Uniform Co-Scheduling Using Object-Oriented Design Techniques *

Nayeem Islam and Roy Campbell [a]

[a]University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Avenue, Urbana, IL 61801

In this paper, we present the object-oriented design of a uniform model of scheduling for groups of processes in loosely coupled and tightly coupled environments. Our purpose is to provide a uniform programming interface for both environments with a large number of policies that may be chosen to suit the application. In our design, we partition functional components into different class hierarchies. Subclasses within a class hierarchy express different policy options. We specify the interactions between the abstract classes in the system that form a framework for scheduling. Finally, we provide preliminary performance results for implementations of the design.

Keyword Codes:D.4;D.4.1;D.4.7
Keywords: Operating Systems,General; Process Management; Organization and Design

1. Introduction

Most system designs separate and make independent the schedulers for distributed and parallel applications. In this paper, we describe a uniform framework for scheduling processes and groups of processes in tightly coupled shared memory systems and loosely coupled distributed systems. Our framework is designed to encompass a large number of different policies and mechanisms, making it general and efficient. Our framework is designed to work well with tightly coupled parallel numerical applications [9, 10] such as FFT, Simplex, Cholesky factorization and Matrix Transpose. Executing such tightly coupled parallel applications requires dispatching each of the processes in the application "simultaneously". This is often referred to as **gang** scheduling. When tightly coupled parallel numerical applications are scheduled asynchronously, processor time may be wasted while processes wait for other parts of the application to respond to messages. Gang scheduling improves application performance [13, 8] in tightly coupled shared memory machines. Its effectiveness for distributed systems is less well understood and, in general, more asynchronous or much looser scheduling policies are used. One asynchronous form of scheduling, called co-scheduling, *attempts* to schedule the processes of the applications at the same time but if it does not succeed, a subset of the processes are run. We will use the terms gang scheduling and co-scheduling interchangeably.

For tightly coupled applications run on a network of workstations, the gang scheduling approach requires that the application have exclusive access to the workstations for the duration of its execution. Gang scheduling resource allocation runs contrary to the view of workstations as dedicated resources for the tasks of the local user (those sitting in front of it and typing into its console) rather than dedicated to a remote task. Tightly coupled applications raise interesting policy and structural issues for an operating system if the operating system also supports other kinds of applications. In our system, loosely coupled and tightly coupled applications tasks may

*This work was supported in part by NSF CDA 8722836 grant number 1-5-30035 and the first author was supported in part by an IBM Graduate Fellowship.

run on multiprocessors, on a group of machines connected by some type of network or on a mixture of both.

Portability of parallel applications suggests that the operating system interface for applications on shared memory multiprocessors should be the same as that for distributed systems. However, there are a number of differences in the systems that make such interfaces difficult to design. In a distributed system, the latencies for synchronization and coordination are large. Distributed and shared memory systems have different failure modes: a node in a distributed system may fail and packets may not be delivered, may be delivered out of order or may contain errors. Node allocation is the primary issue in distributed systems. Processor allocation is the most important issue in shared memory multiprocessors. We have found that an integrated solution is important for high performance applications. In this paper, we will ignore the issue of node failures, and assume messages are delivered reliably. Although the primary contribution of this paper is a uniform model of many scheduling schemes, other important aspects include: an object-oriented design that encourages reuse of code and interfaces, and an abstraction for gangs.

Our experiments are performed on *Choices* [2, 4, 3] an object-oriented operating system written in C++. *Choices* provides both synchronous and asynchronous message passing as well as group communication [3, 9]. *Choices* has MessageContainers (mailboxes) and ContainerGroups (groups of mailboxes). Class names will appear in sans serif font.

2. Motivating example

Many parallel numerical applications may be modeled as consisting of three phases: computing, communicating and waiting. In message based applications on shared memory or distributed memory computers the presence of a blocking receive or send implies that the application may wait to send or receive data. Processes of a parallel program may synchronize explicitly using barrier synchronization for mutual exclusion or they may interact as producers and consumers using communication primitives. In all three cases processes may wait for other processes that are not currently scheduled if there are more processes than processors.

In recent papers, Feitelson et al [8, 7] argue strongly for gang scheduling for fine grained computations on shared memory multiprocessors. They show that busy-waiting with gang scheduling is better than blocking for fine grained applications. The reason is that when the processes of an application frequently interact, if all the processes of the applications are not simultaneously run, the rescheduling overhead increases waiting times.

Model of Application behavior

The following code fragment illustrates the types of programs we will study.

```
for i =1 to N ( --- gang size)

for j =1 to K/N  (-- work per process)
compute for time t (-- grain size )
broadcast sync or all processes communicate ( communication overhead )
```

We assume a SPMD model for our parallel applications, where there are N processes, K amount of work, and each process executes K/N iterations of the problem. At the end of each iteration each process communicates with some subset of the other processes. In our example, we assume broadcast communication involving all the processes of the application or some form of communication where all the processes are involved. One example of the former is the Simplex application and an example of the latter is the FFT application which uses butterfly style

communication. The pattern of interaction is less important than the basic requirement that all processes communicate at the end of each iteration. Clearly, the waiting time of applications is increased if some of the application processes are descheduled. These parallel applications benefit from gang scheduling in both shared memory and distributed systems.

3. Related Work

The Mach kernel [15, 1] provides a user adjustable policy for gang scheduling. Our work differs from the server based approach advocated by Mach [1] for using different scheduling algorithms. In our approach, different scheduling techniques are incorporated through subclassing. The Medusa [13] operating system was one of the first to employ gang scheduling. It implemented co-scheduling, a policy that attempts to provide all the processors requested by a gang but makes do by scheduling a sub-group if all members of the gang cannot be scheduled.

The V [5, 18] distributed operating system has facilities for preemptable remote execution of individual processes and not gangs. The V program manager maintains a cache of information about the resources of machines that may be used for remote execution. Stumm has studied the dissemination of resource usage information in [16]. Condor [11] and Marionette [17] are two systems that provide facilities for scheduling on distributed systems by locating idle workstations that are connected by a local area network. They are implemented on top of the UNIX operating system. Condor's primary interest is in scheduling a single process (not a group) and completing the application's computation in a fault tolerant manner. Marionette presents a master/slave model for parallel distributed computation. Slave processes do not communicate with each other. This helps fault tolerance but greatly restricts the types of parallel applications that may be written. It does not have facilities for locating groups of workstations based on resource usage or machine availability.

Some library packages such as Express allow programs to be loaded and then started, thus providing minimal support for tightly coupled co-scheduling of applications. Once started program performance depends on the UNIX scheduler on each workstation. Recent simulation studies evaluate the effects of co-scheduling on a network of workstations [6, 12].

4. Components of the Scheduling Framework

The five major components in the scheduling framework are the Gang, LongTermScheduler, ProgramLauncher, ResourceMonitor and ProcessContainer. Each component in our design is represented by an abstract class. The abstract class is a specification of the interface for all specializations of that component. Specializations are enumerated as class hierarchies. We will use interface tables to show the methods of the abstract class that are inherited, or refined. These tables provide a good indication of code and interface sharing in our design. In sections 6, 7, 8, 5 and 9 we will describe each of these different components. In section 10 we describe the framework for scheduling, which shows how the different components interact with each other to perform scheduling.

5. Processes and Gangs

This section describes the main application interface for programming gangs. The original *Choices* Process class hierarchy [3] was redesigned to accommodate Gangs. Gangs schedule a group of Processes 'simultaneously' on a group of Processors. Different specializations schedule gangs in different ways. For example, gangs may be scheduled on a network of workstations, on a tightly coupled multiprocessor, or on a mixture of both. A Task is a schedulable entity.

Subclasses include Process and Gang. This is an example of software refactoring where methods of the 'old' Process class were moved into the Task class. These methods are now inherited by Gang and its subclasses and by the Process and its subclasses. The new class hierarchy for Tasks is shown in Figure 1.

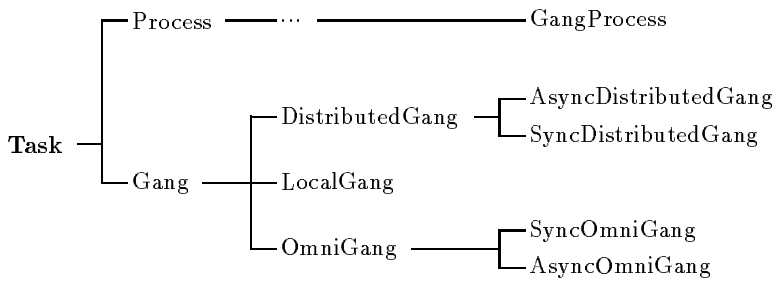

Figure 1. Process and Gang Class Hierarchies

All the methods associated with Task are used for scheduling. Its methods are invoked by ProcessContainers, by LongTermSchedulers and by ResourceMonitors. A Process has methods to transfer control between Processes. A Process has methods block to block the process, giveCPUTo to give the Processor to a specified process, and relinquishProcessor to return the process to the ready queue in order to allow the Processor to run other processes.

A Gang has methods to add and remove members and to schedule its members. The methods of the Gang abstract class and its concrete subclasses are shown in Table 1. The method conc is used to specify the number of processors requested. Our design allows Gangs to be blocked and multiprogrammed. The Gang class has three primary subclasses DistributedGang, LocalGang, and OmniGang. Input parameters to the constructor specify whether or not the Gang requires exclusive access to machines.

A LocalGang is used for gang scheduling on shared memory multiprocessors and OmniGang is used for a network of shared memory multiprocessors and uniprocessors. A DistributedGang is used for a gang that is divided among a number of workstations. All these Gangs may be run in dedicated mode or in multiprogrammed mode. In the dedicated mode, a processor is reserved for each process of the application. In the multiprogrammed mode, there may be fewer processors than application processes on a multiprocessor or in the case of a network of uniprocessors, the Node may be multiprogrammed, that is shared, with other application processes. Moreover, more than one of the processes of the application may be on a particular processor or Node.

OmniGangs and DistributedGangs may be run in either tightly coupled or loosely coupled mode, by refining the scheduleMembers method. In this method, each type of gang may provide some form of 'barrier' synchronization just before processes are scheduled. The added synchronization has to be part of the scheduleMembers method. The classes prefixed by Sync in the class hierarchy provide additional barrier synchronization of the processes of the application by refining the scheduleMembers method whereas those prefixed by Async have no barrier synchronization. The GangMember is a subclass of Process that overrides the die method to invoke the memDone method on the Gang when the process terminates. The done method is called when the last member terminates. This initiates garbage collection of Node allocated resources for the gang. The method NumProcessors gives the number of processors that were actually allocated for the gang. The interfaces for the Task, Process and Gang classes are shown in Table 1.

Method types	
Symbol	Meaning
=0	pure virtual
•	first defined
▽	inherited
△	refined
-	undefined

Interface Protocol inheritance in Task Hierarchy						
Class::method	block	dispatch	ready	remoteDispatch	die	giveCPUTo
Task	=0	•	•	•	-	-
▽ Process	•	▽	▽	▽	•	•
▽ ▽ GangProcess	▽	▽	▽	▽	▽	▽

Interface Protocol inheritance in Gang Hierarchy								
Class::method	block	add	remove	NumProcessors	conc	memDone	done	scheduleMembers
▽ Gang	=0	•	•	•	•	•	•	=0
▽ ▽ DistGang	•	▽	▽	▽	△	△	△	•
▽ ▽ OmniGang	•	▽	▽	▽	△	△	△	•
▽ ▽ LocalGang	•	▽	▽	▽	△	△	△	•

Table 1
Process and Gang Interfaces

6. Long Term Scheduler

The LongTermScheduler controls the degree of multiprogramming in the system [14]. Gang scheduling also performs load balancing [1, 8]. The degree of multiprogramming is specified as an input parameter to the constructor of the LongTermScheduler. All Tasks that are to run, enter the scheduling system through the dispatch queue of the LongTermScheduler. The LongTermScheduler decides whether a Task should be allowed to compete for local and global resources. Tasks enter the dispatch queue from the command line of the system shell, from the network, as the result of a call to the remoteDispatch method on a Task on some other Node or from within an ApplicationProcess.

For console input the commandLine method is called. If the console input runs a program, the method loads the appropriate application process from disk, and then inserts it into the dispatch queue using the add method. A separate process takes processes off the dispatch queue by invoking the remove method and the process consults the ResourceMonitor to determine on which Node to dispatch the Task. In some cases, a remote process may be forwarded once it has been dispatched to another Node. The sender may have had stale state information.

Interface Protocol inheritance in Long Term Scheduler Hierarchy					
Class::method	add	remove	commandLine	Enable/DisableRemoteProcesses	start/stopExclusive
LongTermScheduler	•	•	•	-	-
▽ RemoteExclusive	▽	▽	▽	-	•
▽ UserExclusive	▽	▽	▽	•	-

Table 2
The LongTermScheduler Interface

Table 2 shows two different subclasses that implement different scheduling polices. Each Task records the size of a particular process, whether the process is remote, and whether it requires to be run in dedicated mode. This information is used to locate the best set of processors to run the application. One subclass, UserExclusive does not allow remote processes to run on the local machine making it a dedicated machine for the local user. It exports the methods DisableRemoteProcesses and EnableRemoteProcesses. The user may run the application **NoRemotes** which will destroy all remote applications and disable further applications that

are started remotely. Another subclass RemoteExclusive allows a remote job to acquire the workstation for its exclusive use for a short period of time. It exports the method startExclusive, and stopExclusive, which control when to allow remote processes to run. The duration for which a machine may be acquired in exclusive mode by a remote process is specified using the command **setRemoteExclusive**.

7. ShortTerm Schedulers

ProcessContainers are the short-term schedulers of *Choices*. The ProcessContainers export the methods add, remove and isEmpty. We subclass the existing ProcessContainers to RRGangScheduler to make them understand Gangs. When the last processor has invoked the remove method on the scheduler, the scheduler itself will invoke the scheduleMembers method on the Gang. The short-term scheduler classes are shown in Figure 2.

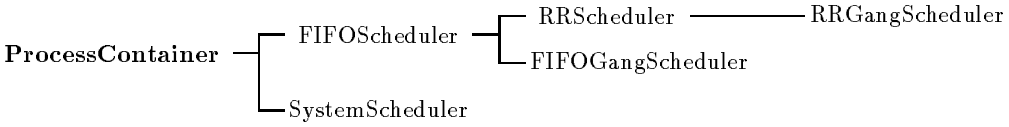

Figure 2. Short Term Scheduler Class Hierarchies

8. Resource Monitoring

The ResourceMonitor class performs two functions, to collect local/global resource usage information and select the best set of sites for a task. Its three methods, updateSchedInfo, updateCPUInfo and updateVMInfo, are invoked by the short term scheduler classes, the CPU class and the Pager class. The individual *Choices* subsystems are changed to update the ResourceMonitor. Optionally, a separate process may be spawned that monitors the various subsystems and then calls these three methods on the ResourceMonitor. These three subsystems represent standard load metrics that have been used for load balancing. Other metrics may be added by subclassing the ResourceMonitor class. The short term schedulers' information changes when its add and remove methods are invoked. These methods invoke the updateSchedInfo of ResourceMonitor to record these changes. The pageout daemon may call updateVMInfo of ResourceMonitor depending upon the paging behavior of the system. Specializations of the ResourceMonitor class may use more or different information from various subsystems being monitored. The frequency of local and distributed updates can be specified using the method setUpdateFreqlocal and setUpdateFreqGlobal. For example, a multi-level feedback queue will provide more information than a simple FIFOscheduler.

ResourceMonitor also has a Process associated with it that is used for acquiring and disseminating global resource information. This process joins a well known ContainerGroup when the ResourceMonitor is created. The class ResourceMonitor exports the method selectSite. The LongTermScheduler will invoke this method for every Task.

Currently there are four subclasses of the ResourceMonitor class that implements resource information consistency across Nodes. The first two subclasses are PublishingResourceMonitor and QueryResourceMonitor. Both of these have further subclasses that are used for setting up hypercube style message passing systems.

QueryResourceMonitor spawns a process which listens on a corresponding ContainerGroup for incoming requests. In the query algorithms, each time a task is scheduled the most up to date resource information is gathered from machines on the network. The steps for the query algorithm are:

```
1. query all members and select best subset.
2. post list of members actually selected.
```

The publishing scheme uses locally cached resource information. The steps for publishing are:

```
1. search local performance cache
2. post list of members actually selected
```

The ResourceMonitor class is abstract and has 4 concrete subclasses. The subclasses must implement all the find procedures as each uses a different mechanism for finding and organizing Nodes. The concrete classes HypQueryResourceMonitor and HypPublishResourceMonitor refine all the metric calculation methods since these use application specific information to find workstations. The interfaces and code sharing between the classes is shown in Table 3. The stars refer to wild cards.

Interface Protocol inheritance in Resource Monitor Hierarchy					
Class::method	update*	*Metric	selectSite	findOptimal*	setUpdateFreq*
ResourceMonitor	•	=0	•	=0	•
▽ QueryResourceMonitor	▽	•	▽	•	▽
▽ ▽ HypQueryResourceMonitor	▽	△	▽	•	▽
▽ PublishingResourceMonitor	▽	•	▽	•	▽
▽ ▽ HypPublishingResourceMonitor	▽	△	▽	•	▽

Table 3
The Interface Table for Resource Monitor and its subclasses

9. ProgramLauncher

The ProgramLauncher class provides process migration facilities. It has one subclass that implements static placement or initial placement called StaticLauncher.

The methods of ProgramLauncher allows one to launch a loosely coupled task by specifying a reference to the program (by invoking launchref)or by passing the task itself (by invoking launchtask). This part of the framework benefits from multicast since both broadcast or point-to-point communication for a large file or program transfer would be very costly.

10. A Framework for Scheduling

In this section, we describe the framework using a class diagram and informally describe the method invocation sequences for the important scheduling operations. Each subclass of an abstract class must implement all the methods of the abstract class either through inheritance or refinement. Figure 3 shows the various abstract classes that represent the components of the scheduling framework and various method invocations between the objects. There are two levels of scheduling in *Choices*. An application process or a group of application processes is first selected by a system level scheduler and then by a processor level scheduler. The presence of a ProcessContainer in Figure 3 indicates places where a policy decision may be made as to

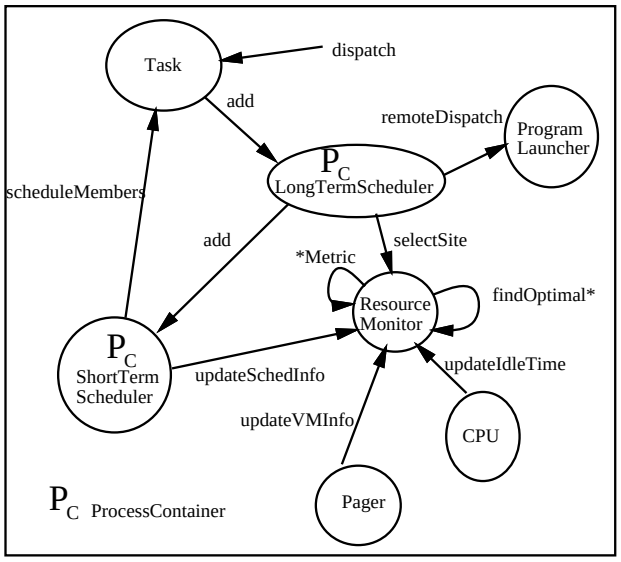

Figure 3. Local Interactions

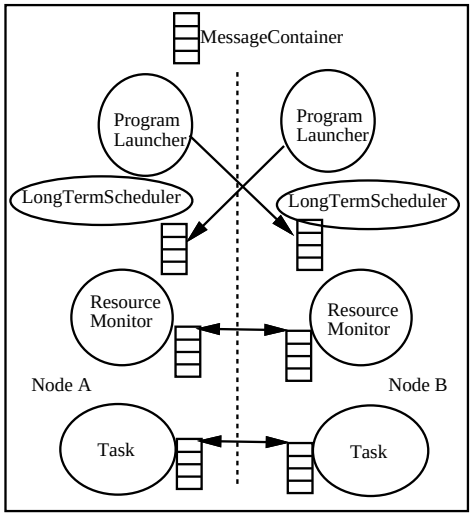

Figure 4. Network Interactions

which Task in the ProcessContainer will be processed next. The Task may experience queuing delays at these points.

When a Task is to be run, it invokes the dispatch method on itself, which adds the Task to the dispatch queue of the LongTermScheduler on the local Node. The LongTermScheduler, invokes the selectSite method on the ResourceMonitor, passing appropriate resource usage requirements of the Task, and is returned a SchedulingVector. The SchedulingVector describes where to schedule the Task. The ResourceMonitor uses one or more of the policies described earlier to maintain global scheduling information. The LongTermScheduler uses the information in the SchedulingVector, to invoke the remoteDispatch method to run a Task remotely or invoke the ready method on the Task if it is to be run locally. The remoteDispatch migrates the Task to the appropriate machine(s). It uses the ProgramLauncher for task migration. The Task and the SchedulingVector are passed to the ShortTermScheduler. The ShortTermScheduler then attempts to schedule the Task. Once a task is in the ShortTermScheduler, it waits for a local processor or processor set to run it. It will then invoke the schedulerMembers method on the Task. As Processors execute the remove method on the ShortTermScheduler they are collected and asked to run the processes from the Gang. Closer startup times are obtained between processes on different machines by a call to scheduleMembers. The code fragments for scheduling are shown below.

```
On local machine:                          On the remote machine:
-----------------                          ----------------------
1.dispatch                                 1.create remoteGang
2.findOptimal*                             2.dispatch
3.load program                             3.findOptimal*
4.if(local)ready,(optional added sync)     4.if (remoteGang not loaded)
else remoteDispatch                        load program
                                           5.if(local)ready,(optional added sync)
                                           else remoteDispatch
```

The methods updateSchedInfo, updateVMInfo and updateIdleTime are called by the short term

schedulers, Pager and the CPU classes to maintain the local resource usage information. Figure 4 depicts the remote interactions between classes on two separate machines. This arrows show in the diagram shows three types of communication: one for process migration, one for resource information dissemination and the last for fine grained synchronization for initial process execution. The arrows show the direction in which messages are sent.

Once a Task enters the short term scheduler, it stays there until it is suspended or it runs to completion. When a Task terminates, the data structures scheduling and processor scheduling are freed.

11. Implementation Performance

In this section, we discuss the performance of a LocalGang and SyncDistributedGang run on dedicated processors and Nodes. The SyncDistributedGang requires the use of the RemoteExclusive LongTermScheduler. This scheduling framework has been implemented on the *Choices* port to a network of SPARCStationII's and a network of Encore multimax 350s. Where application specific data are important, results were obtained for a parallel version of the FFT application. The experiments were conducted when the network was lightly loaded. The results represent the basic costs of the scheduling framework. There are two parts to the performance model for scheduling gangs. The first part entails gang membership management and its performance is shown in Table 5. Row 1 of Table 5, shows how the time to create a gang and add members to it on the Multimax varies slightly as the number of members increases. The cost to add members to the gang increases linearly and is the cost to access sequentially the gang member list structure. Row 2 shows the same numbers for the SPARCStationII. The much faster numbers for the SPARCStationII are a result of its faster processor and uni-threaded memory allocator.

The second part entails the time spent in the various classes in the scheduling framework. The time to schedule a gang is: $Scheduling\ Time = (HOPS + 1) * (T_{LTS} + T_{RM}) + HOPS * T_{PL}) + T_{STS}$

HOPS is the number of times the process is moved between nodes. The subscript LTS is for LongTermScheduler, RM for ResourceMonitor, PL for ProgramLauncher and STS for short term scheduler. The model is abstract and specializations can be used to describe the performance of various implementations and alternatives.

Gang scheduling overhead in ms				
Number of gang members	n =2	n=4	n=6	n=8
SPARCStationII				
Create gang and add to list	2.5	2.6	2.6	2.8
Encore Multimax 350				
Create gang and add to list	78	85	90	98
1.Schedule gang & collect cpus	4.0	4.2	4.4	5.0
2.(a)Ready to running(min)	3.0	3.1	3.8	4.3
2.(b)Ready to running(max)	3.4	3.5	4.6	5.0

Figure 5. Gang scheduling parameters

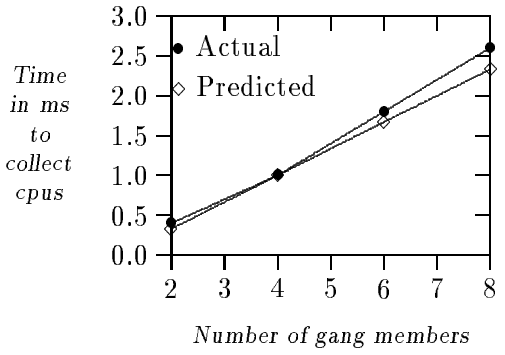

Figure 6. Processor collection

11.1. Multiprocessor Gang Scheduling: Performance results

T_{STS} is the most important parameter for scheduling a LocalGang. The dispatcher will not schedule the LocalGang unless a multiprocessor is found that will accommodate the LocalGang. Processors are self-scheduling and look for processes to run from their ready queues. These processors are 'collected' by changing the pointer to the system ready queue to a special FIFO-GangScheduler. When the requisite number has been collected, the gang processes are added to the FIFOGangScheduler and are run on the collected processors one by one.

The values of the parameters are: $T_{LTS} = 876$ μs, $T_{PL} = 0$ μs (no process relocation) and $T_{RM} = 120\mu$s. The two parameters of interest for short term scheduling are: (1)the time to schedule a gang collect N processors and (2) the time to dispatch N processes on N processors.

Figure 5 presents the values for the above parameters in the absence of any multiprogrammed workload and shows there is fixed overhead for scheduling the gang that does not change with the number of gangs members and a variable component for collecting the N processors for the gang which increases linearly with the number of gang members. The time taken to run processes that are ready is larger than the corresponding uniprocessor dispatch time (2.0 ms). The performance is similar to that of Mach[1].

The performance of collecting a gang of processes in a timeshared, multiprogrammed environment is highly dependent upon the multiprogramming workload and timeslice, unless preemption is permitted. The theoretical time to collect a gang of size p, with N processors in the system and a time slice time of TS seconds is given by the functions $First$ (time to collect the first processor) and $Collect$ (time to collect remainder). $First$, is $TS/(2 \times N)$ if the timeslicing interrupts of the processes are distributed uniformly and independently between 0-TS seconds. The first processor runs the process that collects the processors, reducing the problem to collecting $p-1$ out of $N-1$ processors. Each of the $p-1$ processors needed for the collection can be chosen out of the next $p-1$ processors that are time-sliced. Assuming the remaining $N-1$ processor time-slice interrupts are independent of one another, each collected processor contributes a waiting time of $TS/(2 \times (N-1))$ to the collection time. Therefore, $Collect(TS, N, p) = (TS \times (p-1))/(2 \times (N-1))$. Figure 6 shows the expected and the measured times for collecting processes as a function of p, with a timeslice of 10 ms, averaged over several experiments. $Collect(TS, N, p)$ predicts the measurements made fairly accurately for small numbers of processors. Each time-slice interrupt incurs a slight overhead that is not accounted for by the estimated collection time. As the number of processes increases, the sum of these overheads increase linearly. These numbers are greater than those reported in Figure 5, because the measurements were taken in a multiprogrammed environment.

11.2. Distributed Gang Scheduling: Performance results

When a new SyncDistributedGang is created a corresponding ContainerGroup is allocated. This ContainerGroup is used for tightly coupled synchronization. We will refer to the Node where the SyncDistributedGang is originally created as the originating node. The originating node becomes a centralized scheduler. The ResourceMonitor uses either the query or the publish algorithm to set up the group of processors. Once the group of Nodes is determined, the N processes of the group are relocated to the new machines. The call to scheduleMembers will perform added synchronization before the processes of the gang are executed.

The values of the parameters of the performance model are $T_{STS} = 1.40$ for ms, $T_{LTS} = 140$ μs, $T_{PL} = 500$ μs (this result is application size dependent). The size of the file is 98304 bytes. The table below gives the time for T_{RM}. The cost of the QueryResourceMonitor and PublishResourceMonitor algorithms are shown in Table 4. The results show a tradeoff between using the most recent results and cached information. The cost of the querying mechanism increases as

the number of machines is increased. The publishing algorithm scales better although the effects of updating the cache are not evident from our experiments. The costs of the query mechanism are dominated by the costs of collecting machines across the network, which increase with the number of machines.

Performance of query and publish (ms)			
Number of members	1	2	4
QueryResourceMonitor			
1.MultiRPC	2.45	2.9	4.7
2.Multicast+Bookkeeping	1.0	1.0	1.0
3.Total	3.45	3.9	5.6
PublishResourceMonitor			
1.Search Local Cache	.1	.1	.1
2.Multicast + Bookkeeping	1.0	1.0	1.0
3.Total	1.1	1.1	1.1

Table 4
The cost of Query and Publishing

FFT execution times ms		
Scheduling	gang	uncoordinated
Loosely coupled(SPARCStationII)	72	101
Tightly coupled(Encore Multimax)	1,223	2,136

Table 5
FFT with and without gang scheduling

12. Application Performance

Table 5 clearly demonstrates the tremendous benefit of gang scheduling for a parallel version of the FFT Application on the two types of systems. The scheduling parameters of the last section were used in these experiments. The results are for 4 processors, with a FFT data set size of 1024.

13. Conclusions

We have presented the design and preliminary performance results for a framework for parallel scheduling on tightly coupled and distributed systems. This framework has allowed us to implement a variety of scheduling algorithms and gang scheduling mechanisms rapidly by reusing code and interfaces extensively. We have also been able to compare their performance in a uniform manner. We have discussed some of the algorithms involved and discussed their performance and how they scale.

For some applications and a particular data set size the scheduling overhead may be high compared to the total execution time of the parallel application. As the data set size increases the scheduling costs form a smaller fraction of the total time (scheduling costs + application execution time). Similarly, as the number of nodes increase the cost of scheduling increases, but total execution time of a parallel application decreases. In the future, we will explore these relationships more fully.

REFERENCES

1. David Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *COMPUTER*, pages 35–43, 1990.
2. Roy Campbell, Nayeem Islam, Peter Madany, and David Raila. Experiences Building an Object-Oriented Operating System in C++. In *Communications of the ACM(to appear)*, 1993.
3. Roy H. Campbell and Nayeem Islam. Choices: A Parallel Object-Oriented Operating System. In Gul Agha, Akinori Yonezawa, and Peter Wegner, editors, *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.
4. Roy H. Campbell, Nayeem Islam, and Peter Madany. *Choices*, Frameworks and Refinement. *Computing Systems*, 5(3):217–257, 1992.
5. David Cheriton. The V Distributed System. *Communications of the ACM*, pages 314–334, 1988.
6. Kemal Efe and Margaret Schaar. Performance of Co-Scheduling on a Network of Workstations. In *Proceedings of 13th International Conference on Distributed Computing Systems*, 1993.

7. Dror Feitelson and Larry Rudolph. Distributed Hierarchical Control for Parallel Processing. *IEEE Computer*, 1990.
8. Dror Feitelson and Larry Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.
9. Nayeem Islam and Roy H. Campbell. "Design Considerations for Shared Memory Multiprocessor Message Systems". In *IEEE Transactions on Parallel and Distributed Systems*, pages 702–711, November 1992.
10. Nayeem Islam, Robert E. McGrath, and Roy Campbell. "Parallel Distributed Application Performance and Message Passing: A case study". In *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, September 1993.
11. Michael Litzkow, Miron Livny, and Matt Mutka. Condor: a Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111. IEEE, 1988.
12. M.J.Atallah, C. Lock, D.C. Marinescu, H.J. Seigel, and T.L. Casavant. Co-Scheduling Compute Intensive tasks on a Network of Workstations:Models and algorithms. In *Proceedings 11th International Conference on Distributed Computing Systems*, pages 344–352, 1991.
13. John K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Third International conference on Distributed Computing Systems*, pages 22–30, July 1982.
14. James L. Peterson and Abraham Silberschatz. *Operating System Concepts.* Addison-Wesley, Reading, Massachusetts, 1985.
15. Richard Rashid. Threads of a New System. *UNIX Review*, 1986.
16. Michael Stumm. The Design and Implementation of a Decentralized Scheduling Facility for a Workstation Cluster. In *Proceedings of the 2nd IEEE Conference on Computer Workstations*, pages 12–22, 1988.
17. Mark Sullivan and David Anderson. Marionette: a System for Parallel Distributed Programming. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 181–188. IEEE, 1989.
18. Marvin Theimer. *Preemptable Remote Execution Facilities for Loosely Cooupled Distributed Systems* . PhD thesis, Stanford, June 1986.