ClusterSchedSim: A Unifying Simulation Framework for Cluster Scheduling Strategies

Yanyong Zhang
Dept. of Electrical & Computer Engineering
Rutgers University
Piscataway, NJ 08854
yyzhang@ece.rutgers.edu

Anand Sivasubramaniam
Dept. of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802.
anand@cse.psu.edu

1 Introduction

With the growing popularity of clusters, their usage in diverse application domains poses interesting challenges. At one extreme, we find clusters taking on the role of supercomputing engines to tackle the ”grand challenges” at different national laboratories. At the other extreme, clusters have also become the ”poor man’s” parallel computer (on a smaller scale). In between, we find a diverse spectrum of applications - graphics/visualization and commercial services such as web and database servers - being hosted on clusters.

With the diverse characteristics exhibited by these applications, there is a need for smart system software that can understand their demands to provide effective resource management. The CPUs across the cluster are amongst the important resources that the system software needs to manage. Hence, scheduling of tasks (submitted by the users) across the CPUs of a cluster is very important. From the user’s perspective, this can have an important consequence on the response times for the jobs submitted. From the system manager’s perspective, this determines the throughput and overall system utilization, which are an indication of the revenues earned and the costs of operation.

Scheduling for clusters has drawn, and continues to draw, a considerable amount of interest in the scientific community. However, each study has considered its own set of workloads, its own underlying platform(s), and there has not been much work in unifying many of the previous results using a common infrastructure. The reason is partially the lack of a set of tools that everyone can use for such a uniform comparison. At the same time, the field is still ripe for further ideas/designs that future research could develop for the newer application domains and platforms as they evolve. Already, the design space of solutions is exceedingly vast to experimentally (on actual deployments) try them out and verify their effectiveness.

All these observations motivate the need for cluster scheduling tools (for a common infrastructure to evaluate existing solutions, and to aid future research) that can encompass a diverse range of platforms and application characteristics. If actual deployment is not a choice, then these tools should either use analytical modes or simulation. While analytical models have been successful in modeling certain scheduling mechanisms, their drawbacks are usually in the simplifying assumptions made about the underlying system and/or the applications. It is not clear whether those assumptions are valid for future platforms and/or application domains.

Instead, this paper describes the design of a simulator, called *ClusterSchedSim*, that can be used to study a wide spectrum of scheduling strategies over a diverse set of application characteristics. Our model includes workloads of supercomputing environments that may span hundreds of processors and can take hours (or even days) to complete. Such jobs have typically higher demands on the CPU and the network bandwidth. We also include workloads representative of commercial (server) environments where a high load of short-lived jobs (with possibly higher demands on the I/O subsystem) can be induced. Previous research [19, 12, 27, 23, 17, 7, 6, 22, 8, 15, 16, 1, 21, 3] has shown that not one scheduling mechanism is preferable across these diverse conditions, leading to a plethora of solutions to address this issue. Our simulator provides a unifying framework for describing and evaluating all these earlier proposals. At the same time, it is modular and flexible enough to allow extensions for future research.

An overview of *ClusterSchedSim* is given in Figure 1. *ClusterSchedSim* consists of the following modules:

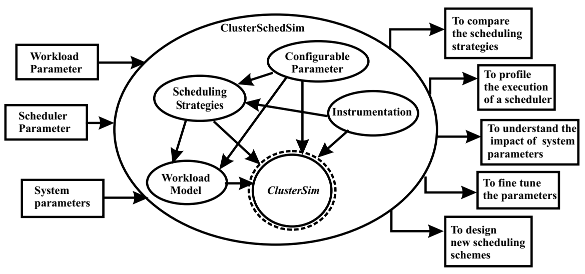

ClusterSchedSim
Workload Parameter
Scheduler Parameter
System parameters
Configurable Parameter
Scheduling Strategies
Instrumentation
Workload Model
ClusterSim
To compare the scheduling strategies
To profile the execution of a scheduler
To understand the impact of system parameters
To fine tune the parameters
To design new scheduling schemes

Figure 1: Overview of *ClusterSchedSim*

- *ClusterSim*: a detailed simulator of a cluster system which includes the cluster nodes and interconnect. It simulates the OS functionality as well as the user-level application tasks on each cluster node.
- Workload package. As mentioned, we provide a diverse set of workloads including those at the supercomputing centers, those for a commercial server, and even some multimedia workloads though this is not explictly described within this paper. This package is implemented on top of the cluster model.
- Scheduling strategy package. This package is implemented on top of the cluster model and workload model. It includes a complete set of scheduling strategies that are designed for various workloads. It includes both the assignment of tasks to nodes of the cluster (spatial scheduling) and the temporal scheduling of tasks at each cluster node.
- Instrumentation package. This package is implemented on top of the cluster model and scheduling strategies. It can instrument the executions at the application level, scheduler level, and even at the operating system level to obtain an accurate profile of the execution. The instrumentation can be easily turned on, turned off, or partially turned on based on the user needs.
- Configurable parameter package. We provide numerous configurable parameters to specify the system configuration (e.g., number of cluster nodes, context-switch cost, etc), scheduler behavior (e.g., time quanta), and overheads for different operations.

In the rest of this paper, we go over the details of the implementation of these different features within *ClusterSchedSim*. This simulator, as mentioned earlier, can be useful for several purposes (Figure 1). In this paper, we specifically illustrate its benefits using three case studies:

- To pick the best scheduler for a particular workload. *ClusterSchedSim* consists of a complete set of cluster scheduling strategies, and we can compare these strategies for a particular workload type and choose the best. For instance, we show gang scheduling [17, 7, 6, 22, 8] is a good choice for communication intensive workloads, while dynamic co-scheduling strategies such as Periodic Boost [15, 32, 33] and Spin block [1] are better choices otherwise.
- To profile the execution of a scheduler. In order to understand why a scheduler may not be doing as well as another, a detailed execution profile is necessary. Using the execution profiles, one can understand the bottleneck in the execution, and thus optimize the scheduler. For instance, we find that gang scheduling incurs more system overheads than some dynamic co-scheduling schemes such as Periodic Boost.
- To fine tune parameters for a particular scheduler. The schedulers, together with the underlying cluster platform, have numerous parameters, which may impact the performance significantly. *ClusterSchedSim* makes all these parameters configurable, and one can use these to tune the setting for each scheduler. For instance, our experiments show that an MPL level between 5 and 16 is optimal for some dynamic co-scheduling schemes [32].

The rest of the paper is organized as follows. The next section describes the system platform *ClusterSchedSim* tries to model and the workload model we use in the simulator. Section 3 explains how the core cluster simulator

(*ClusterSim*) is implemented. Section 4 presents all the scheduling strategies and their implementations. The instrumentation and parameter packages for the simulator are discussed in Section 5. Section 6 illustrates the usages of *ClusterSchedSim*, and finally Section 7 summarizes the conclusions.

2 System Platform and Workloads

Before we present our simulation model, we first describe the system platform and workloads we try to model.

2.1 System Platform

We are interested in those clusters used to run parallel jobs because scheduling on such systems is particularly challenging. The following features are usually common to these clusters, and they are adopted in our simulator:

- **Homogeneous nodes**

  Each cluster node has the same hardware configuration and same operating system. Further, in order to boost the performance/cost ratio, most of today's clusters have either single processor or dual processors per node. On the other hand, scheduling on SMPs has been extensively studied earlier [4, 9, 10, 11], and it is thus not the focus of this paper. Our simulator considers one processor per node. Also, we use *nodes* and *processors* interchangeably unless explicitly stated.

- **User level networking**

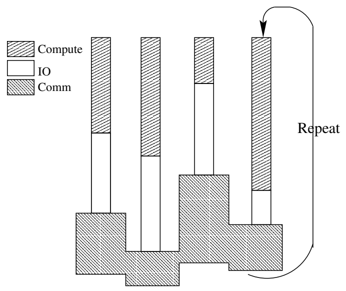
  Traditional communication mechanisms have necessitated going via the operating system kernel to ensure protection. Recent network interface cards (NIC) such as Myrinet, provide sufficient capabilities/intelligence, whereby they are able to monitor regions of memory for messages to become available, and directly stream them out onto the network without being explicitly told to do so by the operating system. Similarly, an incoming message is examined by the NIC, and directly transferred to the corresponding application receive buffers in memory (even if that process is not currently scheduled on the host CPU). From an application's point of view, sending translates to appending a message to a queue in memory, and receiving translates to (waiting and) dequeuing a message from memory. To avoid interrupt processing costs, the waiting is usually implemented as polling (busy-wait). Experimental implementations of variations of this mechanism on different hardware platforms have demonstrated end-to-end (application-to-application) latencies of 10-20 microseconds for short messages, while most traditional kernel-based mechanisms are an order of magnitude more expensive. User-level messaging is achieved without compromising protection since each process can only access its own send/receive buffers (referred to as an *endpoint*). Thus, virtual memory automatically provides protected access to the network.

  Several ULNs [25, 26, 18] based on variations of this paradigm have been developed.

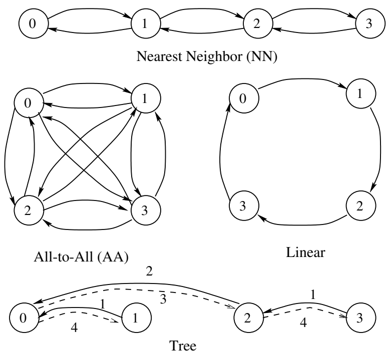
  User-level messaging, though preferable for lowering the communication overhead, actually complicates the issue from the scheduling viewpoint. A kernel-based blocking receive call, would be treated as an I/O operation, with the operating system putting the process to sleep. This may avoid idle cycles (which could be given to some other process at that node) spent polling for message arrival in a user-based mechanism. Efficient scheduling support in the context of user-level messaging thus presents interesting challenges.

2.2 Parallel Workloads and Performance Metrics

We are interested in the environments where a stream of parallel jobs dynamically arrive, each requiring a number of processors/nodes. The job model we develop in this study is shown in Figure 2(a). A parallel job consists of several tasks, and each task executes a number of iterations. During each iteration, the task computes, performs I/O, sends messages and receives messages from its peer tasks. The chosen structure for a parallel job stems from our experiences with numerous parallel applications. Several scientific applications, such as those in the NAS benchmarks [2], and Splash suite [20], exhibit such behavior. For instance, computation of a parallel multidimensional FFT requires a processor to perform a 1-D FFT, following which the processors exchange data with each other to implement a transpose, and the sequence repeats iteratively. I/O may be needed to retrieve the



(a) application model



(b) communication pattern

Figure 2: Job Structure

data from the disk during the 1-D FFT operation since these are large datasets. A parallel SOR similarly has each processor compute matrix elements by averaging nearby values, after which the processors exchange their latest values for the next iteration. Even when one moves to newer domains such as video processing which requires high computational and data speeds to meet real-time requirements, each processor waits for a video frame from another processor, processes the frame, and then streams the result to the next processor in a pipelined fashion. All of these application behaviors can be captured by our job structure via appropriate tuning of the parameters.

Specifically, every job has the following parameters:

- arrival time.
- number of iterations it has to compute.
- number of nodes/processors it requires.
- for each task of the job, we have the following parameters:
  - distribution of the compute time during each iteration. Each task may follow a different distribution.
  - distribution of the I/O time during each iteration. Each task may follow a different distribution.
  - communication pattern. The four common communication patters are illustrated in Figure 2(b).

A parallel workload consists of a stream of such jobs. By varying these parameters, we can generate workloads with different offered load and job characteristics (e.g., its communication intensity, I/O intensity, CPU intensity or skewness between tasks).

From our simulator, we can calculate various performance metrics. To name just a few, the following metrics are important from both the system's and user's perspective:

- *Response Time:* This is the time difference between when a job completes and when it arrives in the system, averaged over all jobs.
- *Wait Time:* This is the average time spent by a job waiting in the arrival queue before it is scheduled.
- *Execution Time:* This is the difference between *Response* and *Wait* times.
- *Slowdown:* This is the ratio of the *response time* to the time taken on a system dedicated solely to this job. It is an indication of the slowdown that a job experiences when it executes in multiprogrammed fashion compared to running in isolation.
- *Throughput:* This is the number of jobs completed per unit time.
- *Utilization:* This is the percentage of time that the system actually spends in useful work.
- *Fairness:* The fairness to different job types (computation, communication or I/O intensive) is evaluated by comparing (the coefficient of variation of) the response times between the individual job classes in a mixed workload. A smaller variation indicates a more fair scheme.

3 *ClusterSim*: The Core Cluster Simulator

3.1 CSIM Simulation Package

ClusterSchedSim is built using CSIM [14]. CSIM is a process-oriented discrete-event simulation package. A CSIM program models a system as a collection of CSIM *processes* that interact with each other. The model maintains simulated time, so that we can model the time and performance of the system. CSIM provides various simulation objects. In *ClusterSchedSim*, we extensively use the following two objects: CSIM *processes* and *events*.

CSIM processes represent active entities, such as the operating system activities, the application tasks, or the interconnect between cluster nodes. In this paper, we use *tasks* to denote the real operating system processes, and *processes* for CSIM processes. At any instant, only one task can execute on the CPU, but several tasks can appear to execute in parallel by time-sharing the CPU at a fine granularity. Tasks relinquish the CPU when their time-slices expire or they are blocked on some events. Similarly, Several different CSIM processes, or several instances of the same CSIM process, can be active simultaneously. Each of these processes, or the instances, appear to run in parallel in terms of the simulated time, but they actually run sequentially on a single processor (where the simulation takes place). The illusion of parallel execution is created by starting and suspending processes as time advances and as events occur. CSIM processes execute until they suspend themselves by doing one of the following actions:

- execute a *hold* statement (delay for a specified interval of time),
- execute a statement which causes the processes blocked on an event, or
- terminate.

Processes are restarted when the time specified in a hold statement elapses or when the event occurs. The CSIM runtime package guarantees that each instance of every process has its own runtime environment. This environment includes local (automatic) variables and input arguments. All processes have access to the global variables of a program.

The CSIM processes synchronize with each other via CSIM events. A CSIM event is similar to a conditional variable provided by the operating system. An event can have one of the two states: occurred and not occurred. A pair of processes synchronize by one waiting for the event to occur (by executing the statement `wait(event)`) and the other changing its state from not occurred to occurred (by executing the statement `set(event)`). When a process executes `wait(event)`, it is put in the wait queue associated with the event. There can be two types of wait queues: ordered and non-ordered. Only one process in the ordered queue will resume execution upon the occurrence of the event, while all the events will resume executions in the non-ordered queue. Further, a process can also specify a time-out limit on how much time it will wait. In such case, the process will resume execution if either of the following two conditions are true: (1) the event occurs within the bound; or (2) the time-out limit is reached.

3.2 Structure of *ClusterSim*

A cluster consists of a number of nodes that are connected by the high-speed interconnect. *ClusterSim* models such a system. As shown in Figure 3, it has the following modules:

- Cluster node module. A cluster node has the three major modules: the CPU, memory, and network interface card (NIC). Further, the CPU hosts both application tasks and operating system daemons.
  - Application Tasks
    A parallel job consists of multiple tasks, with each task mapping to an application task of a cluster node. As observed in many parallel workloads [2, 20], a task alternates between computation, I/O, and communication phases in an iterative fashion. In the communication phase, the application task sends messages to one or more of its peers, and then receives messages from those peers. An application task is implemented using a CSIM process. The following pseudo-code of the CSIM process describes the typical behavior of an application task:

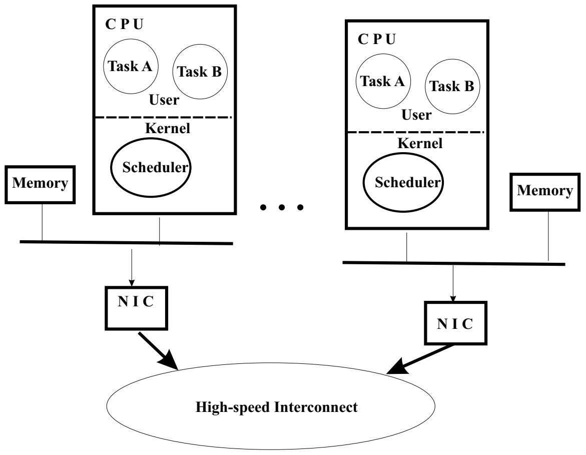


Figure 3: Structure of the Core Cluster Simulator

```
DoAppTask{
    for (i=0; i<NumIterations; i++){
        DoCompute(t);

        DoIO(t);

        for (j=0; j<NumMsgs; j++){
            DoSend();
        }

        for (j=0; j<NumMsgs; j++){
            DeReceive();
        }
    }
}
```

In the compute phase `DoCompute(t)`, the task will use the CPU for t units of simulated time. However, the actual gap between the end of the compute phase and the beginning may be larger than t because of the presence of other tasks on the same node. At the beginning of the compute phase, the application task registers the desired compute time with the CPU scheduler, and then suspends itself until the compute time has been achieved by waiting on the CSIM event `EJobDone`. The event will be set by the CPU scheduler after the computing is done. The details about how the CPU scheduler keeps track of the compute time for each task will be discussed below when the CPU scheduler is presented.

In the I/O phase `DoIO(t)`, the task will relinquish CPU and waits for the I/O operation (which takes t units of simulated time) to complete by executing `hold(t)`. After the I/O operation is completed, it will be put back in the ready queue of the CPU scheduler.

In the communication phase, a task sends out messages to its peers, and then receives messages. From the perspective of a task, sending out a message (`DoSend(msg)`) involves composing the message, appending it to the end of its outgoing message queue, and then notifying the NIC by setting the event `EMsgArrivalNIC`. Then the message will be DMA-ed to the NIC. The application task experiences the overhead of composing the message, and the DMA overhead will be experienced by the NIC (the details are presented below when the network interface module is discussed) . The overhead of composing the message is modelled by `DoCompute(overhead)` because the task needs CPU to finish this operation.

When a task tries to receive a message, it first checks its incoming message queue to see whether the message has arrived. If the message has not yet arrived, it will enter the busy-wait phase. With user-level networking, the task will not relinquish the CPU while waiting for a message. Instead, it polls the message queue periodically. However, polling is an expensive operation in terms of the simulation cost (the time taken to complete the simulation) because each polling requires CSIM state changes. Instead, we use the interrupt-based approach to model the busy-wait phase: the CSIM process that implements the task will suspend its execution by executing `Wait(EMsgArrival)` and will be woken up later when the message arrives. Please note that this is just an optimization to make the simulation more efficient, and the task is still running on the CPU until either its quantum expires or the message arrives. After the message arrives, the task decomposes the message. The overhead of decomposing a message is modelled by `DoCompute(overhead)`.

The pseudo-codes for routines `DoCompute(t)`, `DoIO(t)`, `DoSend(msg)`, and `DoReceive(msg)` are as follows:

```
DoCompute(t){
    WaitUntilScheduled();
    Register t with Scheduler;
    wait(EComputeDone);
}

DoIO(t){
    WaitUntilScheduled();
    remove the job from the ready queue;
    hold(t);
    insert the job to the ready queue;
}

DoSend(msg){
    WaitUntilScheduled();
    DoCompute(composition overhead);
    Compose the message;
    Append the message to the message queue;
    set(EMsgArrivalNIC);
}

DoReceive(msg){
    WaitUntilScheduled();
    if (message has not arrived){
        wait(EMsgArrival);
    }
    WaitUntilScheduled();
    DoCompute(decomposition overhead);
    Decompose the message;
}
```

In the above pseudo-codes, we frequently use the function `WaitUntilScheduled()`. When the task returns from this function, it will be running on the CPU. This function makes sure that all the application operations that need CPU only take place after the task is being scheduled.

The frequency and duration of the compute, I/O and communication phases, and the communication patterns are determined by the workloads (Section 2.2).

– CPU Scheduler

The CPU scheduler is the heart of a cluster node. Tasks on the same CPU implicitly synchronize with each other via the CPU scheduler. The CPU scheduler manages the executions of all the application tasks, and it is implemented by a CSIM process. In a real operating system, the scheduler becomes active whenever the timer-interrupt is raised (e.g., 1 millisecond in Sun Solaris, and 10 milliseconds in Linux) to check whether preemption is needed. This corresponds to the *polling* method. This polling method can be easily implemented by making the scheduler process blocked on the timer event. Using this method, if the simulated time is 1000 seconds (which is much shorter than a typical simulated time) and the timer interrupt becomes active every 1 millisecond, then the scheduler process will become active 1000000 times. In CSIM, waking up a process is a costly operation, and this approach will lead to an unreasonably long simulation time. In order to reduce the simulation time, and to improve the scalability of the simulator, we use an *interrupt* method instead. The scheduler becomes active only when the currently running task needs to be preempted, either because its time slice expires, or because a higher-priority task is made available. This interrupt-based method can be implemented by executing the statement `timed_wait(EScheduler, timer)`. The timer will be set to the duration of a time slice. The event `EScheduler` will be set when new tasks that have higher priorities become ready to execute. Either these tasks are either newly allocated to the node, or they just become ready after waiting for some events (e.g., I/O completion). However, in a real operating system, the preemption does not take place as soon as such tasks are available. Instead, it will happen when the next timer interrupt arrives. In order to model this behavior, we delay signalling the event

`EScheduler` until the next timer interrupt. When the scheduler becomes active, it will preempt the currently running task, and pick the task that has the highest priority to schedule next.

After presenting the basic operation of the CPU scheduler, we next discuss a few details:

∗ Compute time. If the task that will run next is doing computation (i.e., waiting for the event `EComputeDone`) , then the value of `timer` is the smaller one between the time quantum and the remaining compute time. Next time when the scheduler becomes active, it will update the running task's remaining compute time, and its remaining time quantum. If the task is done with its computation, it will notify the application task by executing `set(EComputeDone)`. Further, if the time quantum has not expired, then the application will continue its execution until either quantum expiration or preemption by a higher-priority task.

∗ Context-switches. If a different task will be scheduled next, then a context-switch overhead must be paid. During the context switch, the application task will not make any progress. Hence, the context switch is modelled by executing `hold(overhead)`.

As a result, the high-level behavior of the scheduler process is described by the following pseudocode.

```
DoScheduler(){
    while(1){

        timer = min(quantum, compute time);
        timed_wait(EScheduler, timer);


        if(timer expires && compute time > 0 && compute time < quantum){
            update the compute time;
            if(compute time == 0) set(EComputeDone);
        }

        if(EScheduler || compute time == 0 || compute time > quantum){
            if (compute time > 0) update the compute time;
            inserts the current task to the ready queue;
            pick the candidate task to schedule;
            if (current task != candidate tsak)
                hold(overhead);
        }
}
```

Different CPU schedulers manage the ready tasks in different ways. In *ClusterSim*, we model the CPU scheduler of Sun Solaris which is based on the multi-level feedback queue model. There are 60 priority levels (0 to 59 with a higher number denoting a higher priority) with a queue of runnable/ready tasks at each level. The task at the head of the highest priority queue will be executed. Higher priority levels get smaller time slices than lower priority levels, that range from 20 milliseconds for level 59 to 200 milliseconds for level 0. At the end of the quantum, the currently executing task is degraded to the end of the queue of the next lower priority level. Task priority is boosted (to the head of the level 59 queue) when they return to the runnable/ready state from the blocked state (completion of I/O, signal on a semaphore etc.) This design strives to strike a balance between compute and I/O bound jobs, with I/O bound jobs typically executing at higher priority levels to initiate the I/O operation as early as possible.

– Memory

A parallel job often requires a large amount of memory. This problem is accentuated when we multiprogram a large number of tasks which can involve considerable swapping overheads. For this reason, several systems (e.g., [29, 32]) limit the multiprogramming level (*MPL*) so that swapping overheads are kept to a minimum. Under the same rationale, we use the multiprogramming level directly to modulate the memory usage rather than explicitly model the memory consumption.

– Network Interface Card (NIC)

A NIC connects its host CPU to the interconnect. It sends the outgoing messages to the interconnect, and deposits the incoming messages to the appropriate endpoints in the host memory. We implement the NIC module using a CSIM process.

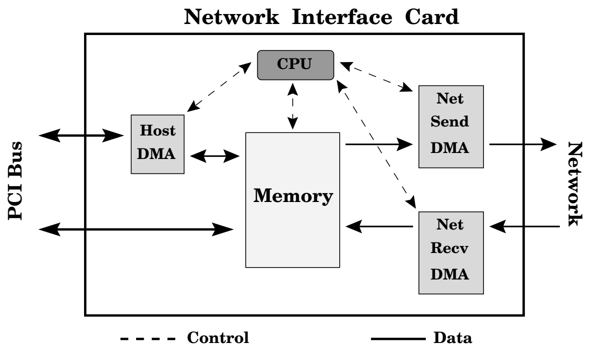


Figure 4: Overview of the network interface card

As illustrated in Figure 4, the NIC has (at least) the following active entities: the CPU, the host DMA engine, the net send DMA engine and the net receive DMA engine. The CPU programs the DMA engines, and the DMA engines complete the DMA operations.

After an application task appends an outgoing message to its message queue (in the memory), it wakes up the NIC CPU, which in turns programs the host DMA engine, and the host DMA engine will DMA the message from the host memory to the outgoing message queue that resides on the NIC. Then the net send DMA engine will DMA the message to the interconnect. Similarly, after the NIC CPU is woken up by the interconnect network for an in-coming message, it will first program the net receive DMA engine, and then the host DMA engine. These two engines will DMA the message to the corresponding endpoints that reside in the host memory. The host operating system is completely bypassed in this whole process.

A straightforward way of implementing the NIC is to have a CSIM process for each of the active entities. However, this approach will lead to a high simulation overhead because CSIM processes are expensive both time-wide and space-wide. Instead, we simplify the model slightly, but without compromising the accuracy. In our approach, we use one CSIM process to implement the entire NIC. The NIC process will be idle if there is no messaging activity by executing `wait(EMsgArrivalNIC)`. After it is woken up by either the host CPU or the network, it will check both the outgoing and the incoming message queues. For each outgoing message, it first executes `hold(DMA overhead)` to model the DMA operation, and then sends the message to the interconnect. For each incoming message, the NIC first identifies its destination task, deposits the message to the appropriate endpoint, and then notifies the task by executing `set(EMsgArrival)`. We include the DMA overhead to/from the interconnect in the end-to-end latency which will be experienced by the interconnect module. This way the timing for both the outgoing messages and the incoming messages are correct since we only have one CSIM process.

The pseudo-code for the network interface process is as follows.

```
DoNIC(){
    while(1){
        wait(EMsgArrivalNIC);
        while(out-going queue not empty){
            hold(DMA overhead);
            deliver the message to the interconnect network;
        }
        while(in-coming queue not empty)
            deliver the message to the application process;
    }
}
```

- High-speed interconnect model

  For the interconnect, we need to model the time it takes to exchange messages between any pair of nodes. Ideally, one would like to create a CSIM process between each pair (link) of nodes to model the details and overheads of the transfer. While we allow this functionality in our simulator, we observed that the overheads of creating a large number of CSIM processes considerably slows down the simulation. Instead, we provide an alternate model that a user can choose which contains only one CSIM process. This process receives all messages (between any pair of nodes), orders them based on anticipated delivery time (that is calculated using models as in [18]), waits for the time between successive deliveries, and then passes each message to its appropriate destination NIC at the appropriate time. Note that this is mainly for simulation overhead (we observed that it does not significantly affect the results since software overheads at the CPU scheduler and its efficiency are much more dominant) and if necessary, one can use the detailed model.

As mentioned above when NIC is discussed, the interconnect will calculate the time when a message will arrive at the message queue on the destination NIC (we call it the *completion time* of the message). This delay consists of two parts: the end-to-end latency and the NIC DMA overhead. We adopt a linear model to determine the end-to-end latency. The interconnect can serve several messages simultaneously. However, the network receive DMA engine of the destination NIC can only complete the DMA operations sequentially. For instance, message m that is addressed to node n arrives at the interconnect at time t. It will be ready for the NIC n to pick up (DMA) at time $t + L$ where L is the interconnect latency for m (L is a function of the message size). Suppose that some other messages are being DMA-ed or waiting to be DMA-ed to the NIC n, then m will be DMA-ed only when it is the first one in the message queue (which will be later than $t + L$).

The CSIM process that implements the interconnect manages all the messages before their completion. As soon as a new messages arrives, it calculates the completion time as described above, and inserts the message into its queue. When the queue is not empty, the interconnect process keeps track of the gap before the next message's completion time, and executes `timed_wait(EMsgArrivalNetwork, timeout)`, which enables the interconnect to pre-empt if a new message with an earlier completion time arrives. If its queue is empty, the interconnect process will again execute `timed_wait(EMsgArrivalNetwork, timeout)`, with `timeout` being set to a very large value.

The pseudo-code for the interconnect process is as follows:

```
DoInterconnect(){
    while(1){
        if (network is busy) timeout = infinity;
        else timeout = completion time of the first message - now;

        wait(EMsgArrivalNetwork, timeout);

        if (EMsgArrivalNetwork occurs){
            set network busy;
            calculate the completion time of the message;
            insert the message into the queue;
        }

        if (timeout){
            if (queue empty) set network idle;
        }
    }
}
```

4 Scheduling Strategy Package

We have looked at our cluster simulation model and the considered workloads. In this section, we discuss how to implement a wide range of scheduling strategies on top of these two components.

4.1 Summary of Cluster Scheduling Strategies

A parallel job consists of more than one task. Each task will be mapped to a different cluster node, and will be communicating with other peer tasks (Section 2.2). Communication between tasks includes sending messages to and receiving messages from its peers. User-level networking bypasses the operating system, thus leading to higher data rates, shorter one-way latencies and lower host CPU utilization. However, user-level networking complicates the scheduling. If a task is waiting for a message from one of its peer tasks, but that task is not being scheduled, then the CPU resource will be wasted because the CPU scheduler is unaware of that the task which is currently running is not doing useful work. As a result, the key to the efficiency of a scheduler lies in its ability to co-schedule the communicating tasks across different cluster nodes.

In order to achieve this goal, over the past decade, significant research literature has been devoted to scheduling parallel jobs on cluster platforms. A large number of strategies have been proposed, which can be broadly categorized into the following three classes based on how they co-schedule tasks across nodes:

- **Space Sharing Schemes**

  Space sharing schemes [13, 24] assign jobs side by side. Each node is dedicated to a job until it completes. Hence, tasks from the same job are always co-scheduled. The downside of these schemes is high system fragmentation and low utilization.

- **Exact Co-Scheduling Schemes**

  In order to address the inefficiencies of space sharing, exact co-scheduling, or gang scheduling [17, 7, 6, 22, 8], allows time-sharing on each node. It manages the scheduling via a two-dimensional scheduling matrix (called the *Ousterhout matrix*), with columns denoting number of nodes in the system, and rows denoting the time slices. Tasks from the same job are scheduled into the same row to guarantee the co-scheduling. At the end of each time slice, every node synchronizes with each other and switches to the next slice simultaneously. Figure 5 illustrates such a scheduling matrix which defines eight processors and four time slices. The number of time-slices (rows) in the scheduling matrix is the same as the maximum MPL on each node, which is in-turn determined by the memory size. As mentioned before, we usually keep it at a low to moderate level. In order to offset the synchronization cost across the entire cluster, exact co-scheduling schemes usually employ relatively large time slices, which will still lead to low system utilization.

	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
time-slice 0	J_1^0	J_1^1	J_1^2	J_1^3	J_1^4	J_1^5	J_1^6	J_1^7
time-slice 1	J_2^0	J_2^1	J_2^2	J_2^3	J_2^4	J_2^5	J_2^6	J_2^7
time-slice 2	J_3^0	J_3^1	J_3^2	J_3^3	J_4^0	J_4^1	J_5^0	J_5^1
time-slice 3	J_6^0	J_6^1	J_6^2	J_6^3	J_4^0	J_4^1	J_5^0	J_5^1

Figure 5: The scheduling matrix defines spatial and time allocation. J_i^k denotes the kth task of job i.

- **Dynamic Co-Scheduling Schemes**

  Dynamic co-scheduling schemes are proposed to further boost the system utilization [5, 1, 21, 15]. These schemes allocate multiple tasks (from different jobs) to a node, and leave the temporal scheduling of that node to its local CPU scheduler. No global synchronization is required, so co-scheduling is difficult to realize. However, they propose heuristics to re-schedule tasks that can use the CPU for useful work (computation, handling messages, etc) as much as possible. Based on when the re-scheduling is done, we can classify the dynamic co-scheduling schemes into the following two broad categories:

  - Re-scheduling on Demand. Schemes in this category try to re-schedule the tasks whenever certain events occur. The most common triggering events include:
    - No message arrival within a time period. After a task waits for a message for some time, and the message has not arrived within that time, the scheduling schemes suspect that its counter part is not being scheduled, so they will remove this task from the CPU and initiate a re-scheduling.
    - Message Arrival. The rationale here is that an incoming message indicates the counter part is being scheduled, so that the scheduler must schedule its destination task immediately if it is not already running.
  - Periodically Rescheduling. These schemes do not require the scheduler to react to every event in order to avoid thrashing. Instead, they periodically examine the running task and all the ready tasks,

and reschedule if the running task is busy-waiting while some other ready tasks have useful work to do.

Dynamic co-scheduling schemes involve low scheduling overhead, but the downsize is that it cannot guarantee co-scheduling.

4.2 Implementing the Cluster Scheduling Strategies

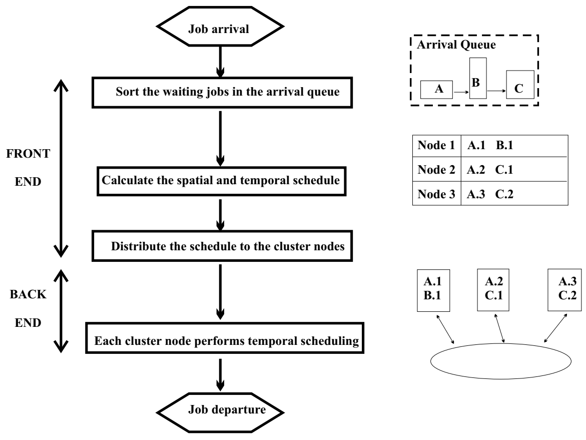

Job arrival
Sort the waiting jobs in the arrival queue
FRONT
END
Calculate the spatial and temporal schedule
Distribute the schedule to the cluster nodes
BACK
END
Each cluster node performs temporal scheduling
Job departure
Arrival Queue
A
B
C
Node 1 A.1 B.1
Node 2 A.2 C.1
Node 3 A.3 C.2
A.1 B.1
A.2 C.1
A.3 C.2

Figure 6: The flow chart of the cluster scheduler framework.

Numerous scheduling strategies have been proposed by earlier studies. However, this work is the first attempt to implement these different variations within a unified framework. Figure 6 summarizes the flow of this framework, and it also includes an example of the flow (shown on the right side). In order to facilitate the schedulers, we need to add a front-end logic to the system, while *ClusterSim* can serve as the back-end. The front-end can run on a different machine, or can run on any one of the cluster nodes. All the incoming parallel jobs are accepted by the front-end at first. As mentioned in Section 3, we adopt a low to moderate maximum MPL on each node. Hence, many jobs will wait before they can be scheduled, especially under the high load. The front-end sorts the waiting jobs according to their priority order. It also calculates the spatial schedule (i.e., where a task will be scheduled) and, in some cases, the temporal schedule (i.e., how the tasks on the same node will be scheduled). Finally, it will distribute the schedule to each back-end cluster node. If the temporal schedule is calculated by the front-end (as in exact co-scheduling), the back-end cluster node will just follow the schedule. Otherwise, it will make its own scheduling decision (as in dynamic co-scheduling).

4.2.1 Implementing the Front End

The meta scheduling is conducted by the front-end. All the incoming jobs are first accepted by the front-end. We use a CSIM process to implement the front-end logic.

First, the front end process must handle job arrivals and departures. The simulator can be driven by either a job trace or a synthetically generated workload. In either case, the front end knows a priori when the next arrival will take place, and let us assume `timer` denotes the gap until the next arrival. However, the front-end does not know when the next departure will happen. As a result, it waits for the event by executing `timed_wait(ESched, timer)`. The CSIM event `ESched` is set by one of the back-end cluster schedulers when one of the tasks running on that node completes. Then the front-end process will check if all the tasks from that job have completed.

Upon a job arrival, the front end process first inserts the job into the arrival queue. Different queue service disciplines are included in *ClusterSchedSim*, e.g., FCFS, Shortest Job First, Smallest Job First, First Fit, Best Fit, Worst Fit.

Upon a job arrival, as well as a task completion, the front end will re-calculate the spatial schedule by trying to schedule more waiting jobs into the system. Calculating the spatial schedule for space sharing and dynamic co-scheduling is rather simple: we need to just look for as many nodes that are having less than maximum MPL tasks as required by the job. On the other hand, it is much more challenging to calculate the schedule for exact co-scheduling schemes because its schedule determines both spatial and temporal aspects, and the schedule will determine the efficiency of the scheme. Further, the tasks from the same job must be scheduled into the same row, which makes it more challenging. We have implemented the exact co-scheduling heuristics that are proposed in our earlier work [30, 29, 28, 33, 32].

After the schedule is calculated, the front-end process will distribute it to each of the affected cluster nodes. If a cluster node needs to execute a new task, the front-end will create a CSIM process to execute the new application task, insert the new task into the task queue, and then notify the CPU scheduler if the new task has a higher priority.

The pseudo-code for the front end process is as follows.

```
DoFrontEnd(){
    while(1){

        timer = next job arrival - now;
        timed_wait(ESchedule, timer);

        if(timer expires){ // job arrival
            Insert into the waiting queue;
        }

        re-calculate the schedule;
        distribute the schedule to affected cluster nodes;
    }
}
```

4.2.2 Implementing the Back End using *ClusterSim*

In this section, we discuss how to enhance *ClusterSim* to implement the back-end for the three classes of scheduling schemes.

- **Space Sharing**

  We do not need to make any modifications to implement space sharing schemes. As soon as a back-end cluster node gets notified by the front-end, it will start executing the new task until it completes.

- **Exact Co-Scheduling**

  Figure 5 depicts the global scheduling matrix for exact co-scheduling schemes. Each node will get the schedule of the corresponding column. For instance, node P_0 will be executing tasks J_1^0, J_2^0, J_3^0, and J_6^0, in the specified order, spending T seconds to each task (T is the time slice).

  We must modify the CPU scheduler on each cluster node to execute the temporal schedule. Firstly, the time slice for each task will become T, not the default value from the original scheduler. Secondly, at the end of each time slice, the task will be moved to the end of the priority queue of the same level.

- **Dynamic Co-Scheduling**

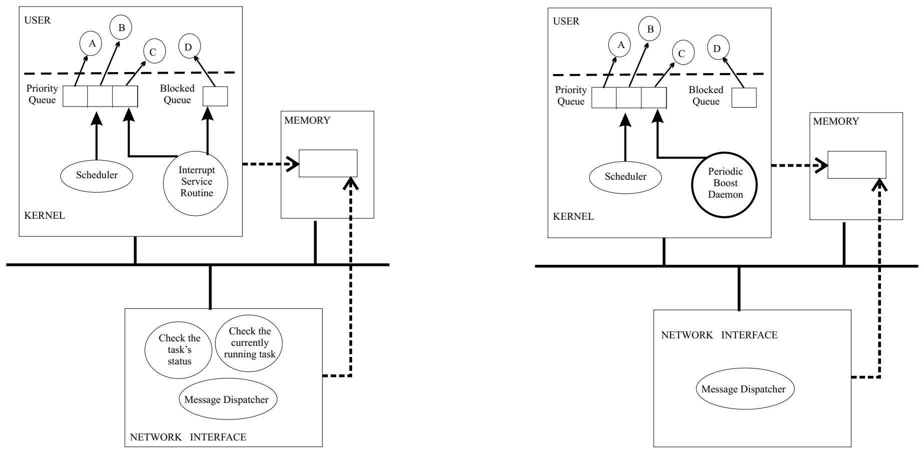
  In order to accommodate the re-scheduling of the tasks, the dynamic co-scheduling schemes need to modify the implementations of the application task, the NIC, and the operating system.

  As mentioned in Section 4.1, dynamic co-scheduling heuristics employ one or more of the following techniques: (1) on-demand re-scheduling because of no message within a time period, (2) on-demand re-scheduling because of a message arrival, and (3) periodic re-scheduling.

  The first technique requires modifications to the implementation of the application tasks. If the message has not arrived within a specified time period since the task starts waiting, the application task will block itself to relinquish the CPU while waiting for the message. This leads to a new implementation of `DoReceive`, which is shown below:

```
DoReceive(Message msg){

    WaitUntilScheduled();
    If (msg is not in the message queue){

        wait(EMsgArrival, timer);
        WaitUntilScheduled();
        if(event occurs){
            DoCompute(overhead of decomposing msg);
            Decompose the message;
        }

        if(timer expires){
            remove the task from the running queue;
            change the task status from running to blocked;
            wait(EMsgArrival);

            WaitUntilScheduled();
            if(event occurs){
                DoCompute(overhead of decomposing msg);
                Decompose the message;
            }
        }
    }
}
```

Please note that in the above pseudo-code, the task executes `wait(EMsgArrival)` twice. The first *wait* is a CSIM optimization to avoid polling (Section 3), and the second implements the operating system blocking.



(a) On-demand re-scheduling due to message arrival (b) Periodic re-scheduling

Figure 7: The enhancements of *ClusterSim*

If the second technique is employed, after the NIC process receives a message from the network, it first identifies its destination task. It then checks the status of that task by accessing a certain memory region. If the task is in block state (as a result of the first technique), the NIC will raise an interrupt, and the interrupt service routine will wake up the task, remove it from the block queue, boost its priority to the highest level, and insert it to the head of priority queue. If the destination task is in the ready queue, but not currently running, the network interface process will raise an interrupt as well. This interrupt service routine will boost the task to the highest priority level, and move it to the head of the priority queue.

In order to implement the second technique, we must add more modules to the NIC and the operating system. Figure 7(a) shows the new modules. In Figure 7(a), we refer to the NIC module in the original *ClusterSim* as the *message dispatcher*. In addition to the message dispatcher , we need two more modules to check the running task and the task status. These two operations involve very low overheads (around 2-3 ns), and the overheads will be overlapped with the activities of the message dispatcher. As a result, we do not use CSIM processes to implement them, and can save a significant amount of simulation time. The operating system in the original *ClusterSim* only has the CPU scheduler module. Now it will have an interrupt service routine module as well. The interrupt service routine is modelled using a CSIM process because its overheads are much higher (around 50-60 ns), and it must preempt other tasks to use the CPU. It has a higher priority than any application task. The interrupt service routine process will be waiting for the event `EInterrupt`, which will be set by the NIC. The overhead of the interrupt service routine is accommodated using `DoCompute(overhead)`.

The third technique re-schedules the tasks periodically. It requires a new module in the operating system: periodic re-scheduling daemon, which is implemented using a CSIM process (Figure 7(b)). The CSIM process wakes up periodically by executing `hold(period)`. Upon its wake up, if the currently running task is not doing anything useful (e.g., it is waiting for a message), then the daemon will examine every task in the ready queue, and pick one that has useful work to do. The overhead of the daemon is also accommodated by executing `DoCompute(overhead)`.

5 Performance Instrumentation and Configurable Parameters

ClusterSchedSim provides a detailed performance instrumentation package and a complete set of configurable parameters.

5.1 Instrumentation Package

ClusterSchedSim includes a set of instrumentation patches that offer detailed execution statistics at different levels. Using these statistics, one can easily locate the bottleneck of a scheduling strategy under certain system and workload configurations. To name just a few, from the job's perspective, we can obtain the following statistics:

- **Wait time vs. execution time**. From the user's perspective, the response time is an important performance measurement. Further, a job's response time can be broken down into two parts: the time between its arrival and its starting execution (wait time), and the time between its execution and its completion (execution time). Quantifying the time a job spends in these two phases can indicate the relative efficiencies of both the front-end and the back-end of the scheduler. This information is especially useful in comparing different classes of scheduling strategies.

- **Degree of co-scheduling** quantifies a scheduler's effectiveness in co-scheduling the tasks, especially for dynamic co-scheduling heuristics. In order to obtain this information, we define the *scheduling skewness*, which is the average interval between the instant when a task starts waiting for a message from its counterpart (at this time, its counterpart is not being scheduled) and the instant when its counterpart is scheduled. If a scheduler has a larger scheduling skewness, its degree of co-scheduling is poorer.

- **Pace of progress** defines the average CPU time a job receives over a time window W. We use this figure to measure the fairness of different dynamic co-scheduling heuristics. If a heuristic favors a particular type of jobs (e.g., those that are communication intensive), then those jobs will have a much higher pace of progress. On the other hand, heuristics that are fair will lead to uniform pace of progress for each job.

From the system's perspective, we can obtain many interesting statistics as well by detailed instrumentation:

- **Loss of capacity** measures the unutilized CPU resource due to the spatial and temporal fragmentation of the scheduling strategies. While jobs are waiting in the arrival queue, the system often has available resources. However, due to the fragmentation a scheduler has, these resources cannot be utilized by the waiting jobs. We call job arrivals/depatures the scheduling events. t_i is the gap between events i and $i-1$. f_i is 1 if there are waiting jobs during t_i, and 0 otherwise. n_i denotes the number of available slots in the system. If the maximum MPL is M, and there are m tasks on a node, then it has $M - m$ available slots. Suppose T is the temporal span of the simulation, and N is the cluster size, then loss of capacity is defined as $\frac{\sum f_i \times n_i \times t_i}{T \times N}$.

- **Loss of CPU resource** measures the ratio of the CPU resource, though utilized, but spent on non-computing activities. Specifically, computing includes the time tasks spend in their compute phase and the time they spend in composing/decomposing messages. Non-computing activities include context-switches, busy-waiting, interrupt overheads, and scheduling overheads. Compared to the loss of capacity, it says, from a fine granularity, how efficient the temporal scheduling phase of a scheduler (especially a dynamic co-scheduling heuristics) is.

5.2 Configurable Parameters

As mentioned earlier, *ClusterSchedSim* provides a complete set of system and scheduler parameters, which can be tuned to represent different system configurations, and to study the optimal settings for different scheduling strategies. To name just a few, the system parameters that can be configured are as follows:

- **Cluster size**. We can vary this parameter to study a cluster with thousands of nodes that represent the supercomputing environment, or a cluster with tens of nodes that represent a smaller but more interactive environment. Varying the cluster size can help one understand the scalability of different schemes.
- **Maximum multiprogramming level**. Intuitively, higher multiprogramming level can reduce a job's wait time, but it can also reduce the probability of co-scheduling especially for dynamic co-scheduling schemes. By varying this parameter, one can study how well different schedulers adapt to higher multiprogramming level.
- **System overheads**. We can also vary the overheads that are involved in various operating system activities such as context-switches, interrupts, and scheduling queue manipulations. These parameters can check how sensitive a scheduling scheme is to the system overheads.

As far as the scheduling schemes are concerned, we can vary the following parameters:

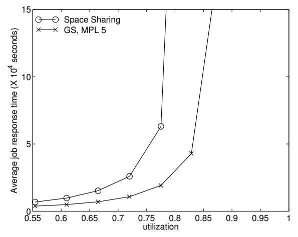
- **Time quanta**. For scheduling schemes that employ time-sharing on each node, time quanta play an important role in determining the performance. Time quanta that are too small will lead to thrashing, while overly large time quanta will have higher fragmentation.
- **Busy wait duration**. Under some dynamic co-scheduling schemes, a task blocks itself after busy-waiting for a message for a time period (the threshold). The duration of this busy-waiting period is essential to the success of the scheduler. If the duration is too short, then interrupts are needed later to wake up the message even though they will arrive shortly; if the duration is too long, then more CPU resource will be wasted.
- **Re-scheduling frequency**. In order to avoid wasting CPU resources, certain dynamic co-scheduling heuristics re-schedule the tasks periodically. This frequency determines the efficiency of such heuristics. If it takes place too often, then it may lead to thrashing, while too infrequent re-scheduling may lead to poor CPU utilization.

6 Case Studies: Using *ClusterSchedSim*

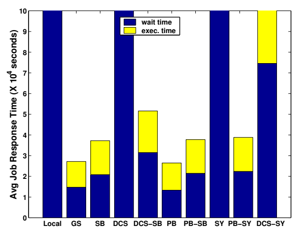
ClusterSchedSim is a powerful tool. We can use this simulation package: (1) to determine the best scheduler for a particular workload and system setting; (2) to profile the execution for a particular scheduler to locate its bottleneck; (3) to understand the impact of the emerging trends in computer architecture; (4) to tune the system and scheduler parameters to the optimal setting; and (5) to design and test new schedulers. In this section, we present case studies for the first three usages.

- **Comparison of scheduling strategies**

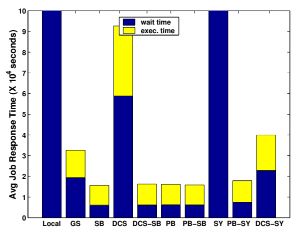
  It is an important problem to choose the best scheduling strategy for a particular workload under a certain system configuration. Using *ClusterSchedSim*, we can easily configure the system and workloads, and compare different heuristics under these configurations. Figures 8(a), (b) and (c) show such examples. In Figure 8(a), we use a real job trace collected from a 320-node cluster at the Lawrance Livermore National Laboratory to compare the space sharing scheme and gang scheduling. It clearly shows that gang scheduling performs better for this workload. In figures 8(b) and (c), we configure a much smaller cluster (32 nodes) and run two synthetic workloads, one communication intensive and the other I/O intensive. The results show that gang scheduling is better for communication intensive workloads (some dynamic co-scheduling schemes are very close though), and dynamic co-scheduling schemes are better for I/O intensive workloads.



(a) Space sharing vs. exact co-scheduling against a real job trace



(b) Exact co-scheduling vs. dynamic co-scheduling against a synthetic workload that is communication intensive



(c) Exact co-scheduling vs. dynamic co-scheduling against a synthetic workload that is I/O intensive

Figure 8: A comparison between different scheduling schemes

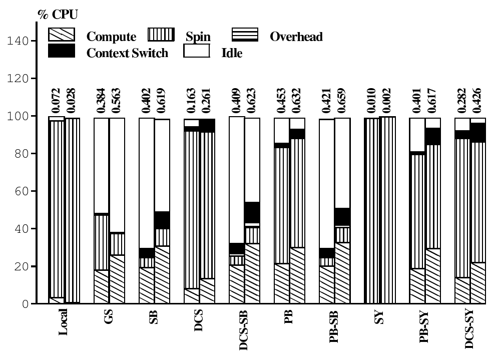


Figure 9: A detailed execution profile for gang scheduling and a set of dynamic co-scheduling schemes

- **Execution Profile**

  In order to explain why one particular scheme performs better than another, one needs to obtain the detailed execution profiles to understand the bottlenecks of each scheme. Using *ClusterSchedSim*, one can easily get such profiles. An example is shown in Figure 9, which shows how much time a CPU spends in computing, spinning (busy-wait), interrupt overheads, context-switch overheads and being idle respectively. Focusing on the two schemes: gang scheduling and periodic boost, we find that gang scheduling spends a smaller fraction of its execution in spinning (because of exact co-scheduling) but a much larger fraction being idle (poorer per node utilization). These statistics can easily explain the results presented in Figure 8. Such statistics can be easily obtained using a simulation tool, and neither an actual experimentation nor an analytical model can do the same.

- **Impact of the system parameters**

Scheme	$Q = 200ms, CS = 2ms$	$Q = 100ms, CS = 1ms$	$Q = 100ms, CS = 2ms$
GS	1.000	0.896	0.977
	$CS = 200us, I = 50us$	$CS = 100us, I = 25us$	$CS = 100us, I = 50us$
SB	1.000	0.648	0.642
DCS	1.000	0.902	0.904
DCS-SB	1.000	0.710	0.804
PB-SB	1.000	0.670	0.687
PB	1.000	0.89	

Table 1: Impact of System Overheads on Response Time

We do not only need to study the performance of a scheduler under one system setting, but it is also important to evaluate how the scheduler fares when the system parameters vary. *ClusterSchedSim* provides a set of configurable parameters, which can facilitate such studies. An example is shown in Table 1 which compares how gang scheduling and dynamic co-scheduling schemes perform when system overheads (e.g., context switch overheads, time quanta, and interrupt costs) change. It shows that gang scheduling will benefit from a smaller time quantum because it can reduce the system fragmentation. On the other hand, dynamic co-scheduling schemes that employ on-demand re-scheduling technique (e.g., SB ,DCS) benefit more from a lower interrupt cost or a lower context-switch cost because they incur a larger number of interrupts and context-switches compared to other schemes that employ periodic re-scheduling (e.g., PB).

7 Concluding Remarks

Last few decades have witnessed the rise of clusters among diverse computing environments, ranging from supter-computing centers to commercial (server) settings. The diversity in the workload characteristics and workload requirements has posed new challenges in job scheduling for such systems. A plethora of scheduling heuristics have been proposed. It is thus critical to conduct a comprehensive and detailed evaluation of these schemes. Due to the numerous parameters and its complexity, both actual implementations and analytical models are not appropriate to perform the evaluation.

We have developed *ClusterSchedSim*, a unified simulation framework, that models a wide range of scheduling strategies for cluster systems. The core of this framework lies in a detailed cluster simulation model, *ClusterSim*. *ClusterSim* simulates nodes across the cluster and the interconnect. Based on this core, we have built the following modules: (1) a set of parallel workloads that are often hosted on clusters; (2) scheduling strategies, including space sharing, exact co-scheduling and dynamic co-scheduling strategies; (3) detailed instrumentation patches that can profile the executions at different levels; and (4) a complete set of configurable parameters, both for the scheduling schemes and the system settings.

ClusterSchedSim is a powerful tool. It can be used to perform various studies in cluster scheduling. For example, one can determine the best scheduler under a certain workload and system setting; one can profile the execution of a particular scheduler to locate its bottleneck; one can quantify the impact of system parameters on a scheduler; one can tune the system and scheduling parameters to the optimal setting; one can design and test new scheduling schemes. On the other hand, *ClusterSchedSim* is modular enough that it can be easily extended to accommodate new modules and new scheduling strategies. For example, we have extended it by incorporating scheduling pipelined real-time workloads and a mixed stream of workloads with diverse quality-of-service requirements [31]. We have made *ClusterSchedSim* publicly available at site (*www.ece.rutgers.edu/ ∼yyzhang/ clusterschedsim*).

References

[1] A. C. Arpaci-Dusseau, D. E. Culler, and A. M. Mainwaring. Scheduling with Implicit Information in Distributed Systems. In *Proceedings of the ACM SIGMETRICS 1998 Conference on Measurement and Modeling of Computer Systems*, 1998.

[2] D. Bailey et al. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[3] M. Buchanan and A. Chien. Coordinated Thread Scheduling for Workstation Clusters under Windows NT. In *Proceedings of the USENIX Windows NT Workshop*, August 1997.

[4] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos. Multiprogramming on multiprocessors. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, pages 590–597, 1991.

[5] A. C. Dusseau, R. H. Arpaci, and D. E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proceedings of the ACM SIGMETRICS 1996 Conference on Measurement and Modeling of Computer Systems*, pages 25–36, 1996.

[6] D. G. Feitelson and L. Rudolph. Coscheduling based on Run-Time Identification of Activity Working Sets. Technical Report Research Report RC 18416(80519), IBM T. J. Watson Research Center, October 1992.

[7] D. G. Feitelson and L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grained Synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, December 1992.

[8] H. Franke, J. Jann, J. E. Moreira, P. Pattnaik, and M. A. Jette. Evaluation of Parallel Job Scheduling for ASCI Blue-Pacific. In *Proceedings of Supercomputing*, November 1999.

[9] D. Ghosal, G. Serazzi, and S. K. Tripathi. The processor working set and its use in scheduling multiprocessor systems. *IEEE Transactions on Software Engineering*, 17(5):443–453, May 1991.

[10] M. Herlihy, B-H. Lim, and N. Shavit. Low contention load balancing on large-scale multiprocessors. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, pages 219–227, June 1992.

[11] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *13th Symp. Operating Systems Principles*, pages 41–55, October 1991.

[12] S. T. Leutenegger and M. K. Vernon. The Performance of Multiprogrammed Multiprocessor Scheduling Policies. In *Proceedings of the ACM SIGMETRICS 1990 Conference on Measurement and Modeling of Computer Systems*, pages 226–236, 1990.

[13] D. Lifka. The ANL/IBM SP Scheduling System. In *Proceedings of the IPPS Workshop on Job Scheduling Strategies for Parallel Processing*, pages 295–303, April 1995. LNCS 949.

[14] Microelectronics and Computer Technology Corporation, Austin, TX 78759. *CSIM User's Guide*, 1990.

[15] S. Nagar, A. Banerjee, A. Sivasubramaniam, and C. R. Das. A Closer Look at Coscheduling Approaches for a Network of Workstations. In *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 96–105, June 1999.

[16] S. Nagar, A. Banerjee, A. Sivasubramaniam, and C. R. Das. Alternatives to Coscheduling a Network of Workstations. *Journal of Parallel and Distributed Computing*, 59(2):302–327, November 1999.

[17] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22–30, May 1982.

[18] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*, December 1995.

[19] S. K. Setia, M. S. Squillante, and S. K. Tripathi. Analysis of Processor Allocation in Multiprogrammed, Distributed-Memory Parallel Processing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):401–420, April 1994.

[20] J. P. Singh, W-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, 1991.

[21] P. G. Sobalvarro. *Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, January 1997.

[22] Thinking Machines Corporation, Cambridge, Massachusetts. *The Connection Machine CM-5 Technical Summary*, October 1991.

[23] A. Tucker. *Efficient Scheduling on Shared-Memory Multiprocessors*. PhD thesis, Stanford University, November 1993.

[24] L. W. Tucker and G. G. Robertson. Architecture and Applications of the Connection Machine. *IEEE Computer*, 21(8):26–38, August 1988.

[25] Specification for the Virtual Interface Architecture. `http://www.viarch.org`.

[26] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995.

[27] J. Zahorjan and C. McCann. Processor Scheduling in Shared Memory Multiprocessors. In *Proceedings of the ACM SIGMETRICS 1990 Conference on Measurement and Modeling of Computer Systems*, pages 214–225, 1990.

[28] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam. The Impact of Migration on Parallel Job Scheduling for Distributed Systems . In *Proceedings of 6th International Euro-Par Conference Lecture Notes in Computer Science 1900*, pages 245–251, Aug/Sep 2000.

[29] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam. Improving Parallel Job Scheduling by Combining Gang Scheduling and Backfilling Techniques. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 133–142, May 2000.

[30] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam. An Integrated Approach to Parallel Scheduling Using Gang-Scheduling, Backfilling and Migration. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):236–247, March 2003.

[31] Y. Zhang and A. Sivasubramaniam. Scheduling Best-Effort and Pipelined Real-Time Applications on Time-Shared Clusters. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 209–218, July 2001.

[32] Y. Zhang, A. Sivasubramaniam, J. Moreira, and H. Franke. A Simulation-based Study of Scheduling Mechanisms for a Dynamic Cluster Environment. In *Proceedings of the ACM 2000 International Conference on Supercomputing*, pages 100–109, May 2000.

[33] Y. Zhang, A. Sivasubramaniam, J. Moreira, and H. Franke. Impact of Workload and System Parameters on Next Generation Cluster Scheduling Mechanisms. *IEEE Transactions on Parallel and Distributed Systems*, 12(9):967–985, September 2001.