user might have jobs serviced at a certain time.

In UNICOS systems, NQS had an elaborate entitlement analysis capability that mimicked that of the fair−share component in the kernel. The result of this analysis was a rank that had meaning when compared with the ranks of other users. This helped NQS decide which jobs to initiate. Although this analysis worked well, it was complex and required tinkering as the underlying operating system evolved. The influence of small analysis errors can be subtle and not at all apparent in every environment. Thus, it was not always obvious whether the entitlement assessment conformed with that of the host system.

When PScheD was in its early design stages, the NQS/NQE developers lobbied strenuously for an interface to the system which would provide an externally useful entitlement assessment upon request. It was becoming clear that writing an external analyzer similar to that used with UNICOS would be a difficult job for Unicos/mk because of its much more complex organization and non−uniform service regions. The *MUSE factor* was created to fill this need.

The MUSE factor is a machine−independent measure of a user’s effective entitlement. This means a service provider can compare the

MUSE factors of competing users and decide how to rank them for a single machine as well as determine which of a number of machines able to process the job would offer the best service[15].

Figure 8 demonstrates the MUSE factor for a user and shows how it represents a user's effective entitlement. A MUSE factor of 1.0 means the user could consume all the resources of the machine for some period. A MUSE factor of 0.0 means the user has consumed at least all the entitled resources and the system will give other users with non−zero MUSE factors a higher priority. The numeric value of a MUSE factor is meaningful with respect to the MUSE factors of other users on either the same or different machines.

In Figure 8 the line labeled *Usage* shows the cumulative resources consumed by user ***r***. This is shown only to describe the way the user consumed resources over time. The line labeled *Decayed usage* is calculated by MUSE and is the effective usage, modified by the decay factor, that characterizes this user with respect to the user's entitlement. The MUSE factor ***M*** for user ***r*** (labeled *MUSE factor* in Figure 8) is

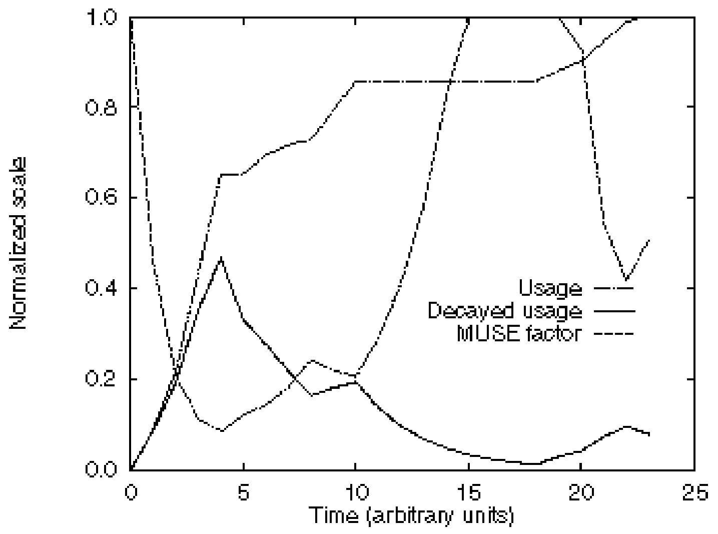
$$Mr = Er \times Er / Ur$$

where **Er** is the user's normalized entitlement and **Ur** is the user's normalized decayed usage. The MUSE factor is clipped at 1.0 since values higher than that decay very rapidly with small amounts of usage.

The MUSE factor incorporates a user's relative entitlement among machines as well as the ratio of entitlement to usage. This allows a selection of which machine in a network might best process a user's work given roughly equal performance capabilities. For example, if a user had a 20% entitlement on machine *A* and a 5% entitlement on machine *B* but on both machines had a entitlement to usage ratio of 0.4, the MUSE factors would be 0.08 for machine *A* but only 0.02 for machine *B*. That might mean that the job would best run on machine *A*.

An interface is provided to PScheD which takes as input a list of one or more UID, ACID pairs (either numeric IDs or user or account names) and returns the associated MUSE factors. (Both UID and ACID must be given since the requestor should not need to know whether MUSE is configured for UIDs or ACIDs.) For example, if a service provider wanted to know the current MUSE factors for users with UID/ACIDs 111, 8456 2345, 8855 and 6789, 9445 it would compose a MUSE factor request with the string `<111,8456 2345,8855 6789,9445>` and MUSE in configured for UIDs could respond `<111=0.0022 2345=0.1577 6789=0.0399>`. MUSE will return only the one ID of each pair that corresponds to its configured mode. The ID returned will be either the name or number that was used on the request. MUSE values change over time so the service provider must refresh its notion of the MUSE factors of active IDs from time to time.

Other features also acquire the MUSE factors to order gang scheduling and load balancing candidates in cases where entitlement is a factor in their decision.



MUSE Factor

$$M = \frac{e^2}{u}$$

Where e is the user's normalized entitlement and u is the user's normalized decayed usage. M is limited to the range $0.0 \leq M \leq 1.0$.

Figure 8

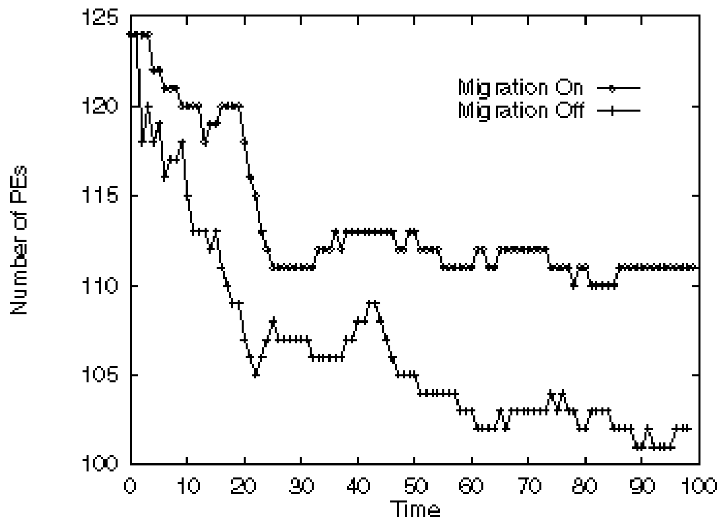
7 Does Migration Improve Performance?

The requirement that applications occupy consecutive logical PEs has often been mentioned as a factor which could limit the utilization of the CRAY T3E. This is certainly intuitive and has been seen to occur, especially on small machines, with some frequency.

The Load Balancer attempts to reduce fragmentation by removing unoccupied gaps between applications. The result of this process is to make larger contiguous spaces in which to allocate new work. To test whether migration is an effective way to improve PE utilization in the CRAY T3E, we developed a workload for a machine with 128 application PEs that we could run both on the Load Balancing Simulator and the hardware. The results reported below are from the Load Balancing simulator. The configuration was set to run one application per PE.

A workload consisting of an unlimited source of applications is introduce into the system. Selected application sizes ranging from eight to 128 PEs with run times from 180 to 2048 seconds are used. Up to 33 applications are either active or queued for initiation at one time. We wanted a reasonable backlog in order to give GRM some choice in its ability to fill the machine. The test is run until the PE utilization value appears to stabilize. An identical application stream is used for both migration and non−migration scenarios.

The top graph in Figure 9 shows that with this workload, the average PE utilization with migration active stabilizes at about 112 PEs while without migration the stable level of utilization is about 103 PEs. This means that migration delivered about nine more PEs to the users or about seven percent more effective PEs. In this graph *Time* is expressed as a percent of the entire test period. The lower graph in Figure 9 shows the relative amount of time applications wait to be initiated. Migration tends to slightly increase the wait time for small applications but significantly reduces the wait time for large ones.



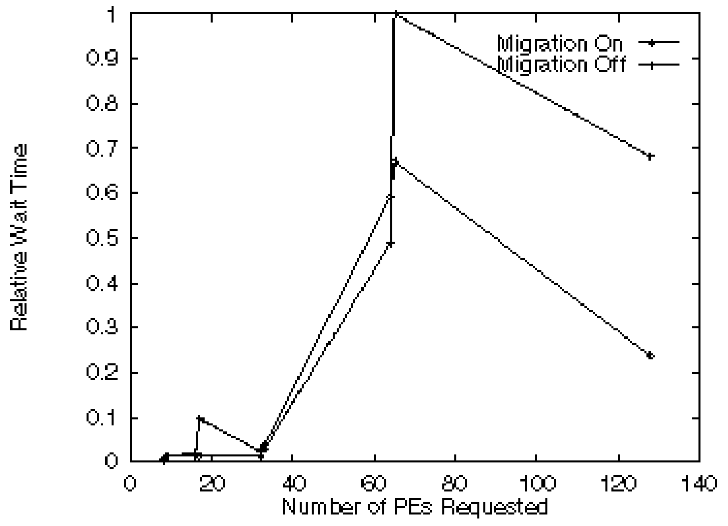


Figure 9

We have found that it is fairly easy to design a workload to demonstrate almost any type of behavior. The performance simulations reported here attempted to create a reasonably realistic environment that would show how migration can increase machine utilization and decrease the time users wait for results. The results are modest but we believe them to be realistic in typical environments.

8 Summary

PScheD has been conceived to support solutions for the general administrative needs of large CRAY T3E systems and future machines. As of this writing, the feature is powerful enough to support the requirements of system scheduling. Special system enhancements to support political scheduling are minimal. This was a primary goal since we did not want to require complex kernel assistance as this makes future system development more difficult to design, implement and test. Unicos/mk will run without PScheD, but not with the same degree of hardware utilization and administrative control as when it is present.

It has not been feasible to describe all of the features of PScheD in detail, especially those capabilities intended to provide insight into what decisions are being made, within the scope of this paper, but we trust enough detail has been provided to communicate the flavor of our design.

PScheD is an evolving tool intended to support current as well as anticipated future hardware and system advances. The flexibility of feature management and configuration means additional capabilities can be added without the need to rewrite the existing features. It is also easy to remove unwanted feature bindings when PScheD is configured so only the features needed to handle the present needs of an installation are active. For example, if only Gang Scheduling is needed, other features such as MUSE or Load Balancing can remain unbound and inactive. Features which are initially unbound may at any time be bound if their configuration information has been set up.

Footnotes

[1] Additional PEs may be present to support operating system needs
[2] The number is determined by the size of the machine and the type of workload.
[3] Generally, the command region has no restrictive attributes.
[4] Migration is managed by the political scheduler.
[5] Batch and interactive job initiators, for example, have different service provider types.
[6] An early attempt to globally manage memory dramatically clarified this issue.
[7] The design assumption is that the machine is to be used more for multi−PE applications than single−PE work.
[8] A command region and a single application region.
[9] The `policy()` system call.
[10] See Filtering Evaluation Candidates.
[11] See MUSE.
[12] The three factors are quite different so typically only one of them has a dominant weight.
[13] This could have devastating consequences with large multi−PE applications.
[14] Either share−by−UID or share−by−account may be selected. Share−by−UID is assumed here.
[15] Not necessarily fastest turn−around, though, since the MUSE factor does not indicate the performance capability of a machine.

We wish to acknowledge the many people who helped to review and critique this work. Even with the best scrutiny, errors will creep into any work. The author assumes responsibility for all misleading or incorrect content.