Preprint
UCRL-JC-138583-REV-1

A Coscheduling Technique for Symmetric Multiprocessor Clusters

A.B. Yoo and M.A. Jette

This article was submitted to
15th Annual International Parallel and Distributed Processing Symposium, San Francisco, California, April 23-27, 2001

September 18, 2000

A Coscheduling Technique for Symmetric Multiprocessor Clusters*

Andy B. Yoo and Morris A. Jette

Lawrence Livermore National Laboratory
Livermore, CA 94551
e-mail: {yoo2 | jette1}@llnl.gov

Abstract

Coscheduling is essential for obtaining good performance in a time-shared symmetric multiprocessor (SMP) cluster environment. However, the most common technique, gang scheduling, has limitations such as poor scalability and vulnerability to faults mainly due to explicit synchronization between its components. A decentralized approach called dynamic coscheduling (DCS) has been shown to be effective for network of workstations (NOW), but this technique is not suitable for the workloads on a very large SMP-cluster with thousands of processors. Furthermore, its implementation can be prohibitively expensive for such a large-scale machine. In this paper, we propose a novel coscheduling technique based on the DCS approach which can achieve coscheduling on very large SMP-clusters in a scalable, efficient, and cost-effective way. In the proposed technique, each local scheduler achieves coscheduling based upon message traffic between the components of parallel jobs. Message trapping is carried out at the user-level, eliminating the need for unsupported hardware or device-level programming. A sending process attaches its status to outgoing messages so local schedulers on remote nodes can make more intelligent scheduling decisions. Once scheduled, processes are guaranteed some minimum period of time to execute. This provides an opportunity to synchronize the parallel job's components across all nodes and achieve good program performance. The results from a performance study reveal that the proposed technique is a promising approach that can reduce response time significantly over uncoordinated time-sharing and batch scheduling.

1 Introduction

The most prevailing machine architecture for large-scale parallel computers in recent years has been the cluster of symmetric multiprocessors (SMPs), which consists of a set of SMP machines interconnected by a high-speed network. Each SMP node is a shared-memory multiprocessor running its own image of an operating system (OS) and often constructed using commodity off-the-shelf (COTS) components mainly due to economic reasons [1]. Continuous decrease in the price of these commodity parts in conjunction with the good scalability of the cluster architecture has made it feasible to economically build SMP clusters that have thousands of processors and total physical memory size on the order of Terabytes. The most prominent example of such very large-scale SMP clusters is the Department of Energy (DOE) Accelerated Strategic Computing Initiative (ASCI) project [5] machines [3, 4, 6]. For instance, the Lawrence Livermore National Laboratory (LLNL) ASCI SKY machine is an IBM SP2 with 5856 processors, total system memory size of 2.6 Terabytes and peak performance of 3.9 Teraflops.

*This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Efficiently managing jobs running on parallel machines of this size while meeting various user demands is a critical but challenging task. Most supercomputing centers operating SMP-clusters rely on batch systems such as LoadLeveler [15, 26] for job scheduling [32]. We may utilize a system efficiently using these batch systems, but high system utilization usually comes at the expense of poor system responsiveness with a workload consisting of long running jobs, as is typical of many large scale systems [11]. In a worst case scenario, for example, a user who wants to execute a simple debugging job that runs for only a few minutes may have to wait for several hours before sufficient resources are available to initiate the job. An alternative scheduling technique that improves the system responsiveness while improving fairness and freedom from starvation is *time-sharing*. With time-sharing, we can create virtual machines as desired to provide the desired level of responsiveness.

Due to the SMP-cluster architecture and the popularity of standardized message passing libraries such as MPI [20, 31], most applications running on an SMP-cluster follow the message-passing programming model. An important issue in managing message-passing parallel jobs in a time-shared cluster environment is how to coschedule the processes (or tasks) of each running job. Coscheduling here refers to a technique that schedules the set of tasks constituting a parallel job at the same time so that they can run simultaneously across all nodes on which they are allocated. When a parallel job is launched on an SMP-cluster, a set of processes are created on the nodes allotted to the job. These processes of the job usually cooperate with each other by exchanging messages. In most cases, two communicating processes do not proceed until both processes acknowledge the completion of a message transmission. Therefore, the interprocess communication becomes a bottleneck which may prevent the job from making progress if both sending and receiving processes are not scheduled at the time of the message transmission. Without coscheduling, the processes constituting a parallel job suffer high communication latencies due to spin-waiting periods and context switches. The ill effect on system performance of running multiple parallel jobs without coscheduling has been well documented [23]. It is very difficult to coschedule parallel jobs in a time-shared environment using local operating systems running independently on each node alone. A new execution environment is required in which parallel jobs can be coscheduled.

A few research efforts have been made to develop a technique with which the coscheduling can be achieved efficiently for SMP-clusters and networks of workstations (NOW). The simplest approach to coscheduling is a technique called *gang scheduling* [12, 13, 16, 17, 18, 19]. In gang scheduling, a matrix called gang matrix (or Ousterhaut matrix), which explicitly describes all scheduling information, is used. Each column and each row of a gang matrix represent a processor in the system and a time slice during which the processes in the row are scheduled to run, respectively. In other words, the entry in the ith row and jth column of the gang matrix contains the information on the process which is assigned to the ith processor and scheduled during the jth time slice. The coscheduling is achieved by placing all the processes of a job on the same row of the gang matrix. The gang matrix is usually maintained by a central manager running on a separate control host. Alternately, distributed database techniques may be used to maintain the gang matrix [18]. The gang matrix is distributed periodically or whenever there is a change. Each node receives a portion of the gang matrix that is pertinent to that node instead of the entire matrix. A small daemon process running on each node follows this well-defined schedule to allocate resources to processes on that node. This simple scheduling action guarantees coscheduling of parallel jobs due to the way the gang matrix is constructed. The gang scheduling technique is relatively simple to implement. A few successful gang scheduling systems have been developed and operational on actual production machines [16, 18].

However, gang scheduling has limitations, many of which stem from the support of a gang matrix explicitly describing scheduling activities for the entire system. First, correct coscheduling of jobs entirely depends on the integrity of the distributed scheduling information. If any of these schedules, which are transmitted through unreliable network, are lost or altered, it is highly likely that the jobs will not be coscheduled. Second, the gang scheduler's central manager is a single point of failure. Once the central

manager fails for any reason, the whole scheduling system fails. The last and the most serious drawback of the gang scheduling technique is its poor scalability. As the number of nodes in the system increases, not only the size of the gang matrix but also the number of control messages increases. These control messages convey various information such as the node status, the health of local daemons and the jobs running on each node, and so on. In many cases, the central manager is required to take appropriate actions to process the information delivered by a control message. Due to the excessive load imposed on the central manager, the gang scheduler does not scale well to very large system.

Another method for achieving coscheduling is a decentralized scheme called *dynamic coscheduling* (DCS) [2, 22, 27, 28]. In DCS, the coordinated scheduling of processes that constitute a parallel job is performed independently by the local scheduler, with no centralized control. Since there is no fixed schedule to follow in DCS, the local scheduler must rely on certain local events to determine when and which processes to schedule. Among various local events that a local scheduler can use to infer the status of processes running on other nodes, the most effective and commonly-used one is *message arrival*. The rationale here is that when a message is received from a remote node, it is highly likely that the sending process on the remote node is already scheduled. What this implies is that upon receiving a message, the local scheduler should schedule the receiving process immediately, if not already scheduled, to coschedule both the sending and receiving processes.

A few experimental scheduling systems based on this method have been developed [2, 22, 27]. All of these prototypes are implemented in an NOW environment, where workstations are interconnected through fast switches like Myrinet [21]. Interprocess communication is carried out using high-performance user-level messaging layers that support user-space to user-space communication [24, 29, 30] in these systems to reduce communication latency. The implementation of DCS under such special communication hardware and software involves programming switch firmware and network interface cards and developing communication libraries.

The DCS technique can achieve effective, robust coscheduling of processes constituting a parallel job and overcome the drawbacks of gang scheduling. However, current DCS implementations available may not be suitable for a large-scale SMP-clusters. They do not address the issues related to interprocess communication via shared-memory. Furthermore, the purchase, deployment, and maintenance of the special hardware and software for an SMP-cluster with thousands of processors is both difficult and expensive. Finally, fine-grain scheduling of the DCS scheme, which aims at quickly establishing coscheduling among communicating processes may work well in an NOW environment where context switches among processes of interactive and parallel jobs are very frequent, but it is not suitable for a large-scale SMP-cluster where interactive jobs are usually executed on a separate pool of reserved nodes. Frequent context switches among processes of parallel jobs will only hurt the performance due to increased communication latencies and memory management overhead, including both cache refresh and potentially paging.

In this paper, we propose and evaluate a novel coscheduling technique for an SMP-cluster. Our goal in this study is i) to design a technique that overcomes the shortcomings of previous approaches and can still achieve coscheduling efficiently on a large-scale SMP-cluster, and ii) to verify its viability through experiments.

Design criteria we used are as follows;

- Good scalability. The scheduler should be easily scalable to a very large SMP-cluster.
- Cost-effectiveness. Its implementation should not require to purchase any additional hardware or software.
- Portability. The system should be portable, and hence we should not make any changes to proprietary software such as local operating system or device driver.

- Low-overhead. The design has to be simple and efficient so that only minimal run-time overhead occurs.

To achieve these design goals, we have adopted the DCS approach, which allows us to eliminate any form of centralized control. The primary concern of the previous DCS schemes is boosting the priority of a receiving process as quickly as possible on a message arrival to establish immediate coscheduling. To accomplish this, they program network devices so that an incoming message can be trapped long before the receiving process gets scheduled. We believe that what is more important to improve overall performance is not reacting immediately to incoming messages but keeping the communicating processes coscheduled while they are running. In the proposed scheme, therefore, a process of a parallel job, once scheduled, is guaranteed to remain scheduled for certain period of time assuming that other processes of the job are either already scheduled or getting scheduled through message exchanges.

A mechanism to detect message arrivals is embedded into a message-passing library whose source code is freely available to the public, making the design portable. On a message arrival, the receiving process reports this to a local scheduler which makes appropriate scheduling decisions. Processes that are not scheduled need to be run periodically to trap incoming messages. An adverse effect of this sporadic execution of non-scheduled processes is that they may send messages triggering preemption of other coscheduled processes. This problem is resolved by attaching the status of sending process to each outgoing message.

We implement and evaluate the proposed coscheduling technique on a Compaq Alpha cluster testbed at LLNL. The results from our measurements show that the proposed coscheduling technique can reduce job response time as much as 50% over traditional time-sharing scheduling. The effect of various system parameters on performance is also analyzed in this study.

The rest of the paper is organized as follows. Section 2 describes the proposed technique and its implementation. Experiment results are reported in Section 3. Finally, Section 4 draws conclusions and presents directions for future research.

2 Design and Implementation

2.1 Basic Design

The proposed coscheduler for SMP-clusters is designed based on two principles. First, it is essential for achieving coscheduling to make correct decisions on when and which processes on each node to schedule. Second, it is crucial to maximize coscheduled time as a portion of scheduled time for the processes on each node. If preemption occurs too frequently, the parallel job's throughput will suffer from an increase in spin-wait time at synchronization points, cache refresh delays, and potentially paging delays.

A key factor in scalable coscheduler design is decentralization of scheduling mechanism. An ideal scalable coscheduler should not employ any centralized control or data structures, but completely rely on autonomous local schedulers. Our coscheduling technique also follows such decentralized approach. Without any global information on the status of all the processes in the system, each local scheduler has to determine the status of remote processes and coschedule the local processes with their remote peers. Exchanging control messages that contain process status information among local schedulers is not a scalable solution. An alternative is to use certain implicit local information to infer the status of remote processes. Such implicit information includes response time, message arrival, and scheduling progress [2].

Like all the previous work [2, 22, 27, 28], our coscheduler also depends on message arrival to acquire status information of remote processes. The message arrival refers to the receipt of a message from a remote node. When a message is received, this implies the sending process is highly likely to be currently scheduled. Therefore, it is crucial to quickly schedule the receiving process to achieve coscheduling.

In order to implement this idea, we need a mechanism which detects the arrival of a message and reports this to the local scheduler. This message trapping mechanism is performed at user-level in our design to fulfill one of our design goals: cost-effectiveness. The implementation can be easily done by inserting a few lines of code into a small number of application program interfaces (APIs) provided by an open-source message-passing libraries like MPICH [14]. This code notifies the local scheduler of message arrival through an interprocess communication (IPC) mechanism. The user-level message trapping allows us to avoid the purchase of additional hardware and software and the need to do any device programming. In addition, the use of publicly available software makes our design more portable.

The functions of local scheduler include maintaining information such as the process ID (pid) and the status of processes assigned to the node and scheduling appropriate processes for coscheduling. When a process is about to start or terminate execution, the process reports these events to the local scheduler along with its own pid. When notified of these events, the local scheduler adds/removes the pid received to/from the data structure it manages. Similarly, when a message arrives, the receiving process reports this with its pid to the local scheduler, which then responds by performing appropriate scheduling operations. Here the report of message arrival serves as a request to local scheduler to schedule the receiving process.

The group of processes constituting the same parallel job on each node serve as a scheduling unit. That is, whenever a process is scheduled, its peer processes on the same node are also scheduled together. This is to establish the coscheduling more quickly. Since the peer processes of a recently scheduled process will be eventually scheduled via message-passing, we can reduce the time that takes to establish the coscheduling by scheduling the entire group of peer processes at once instead of scheduling them individually. More importantly, this strategy may increase the number of messages to other unscheduled processes on remote nodes and hence achieve the coscheduling more quickly.

In an attempt to reflect the second design principle, we ensure that all the newly scheduled processes run for a certain period of time without being preempted. This guarantees that each parallel job, once coscheduled, runs being coscheduled at least for the given time. We use a predetermined threshold value for the *guaranteed minimum execution time* (GMET), but the value may be calculated dynamically as well. Receiving a scheduling request from a user process, the local scheduler checks if the currently scheduled processes have run at least for the GMET. If so, a context switch is performed. Otherwise, the request is ignored.

While message arrivals cause user process to send scheduling requests, this can result in job starvation. The starvation is prevented by a timer process that periodically sends a context switch request to the local scheduler. The local scheduler, on receiving this request, performs a context switch in a similar fashion to a scheduling request from a user process. In this case, however, the local scheduler selects a new job to run. During a context switch, the local scheduler selects a job which hasreceived the least CPU time as the next job to run to maintain fairness. The local scheduler keeps track of the CPU time each job has consumed to facilitate this scheduling process. We use a time-slice on the order of seconds in this research, adhering to the second design principle. The rationale behind such a long time-slice is to insure the job establishes coscheduling and executes coscheduled for some minimum time.

There is a critical issue in conjunction with the user-level message trapping that needs to be addressed. In order for a user process to trap incoming messages, the process itself has to be scheduled. Otherwise, message arrivals will never be detected and reported to the local scheduler. The local scheduler in our design, therefore, periodically schedules all the jobs for a very brief period of time to detect any message arrival. A serious side effect of this simple approach is that the local scheduler may receive *false scheduling requests*. A false scheduling request can be sent to the local scheduler when a user process receives a message from a remote process which is scheduled for the message-trapping purpose. These false scheduling requests may results in wrongful preemption of coscheduled processes and significant performance degradation. We solve this problem by attaching the status of sending process to every

outgoing message. With the status of sending process available, the receiving process can easily decide whether a context switch is needed or not on each message arrival. The design of the coscheduler is shown in Fig. 1.

2.2 Implementation

The proposed coscheduler described has been implemented and evaluated on an eight-node Compaq Alpha cluster testbed running Tru64 Unix 5.0 at LLNL. Each node has two Compaq Alpha EV6 processor operating at 500 MHz with 1 GB of main memory. The implementation exercise has involved only minor modifications to a user-level message-passing library and the development of two very simple daemon processes. The implementation of the proposed coscheduler is described in detail in what follows.

2.2.1 MPICH Library

We have modified an open-source message-passing library, MPICH [14], to implement the user-level message trapping as well as the process registry operations. The MPICH is a freely-available, high-performance, portable implementation of the popular MPI Message Passing Interface standard. We have chosen the MPICH library mainly due to its popularity and easy access to its source code.

A few new functions are added to the MPICH library in this implementation. These functions notify the local scheduler when certain events occur through IPC. Those requests are accompanied by the pid of sending process. The functions are summarized in Table. 1.

MPI_Register is invoked during the initialization phase of an MPI process. The MPI_Register, when invoked, sends a CMDREG request to local scheduler. An MPI application id is also sent along with the request to notify the local scheduler of which MPI job the process belongs to. The local scheduler creates a small shared-memory region at the time a process is registered through which the process learns its scheduling status. Similarly, MPI_Terminate is invoked during the finalization phase of the MPI process and sends CMDOUT request to the local scheduler. The terminating process is then removed from the list of processes assigned to the local scheduler. MPI_Schedule sends CMDSCH request along with its pid to local scheduler in an attempt to schedule itself.

A few MPICH functions need to be modified as well to incorporate the capability to handle messages carrying process status information. These functions are `net_send`, `net_recv`, and `net_recv_timeout`. We have modified `net_send` in such a way that a single byte representing the status of sending process is attached to each outgoing message. The actual length of the message is increased by one byte. The additional byte is prefixed to the message, because the user can specify arbitrary message length at the receiving end. If we postfix the status information to an outgoing message, and a different message length is given in a receiving routine, the information can be lost or even worse, incorrect status information can be extracted by the receiving process. By always sending the status information before actual message body, we can preserve and retrieve correct status information regardless of the message length specified by a user.

With the modifications made to `net_recv` and `net_recv_timeout`, the status information is separated from each incoming message and actual message is passed to whichever routine invoked these functions. An early scheduling decision, which is whether a context switch is needed or not, is made at this level using the status information received. That is, if the sending process is currently scheduled and the receiving process is not, a context switch is necessary. A request for context switch is sent to the local scheduler by calling MPI_Schedule.

2.2.2 Class Scheduler

In our implementation, we use the Compaq Tru64 UNIX priority boost mechanism called *class scheduler* [7] to schedule processes of a parallel job. With the class scheduler, we can define a class of system entities and assign certain percentage of CPU time to the class. The class scheduler ensures that access to the CPUs for each class does not exceed its specified limit. The entities that constitute a class can be users, groups, process groups, pids, or sessions. There may be a number of classes on a system. A database of classes, class members, and the percentage of CPU time for the class is maintained by the class scheduler. The database can be modified while the class scheduler is running, and the changes take effect immediately.

The kernel has very little knowledge of class scheduling. A class, in the kernel, is an element in an array of integers representing clock ticks. A thread that is subject to class scheduling has knowledge of its index in the array. Each time the thread uses CPU time, the number of clock ticks used is subtracted from the array element. When the count reaches zero the thread is either prevented from running altogether or, optionally, receives the lowest scheduling priority possible.

When class scheduling is enabled, a daemon is started. The daemon wakes up periodically and calculates the total number of clock ticks in the interval. Then, for each class in the database, it divides the total by the percentage allocated to the class and places the result into an array. When finished, the array is written to the kernel.

The class scheduler provides APIs which system developers can use to enable and disable class scheduling, create and destroy a class, add and remove a class member, change the CPU percentage allotment for a class, and so on. Using these APIs, we define a class of pids for each group of processes constituting the same MPI job. We use the application id of the MPI job as the name of the class. Processes of an MPI job can be scheduled at the same time to the class representing those processes. For example, if we allocate 100% of CPU time to a class, only the processes defined in the class will receive CPU time. The local scheduler performs a context switch by swapping the CPU percentage of two classes of processes that are being context-switched.

It was mentioned that all the processes, whether currently scheduled or not, need to receive some CPU time periodically to trap incoming messages at the user-level. One way of doing this is to let the local scheduler periodically allocate 100% of CPU time to each of the classes in the system for a very short time. This is a feasible solution, but it may burden the local scheduler as the number of jobs assigned to the node increases. Therefore, we rely on the class scheduler to achieve the user-level message trapping. In our implementation, 1% of CPU time is allocated to each unscheduled class so that the processes in the class are executed for very short periods of time, and remaining CPU percentage is allocated to a scheduled class. Therefore, if there are n classes in the system, $(n-1)\%$ of CPU time is allocated to $n-1$ classes, and a scheduled class receives $(100 - n + 1)\%$ of CPU time. The class scheduler is configured to strictly adhere to these percentage allocations and time allocated to a class which is not used by that class is not used by other job classes. Whenever a class is created or destroyed, the CPU allotment to the scheduled class is adjusted accordingly.

2.2.3 Daemons

Two daemons are used to perform process scheduling in this implementation, a timer and a scheduler daemon. The task of the timer daemon is to periodically send a request for context switch to scheduler daemon to enforce time-sharing. The timer daemon simply repeats the process of sleeping for a predetermined interval, which works as time-slice, followed by sending the context-switch request to the scheduler daemon.

The scheduler daemon performs key scheduling operations such as managing process and MPI job

status and changing the priority of processes. The scheduler daemon is a simple server that acts upon requests from either user process or the timer daemon. Those requests are sent to the scheduler daemon via shared-memory IPC, since the IPC occurs only within a single node and the shared-memory provides the fastest IPC mechanism. A shared-memory region, through which requests are sent, is created when the scheduler daemon starts execution.

The main body of the scheduler daemon consists of a loop in which the daemon waits for a request and then execute certain operations corresponding to the request received. There are five requests defined for the scheduler daemon: CMDREG, CMDOUT, CMDCSW, CMDSCH, and CMDDWN.

The CMDDWN request terminates the scheduler daemon. On receiving this request, the scheduler daemon removes the shared-memory region created for IPC and then exits. CMDREG and CMDOUT requests are associated with the process management operations. An MPI process sends CMDREG to notify that the process is about to start execution. When receiving this request, the scheduler daemon creates an entry in the process table it maintains. An entry in the process table contains information about a process such as its pid and the MPI job that the process belongs to. The table also contains scheduling information on the MPI job assigned to the node. Such information on an MPI job includes the job id, the number of member processes, the time when the job was scheduled and preempted, and a pointer to a shared-memory region from which processes of the job read the job's status. The table is organized in such a way that there is a link between each MPI job and all the processes that constitute the job. When an MPI job is registered for the first time, the scheduler daemon performs two things. First, it creates an entry for the job in the process table. Next, a class is created using the job's application id as the class name. The pid of the requesting process is added to the table and the class created. A newly created class receives 1% of CPU time initially. The CPU time allotment of scheduled class is adjusted accordingly when a new class is created.

CMDOUT, a request issued upon process termination, does the reverse of CMDREG. Receiving CMDOUT request, the scheduler daemon removes the pid of the sending process from the process table and the corresponding class. When the last process terminates, corresponding process table entries and class defined for the terminating job are destroyed, and the CPU time allotment of scheduled class is adjusted.

The CMDCSW request is issued by the timer daemon. Upon receiving this request, the scheduler daemon simply swaps the CPU time allotment of currently scheduled job with that of the next job to be selected. The CMDSCH request also causes a context switch, but it is issued by a user process upon a message arrival. The scheduler daemon, upon receiving this request, first determines whether the context switch is allowed by checking if currently scheduled job has consumed at least the GMET. If so, the requesting job is scheduled by adjusting the CPU time allotment. Otherwise, the request is discarded.

The pseudo codes for the daemons are given below.

Timer Daemon:

1. Create a pointer to a shared-memory region for IPC.
2. loop
   Sleep for n seconds, where n is predetermined value for time-slice.
   Send CMDCSW to scheduler daemon.
   end loop

Scheduler Daemon:

1. Create a shared-memory region for IPC.
2. Initialize process table and system queue.
3. loop

```
Wait for a request.
switch (request)
  case CMDDWN:
    Destroy classes, if there are any.
    Remove the shared-memory region.
    Exit.
  case CMDREG:
    if there is no entry for MPI job corresponding to the requesting process, then
      Create an entry in the process table and perform initialization for the job.
      Create a new class for the job and assign 1% of CPU time to the class.
      Create a shared-memory region for the communication of job status.
      if there are no other job in the system, then
        Schedule the newly created job.
      else
        Adjust the CPU time allotment of a scheduled job.
      end if
    end if
    Add the sending process to the process table and corresponding class.
  case CMDOUT:
    Remove requesting process from the process table and the class the process belongs to.
    if the number of processes in an MPI job corresponding to the requesting process is zero, then
      Destroy the entry and the class defined for the MPI job.
      if the job is currently scheduled, then
        Schedule the next job in the queue, if there is one.
      else
        Adjust the percentage of CPU time allocated to a scheduled job.
      end if
    end if
  case CMDCSW:
    Schedule a job that has received the least CPU time by adjusting the CPU time allotment.
  case CMDSCH:
    if currently scheduled job, if exists, has run at least for the GMET, then
      Schedule the requesting job by adjusting the CPU time allotment.
    end if
end switch
```

---

3 Experimental Results

3.1 NAS Parallel Benchmarks

We have selected the NAS Parallel Benchmarks (NASPBs) [8, 9, 10, 25] to study the performance of proposed coscheduling technique. The NASPBs are a widely-recognized suite of scientific benchmarks developed at NASA Ames Research Center to study the performance of parallel supercomputers on scientific applications. The benchmarks consist of eight benchmark programs, each of which focuses on some important aspect of highly parallel supercomputing for aerophysics applications.

Most of those benchmark programs are written in Fortran. There are three standard sizes for the

NASPBs, known as classes A, B, and C. The nominal benchmark sizes for these classes can be found in [25]. Class A and class C represent the smallest and the largest problem sizes, respectively. The NASPBs can be compiled for different number of processors as well. The eight problems consist of five kernels and three simulated computational fluid dynamics (CFD) applications. These benchmarks are briefly described in what follows, where five kernel benchmarks are discussed first.

Embarrassingly Parallel (EP) The first of five kernel benchmarks is an embarrassingly parallel problem. In this benchmark, two-dimensional statistics are accumulated from a large number of Gaussian pseudo-random numbers, which are generated according to a particular scheme that is well-suited for parallel computation. This is typical of various Monte Carlo applications. This problem involves minimal communication amongst the processes.

Multigrid (MG) The second kernel is a simplified multigrid benchmark that solves a three-dimensional Poisson partial differential equation (PDE). This problem is simplified in the sense that it has constant instead of variable coefficients as in a more realistic application. This code is a good test of both short and long distance highly structured communication.

Conjugate Gradient (CG) In this benchmark, a conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel is typical of unstructured grid computations in that it tests irregular long-distance communication.

FFT PDE (FT) In this benchmark a three-dimensional PDE is solved using FFTs. This kernel performs the essence of many spectral methods. It is a good test of long-distance communication performance.

Integer Sort (IS) This benchmark tests a sorting operation that is important in particle method codes. This type of application is similar to particle-in-cell applications of physics, wherein particles are assigned to cells and may drift out. The sorting operation is used to reassign particles to the appropriate cells. This benchmark tests both integer computation speed and communication performance. This is the only code implemented in C in NASPB.

Lower-Upper Diagonal (LU) The first of simulated CFD applications is the LU solver. It does not perform an LU factorization, but employs a symmetric successive over-relaxation (SSOR) numerical scheme to solve a regular-sparse, block lower and upper triangular system. The way the SSOR procedure operates requires the use of a relatively large number of small messages.

Scalar Pentadiagonal (SP) and **Block Tridiagonal (BT)** In the SP and the BT benchmarks, multiple independent systems of nondiagonally dominant scalar pentadiagonal equations and block tridiagonal equations are solved. SP and BT are representative of computations associated with the implicit operations of CFD codes. In both benchmarks, the granularity of communications is kept large and fewer messages are sent.

3.2 Performance Evaluation

In this research, we have conducted a performance study on an 8-node Compaq Alpha SMP cluster testbed to evaluate the proposed coscheduler. Three workloads, each exhibiting different degree of communication intensity, are used to evaluate the performance under various message traffic conditions. Here, the communication intensity of a job is measured by the number of messages exchanged during the course of execution. The first workload consists of randomly selected class A and class B NASPBs and represents a workload with moderate message traffic, under which the communication intensity of jobs varies to a great extent. The second workload is constructed from the three most communication-intense NASPBs (LU, SP, and BT) to represent a workload with heavy message traffic. The third workload consists of

only the EP NASPB in which there is little communication between processes. The three workloads are summarized in Table 2. The performance measure of interest in this study is mean job response time.

Fig. 2 compares the performance of the new coscheduling technique with that of uncoordinated time-sharing. The time slice and GMET used in this experiment are 15 and 5 seconds, respectively. For all three workloads, the new coscheduler shows better or comparable response time behavior compared to the uncoordinated time-sharing. As expected, the best performance is achieved when the message traffic is heavy (Workload 2). Here, the mean job response time is reduced by 50% when the proposed coscheduling technique is used. The measures for mean job response time are almost identical for the Workload 3. This is because the effect of uncoordinated scheduling of the processes constituting a parallel job on performance is not significant when the message traffic is light. These results are a strong indication that the proposed technique is a promising approach to coscheduling, which can efficiently improve the performance of parallel jobs under various message traffic conditions.

Fig. 3 shows the response-time behavior of the proposed coscheduling technique and uncoordinated time-sharing scheduling for varying degree of multiprogramming (DOM). The time-slice and the GMET lengths are the same as in Fig. 2. The workloads used in this experiment are summarized in Table 3. We increase the load to the system by adding a new set of randomly selected NASPBs to existing workload, as DOM increases. In this experiment, only class A benchmarks are considered to minimize the effect of paging overhead. As Fig. 3 indicates, the proposed coscheduling scheme obtains the best performance gain (85 % reduction in response time) when the DOM is 2. This is because without coordinated scheduling, processes of parallel jobs tend to block very frequently waiting for their communicating peers to be scheduled, whereas our technique minimizes the blocking time considerably through coscheduling of the processes. However, the performance gain decreases as the DOM increases. The reason for this is that as the number of time-shared jobs increases, the waiting time due to blocking is compensated by increased computation and communication interleave, while coscheduling the parallel jobs becomes increasingly difficult. This experiment highlights the need for an additional process allocation mechanism, which can control and limit the DOM of the system.

Fig. 4 plots the average job wait time under batch scheduling and proposed coscheduling technique with varying time slice length and DOM. In this experiment, we submitted 100 NASPBs to the system at once and measured the wait (or queueing) time of each job. The workload consists of 98 class A NASPBs and two class C NASPBs (LU). GMET is set to 2 seconds in this experiment. A separate script starts new jobs in such a way that desired DOM is maintained. Fig. 4 shows that the proposed coscheduling technique reduces the average job wait time by as much as 41% over simple batch scheduling. The poor performance of the batch scheduling is due to what is known as the 'blocking' property of the first come first served (FCFS) scheduling discipline [33]. That is, under the FCFS policy a job has to wait until all preceding jobs finish their execution, and therefore, its wait time is the total of the execution time of all the preceding jobs. On the other hand, the proposed technique, with its time-sharing and coscheduling capability, is not affected by the blocking property and hence performs very well in this experiment. Furthermore, closer examination reveals that the average job wait time increases as the DOM increases. As already discussed in Fig. 3, this is because it becomes increasingly difficult to establish coscheduling as the DOM increases.

Figures 5 and 6 examine the effect of the GMET and the time-slice lengths on performance of the proposed coscheduler, respectively. Fig. 5 shows the response-time behavior of the coscheduler for three workloads described in Table 2 as the length of GMET varies. The time-slice length in this experiment is set to 30 seconds. The results reveal that the GMET length does not affect the performance of the coscheduler for workloads 1 and 3, where the communication intensity is relatively low. On the other hand, the GMET length has significant effect on the system performance for the workload 2 in which the communication intensity is high. If the GMET length is set too small for such a workload with high communication intensity, coscheduling a parallel job is extremely difficult because some of the processes

that constitute the parallel job are highly likely to be preempted before the coscheduling is established due to the increased message traffic. If the length of GMET is too large, the coscheduler fails to quickly respond to incoming context-switch requests from remote processes, and this degrades the performance. However, the performance degradation in this case is not as severe as in the previous case, since the large GMET length still prevents excessive context-switches. This is clearly visible in Fig. 5, where the response-time curve for the workload 2 sharply drops and then increases as the GMET length changes from 2 through 5 seconds. For the GMET lengths greater than 5 seconds, the response-time behavior remains almost unchanged, since most of context-switch requests are discarded with such long GMETs and the performance is strictly governed by the length of the time slice used.

Fig. 6 plots the changes in response time as the time-slice length varies for the three workloads. The GMET length is set to 5 seconds. As expected, the performance of the coscheduler is hardly affected by the time-slice length for workload 3. However, the response time continuously increases for both workloads 1 and 2 with time-slices greater than 15 seconds. This can be explained in conjunction with the results from the previous experiment. Since there is no global control in our design, which could schedule all processes of a parallel job concurrently, a situation in which scheduled processes that constitute different parallel jobs contend for scheduling of their communicating peers occurs quite frequently. If the GMET length is set too large (as in this experiment), the context-switch requests through messages sent to remote nodes are discarded and hence the parallel jobs eventually stall until a context-switch is initiated by one of the timer daemons. Consequently, the waiting time of each job increases as the time-slice length increases.

As shown in Fig. 5 and Fig. 6, the GMET and the time-slice lengths can have significant effect on performance and hence, selecting optimal values for these parameters is critical. However, such optimal values are highly workload-dependent and therefore, careful workload analysis must be conducted. The experiment results also suggest that in general short time-slice and long GMET lengths are favorable to obtaining good system performance.

4 Concluding Remarks and Future Study

Efficiently coscheduling processes of message-passing parallel jobs on a time-shared cluster of computers poses great challenges. In this paper, we propose a new technique for a cluster of SMP machines, which offers a scalable, portable, efficient, and cost-effective solution for achieving coscheduling. The proposed technique uses message arrivals to direct the system towards coscheduling and hence requires no explicit synchronization mechanism. Unlike other coscheduling schemes based on message arrivals, however, incoming messages are caught by user processes to avoid any need for additional hardware and software. The status of a sending process is attached to each outgoing message so that better scheduling decisions can be made at the receiving end. Processes are guaranteed to run at least for a certain period of time once scheduled to ensure that each parallel job makes progress while being coscheduled. This design principle is the key to the success of our coscheduler in obtaining high performance. Experimental results indicate that the proposed technique is a promising and inexpensive approach to efficient coscheduling, which can improve the performance significantly over uncoordinated time-sharing and batch scheduling.

There are two interesting directions for future research. The performance of our coscheduler is greatly affected by the length of time-slice and GMET. The results from a preliminary analysis reveal that short time-slice and long GMET lengths are beneficial to achieving good system performance. We plan to conduct more rigorous study on the effect of these parameters on performance in the future study. In addition, tests of this technique in heterogeneous computing environment could provide the ability to execute even larger problems.

References

[1] T. E. Anderson, D. E. Culler, and D. A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54 64, Feb. 1995.

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

[3] ASCI Blue Mountain. `http://www.lanl.gov/asci/bluemtn/bluemtn.html`.

[4] ASCI Blue Pacific. `http://www.llnl.gov/platforms/bluepac`.

[5] ASCI Project. `http://www.llnl.gov/asci`.

[6] ASCI Red. `http://www.sandia.gov/ASCI/Red`.

[7] Class Scheduler. `http://www.unix.digital.com/faqs/publications/base_doc`.

[8] D. H. Bailey et al. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5:63 73, 1991.

[9] D. H. Bailey et al. The NAS Parallel Benchmarks. Technical Report NASA Technical Memorandom 103863, NASA Ames Research Center, 1993.

[10] D. H. Bailey et al. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Dec. 1995.

[11] D. H. Bailey et al. Valuation of Ultra-Scale Computing Systems: A White Paper, Dec. 1999.

[12] D. J. Feitelson and M. Jette. Improved Utilization and Responsiveness with Gang Scheduling. In *IPPS'97 Workshop on Job Scheduling Strategies for Parallel Processing, Vol. 1291 of Lecture Notes in Computer Science*, pages 238–261. Springer-Verlag, Apr. 1997.

[13] H. Franke, P. Pattnaik, and L. Rudolph. Gang Scheduling for Highly Efficient Multiprocessors. In *Proc. Sixth Symp. on the Frontiers of Massively Parallel Processing*, Oct. 1996.

[14] W. Gropp and E. Lusk. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22:54 64, Feb. 1995.

[15] IBM Corporation. *LoadLeveler's User Guide, Release 2.1.*

[16] J. E. Moreira et al. A Gang-Scheduling System for ASCI Blue-Pacific. In *Proc. Distributed Computing and Metacomputing (DCM) Workshop, High-Performance Computing and Networking '99*, Apr. 1999.

[17] M. Jette. Performance Characteristics of Gang Scheduling in Multiprogrammed Environments. In *Proc. SuperComputing97*, Nov. 1997.

[18] M. Jette. Expanding Symmetric Multiprocessor Capability Through Gang Scheduling. In *IPPS'98 Workshop on Job Scheduling Strategies for Parallel Processing*, Mar. 1998.

[19] M. Jette, D. Storch, and E. Yim. Timesharing the Cray T3D. In *Cray User Group*, pages 247–252, Mar. 1996.

[20] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.

[21] N. J. Boden et al. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[22] S. Nagar, A. Banerjee, A. Sivasubramaniam, and C. R. Das. A Closer Look At Coscheduling Approaches for a Network of Workstations. In *Proc. 11th ACM Symp. of Parallel Algorithms and Architectures*, June 1999.

[23] J. K. Ousterhout. Scheduling Technique for Concurrent Systems. In *Int'l Conf. on Distributed Computing Systems*, pages 22–30, 1982.

[24] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Meessages (FM). In *Proc. Supercomputing '95*, Dec. 1995.

[25] S. Saini and D. H. Bailey. NAS Parallel Benchmark (Version 1.0) Results 11-96. Technical Report NAS-96-18, NASA Ames Research Center, Nov. 1996.

[26] J. Skovira, W. Chan, H. Zhou, and D. Lifka. The Easy-LoadLeveler API Project. In *IPPS'96 Workshop on Job Scheduling Strategies for Parallel Processing, Vol. 1162 of Lecture Notes in Computer Science*, pages 41–47. Springer-Verlag, Apr. 1996.

[27] P. G. Sobalvarro. *Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors.* PhD thesis, Dept. of Electrical Engineering and Compuer Science, Massachusetts Institutute of Technology, 1997.

[28] P. G. Sobalvarro and W. E. Weihl. Demand-based Coscheduling of Parallel Jobs on Multipr ogrammed Multiprocessors. In *Proc. IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 63–75, Apr. 1995.

[29] T. von Eicken and A. Basu and V. Buch and W. Vogels. U-Nnet: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. 15th ACM Symp. on Operating System Principles*, Dec. 1995.

[30] T. von Eicken and D. E. Culler and S. C. Goldsten and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proc. 19th Annual Int'l Symp. on Computer Architecture*, Dec. 1995.

[31] The MPI Forum, May 1995. `http://www.mcs.anl.gov/mpi/standard.html`.

[32] Top 500 Supercomputer Sites. `http://www.netlib.org/benchmark/top500.html`.

[33] B. S. Yoo and C. R. Das. A Fast and Efficient Processor Management Scheme for k-ary n-cubes. *Journal of Parallel and Distributed Computing*, 55(2):192–214, Dec. 1998.

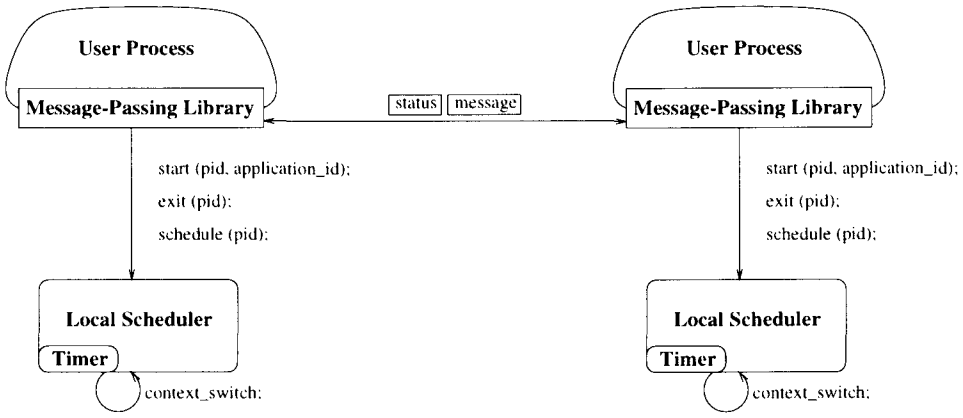

Figure 1: The design of proposed coscheduler.

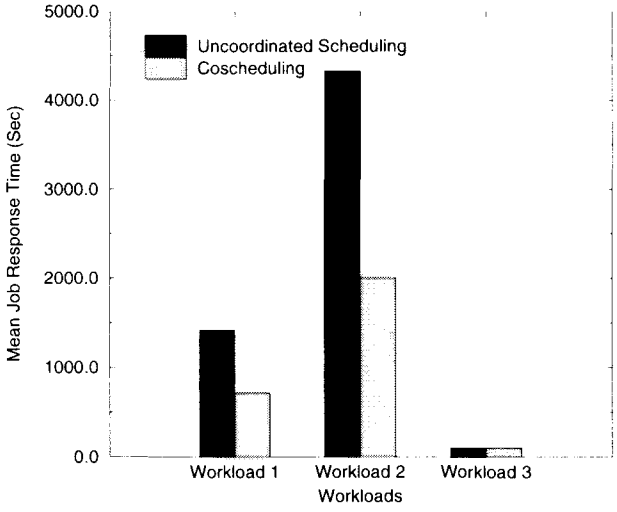

Figure 2: Comparison of mean job response time for different workloads (Time slic e = 15 seconds and GMET = 5 seconds).

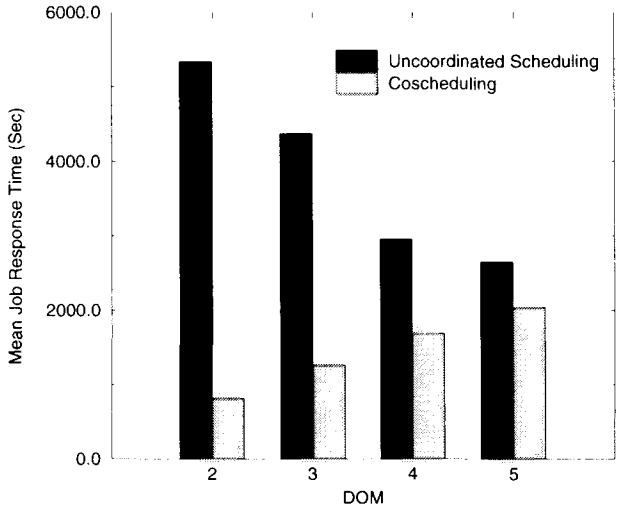

Figure 3: Comparison of mean job response time for different degree of multiprogramming (DOM) (Time slice = 15 seconds and GMET = 5 seconds).

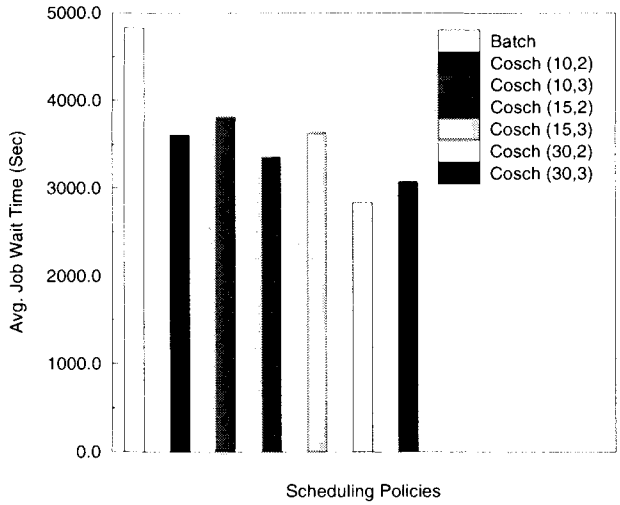

Figure 4: Comparison of average job wait time under batch and proposed coscheduling technique with different time slice legth and DOM (Cosch (time slice, DOM)).

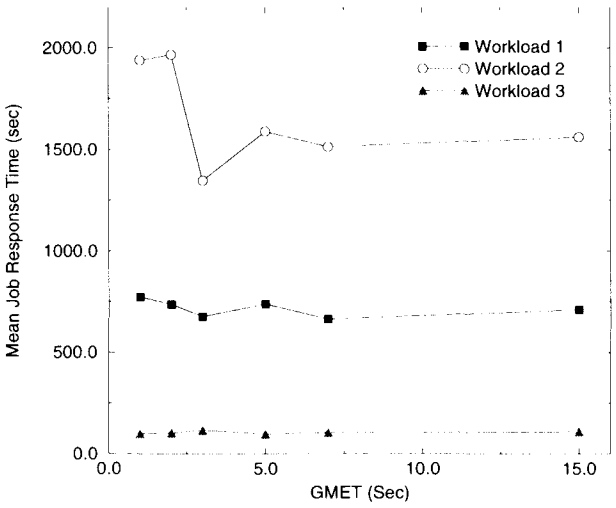

Figure 5: The effect of the GMET on performance (Time slice = 30 seconds).

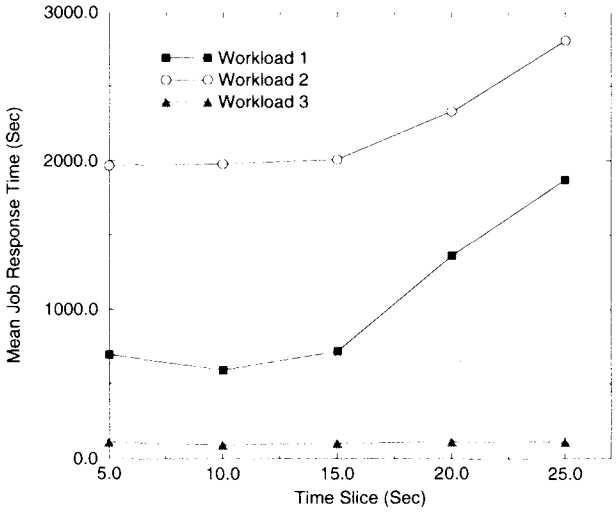

Figure 6: The effect of time slice on performance (GMET = 5 seconds).

Function	Request	Event	Local Scheduler Action
MPI_Register	CMDREG	Process Initialization	Register requesting process
MPI_Terminate	CMDOUT	Process Termination	Remove requesting process
MPI_Schedule	CMDSCH	Message Arrival	Schedule requesting process, if allow ed

Table 1: Summary of newly defined MPI functions.

Workload	Benchmarks
Workload 1	bt.B.4, ep.B.8 (2), bt.A.4, sp.A.9, mg.A.2, lu.B.4
Workload 2	bt.A.4 (2), lu.A.2 (2), sp.B.9, sp.A.9, sp.A.4, lu.B.2
Workload 3	ep.A.2 (2), ep.A.4 (2), ep.B.8, ep.B.4 (2), ep.A.8

Table 2: Three workloads used.

DOM	Benchmarks
2	sp.A.16, sp.A.9
3	sp.A.16, sp.A.9, lu.A.8
4	sp.A.16, sp.A.9, lu.A.8, cg.A.16, ft.A.8
5	sp.A.16, sp.A.9, lu.A.8, cg.A.16, ft.A.8, ep.A.8

Table 3: The workloads used for each DOM.