Scheduler Testbed System Design

Eitan Frachtenberg and Fabrizio Petrini
Parallel Architectures Team
CCS-3 Modeling, Algorithms, and Informatics Group
Computer and Computational Sciences (CCS) Division
Los Alamos National Laboratory

3 August, 2001
Technical Report

Contents

1 Preamble **4**

1.1 Objective 4

1.2 Audience 4

1.3 Document Organization 4

2 Overview of Schedulers **5**

2.1 Introduction to Gang-Scheduling 5

2.2 Dynamic and Implicit Coscheduling 7

2.3 Buffered Coscheduling 8

2.3.1 Overview 8

2.3.2 Scheduling 9

2.3.3 Example 10

2.4 Flexible Coscheduling 11

2.4.1 Overview 11

2.4.2 Process characterization 11

2.4.3 Scheduling 12

2.4.4 Characterization Heuristics 13

2.5 Local Scheduling 14

2.6 First-Come-First-Serve Scheduling 14

2.7 The Parpar Gang-Scheduler 14

3 Technical Description of RMS **16**

3.1 Overview 16

3.2 The Partition Manager (pmanager) 18

3.2.1 Resource Allocation 18

3.2.2 Communication With Node Daemons 22

3.3 The Node Daemon (rmsd) 23

3.3.1 Mode of Operation 23

3.3.2 Scheduling Mechanism 23
3.3.3 Communication With the RMS Host 23

4 Technical Design **25**
4.1 General 25
4.2 Architectural Overview 25
4.3 Scheduler Framework 26
4.3.1 The machine manager 27
4.3.2 The node manager 28
4.3.3 The program launcher 28
4.4 Implementation of Dynamic and Implicit Coscheduling 29
4.5 Implementation of Buffered Coscheduling 29
4.5.1 Local Scheduler 30
4.5.2 Communication Thread 30
4.6 Implementation of Flexible Coscheduling 32
4.6.1 Process Classification 32
4.6.2 Scheduling Implementation 33
4.7 Implementation of Local Scheduling 33
4.8 Implementation of FCFS Scheduling 33
4.9 Implementation of ParPar 34

5 Benchmarks **35**
5.1 LANL Applications 35
5.2 External Benchmarks 35
5.3 Synthetic Benchmark 36
5.4 Testing platform 36

6 Work Breakdown Structures (WBS) **37**

1 Preamble

1.1 Objective

The Parallel Architectures & Performance Team at the Los Alamos National Laboratory investigates the issues that contribute to optimal application and computer system performance on extreme-scale advanced architectures. As part of this effort, we are interested in studying the performance of job scheduling strategies for parallel systems, and in particular the performance of gang-scheduling methods, which show a great promise for enhancing system performance, utilization and even reliability for large scale systems. To this end, we intend to implement a testbed for several current (and hopefully future) gang scheduler algorithms, based on the RMS scheduler. The objective of this document is to offer a high level description and implementation design for the testbed we propose.

1.2 Audience

This document is primarily designated to the implementers of the testbed's modules. It should be reviewed by other members of the team, as well as technical people from QSW. We assume the reader of this document has some familiarity with the following subjects: job scheduling for parallel and distributed architectures, inter-process communication (IPC), communication concepts for parallel jobs, as well as a working knowledge of C++.

1.3 Document Organization

This document is organized as follows. Section 2 offers a brief introduction to gang schedulers in general, and to those we plan to implement in particular. Since the testbed is based on Quadrics' RMS to a large extent, we detail some of the technical aspects of RMS in Section 3. Next, we proceed to detail the technical design of the testbed in Section 4 . Finally, Section 5 discusses the benchmarks that will be used to evaluate the different schedulers.

2 Overview of Schedulers

This section describes the basic ideas behind gang-scheduling (GS), as well as details of each of the gang-scheduling algorithms we intend to implement: buffered coscheduling (BCS), flexible coscheduling (FCS), dynamic coscheduling (DCS), and implicit coscheduling (ICS). Also, several basic schedulers could be implemented for comparison, and these are described here as well. These algorithms are only detailed to the level required for understanding the rest of this document, and their description represents by no means a comprehensive or particularly exact one. The interested reader is referred to the original papers wherever applicable. Note: during the rest of this paper, the terms CPU (central processing unit) and PE (processing element) are used interchangeably

2.1 Introduction to Gang-Scheduling

Gang-scheduling [8, 15, 20] was introduced as an efficient way to multiprogram jobs on a parallel or distributed computer by *coscheduling* processes (running all processes of a parallel job concurrently). The basic idea behind gang-scheduling is that application can not only space-share the compute resources, but also time-share them, the same way that processes time-share in a uniprocessor machine running a multitasking operating system. Typically, a gang-scheduling system consists of a *master daemon* (which can be distributed) and *node daemons* that run on compute nodes (whether SMPs or uniprocessors). The master daemon allocates space resources for arriving jobs, and changes the active resource allocations (a "multi-context-switch") at a regular interval called timeslice quantum, or simply timeslice. Thus, a resource allocation decision only impacts the scheduling slot to which it pertains; other slots are available to handle other jobs and future arrivals.

Another key concept of gang-scheduling is the allocation-matrix or Ousterhout-matrix. An Ousterhout-matrix is a representation of resource allocation of space and time. Figure 2.1 shows an example of an Ousterhout-matrix for an eight-node uniprocessor cluster. The diagram shows the resource allocation to seven jobs on an eight-node machine during

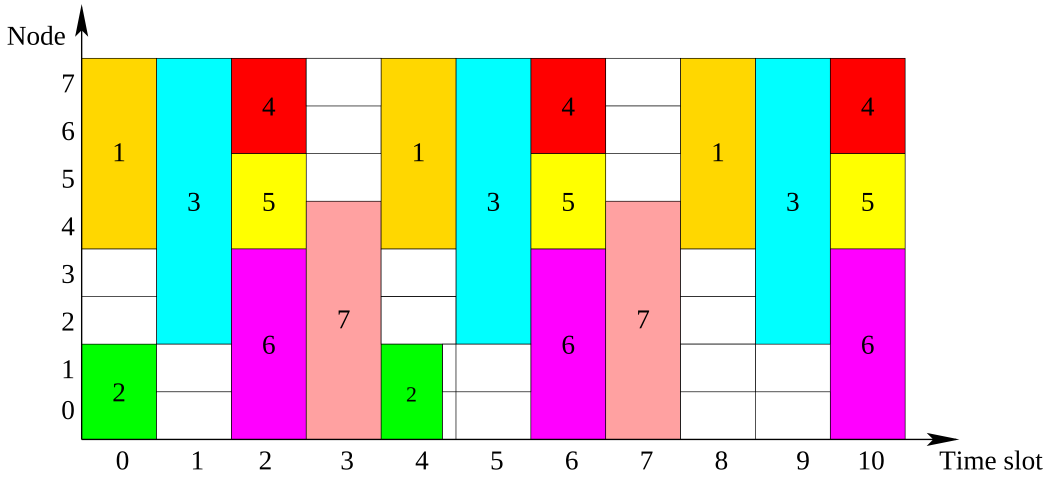

Figure 2.1: An Ousterhout-matrix example

eleven time slots, with a cycle of four time slots. In slot 0 we see two jobs, using four and two nodes respectively, while nodes 2 and 3 remain idle. On slot 2, the resource space is fully shared between three jobs, and no nodes remain idle. From slot 4 and up we see a repetition of the allocation pattern, except that job 2 terminates on slot 4, before the context-switch, and does not run any more. It is relatively simple to increase the utilization of the system using *alternate scheduling*, where jobs can "spill over" the next time slot(s) if some or all of the resources they use are idle in those time slots.

Figure 2.2 shows the same allocation matrix using alternate scheduling. If the jobs have a fine communication granularity, alternate scheduling is most effective if all the processes of a job continue running, as is the case for jobs 2 and 4 in this example. Other jobs, could have synchronization problems because not all their processes are running (e.g. jobs 5 and 7). It is more common to find schedulers that would use partial alternate scheduling to fit the request that only jobs that can have all their CPUs allocated will continue to run on other timeslots.

Furthermore, it might scale poorly if the master daemon becomes a bottleneck. Gang-scheduling is sometimes referred to as *Explicit coscheduling*, since the processes that are required to be coscheduled are explicitly known to the system from the moment they are launched as a single job. In contrast, The schedulers we describe in sections 2.2 use *Implicit coscheduling*, where the communication pattern of processes affects which processes are coscheduled.

Gang-scheduling was shown to significantly increase system utilization, improve system responsiveness and adapt well to varying workloads [8, 15]. On the other hand, gang-

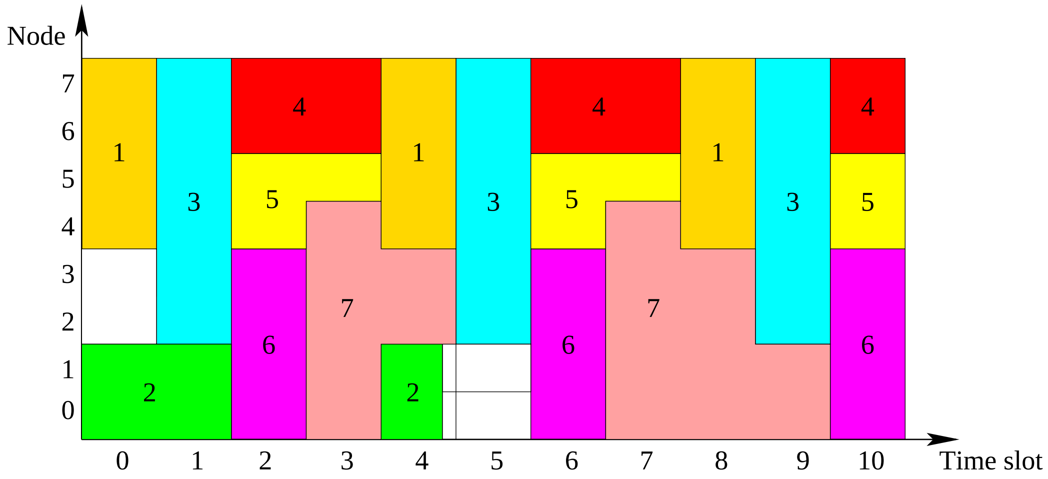

Figure 2.2: An Ousterhout-matrix example with alternate scheduling

scheduling introduces system overhead, because context-switching incurs cache penalties, working set changes, and possibly communication buffer flushing. It is also not very well suited to synchronization problems within a parallel job, thus allocation resources to processes that are blocked, waiting for their peers. To address some of these problems, several new techniques have been proposed. Sections 2.2-2.4 describe those methods we intend to use for out testbed. We also intend to compare these scheduling methods with some simple schedulers, which are described in sections 2.5-2.7.

2.2 Dynamic and Implicit Coscheduling

Dynamic coscheduling (DCS) was proposed as a way to reduce the amount of global synchronization of coscheduling, and thus decrease the system's overhead and improve its scalability [19, 26, 27]. The philosophy behind DCS is demand-based coscheduling:

- coordination is achieved by observing the communication between processes, and not by a master daemon,
- Communication between processes is used to deduce which processes should be coscheduled and to effect coscheduling. Thus, when a processes receives an incoming message it immediately receives a priority boost. This can effectively cause an immediate local context-switch to this process, depending on fairness policy and system load, and
- Processes are otherwise scheduled normally by the operating system.

The reason this method results in robust coscheduling is its underlying assumption: if a process receives an incoming message, its peer(s) must be currently running on other nodes, so the process should be prioritized for immediate execution.

Several aspects and parameters can be tuned for different assumptions or workloads, e.g. when to make a preemption decision [26]. The design of the testbed should allow room for configuration and tuning of these parameters.

Like DCS, Implicit coscheduling (ICS) makes local scheduling decisions based on monitoring communication activity [5, 6]. Processes waiting for a communication action to complete use a spin-block mechanism to relinquish control of the CPU, where first the process spins (actively waits) some time for the communication to complete, and then blocks, which consequently causes a context-switch if another process is ready to run. When a communication activity of a blocked message completes, it receives a priority boost, much in the same way it would in DCS. The main difference is that DCS explicitly treats every incoming message (not just those for blocked processes) as a demand for coscheduling, causing an immediate scheduling of the receiving process as soon as it would be fair to do so. Also, in DCS a waiting process always spins and does not block (it could be preempted though, immediately when another process receives an incoming message). In [26] it is claimed that while ICS is well-suited to "bulk-synchronous" applications (those that perform regular barriers, possibly with other communication taking place in between barriers), DCS is more suited for less-regular applications.

A comparative study of DCS, ICS, and other variations on implicit coscheduling techniques is presented in [4]. These techniques include different actions for waiting processes, as well as for those receiving messages. In some cases, the author finds that simpler coscheduling mechanism can outperform ICS and DCS, depending on the OS abilities. The design proposed in this document should be flexible enough to allow the implementation of some of these variations, should we want to measure their performance as well. On the other hand, this paper and others also study the effect of the local UNIX scheduler on implicit gang-schedulers. We do not intend at this stage to modify the OS kernel, so the comparison between different local-scheduling policies will be confined to the operating systems available for our testing.

2.3 Buffered Coscheduling

2.3.1 Overview

Buffered coscheduling (BCS) represents a different approach to coscheduling [10, 11, 12, 13, 21]. In BCS, local scheduling decisions are based on global information of the system's status, essentially converting an on-line problem of coordinating jobs to an offline problem. Global knowledge of the system's status can potentially be costly and unscalable, so BCS offers a technique for seamless integration of information exchange with the scheduler. Although BCS has the potential to address many system-wide performance and reliability issues, this document is mainly concerned with the parts relevant to this design, i.e. the jobs scheduling subsystem.

2.3.2 Scheduling

In the BCS model, each timeslot has two phases, computation and communication. During the computation phase the current job runs normally, except that all the outgoing communication is buffered for later execution. If the communication is of a blocking type, the process is preempted and another process is chosen and scheduled (in a way that will be explained shortly). The communication phase is divided into three parts: first, an exchange of information occurs between the nodes, after which every node has global knowledge of the information that pertains to it pending incoming- and outgoing-communication, and possibly other information such as load, availability and status of other nodes. During the second part of the phase, the communication between the nodes is scheduled for optimal use of the network without exceeding the phase's time limit. Lastly, the messages are sent and received according to the schedule. It should be noted that the availability of advanced network hardware can enable a communication phase that is so fast, that several of those can fit in a timeslot. In that case, the model can change to accommodate communication phases that service the currently running computation phase, thus reducing the need for a premature context-switch.

To use the machine effectively, the computation and communication phases are overlapped, so that the communication phase for computation phase N runs concurrently with computation phase $N+1$. This requires that network interface card (NIC) have its own processing capabilities, as is the case with the Elan NIC. Obviously, this only makes sense when there are more than one job running, which is the typical case for supercomputers. When only one job is running on a node, two strategies are available:

1. Continue with the BCS mechanism, sacrificing some idle time on the CPU while it is waiting for blocking communication to go through the communication phase

2. Work in a dedicated mode, where messages are not buffered or scheduled, and no exchange of information is done by the NICs.

This issue requires further investigation to determine the feasibility of each alternative.

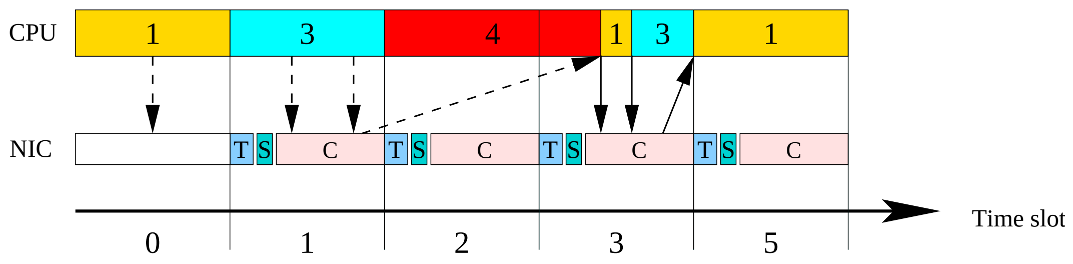

Figure 2.3: Buffered Coscheduling example. Dashed arrows represent non blocking communication messages; regular arrows represent blocking messages. The communication phase in the NIC is divided in three parts: (T)otal exchange, (S)cheduling and (C)ommunication.

2.3.3 Example

Figure 2.3 shows an example of buffered coscheduling on the first five timeslots of node 7 in the Ousterhout matrix shown in Figure 2.2. On the first time slot, the CPU of the node runs job number 1's process, while the NIC is relatively idle, since there were no previous computation phases. The only thing it does is to receive a non-blocking message from the process, and buffer it for sending in the next communication phase. In time slot 1, Job number 3 is running in the CPU, sending (and buffering) two non-blocking messages, while the NIC is going through the three steps of the communication phase: total exchange of information with other nodes (which is not necessarily implemented as a total exchange), scheduling communication (in this case, job 1 incoming and outgoing messages), and executes the communication. We can see that a non-blocking message for job 1 has been received, and will be waiting for it when it is next scheduled in time slot 3. During the next time slot job number 4 is running without communicating while the NIC processes job 3's communications. Job number 4 continues into the next time slot due to the alternate scheduling, but does not complete it: it sends a blocking message which causes its preemption the the next job in the queue, job 1. This job too performs a blocking receive operation before completing the time slot, and is descheduled. Job three then uses the rest of the time slots until a global context-switch causes its preemption. During this time slots the NIC receives the message job 1 expects, so it can be re-launched in time slot 5, as determined by the Ousterhout-matrix.

2.4 Flexible Coscheduling

2.4.1 Overview

Flexible Coscheduling (FCS) is currently being developed as a research project at Hebrew University by E. Frachtenberg, under the supervision of Dr. D. G. Feitelson. FCS augments gang-scheduling with dynamic process classification and local scheduling to better handle problems arising from system heterogeneity and load imbalances. Applications with fine-grained communication or synchronization suffer most from these conditions, since some processes are often found blocked, waiting for their peers which could either run on a slower platform or have a higher load then themselves. This in turn inflicts performance penalties on the system (lower utilization and higher overhead if applications are constantly preempted) and the running jobs (higher turnaround time and less responsiveness).

2.4.2 Process characterization

To address these problems FCS employs online process characterization and schedules processes using this information. Processes are categorized into one of these three classes:

1. CS (coscheduling): These processes require coscheduling with their peers, and are currently successfully coscheduled. We will see shortly how this success is measured.
2. F (frustrated): These processes require the synchronization gains obtained with coscheduling, but coscheduling them is unsuccessful. These are typically those processes that suffer for system heterogeneity of load imbalance, resulting for example from uneven decomposition of the data set).
3. DC (don't-care): These processes rarely synchronize, and can be scheduled in almost any possible way without penalizing the system's utilization. For example, a job using a coarse-grained workpile model would be categorized as DC.

Another class of process is the RE (rate-equivalent) category, for processes that have low synchronization requirements, but have coarse-grain load balancing, so they require the same amount of normalized CPU time, even if not necessarily in the same time slots. For all practical purposes, we characterize these processes as DC. Processes of the same job will typically, but not always, be of the same class. Some local traffic patterns can create subgroups of processes with their own synchronization patters. To allow for these cases, and to avoid global exchange of information, processes are categorized on a per-process basis, instead of a per-job process.

2.4.3 Scheduling

The principle behind scheduling in FCS is this: CS processes should be coscheduled and should not be preempted; F processes should be coscheduled but can be preempted when synchronization is not achieved; and lastly, DC processes make no requirements on scheduling. Like with BCS, the node daemon in FCS receives global context-switch messages from the master daemon, but also has an autonomy of its own. Algorithm 1 shows

Algorithm 1 Context switch algorithm for FCS

```
procedure context-switch (current-process, next-process)
begin
    if current_process == next_process then return
    switch depending on type of next_process
        if $CS$ then
            run next-process for entire time slot
        if $DC$ then
            let local OS scheduler schedule among all
            $DC$ processes for entire time slot, and if
            there are no $DC$ processes waiting to run,
            allow local scheduler to choose from $F$ and
            then $CS$ processes
        if $F$ then
            loop for entire time slot
                1) run next-process until it blocks for communication
                2) Do until communication unblocks for next-process:
                        let local OS scheduler schedule among all
                        $DC$ processes for entire time slot, and if
                        there are no $DC$ processes waiting to run,
                        allow local scheduler to choose from $F$ and
                        then $CS$ processes
end
```

the basic algorithm of the node daemon upon receipt of a context-switch message. The basic idea is to allow the local operating system freedom to schedule DC processes according to its usual criteria (fairness, I/O considerations, etc.), and also use DC processes as "Lego blocks" to fill in the gaps that F processes have because of their synchronization idiosyncrasies to gain better machine utilization. An F process that waits for pending communication should probably not block immediately, but rather spin for some time to avoid unnecessary context-switch penalties. The fact that FCS collects communication statistics for each process allows the scheduler to determine a competitive spinning time on a per-process basis. Two more principles differentiate this scheduling method from DCS and ICS: (1) An CS process cannot be preempted before the time slot expires even if an incoming message arrives for another process (CS processes have "proven" that it is not worthwhile to deschedule them in their time slot). (2) The local scheduler's freedom to choose among processes in the DC time slots and F gaps is affected by the communication characterization of processes, which could lead to less-blocking processes and higher utilization of resources.

2.4.4 Characterization Heuristics

FCS uses dynamic characterization of processes based on their communication behavior, to continually adapt processes' classification and associated parameters over time. The local scheduler monitors four parameters for every process:

1. C_{CS} - The count of communication attempts (in and out) the process performed while being coscheduled (i.e. scheduled as either CS or F).
2. C_{DC} - The count of communication attempts (in and out) the process performed while **not** being coscheduled (i.e. scheduled as DC).
3. W_{CS} - The total amount of time spent waiting for communication attempts to complete while being coscheduled.
4. W_{DC} - The total amount of time spent waiting for communication attempts to complete while **not** being coscheduled.

Initially, when a process is launched, it is tagged as CS, and is scheduled as such for a fixed number of time slots, while incrementing the C_{CS} and W_{CS} counters. Then, the process is tagged as DC for another fixed number of time slots while incrementing the C_{DC} and W_{DC} counters. During this initial information-gathering step some short processes might even terminate, which is why we first schedule them as CS: This way short jobs benefit from the benefits of coscheduling and thus possibly finish early. After this period and for the entire lifetime of the process, the counters are updated and the process scheduling class is determined using these principles:

◆ If the process rarely communicates (both C_{CS} and C_{DC} are lower than a given threshold), the process is either DC or RE and is classified as DC;

◆ Otherwise, If the process has relatively long periods of waiting for communication to complete, it is of type F;

◆ Otherwise, If the process communicates significantly better while being coscheduled than not (as determined by the ratio C_{CS}/C_{DC}), the process is of type CS;

	Short average wait	Long average wait
Low $\frac{C_{CS}}{C_{DC}}$ ratio	DC	F
High $\frac{C_{CS}}{C_{DC}}$ ratio	CS	F

Table 2.1: FCS Scheduling classification using communication information. Processes that rarely communicate are automatically classified as DC.

- Otherwise, the process is of type DC.

These principles are summarized in Table 2.1. The thresholds to determine what “low” and “high” values are will be determined empirically.

2.5 Local Scheduling

Local, or uncoordinated scheduling is the most straight-forward approach to time-sharing on a multi-computer, where no global control of resources is used. A central management system may be used to launch jobs to nodes, but once launched, each node schedules its processes by itself, typically using the UNIX kernel's sched() function. Although its simple and scalable, the attractiveness of this method is severely diminished when considering the performance penalty it entails for parallel applications [5, 6]. We use local scheduling as a base case to compare the performance of the different coscheduling methods

2.6 First-Come-First-Serve Scheduling

First-Come-First-Served (FCFS) Scheduling can be considered as the opposite of local scheduling: there is no time sharing among jobs, and only global resource allocations are allowed. When a new job arrives, it is immediately launches its demand meets the available resources, or queued until such resources become available.

2.7 The Parpar Gang-Scheduler

Parpar is a software-based gang-scheduler for BSD and Linux systems. It was designed to work well with short time slice ($\approx$ 1 sec.) and scale up to 256 nodes. Its basic architecture is similar to that of RMS (which is described in the next section), consisting of a master daemon that controls resources and node daemons that attach jobs to resources in a proper context, with several implementation differences. For example, Parpar can use several efficient communication mechanism, like IP multicast and reliable datagrams to broadcast synchronization messages (this low-latency low-overhead approach enables the small timeslice values). Furthermore, Parpar can use different resource allocation algorithms that are linked from an external library with a simple interface. Two such algorithms were already implemented for it. Parpar's modular design could possibly offer simple integration of some of its components in RMS to allow comparison of implementation choices between the two schedulers, or as a shortcut to implementing features in RMS. For a more detailed view of Parpar, the reader is referred to [1, 9, 14].

3 Technical Description of RMS

3.1 Overview

In this section we describe various technical aspects of RMS, and in particular those that are relevant to job scheduling. We do not intend to use RMS for our prototype implementation except for the initial launching of processes. Therefore, it is not required to read this section to understand the technical design. In the future, as RMS integrates more features that allow it to be extended with external schedulers, this section may become more relevant.

An RMS cluster consists of management node(s) and compute nodes. Compute nodes can be divided into mutually exclusive *partitions* so that each partition can have different properties and policies for resource allocation, and several *configurations* can be defined and switched to allow a different set of properties per partition (e.g., different configurations can exist for day and night operations, allowing larger programs to run at night). At least one node (which can be separate from the compute nodes) is designated as a management node and holds the RMS database, which enables interfacing to the system using standard SQL queries.

The RMS provides a single point of interface to the system for resource management. It includes facilities for gathering information on resources (monitoring, auditing, accounting, fault diagnosis, and statistical data collection) and for resource handling (CPU allocation, access control, parallel jobs support, and execution, and scheduling). RMS is implemented as a set of user-level UNIX commands and daemons that communicate using socket daemons and access the database for storing or retrieving all the system details. Some of the daemons also communicate with the UNIX kernel through the augmentation of the kernel with a few system-calls. Of the set of daemons provided by RMS, two are concerned primarily with parallel job launching and scheduling. The *Partition Manager* (`pmanager`) is a per-partition daemon that runs on the management node. It handles requests for job launching and termination, checks the privileges and priorities

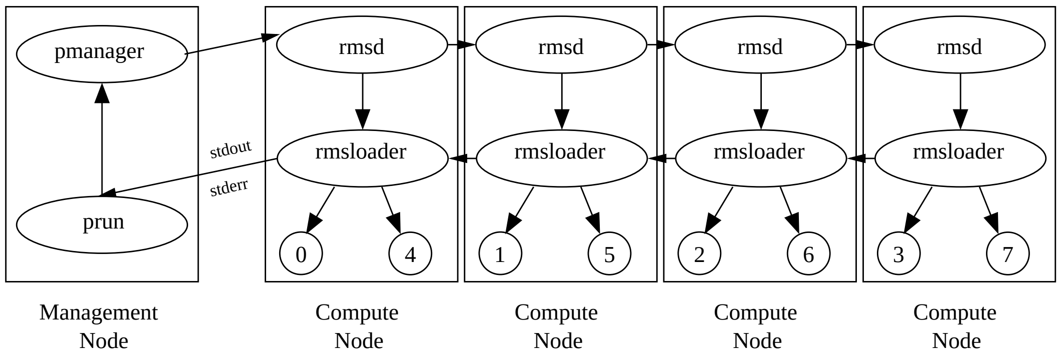

Figure 3.1: An eight-process program on four nodes

allowed for each job, manages and allocates resources within its partition, and schedules the jobs. The *RMS Daemon* (`rmsd`) runs on each computing node in the system. It loads and runs user processes (using the application loader `rmsloader`), creates communication contexts for the application, delivers signals, and monitors resource usage and system performance.

Figure 3.1 shows how the system runs an eight-process job on four two-way SMP nodes. First, a user invokes a program called `prun` on the management node to launch her program, which in turn asks pmanager to allocate PEs and start the job on them. The `pmanager` notifies the `rmsd` processes on the allocated nodes to invoke an `rmsloader` process with the user's program; `rmsloader` also directs the `stdout` and `stderr` streams of the program to `prun`, which forwards it to the controlling terminal or output files.

The RMS scheduler allocates *boxes* (N nodes with a fixed number of PEs per node) to jobs so that they may take advantage of the hardware support of the QsNet for broadcast and barrier operations that operate over a contiguous range of network addresses. More details on the process of resource allocation can be found in 3.2.

Each partition can have its own scheduling policy and parameters (such as timeslice interval, time limit, etc.) The scheduling algorithm used can be one of the following:

1. Gang-scheduling (with alternate scheduling.)

2. Regular (local) UNIX scheduling with the addition of simple load balancing.

3. FCFS scheduling.

The next two subsections continue the description of `pmanager` and `rmsd` to the level of detail required to understand the technical design in Section 4.

3.2 The Partition Manager (pmanager)

3.2.1 Resource Allocation

The partition manager holds information about the partition's CPU and memory resources and allocates them according to availability to scheduling policy. It should be noted that RMS does not hold a representation of an Ousterhout-matrix. Rather, it holds a list of all resources/jobs, whether allocated or not, and a list of the resources allocated for the current timeslice only. Gang-scheduling and context switching is obtained by way of changing jobs' priorities, as will be explained below. We should start by describing the basic data structures (C++ classes) that `pmanager` uses (in bottom-up order):

- *Box*: A Box is a rectangular array of CPUs, where one dimension represents nodes and another represents CPUs. For example, eight four-way SMPs can be represented by an 8x4 Box with a total area (total number of CPUs) of 32.
- *BoxPacker*: A BoxPacker is a collection of Box instances, not necessarily of the same dimensions. Examples for BoxPacker usage are representation of the entire cluster and representation of used/free CPUs per a timeslice. Together, Box and BoxPacker offer services (methods) to allocate/deallocate CPU resources, fit resources to requests and perform geometrical queries.
- *Resource*: A Resource hold all the information on resources allocated to a job, and can be operated on in various ways. A resource contains (among other things) a list of *Segments*, where each Segment contains a range of allocated nodes. A Resource's methods include all the set/get operations on the job's run-time parameters (priority, time limit, memory limit, nodes in use, etc.) and status (running, queued, blocked, etc.).
- *Job*: A Job class is a representation of a parallel application, and is similar to a Resource (a Job contains a pointer to its allocated Resource). Note however that a job does not have to have the same number of processes as CPUs in the Resource (could be more or less). A job also contains scheduling-specific information, like which partition it belongs to and its CPU allocation policy.
- *Partition*:This class holds partition-management information and methods, including methods to report and update the partition's status (e.g add nodes to it), and holds machine and node information. This class is not very relevant to scheduling issues, but it does contain information on the scheduler's features (e.g whether it support gang-scheduling).

- *Scheduler*: This is the main class for the scheduling aspects of the partition manger. It contains methods for initiating a job and allocating resources to it, as well as context-switching between jobs or preempting low-priority jobs as necessary. The important processes of the Scheduler class are outlined below.

`Pmanager` loops indefinitely, waiting for messages from `prun` or the node daemons, or for periodic timeslice wakeup to occur, and handles each of these events in a callback function. When the `pmanager` receives a request to run a new job from `prun`, it undergoes an allocation process which is shown in Algorithm 2. Note that when a job is marked

Algorithm 2 Allocation of a new job in `pmanager`

```
1) Verify all the following, or return error to prun:
       Job has permissions to run
       Partition allows parallel jobs or user is root
       Partition is active or job is allowed to block till it does
       Partition's job queue length does not exceed limit
       The required amount of rails (if any) is available
       The resources requirement is legal (e.g. does not exceed max. CPUs)
2) Create a new resource for job
3) Generate Elan context for job
4) If allocating CPUs to job will exceed CPU usage limit and user != root then
       mark job/resource as "blocked", or fail if request is immediate
   else
       mark job as "queued"
5) Make initial allocated of CPUs to job/resource
6) Schedule resources using Scheduler::scheduleResources()
```

as "blocked", it will not change status until more resources become available (e.g. when another job terminates). The check for available resources is made whenever a scheduling event happens, as we will see shortly. Initial allocation of CPUs per resource (step 5) uses the BoxPacker and Box classes to try to find a continuous range of nodes to allocate, if possible, to allow exploitation of Elan's hardware broadcast. If unavailable, Allocation of disjoint node boxes will be given. If that too fails, and the partition supports gang-scheduling, the system may try to allocate CPUs to the resource in a later timeslot, or create a new one if necessary (this happens transparently as part of the scheduling process).

Whenever a scheduling event occurs (e.g. expiration of a timeslot, arrival of a new job or termination of an old one, suspending and resuming of jobs by the user), the Scheduler's scheduleResources() method is called (Algorithm 3). We note the following properties of this algorithm:

Algorithm 3 Scheduler::schedulerResources()'s algorithm

```
1) Delete the allocation of resources to current timeslice and restart it
2) Build array of runable resources (e.g. not externally suspended)
3) If timeslice just expired then
       increase timeslice count of queued and suspended resources
4) Sort resources according to the following criteria (in descending order):
       - prefer requests submitted by root
       - prefer the higher priority resource
       - prefer the resource that has been waiting longer
       - prefer jobs that are running or queued
       - prefer the job that was submitted first
5) Loop $r$ over sorted list of resources (higher priority first)
   begin loop:
       if CPUs were already initially allocated to $r$ then
           if $r$ is “blocked” (usage limit exceeded) then
               try to see if $r$ can be unblocked, otherwise continue loop
           try to allocate $r$'s CPUs, mark $r$ as “suspended” if already allocated
       else if $r$ will exceed usage limit then
           mark $r$ as “blocked”
       else if initial CPU allocation for $r$ is successful then
           allocate $r$'s CPUs
       else no CPUs are available for $r$ now, mark it as “queued”
   end loop
6) If any changes to the scheduling were made then
       instruct the rmsds to make the changes
```

- Every time a scheduling event happens, including timeslice expiration, the entire process of sorting resources and allocating them to CPUs. This implies relatively high overhead when the timeslice quantum decreases and/or the number of gang-scheduled resources increases. This in part can explain the poor scheduler performance that was observed in [17] for small values of timeslice quanta.

- Since the scheduler does not maintain an Ousterhout-matrix, gang-scheduling with RMS is almost an implicit process. The circular nature of timeslot scheduling is obtained by the prioritizing of resources. When all other things are equal, resources are sorted by the amount of timeslots they used, so a round-robin effect is achieved. Also, there is no fixing of the jobs that occupy the same timeslot, and in theory there is not guarantee that the scheduling will repeat itself. In practice though, when nothing else changes, when it comes to re-allocate jobs to the current timeslot, the deterministic allocator would repeat its decision from the previous time those jobs were allocated.

- Several events, (e.g. the addition or removal of jobs or change of priorities) can change the entire allocation of jobs to timeslots, since the new sorted list of resources can be modified enough to result in a completely different allocation. However, once CPUs are allocated to jobs, they are never moved.

- The default value for the CPU usage limit in an RMS partition is the available number of CPUs in the partition. Therefore, normal (non-root) user jobs of the same priority that together exceed this number will not actually be gang-scheduled: The first first will run to completion while the others are blocked, then the second, and so forth, till all the remaining jobs can be scheduled to run together. Therefore, having the partition gang-schedule this job would require raising the usage limit to some higher value or running them as root.

- A benefit of this algorithm is that we get partial alternate-scheduling for free: If a job that run on timeslot t can use the same CPUs in timeslot t+*1* (after all eligible jobs have been allocated), it will run on that timeslot. However, this will happen only if all the jobs CPU's can be allocated to it, and not just part.

Although this algorithm runs for every timeslice, note that the impact is minor if all runable jobs fit into a single timeslot. In that case, reallocating the jobs would probably not cause changes in the scheduling, and the node daemons will not be notified to make changes, so jobs continue to run uninterrupted. The next subsection describes how the

Field name	Description
cmd	A specific command from an - enumeration of all possible commands
rc	Return code
nob	Number of bytes to transfer
buf	A character buffer with command's data
ptr	A pointer into buf for packing/unpacking
timestamp	Command's timestamp

Table 3.1: The Cmd class

partition manager communicates with the node daemons when a change of scheduling is due.

3.2.2 Communication With Node Daemons

RMS has a generic data structure for communicating messages from `pmanager` to `rmsd` embodied in the Cmd class.

Table 3.1 shows the basic data members of the Cmd class. It also contains various methods for packing and unpacking messages, as well as setting/getting data members, returning error messages, etc. These commands are communicated from pmanager on a communication tree, implemented in the class CommsTree. CommsTree represent a quaternary tree, whose root typically represents the originator of messages and nodes represent the message receivers - in our case the pmanager and rmsds respectively. Messages (Cmd instances) can be propagated down the tree, and acknowledgements are propagated back up using the SocketServer class that manages it. The tree is dynamically maintained by pmanager to reflect changes to the partition (availability of more - or less - nodes). The SocketServer also handles some timeout/retry and flow-control issues, as well as initialization of a partition's tree. Every node in the tree, including root, registers callback functions for the various commands it may get, in the form of implementation of the virtual Cmd action data members. for example, rmsd implements actions for the commands to start and kill jobs in the form of CMD::rpc_start() and CMD::rpc_kill().

3.3 The Node Daemon (rmsd)

3.3.1 Mode of Operation

As described above, `rmsd`'s role in RMS is responding to requests from `pmanager`. It is first called from UNIX's `init` process, launched by a daemon called `rmshd`, which also takes care of re-running it, should it fail. After initializing the connection with the communication tree and the Elan libraries, it enters into into a passive loop, where callbacks are called for each of the registered actions it needs to take care of. These include management commands from `pmanager` (e.g. shutdown, request for statistics, etc.), scheduling commands from `pmanager` (e.g. suspend/resume job, etc.), and signal handlers (e.g. handling of SIGCHLD for processes that terminate and periodic housekeeping functions.) Another role of the node manager is to periodically save its state to a local disk file, so in case of a crash it can reproduce it and re-launch the jobs that were running on the node before the crash. However, applications should have their own checkpointing mechanism to benefit from this feature.

3.3.2 Scheduling Mechanism

When a new job is assigned to a node, `rmsd` receives a Cmd structure that contains, among other things, the new job's ID and its Resource ID. This in turn is used to create a *LoaderDesc* class and an *RDesc* class: the former is used to launch the program and store process information in it, and the latter as a resource descriptor. After the environment for the new process is set (Elan capabilities, environment variables, signal handling, etc.) is set, `rmsd` fork()s and executes `rmsloader`, that in turn executes the program. A destructor method also exists in these classes to stop (kill) a running job.

A scheduling command arriving from `pmanager` is handled in a simple manner. The rpc_schedule() callback parses all the scheduling requests from the Cmd structure, including the Resource IDs. It finds the appropriate RDesc class for each resource, and lets its RDesc::schedule method handle the request by calling the appropriate system calls.

Both types of activities use access to the UNIX/Linux kernel by way of RMS system calls the were added to the kernel. Table 3.2 shows the relevant RMS system calls.

3.3.3 Communication With the RMS Host

There are three ways in which `rmsds` convey messages to the RMS host (either the `pmanager` or to update the database):

System Call Interface	Description of Function
`int rms_prgcreate (int id, uid_t uid, int cpus);`	Creates a new program description. *id* is a job identifier - invariant across nodes *uid* is the user identifier that owns this program *cpus* is the local number of CPUs for this program
`int rms_prgdestroy (int id);`	Destroys an existing program description.
`int rms_prgsuspend (int id);`	Suspends all the processes of a running program.
`int rms_prgresume (int id);`	Suspends all the processes of a running program.
`int rms_prgsignal (int id, int signo);`	Sends a signal to all processes of a program. *signo* is the signal number

Table 3.2: RMS system calls for scheduling

1. Replying to commands: As described in 3.2.2, the Cmd class allows for a return code to be filled with a command's reply, with either a success or an error code. Almost all commands cause the return of this code. Furthermore, a command callback in `rmsd` can fill the Cmd buffer with its own return data and send it back in the reply. For example, the request-for-statistics callback returns the required statistics to `pmanager` this way.

2. Sending Cmds to `pmanager`. There is a separate thread that runs in rmsd, in an object called NodeStats, that periodically sends statistical information to pmanager.

3. The last method of communication is through the *Event* class. This class is mainly used to log errors and messages in the RMS database. The Event class itself is an encapsulation of the Cmd class, and has similar abilities, but is used solely to post events in the database or Pandora.

4 Technical Design

4.1 General

Since our proposed testbed has several nonstandard requirements, we will not be able to base our schedulers on the existing RMS infrastructure without modifying it extensively. Furthermore, since we may expand our testbed requirements in future research, and need a degree of flexibility from the scheduler that RMS was not designed to offer. We therefore intend to implement a rudimentary scheduler framework ourselves, with as little intervention with RMS as possible. The next section describes the general architecture of this framework. Sections 4.4-4.9 cover the implementation details of each of the proposed schedulers. In cases where two implementation alternative exists and none has a clear advantage over the others, both alternatives are presented. Note: some parts of the design rely on process priorities. Whenever compare priorities, we refer to them in the intuitive way rather than the UNIX way, i.e. a process is preferred if it has a higher priority (in UNIX, a negative priority is preferred to a positive priority).

4.2 Architectural Overview

We would like our framework to have the ability to run independent jobs in a normal manner, but retain control of their communication behavior in a way that is transparent to the applications but visible to the scheduler. Our main limiting factor in the existing system software is that RMS was not designed to allow one process control of another process' communication. The main idea behind our testbed is therefore to 'cheat' RMS into thinking all the jobs in the system are actually one application, and these exchange information through the use of helper threads running on the NICs. The 'single job' actually contains independent application jobs, as well as two types of schedulers, as will be explained below.

We try to follow these guidelines in our design

- User-transparency: Applications need not be changed, and regular users should not need to change their mode of work. Applications will only need to be re-linked if they statistically link with Quadrics communication libraries. The only difference will be in the application launching mechanism.

- Portability: The testbed should be portable to any machine that supports Quadrics, independent of operating system. This would enable us to make the tests in different environment, but also poses a limit on the changes that can be made: The OS kernel and user applications should remain untouched.

- Extendability: Since this is a research project, we may come up with new scheduling mechanisms over time. The system should use simple, clean interfaces and export as much of the scheduling mechanism as possible, to allow easier future extensions.

- Minimal intervention: For reasons of portability and ease of implementation and maintenance, it is best that we do not modify RMS or any other of the existing system software.

- Performance: The testbed should incur as little overhead as possible and strive for best scheduler performance so that the measurements can produce meaningful results.

4.3 Scheduler Framework

Our system consists of a single wrapper application that will run different executables, all having the same Quadrics capability, allowing them to communicate among themselves. This "application" can actually be a simple shell script run with `prun`, that in turn executes one of three programs, depending on its RMS_NPROC and RMS_RANK environemt variables. These three types of executables consist of a machine management (MM) process, node managament processes (NM) and program-launcher processes (PL). Generally, the MM is in charge of the initial allocation of resources to programs and the coordination of the NMs; the NM are responsible for local scheduling on each node and communicate with the MM and the PLs; the PLs' only function is to fork() and execute new applications, and report back when these terminate. These processes communicate with each other through helper threads that each run on the Quadrics NIC. This way, the communication incurs almost no penalty to the compute processes, and can benefit from Quadrics' network advatanges, like fast collectives and communication that requires no intervention from the main CPUs. The functions of each process and their help threads is detailed in the following subsections.

4.3.1 The machine manager

The machine manager has two roles: the dispatching of new jobs and initial allocation of resources to them, and the coordination of node manager through heartbeat messages. We intend to feed the workload of jobs to the MM using a workload file, with a simple format that describes, in each line, the following paramaters:

1. Time to run the application (in some predefined units).
2. Number of PEs the application requires.
3. Command line of application to run, including parameters for the application

The MM reads this file, which is sorted by application start time (Item 1), one line at a time, and sleep()s until the time the application should be launched or a heartbeat message should be sent. It would then wake up to perform the communication, and possible to read the next line of the workload file, before going back to sleep. It can also be awaken by a message from one of the NMs, announcing the termination of a process.

For initial allocation of resources, we intend to use the buddy-tree allocation algorithm [20]. The ParPar scheduler has a software module that can be use for this puprose with relative ease. It's generic design allows for plugging into different schedulers and use various allocation schemes. It is important though that we keep its internal data structures updated by forwarding to it information on the status of the nodes and jobs. This will be done by propagating the appropriate messages (e.g. process termination) from the NM to the MM, which will then call the module's appropriate callback function.

Communication with the node manager is performed through helper threads that run in the Elan NIC. These threads can broadscast information to the NMs (e.g. new jobs or heartbeat message), and receive messages from the NMs (e.g. process termination notification).

At this stage of the testbed design, we do not address the important issue of fault-tolerance. One of the issues that we do not address right now is that the MM provides a single point of failure for the machine, and if this process fails for whatever reason, the entire testbed becomes unusable. In the future, we may solve this by initally allocating some spare process that can become the MM should the original one fail. These can even run on other nodes, but we must implement a means to transfer the MM's internal information to it by regular checkpoints.

4.3.2 The node manager

Node managers are responsible for the launching and scheduling of processes on each node. For simple gang-scheduling, the role of the NMs is limited to excecuting commands issued by the MM: launching and preempting jobs. With the other scheduling schemes, where local scheduling is made based on locally-collected information, the NMs will perform the information collection and the local scheduling decisions.

The NMs will hold a representation, through messages from the MM, of the Ousterhout-matrix information pertaining to their node. This will allow them to know which processes to run on a context-switch, and also to make more informed local-scheduling decisions. For some of the scheduling schemes, the NM also requires information about communication among the processes. Information regarding local processes will be gathered by augmenting the ADI level of the MPI library to inform the NM of relevant communication events. A small library using BSD message queues was implemented for this purpose. For information regarding remote processes, the NM will use its help thread in the NIC to exchange information with remote NMs at regular intervals. For more details on this mechanism, see 4.5.

For sake of implementation simplicity, we intend to run one NM per PE (as opposed to per node). This may make local scheduling slightly less efficient, but will make initial job allocation and the implementation of the Ousterhout-matrix mechanism much simpler. At a future point, we may reconsider this detail if siginificant performance improvements can be gained by having a single NN per SMP node.

4.3.3 The program launcher

The program launcher has a very simple role: execute commands from the NM to launch programs. Each PL is associated with one NM with which it communicates through help threads. The number of PLs determines in effect the maximum level of time sharing we allow in the system, since a program can only be launched if an available PL exists.

An available PL simply waits for an event from the NM, giving it the details of a program to run (command line, etc.), and then fork()s, and exec()s the program. It is important that the PL closes all the open Quadrics handlers it has before executing the user program, so that they become available to the program. Further, some changes are required to the MPI_Init() call of the MI library, to trick application to see a world of MPI processes that consists only of their peer processes, and not all the NMs, PLs, and running applications. This in turn requires that information about the program[1]

[1]Specifically, information on the program's row in the Ousterhout-matrix

be available to the MPI layer. This can be done by the NM posting this information in some shared-memory region where it can be read by all programs.

When the user process terminates, the PL, which has been blocked waiting for it, wakes up and re-establishes the connection with the NM. At this point, it will notify the NM of the process termination and becomes available for a new program again.

4.4 Implementation of Dynamic and Implicit Coscheduling

Dynamic and implicit coscheduling are for the most part priority-based scheduling schemes, where processes' priority is dynamically updated based on their communication behavior. The main role of the NM for these schedulers is therefore not to actually schedule and deschedule processes, but to modify the priorities and let the local UNIX scheduler do the rest. To that end, there is no point in heeding to the MM's heartbeat commands, and these can be ignored or discarded all-together (or not sent at all). Another important implementation detail of the NM for DCS is that whenever an incoming message arrives for a non-running process, its priority should be boosted to cause its execution at the soonest time it would be fair to do so. This should be done by notification of incoming messages, either at the MPI level or at the helper-thread level (TBD). Unlike the normal gang-scheduling operation, processes are never suspended; rather, they are preempted when another process gains priority over them (the local UNIX scheduler will gradually lower running processes' priorities). Lastly, we should make some modification to the communication library (MPI), so that it always spins when waiting for a waiting message in DCS, or spin-block in ICS (This are already implemented to some extant at the Elan library level). Other variations, as discussed in 2.2 and [4], can be readily implemented once these three modifications are performed.

4.5 Implementation of Buffered Coscheduling

There are two different parts of BCS that need to be implemented: the local scheduler in the NM, and the communication strobing, in the NM's helper thread running on the NIC. The former is relatively simple to implement under the suggested framework, while the latter is more complicated, requiring complex communication code to be run on the NIC's processor, and a sophisticated algorithm for the exchange of information with other NICs. This algorithm is the subject of future research, and will not be detailed in this document. Note that scheduling decisions are only made and carried out by the local scheduler. However, these decisions are based on information processed and provided by the communication thread, as is outlined below.

4.5.1 Local Scheduler

For the local scheduler, we would need to implement the following features:

1. All the commands received from the MM, including the heartbeat scheduling commands, should be processed as usual.

2. Information from the communication layer should be received and acted upon as fast as possible, especially if the process has issued a blocking communication attempt. In this case, the NM should preempt the process and schedule another process.

3. When making scheduling decisions, the NM should take into account the communication status of processes (blocked/ready). When preempting a process, it can choose the next process from the Ousterhout-row, or just let the UNIX scheduler do its job otherwise. In that case, we must ensure that previously preempted processes are not in suspended status.[2]

4.5.2 Communication Thread

In BCS, NM's helper thread receives the extra role of executing the communication phase of BCS. As described in 2.3.2, the communication phase is divided in three stages, and one of the modifications required to the thread is to schedule the different stages (using a timer). Let us consider each stage separately.

4.5.2.1 Information Gathering and Exchange

In the first stage, information about processes' communication is gathered. This is simple to implement by leaving most of the communication mechanisms intact and making only two minor changes. First, we must delay the execution of message sending: The Elan card and driver are designed to accept message-buffers DMA descriptors as a compact, fast interface to sending messages [22, 23, 24]. All we need to modify is to make sure the Elan thread does not dequeue and execute these messaging commands, but rather they are passed to the NM via shared-memory. Then, we must notify the scheduler (NM)

[2]By default, the NM suspends preempted processes. If we want the UNIX scheduler to run such processes, we should either send them a SIGCONT signal, or change the default NM behavior so that it lowers processes' priorities instead of suspending them.

immediately of blocking messages, so that it can preempt the running process if required. This is given almost for free using the communication mechanism we described in **??**.

At the end of this stage, a logical total-exchange of information between all participating nodes is performed, so that every node has complete knowledge of pending incoming- and outgoing- messages. The helper thread will read all the local information to be exchanged from the NM's memory (they share the same address space). Algorithmical aspects of this total-exchange will be researched in the future, and therefore are not detailed here. However, initially we could use the following representation and data exchange: We will use an $N \times N$ matrix, where N is the number of NICs in the system. The $N_{i,j}$ cell contains the total amount of data that NIC i needs to send to NIC j. Therefore, in an initial implementation of the total-exchange, each NIC i can broadcast the the i-th row and receive the i-th column. Later implementations will minimize the amount of communication and schedule the broadcasts to minimize network contention.

This stage is also a good point to hook a fault-tolerance mechanism. A distributed algorithm can use the information exchange to detect which nodes are not responding, report them to the NM, which in turn can report to the MM for re-allocation of resources. It can also be a good point in time to occasionally checkpoint the network status, which together with memory checkpointing can offer restoring capabilities for crashed processes. The discussion of the details of such a fault-tolerance mechanism are beyond the scope of this document, and may be addressed at a later time of the design.

4.5.2.2 Communication Scheduling

In the second communication stage, the data to be sent is scheduled by each Elan thread, using a distributed algorithm that ensures global agreement on sending order, and minimizes the amount of communication and network contention. This too is a research subject for future implementation. As an initial implementation, we can employ a simple, non-optimal algorithm. For example, we schedule the communication in two parts where first, all the odd-numbered NICs send to even-numbered NICs, and in the second, the reverse traffic takes place.

The product of this scheduling should result in a list of pointers to the DMA descriptors, sorted by the order of the scheduling. If feasible, we may actually sort the original list of descriptors instead of pointing to it (remains to be tested). Also, we can limit the amount of incoming messages by limiting the amount of requests-to-receive messages we disseminate at the information exchange stage. This way, we can guarantee that the amount of incoming data will not be so large that the time to receive it would exceed the timeslot.

4.5.2.3 Communication Execution

In the last stage, the scheduling decisions made in the previous stage are executed. If the previous stage sorted the original descriptor list, then all that is left is for the NM's helper thread to scan this list and send appropriate messages to the applications' helper threads. These in turn will send their messages from their own address space. We have noted in our experiments that we may keep the communicating processes in a running state, but without busy-wiating (i.e. communication is interrupt-driven), and the kernel will do a rather good job of overlapping the communicating processes with the currentl computing task. The NM can abort this stage before all the messages have been sent if the time allocated to it does not suffice for the amount of traffic. In that case, the thread just keeps the remaining list of descriptors, and passes it forward to the next communication phase.

Another important role of this stage is the releasing of blocked processes: When an incoming message or ACK arrives for a process that was blocked waiting for it, the local-scheduler should be notified as soon as possible so that it can re-schedule the blocked process when appropriate.

4.6 Implementation of Flexible Coscheduling

Flexible coscheduling contains two logical parts that require special attention, processes scheduling and process classification. Both parts are implemented and integrated with the NM, and gather process-communication information from the communication layer.

4.6.1 Process Classification

All the information required for the dynamic process classification can be easily gathered by the communication mechanism By adding the appropriate hooks the the MPI ADI layer. we also require a method to store the new process-relevant information and update it frequently. We can achieve that by enhancing the LoaderDesc class (see 3.3.2) to contain the extra information fields, along with the methods to calculate and update them. Such fields can include the process class and the time it was last set, how long has it been spinning, what is its average spin time and how long is it allowed to spin, etc. The methods to update this data include the classification formulae described in 2.4.4 and take into account the initial period until the process' class is stabilized. The most natural place to call these update routines would be in the housekeeper() function, whether when invoked by a timer event or by a communication event.

4.6.2 Scheduling Implementation

The invocation of the scheduling mechanism in FCS can be triggered by three types of events, all having an appropriate callback in our infrastructure: A global context-switch event (heartbeat message), a process waiting for synchronous communication and a process unblocking for same (processes should not actually block for synchronous communication but rather spin.) In effect, the termination of a process will also cause a scheduling decision, but this will be carried out by default by the UNIX scheduler, picking any other process as outlines below. The following principles apply for process scheduling:

1. When a heartbeat message arrives for a CS or F process, its priority is raised to the maximum, so that it has exclusive use of the processor. When a heartbeat causes the descheduling of this process, its priority is returned to the previous one.
2. If an F process spins for more than its allotted spin time, its previous priority is restored, so that the UNIX scheduler can preempt it and run another process in its stead. However, as soon as it is unblocked by the network layer, its priority should be set back to the maximum, if still within its timeslot.
3. When a DC time slot starts, marked by a heartbeat message that schedules a DC process, it is ignored by the NM. All the scheduling is done by the UNIX scheduler.
4. To ensure Processes are scheduled properly during DC timeslots and F gaps, processes are divided into three groups of priorities (below the maximum). DC processes belong to the highest group, F processes run in the second group, and CS have the lowest priority.

By integrating these priority manipulations into the NM (which is also required in 4.4), we can get implement FCS with little effort using the help of the UNIX scheduler.

4.7 Implementation of Local Scheduling

The implementation of local scheduling is particularly simple in our framework. We simply disable all the scheduling functions of the NM, except for initial launching of jobs. The local unix scheduler with do the rest.

4.8 Implementation of FCFS Scheduling

The implementation of FCFS scheduling is relatively simple in our framework. All that is required is to set the amount of rows in the Ousterhout-matrix to one (no parallelism), and queue all the jobs that cannot be immediately allocated, retrying to allocate resources for them whenever a previous job completes. This will require a simple implementation of a queueing mechansim in the MM.

4.9 Implementation of ParPar

The implementation of ParPar over Quadrics would require two significant changes. The first is to replace the internal communication layer between the master daemon and the node daemons to work over Quadrics. This in turn implies that the ParPar application has a network capability, and may require that we launch it with `prun`, much in the same way we plan for the MM and NMs. The other issue to be addressed is the creation of proper capabilities for newly-launched application. This is a rather delicate subject, and will require consultation with Quadrics developers and RMS code.

5 Benchmarks

Part of any testbed is the set of measurement tools used to test and compare the various alternatives. We intend to use three types of tests for evaluating the schedulers: running a workload consisting of typical LANL applications; running a workload consisting of externally developed benchmark applications, and running a synthetic benchmark design explicitly to measure the scheduler's performance.

5.1 LANL Applications

To test the viability of different schedulers for the lab, we intend to run several applications that are representative of the lab's production environment. These applications are mainly from the field of 3D fluid dynamics, and consist of SAGE, SWEEP3D, and several others. The workload model to be used for these applications will be defined in the future, after further study of the current site workloads. However, the reader should keep in mind that these applications are usually not run as time-shared, and exploit most of the machine's physical resources.

5.2 External Benchmarks

There are a few suites of parallel application benchmarks that are used by the research community, and we could potentially use some of these to compare the different schedulers. Notable benchmarks suites in the field are the NAS NPB-2 [7, 2] suite from NASA Ames center, and the SPLASH [25] and Splash-2 [3, 28] suites from Stanford. When the testbed is completed, we will review these benchmarks and choose the most appropriate suites. Note that these were designed with hardware-performance benchmarking in mind, and not so much system-performance, where the scheduler has an immense effect. Therefore, a study of the current state-of-the-art in workload modeling will probably also be required to run applications from these suites concurrently in a realistic way, and not as stand-alone applications.

5.3 Synthetic Benchmark

In a recent research work by Dr. D. Feitelson and E. Frachtenberg, a new benchmarking tool was developed to measure specifically the performance of coschedulers. This tool is based on the model presented in [16], and is a synthetic benchmark that launches parallel jobs of different characteristics: communication granularity and type, size of job, resource requirements, etc. It uses the workload model developed in [18] to create realistic workload for any machine configuration and different loads. It was designed with homogeneous and heterogeneous clusters in mind, and can simulate different CPU speeds even on homogeneous clusters. A visualization tool exists that can visually illustrate the scheduling decisions made during the tested scenario. This tool can be extended to work with any application (and not only the benchmark application), if the network and scheduling layer of the nodes is slightly modified to log the required information.

5.4 Testing platform

The basic testing platform we intend to use for the tests is the 16-node 'crescendo' cluster. Each node in this cluster will contain two 1-GHz Pentium III processors, with 1GB of ECC RAM, a 66 MHz 64-bit PCI bus, and an Elan and Ethernet network connections. At some point we may modify some of the nodes by upgrading or downgrading them if we intend to measure performance in an heterogeneous cluster. We may also perform some measurements on other LANL Quadrics clusters, if such become available for testing.

6 Work Breakdown Structures (WBS)

TBD....

1. Create framework application (MM, NMs, PLs)
2. Create helper threads for MM and NMs, test communication
3. Create helper threads for PLs and NMs, test communication
4. Implement protocols for helper threads (except strobing)
5. Code MM, including rm_dynbt
6. Implement PLs, inc. MPI cheat
7. Test and debug application launching
8. Implement Heartbeats
9. Full GS - test and debug
10. Implement MPI to NM protocol
11. Implement FCS and others
12. Implement benchmarks
13. Measurements

Bibliography

[1] http://www.cs.huji.ac.il/labs/parallel/parpar.shtml.

[2] http://www.nas.nasa.gov/Software/NPB/.

[3] http://www-flash.stanford.edu/apps/SPLASH/.

[4] Cosimo Anglano. A Comparative Evaluation of Implicit Coscheduling Strategies for Networks of Workstations. In *Proceedings of the Ninth International Symposium on High Performance Distributed Computing (HPDC 9)*, Pittsburgh, PA, August 2000.

[5] Andrea C. Arpaci-Dusseau, David Culler, and Alan M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proceedings of the 1998 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, Madison, WI, June 1998.

[6] Remzi Arpaci-Dusseau, Andrea C. Arpaci-Dusseau, Amin Vahdat, Lok T. Liu, Thomas E. Anderson, and David A. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of the 1995 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, pages 267–278, Ottawa, Canada, May 1995.

[7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[8] Dror G. Feitelson and Larry Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *Journal of Parallel and Distributed Computing*, 16(4), 1992.

[9] Yoav Etsion and Dror G. Feitelson. User-Level Communication in a System with

Gang Scheduling. In *Proceedings of the International Parallel and Distributed Processing Symposium 2001, IPDPS2001*, San Francisco, CA, April 2001.

[10] Fabrizio Petrini and Wu-chun Feng. Buffered Coscheduling: A New Methodology for Multitasking Parallel Jobs on Distributed Systems. In *Proceedings of the International Parallel and Distributed Processing Symposium 2000, IPDPS2000*, Cancun, MX, May 2000.

[11] Fabrizio Petrini and Wu-chun Feng. Improved Resource Utilization with Buffered Coscheduling. *Journal of Parallel Algorithms and Applications*, 2000. Accepted for Publication.

[12] Fabrizio Petrini and Wu-chun Feng. Scheduling with Global Information in Distributed Systems. In *Proceedings of the The 20th International Conference on Distributed Computing Systems*, Taipei, Taiwan, Republic of China, April 2000.

[13] Fabrizio Petrini and Wu-chun Feng. Time-Sharing Parallel Jobs in the Presence of Multiple Resource Requirements. In *6th Workshop on Job Scheduling Strategies for Parallel Processing*, Cancun, MX, May 2000.

[14] Dror G. Feitelson, Anat Batat, Gabriel Benhanokh, David Er-El, Yoav Etsion, Avi Kavas, Tomer Klainer, Uri Lublin, and Marc Volovic. The ParPar System: a Software MPP. In Rajkumar Buyya, editor, *High Performance Cluster Computing*, volume 1: Architectures and systems, pages 754–770. Prentice-Hall, 1999.

[15] Dror G. Feitelson and Morris A. Jette. Improved Utilization and Responsiveness with Gang Scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 238–261. Springer-Verlag, 1997.

[16] Dror G. Feitelson and Larry Rudolph. Metrics and benchmarking for parallel job scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1495 of *Lecture Notes in Computer Science*, pages 1–24. Springer-Verlag, 1998.

[17] Eitan Frachtenberg, Fabrizio Petrini, Salvador Coll, and Wu chun Feng. Gang Scheduling with Lightweigth User-Level Communication. In *2001 International Conference on Parallel Processing (ICPP2001), Workshop on Scheduling and Resource Management for Cluster Computing*, Valencia, Spain, September 2001.

[18] Uri Lublin. A workload model for parallel computer systems, 1999. Master's thesis, Hebrew University, 1999. (In Hebrew).

[19] Shailabh Nagar, Ajit Banerjee, Anand Sivasubramaniam, and Chita R. Das. A Closer Look At Coscheduling Approaches for a Network of Workstations. In *Eleventh ACM Symposium on Parallel Algorithms and Architectures, SPAA'99*, Saint-Malo, France, June 1999.

[20] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. *Proceedings of Third International Conference on Distributed Computing Systems*, 1982.

[21] Fabrizio Petrini, Federico Bassetti, and Alex Gerbessiotis. A New Approach to Parallel Program Development and Scheduling of Parallel Jobs on Distributed Systems. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, volume I, pages 546–552, Las Vegas, NV, July 1999.

[22] Quadrics Supercomputers World Ltd. *Elan Kernel Communication Manual*, December 1999.

[23] Quadrics Supercomputers World Ltd. *Elan Programming Manual*, January 1999.

[24] Quadrics Supercomputers World Ltd. *Elan Reference Manual*, January 1999.

[25] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44.

[26] Patrick Sobalvarro, Scott Pakin, William E. Weihl, and Andrew A. Chien. Dynamic Coscheduling on Workstation Clusters. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 231–256. Springer-Verlag, 1998.

[27] Patrick Sobalvarro and William E. Weihl. Demand-Based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors. In *Proceedings of the 9th International Parallel Processing Symposium, IPPS'95*, Santa Barbara, CA, April 1995.

[28] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, , and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.