Coscheduling on Cluster Systems

Xin Liu, AbdulAziz Al-Shammari, Noman Noor Mohammad , Chuong Le

Abstract Coordinated scheduling of parallel jobs across the nodes of a multiprocessor system is known to produce benefits in both system and individual job efficiency. Without coordinated scheduling, the processes constituting a parallel job would suffer high communication latencies because of processor thrashing. With clusters connected by high-performance networks that achieve latencies in the range of tens microseconds, the success of co-scheduling becomes a more important factor in deciding the performance of fine-grained parallel jobs.

We study the existing co-scheduling techniques in a distributed environment, and mainly analyze Implicit and Dynamic Coscheduling as they are more suited to the modern cluster systems like NOW (Network Of Workstations) and HPVM (Hi Performance Virtual Machine). After a careful study of these co-scheduling techniques, we present possible improvements for co-scheduling in cluster systems, integrating the Implicit and Dynamic Co-scheduling approaches to improve performance and introducing a variation of the Stride scheduling technique to provide better fairness among processes. The suggestions seem to be promising; however experiments are needed to verify the their effects in real cluster systems.

1 Introduction

Scheduling parallel applications in a distributed environment, such as a cluster of workstations, remains an important and unsolved problem. Coordinated scheduling of parallel jobs across the nodes of a multiprocessor system is known to produce benefits in both system and individual job efficiency. Without coordinated scheduling, the processes constituting a parallel job suffer high communication latencies because of processor thrashing. With clusters connected by high-performance networks that achieve latencies in the range of tens microseconds, the success of co-scheduling becomes a more important factor in deciding the performance of fine-grained parallel jobs. Also, the shared resources in the cluster allow a mix of workloads such as parallel, interactive and sequential jobs to coexist. This places a new demand on the scheduler for a fair and efficient method of allocating system resources.

Section 2 describes the cluster of workstations environment and the reasons of its increasing popularity. Section 3 defines the co-scheduling technique and points out its necessity in the cluster environment. This section also specifies the main desirable features of an ideal scheduler. Section 4 is a study of the existing co-scheduling techniques for a distributed environment. Section 5 gives a classification of the co-scheduling techniques into two categories: explicit and dynamic, and gives reasons of why the dynamic approaches are more suitable to the modern cluster systems like NOW (Network Of Workstations) and HPVM (Hi Performance Virtual Machine). Section 6 concentrates on the analysis of the dynamic approaches, i.e. Implicit and Dynamic Co-scheduling; it presents their similarities as well as an analysis of their performance and fairness. Section 7 suggests possible improvements for co-scheduling in cluster systems by integrating the Implicit and Dynamic Co-scheduling approaches to improve performance and introduces a variation of the Stride scheduling technique to provide better fairness among processes in the system.

2 Background: Cluster of workstations

First of all, we will give some background on the Cluster of workstations. This will provide a much clearer picture of the challenges involved with coscheduling in such an environment.

A Cluster of Workstations is a network of workstations that is used as a large computer system (a supercomputer) out of small mass-produced computers. It is capable of running a task that is much bigger than that would be feasible on a single workstation. Anderson et al. point out that “NOW provides a means to exploit the full desktop building block, including the operating system and applications. It provides both the interactive performance of a traditional workstation as well as the aggregate resources of the whole network for demanding sequential and parallel programs” [1]. In the context of this survey, “Cluster of Workstations” and “Network of Workstations” refer to the same system.

Since resources in engineering workstations are idle most of the time, we can utilize them and pool all those available resources for more complex tasks. Usually we can harness a large collection of resources on a single program from the resources of the whole network: large memory, large disk, and large processing power. The result is that high performance parallel applications can be run in everyday computing network infrastructure.

In most cases, the operating systems in network stations can be modified or extra layers can be built upon existing operating systems in workstations to provide the operating system for the whole cluster. This is an advantage over Parallel Processor systems that require a new operating system built specifically for that computing system.

The improvement in network latency and the availability of large number of workstations with idle resources are the two factors that make clusters of workstations possible. In [10] Dusseau et al. state that by connecting the workstations on low-latency networks, one can build incrementally-scalable, cost-effective, and highly-available shared servers capable of supporting a large number of users running a variety of workloads.

3 Coscheduling

3.1 Definition of co-scheduling

The supercomputer with the parallel processing power from individual workstations is capable of running parallel programs with concurrent processes running and contributing to the overall solution of a particular problem. Sobalvarro et al. point out that without scheduling, the processes constituting a parallel job suffer high communication latencies because of processor thrashing [9]. The executing processes can be blocked waiting for responses from those that are currently not executing. Without an efficient scheduling technique, the machine can spend most of its time context switching when a process is scheduled to execute and its interacting counterpart is de-scheduled from execution. Therefore, a scheduling policy must be implemented to efficiently exploit this parallel environment by coordinating the individual processes. Among the various scheduling techniques which will be briefly examined in the following part of this survey, co-scheduling proves to be one of the most promising techniques.

Rudolph and Feitelson define co-scheduling as the technique that schedules the interacting activities in an application so that all these activities execute simultaneously on distinct processors [7]. Crovella et al. point out that in co-scheduling, "no process is forced to wait for another that has been preempted, processes can communicate without an intervening context switch" [2]. In general,

co-scheduling will reduce communication latencies and increase system efficiency by reducing spin-waiting periods and context switches [9].

However, there is still concern that in case of more than one application running in the system, all of these applications should be scheduled, and the context switch from one application to another may result in cache flush and reload. Those reloads are expensive. Furthermore utilization may be reduced in case of applications that have a wide varying range of parallelism, or if processes cannot be evenly assigned to time-slices of the machine [9]

Co-scheduling for clusters is challenging because it must balance the two opposing demands of parallel and local computation [9]

3.2 An ideal coscheduling algorithm

After going through the several co-scheduling techniques, we found out that there is are criteria desired for an ideal coscheduling technique. We thought to put this up front, before discussing the techniques, so that the reader has an idea of the goals or aims of the co-scheduler.

The first criterion is efficiency. The scheduling algorithm should schedule processes in such a way that other resources in the cluster as a whole should be efficiently utilized. CPU time should be allocated for processing instead of busy waiting or wastefully context-switching.

Secondly, Sobalvarro et al. consider that the scheduling policy must be non-intrusive. It should not impose on the programmer a particular programming model (i.e. the requirement that all parallel applications must be coded in a multithreaded fashion). It should also be flexible: the exact number of processors on a machine should not be taken into consideration when the programmer or the compiler-writer writes the code. Furthermore, the algorithm should be dynamic: it can adapt to changes in workload and communications among processes [9].

Finally, the scheduler should be fair in the allocation of CPU time to processes: no processes should be scheduled to run most of the time at the expense of other processes. Applications with different degree of parallelism should also be evenly scheduled for CPU time.

4 Existing co-scheduling techniques

Most of the scheduling techniques for multiprogrammed parallel systems can be classified as time-sharing, space-sharing or a combination of both. In time-shared systems all processors are dedicated to a job for a certain time-slice. Although this provides a guaranteed service to the jobs, it leads to an under- utilization of resources for jobs that do not require all processors. In contrast, in a space-shared system the processors are partitioned, and each job is allocated to a certain partition according to its needs. Although this increases utilization of the system resources as compared to time-sharing, just simple space-sharing does not work, as the number of processes is usually much larger than the number of processors.

The co-scheduling techniques described below use a combination of time-sharing and space-sharing techniques with special regards for synchronization of certain processes.

4.1 Distributed Hierarchical Control

In Distributed Hierarchical Control [4], the processors are arranged in a binary tree-like structure with the processors as leaves of the tree and the controllers as roots of the subtrees. Each subtree is

considered as a partition, and the controller of the corresponding subtree is responsible for coordinating and scheduling the processors within the subtree. The jobs are allocated to the smallest partition in which the number of processors is greater than or equal to the number of processes required for that job. This helps achieve load balancing. The algorithm also tries to keep processor fragmentation - the number of idle processors on a partition - low, thus increasing resource utilization. This is achieved by allocating alternate jobs to the processors, in a partition, that are idle.

If global coordination is required, for processes that need to be co-scheduled, the controllers of different partitions coordinate with each other via the controller in the next higher level (i.e. their parent). Otherwise, the local controller can schedule its processes according to local considerations. The processors are space-shared among different partitions on the same level. In addition, time-sharing is done among the partitions whose controllers are at different levels. The controllers on the same level lie on the same time quantum, since the processors to which they are mapped are distinct. Time-sharing also occurs among the different jobs assigned to the same partition.

The distributed hierarchical control structure has certain advantages over a centralized structure with a global scheduler such as it can coschedule alternates and provides the advantages of both time-sharing and space-sharing techniques. However, the multi-context switching that occurs at the end of the time-slice, for processes belonging to the same job, leads to huge overheads. Also, it requires the user to partition the job into threads as part of program development.

4.2 Process Control

The main idea of Process Control [5] is to gradually equate the number of runnable processes of each application to the number of available processors assigned to that application. In this technique, there is a concept of controllable versus uncontrollable application. A controllable application is one that can dynamically control the amount of its runnable processes according to the number of processors assigned to it. On the other hand, an uncontrollable application has no such control on its runnable processes. When a controlled application first executes, the root process of that application notifies the central server about the application's existence. The server then checks to see how many processors are available by subtracting the number of processors assigned to uncontrollable applications from the total number of processors. Then it divides the available processors equally among the controllable applications. The controllable applications then adjust the number of their runnable processes according to the number of processors assigned to them by the server. If there are more applications than the number of processors, then the scheduler time-shares the processors among the applications.

The advantage of Process Control is that there is fewer overheads associated with multi-context switching, as threads of the same application share a common address space. However, this requires the user to develop the application using multi-threads. Also, this technique provides fairness only if the system comprises only of controlled applications. If there are any uncontrolled applications in the system, then they end up consuming most of the processor cycles, and this leads to fairness problems. Another problem is that in client/server applications this multi-threaded approach maybe impractical, as although a lot of communication occurs between the clients and the server, the clients do not have access to all of the server's data.

4.3 Gang Scheduling:

In gang scheduling [6], all the components of a parallel job are executed at the same time. A parallel job is identified as a gang and each of its components is known as a gang member. When a

job is gang-scheduled, each of its gang members is allocated to a distinct processor and thus the gang executes in parallel. A Distributed Hierarchical Control (DHC) structure [4] is used for its global scheduling strategy. The partitions in DHC structure whose controllers are on the same level are identified as a class. A gang is assigned to a class that has the minimum number of processors greater than or equal to the number of its gang members. The classes have a local scheduler, which can have its own scheduling policy. Each class is assigned a fraction of time for which it can utilize the processors. The higher the class is in the hierarchy, the higher the fraction of time it is assigned. If there are more than one gang assigned to a class, then they are time-shared within the fraction of time allocated to that particular class. At the end of a time slice, a gang context switch occurs, in which all the members of that gang are suspended and another gang takes over.

The gang scheduling technique works well for parallel jobs that have a lot of inter-process communication. However, there is a fairness problem. In the case of mix jobs, the parallel jobs having a lot of inter-process communication consume a lot of the processor cycles, even if they are fewer in number to other jobs (e.g. interactive jobs). This is because such parallel jobs fall into the higher classes which are assigned higher fractions of time. Also, the multi-context switching that occurs when the gang members are preempted at the end of their time slice can lead to large overheads.

4.4 Runtime Identification of Activity Working Sets

Runtime identification of activity working sets, presented by Feitelson and Rudolph in [7], is a method that tries to identify the sets of interacting activities at run-time. This identification is done for the purpose of co-scheduling such sets of interacting activities. The method uses certain shared data structures, known as communication objects, which are used in communicating between activities. The user identifies these communication objects. The access rates of the communication objects along with the activities involved are maintained in a separate data structure. If the value of the access rate of a particular communication object is above a specified threshold, then the activities involved are co-scheduled. This is derived from the notion that the activities that access the same objects at a high rate thereby interact frequently, and would therefore benefit from co-scheduling. The set of activities that are co-scheduled are known as *'activity working set.'* The activity working sets change dynamically during the course of execution and reflect the changed communication among the activities.

The activity working sets allocate processors according to the largest set that fits the available processors. The co-scheduled activities continue to run until either their time quantum expires, or all of them are busy waiting for an interaction. In such a situation, multi-context switching occurs, and the other activity working sets are co-scheduled. After all the activity working sets have been co-scheduled, the remaining activities are scheduled in an uncoordinated manner.

Although this method lifts the restriction of developing programs in a particular way, the user still needs to identify the communication objects. Also, the multi-context switching of the co-scheduled activities requires close coordination and thus leads to larger overheads. Another drawback is that this technique uses past experience (access rates) for co-scheduling activities. Thus, this technique will not work effectively in cases where the interaction between activities occurs for a short time, as the case in client/server applications.

4.5 Implicit Coscheduling

Dusseau et al. in [8] give out a distributed co-scheduling algorithm for time-sharing communicating processes in a cluster of workstations. Independent schedulers on each workstation

coordinate parallel jobs through local events that occur naturally within the communicating applications. They infer remote state from local events, such as message arrivals, and make decisions that lead to a global goal. The mechanism that achieves the co-scheduling is '*two-phase spin-blocking'*. With two-phase spin-blocking, a process spins for some amount of time, and if the response arrives before the spin time expires, it continues executing. If the response is not received within the threshold (spin time), the process voluntarily relinquishes the processor so that a competing process can be scheduled. The key is how to use the implicit information to decide the spin time. There are three kinds of implicit information available for implicit co-scheduling to be used for adjusting the spin-time. They are response time, message arrival, and scheduling progress.

The spin-time consists of three components. Firstly, the *baseline* spin-time: a process must spin a baseline spin-time to ensure that processes stay coordinated if already in such a state. In general, the baseline equates to the round-trip time of the network, plus potentially a context switch time. Secondly, the *local* spin-time: the process may need a modified spin-time in accordance with a local *cost-benefit* analysis. For example, if the destination process will be scheduled soon in the future, it may be beneficial to spin longer and avoid the cost of losing coordination and being rescheduled later. The local spin-time can be set to the additional penalty that the local process will pay when it is rescheduled in the future when the desired response arrives: the time to wake-up and schedule a process on a message-arrive. Thirdly, the *pairwise* spin-time: performing cost-benefit analysis not only for a single process, but also between pairs of communicating processes, improves the performance of implicit co-scheduling. Consider a pair of processes: the receiver who is performing a two-phase spin-blocking while waiting for a communication operation to complete, and a sender who is sending a request to the receiver. When waiting for a remote operation, the process spins for the baseline and local spin-time, while recording the number of incoming messages. If the average interval between requests is sufficiently small, the process assumes that it is beneficial in the future to remain scheduled and continues to spin for an addition time of pairwise spin-time.

Performance was evaluated by comparing workload completion times of the entire workload to completion times under an idealized gang scheduler that used spinning message receipt and 500-millisecond time-slices. The results show that implicit co-scheduling algorithm has good performance for both synthetic programs that communicate either continuously or in bulk-synchronous style, as well as real applications with a mix of communication characteristics [8].

However, the conclusion of the paper [8] reports that the scheduler favors more coarse-grained jobs over more fine-grain jobs, and that fairness can be a problem. These issues are discussed in more detailed on section 6.

4.6 Dynamic Coscheduling

Dynamic Coscheduling (DCS) provided by Sobalvarro et al. in [9] also exploits communication between processes to deduce which processes should be co-scheduled and to make co-scheduling more efficient. It is a bottom-up, emergent scheduling approach that exploits the key observation that only those threads which are communicating need to be co-scheduled. Dynamic co-scheduling is straightforward to implement; an arriving message not addressed to the currently running process can trigger an interrupt on the network interface device and then a scheduling decision is made. This decision is based on a wide variety of factors (e.g. system load, last time run, etc.). They also use the spin-blocking synchronization method.

The implementation of Dynamic Coscheduling includes three parts:

1. Monitoring communication/thread activity: when a message arrives, the scheduler should not act if the message is for the current running process, this work is performed on the network interface card.
2. Causing scheduling decisions: if the message received is not for the currently running process and it is fair to preempt the currently running process, the interrupt routine in the device driver of network will raise the destination process's priority to the maximum allowable priority (for user-mode timesharing processes) and place it at the front of the dispatcher queue. Flags are set to ensure that the operating system scheduler runs on exits from the interrupt routine, causing a scheduling decision based on the new priority. This will cause the process receiving the message to be scheduled unless the process that was running had a higher priority than the maximum allowable priority for user mode.
3. Make a decision whether to preempt: In dynamic co-scheduling, an incoming message's process is scheduled only if doing so would not cause unfair CPU allocation. They chose to implement fairness by limiting the frequency of priority boosts, and then limiting the frequency of preemption. To achieve this, it must measure how much CPU time each process has gotten recently.

Performance was evaluated on Illinois Fast Messages (FM), a user-level message layer. They chose three-performance metrics: job response time, CPU time and fairness. Experiment show that performance was close to ideal for the case of fine-grained processes using spin-blocking message receipt, CPU times were nearly the same as for batch processing, and DCS reduced job response times by up to 20% over implicit scheduling while maintaining near-perfect fairness.

There are two limitations in the experiment that must be pointed out: the first is that they can't run multiple parallel jobs on the FM, so the data is mainly about a single parallel job running on a cluster of workstations in competition with serial processes. The second limitation is that they did not implement a means of achieving fairness automatically. In section 6, more detailed will be given concerning these issues.

5 Coscheduling for cluster system

We can classify all these techniques mentioned earlier, into two classes: explicit co-scheduling and dynamic co-scheduling. The former includes Distributed Hierarchical Scheduling, Process Control, Gang Scheduling, and Runtime Activity Working Set (RAWSI). The latter includes Implicit Coscheduling and Dynamic Coscheduling. The criterion is that the former techniques all require simultaneous multi-context switch among processors. In fact, the former schedulers are more suitable for Massively Parallel Processors (MPP) or Symmetric Multi Processors (SMP) systems as opposed to cluster systems such as NOW and High Performance Virtual Machine (HPVM).

Explicit co-schedulers have numerous disadvantages that are accentuated by the cluster environments:

1. Explicit co-scheduling requires a centralized master to determine a fixed schedule for running parallel jobs at the same time across workstation. Ensuring that context switches occur simultaneously on all the processors across the clusters is difficult and will increase the cost of each context switch.
2. Explicit co-scheduling does not interact well with interactive jobs or parallel jobs performing I/O. The reason for this is that in order to get good performance for the coscheduling of parallel jobs, explicit co-scheduling has to use large time-slices to reduce the context switches. This leads to an increase in response times of interactive jobs.

3. Explicit co-scheduling usually requires processes that should be co-scheduled to be identified a priori (except the RAWSI), so it is inflexible and can not be applied to distributed client/server applications.

The promising time-sharing approaches for scheduling parallel applications in a cluster are Implicit Coscheduling or Dynamic Coscheduling. Unlike explicit co-scheduling, both are completely distributed, requiring no global coordination; both have potential for working well with a mix of parallel and interactive jobs and parallel jobs performing I/O; finally, neither requires that communicating processes be statically identified. From these reasons, we conclude that only the dynamic co-scheduling techniques are better for cluster systems. The following sections give a detailed analysis of those two techniques suggest some improvements for their performance and provide a solution to the problem of fairness.

6 Comparison of Implicit & Dynamic Coscheduling: Performance and Fairness

6.1 Similarities of Dynamic & Implicit Coscheduling

At first glance, these two techniques seem to have great differences. Implicit Coscheduling uses messages exchanged between processes to infer the states of remote processing and to make the scheduling decision. Dynamic Coscheduling uses the message arriving as the direct requirement of the process to be scheduled. However after a careful study of their implementations, we found that they are very similar in nature. The similarities exist in the following points.

6.1.1 Both of them use spin-blocking synchrony

For the Implicit Coscheduling, spin-blocking is its key mechanism to achieve co-scheduling. From response time, message arrival, and scheduling progress, the local scheduler can infer remote process states, and thus make decisions that lead to a global goal. There are three components. Firstly, the amount of time a process should spin to keep the coordinated jobs in synchrony. Secondly, the amount of time a process should keep spinning so that the benefit of spinning is greater than blocking. Thirdly, the amount of time the process should continue spinning when it is receiving messages from another process, considering the impact of this process on others in the parallel job. In fact, all the work of Implicit Coscheduling is to decide how long a process should spin before relinquishing the CPU.

Looking through the paper of Dynamic Coscheduling, we find that it also uses the spin-blocking technique to improve performance. The only difference is that in Dynamic Coscheduling, the problem has been simplified by using a fixed spin-time (i.e. 1600us).

6.1.2 Both of them use the priority boost mechanism

For Dynamic Coscheduling, when a message for a process that is not current running arrives and it is fair to preempt current process, DCS will raise the target process's priority to the maximum allowable priority for user-mode timesharing processes and place it at the front of the dispatcher queue. Flags are set to ensure that the [Solaris 2.4] scheduler will make a new scheduling decision based on the new priorities.

Implicit Coscheduling also benefits from priority boost. We know that Implicit Coscheduling is implemented on Solaris 2.6 and the Solaris 2.6 scheduler has the characteristic of priority boost.

Solaris 2.6 priority-decay schedulers maintain a queue for each priority level and service individual run queues in round-robin order, except in the case of transitions from the sleep queues, where the newly awakened process is placed at the end and is given a priority boost. So the destination process of the arriving message has great possibility to be scheduled immediately. Thus the effect of a message arrival, on the destination process, in Implicit Coscheduling partly resembles that in Dynamic Coscheduling due to the priority boost.

So the two key mechanisms of Implicit & Dynamic Coscheduling coexist in both of them and contribute to the performance of co-scheduling. Now we can acclaim that they should have comparable performance.

6.2 Performance analysis of Implicit Coscheduling & Dynamic Coscheduling

Both of them can achieve good co-scheduling of the parallel jobs. But we noticed an interesting difference between them. The Implicit Coscheduling technique works well at corse-grained communication applications, while the Dynamic Coscheduling technique suffers performance degradation with the decrease in communication. At the same time, under Implicit Coscheduling, the coarse-grained applications will get more CPU time when competing with fine-grained applications. This is also seems to be a contradiction. As we know, with the decrease in communication, there is less information for the scheduler to make the co-scheduling decisions; and thus for Dynamic Coscheduling, it is logical to become ineffective at that situation. The same behavior should be exhibited by Implicit Coscheduling.

If we can find why Implicit Coscheduling works well at coarse-grained applications, we can improve the performance of Dynamic Coscheduling. In fact, after careful analysis, we found that Implicit does not work better than Dynamic for the coarse-grained jobs as it seems to be (more detailed is giving in subsection 6.2.2.).

6.2.1. Different Performance Metrics

As we mentioned before, Implicit Coscheduling evaluates the performance by using the slowdown of the last job to be completed in the workload in comparision to an idealized, perfect model of explicit co-scheduling. In the mean time Dynamic Coscheduling uses Job Response Time, CPU Time and Fairness as performance metrics. The different performance metrics make it difficult to compare the performance of the two techniques directly. The most plausible criterion to use is the slowdown in the batch execution of the workload.

6.2.2.Different Workloads definitions

We mainly focus on the definition of a coarse-grained application in both techniques; it is represented by the interval between two communications. In Implicit Coscheduling it is about 100ms while in Dynamic Coscheduling it is about 0.78ms. As we know, with the decrease in communication, it becomes less important to co-schedule the related processes. One may infer that there is a certain threshold for the communication interval, in which, when there is more communication than that threshold, the success of co-scheduling dominates the performance. However, if there is less communication than that threshold, the necessity for co-scheduling becomes trifle. Therefore the above-mentioned contradiction may be because Implicit Coscheduling was used in too coarse applications, and they cannot ensure the threshold value found in Dynamic Coscheduling. We need more detailed test data to verify this inference.

6.3 Fairness

Fairness is a very important issue when there are multiple jobs and when there are interactive applications. Neither of the Implicit or Dynamic Coscheduling techniques resolve this problem very well.

For Implicit Coscheduling, with the default Solaris time-sharing scheduler, an infrequently communicating job is given more of the processor than a competing medium and fine-grain job. This is because fine-grained jobs are more sensitive to the scheduling on remote nodes, and frequently sleep when waiting for communication to compete. If a fine-grained job competes against similar jobs, it soon re-acquires the processor when the other jobs sleep; however, the competing coarse-grained job rarely relinquishes the processor. The Dynamic Coscheduling suffers from the inverse problem, the fine-grained jobs will get more CPU times than the coarse-grained jobs.

This observation can be explained by two facts. One is that the coarse-grained jobs for Dynamic Coscheduling are in fact much finer than that in Implicit Coscheduling. The second reason is that the priority boost in the default Solaris 2.6 scheduler that was used in Implicit Coscheduling experiments is usually not high enough to preempt the current running processing. Whereas in Dynamic Coscheduling, the fine-grained jobs always preempt the current running process and thus end up taking much more of CPU time.

While Dynamic Coscheduling does not provide an automatic way to assure the fairness, Implicit Coscheduling uses an algorithm to improve fairness. We will briefly explain the principle as follows. The priority of each job in the system is raised once a second, regardless of the past allocation of each job, and the priority of a process drops after each executed time-slice. Providing a fair allocation of the CPU requires boosting the priority of a process as determined by its past allocation; that is, since fine-grained jobs are sleeping more frequently, they should be given a higher priority when runnable. So the ideal starve interval, S, should slightly exceed the time-slice, Q, multiplied by the number of jobs, J ($S > JQ$). But it does not work very well. There is still more work to be done on this scheme.

7 Possible improvements in co-scheduling on cluster systems

After studying the major kinds of co-scheduling techniques and carefully analyzing both Implicit and Dynamic Coscheduling, we suggest some ideas to improve the performance of co-scheduling on cluster systems. We can use the message arrival as a demand for co-scheduling, as it is the case in Dynamic Coscheduling; and use the Implicit Coscheduling technique to decide the spin-time before blocking the process. We can also integrate some mechanisms to assure fairness.

7.1 Combining the Dynamic & Implicit Coscheduling

From the analysis of the last section, we found that both Dynamic and Implicit Coschedulers have benefited from each other. Our suggestion is to combine them explicitly.

In Dynamic co-scheduling, we do not depend on the Solaris 2.6 scheduler to provide priority boost, in fact it works better than the default scheduler. And the spin-time analysis in Implicit Coscheduling can make the spin-blocking synchrony more effective. We can integrate them and the combination seem to provide better performance. Now we should concentrate more on the issue of fairness. In fact, since time-sharing schedulers are designed to provide a compromise between interactive response time and fairness, it is not an ideal platform for co-scheduling, stride schedulers or other proportional share schedulers seem more suitable.

To assure the fairness of the scheduling, we can use a kind of proportional scheduler as local scheduler on each node. We choose the Stride scheduler, which is described in the next subsection.

7.2 Stride scheduling

Stride scheduling [10] is a deterministic allocation algorithm that encapsulates resource rights with tickets. Resources are allocated to competing clients in proportion to the number of tickets they hold. For example, if a client has *t* tickets in a system with T total tickets, the client receives *t*/T of the resources in that system.

In the original Stride Scheduler, there is no convention of preempts, and each process can use its whole share of time-slice according to the tickets it holds. But interactive process will give up the CPU time when waiting for I/O. So in order to assure that the I/O processes can get their fair share of proportional CPU time, Stride scheduling introduces two techniques: Loan & Borrowing and System Credit [10].

Loan & Borrowing:

In this technique, processes can borrow tickets from sleeping processes and then return them back when the sleeping process wakes up.

System Credit:

In this technique, the system gives credit, in the form of tickets, to a process that sleeps since he did not get its fare share of resources and was blocked.

The disadvantage of loan & borrowing is that it introduces an excessive amount of computation into the scheduler on every sleep & wake up event, especially as there can be a lot of processes involved. However, the advantage of this scheme over system credit is that a process can borrow tickets when it is running. This aspect is very important in terms of co-scheduling. We will combine these two techniques and introduce some of our own ideas for guaranteeing fairness in our improvements that we introduce for the co-scheduling algorithm.

7.3 Possible Improvements on co-scheduling algorithm for cluster systems

7.3.1 Using a modified Stride scheduler as local scheduler

Stride scheduler is originally used to fairly allocate resources on a single node. If we try to use the stride scheduling as a building block of our co-scheduling technique on cluster systems, some modifications are required:

(1) Sleep Credit:
Because the communicating processes, like interactive processes, will often sleep to give up the CPU by themselves, they should receive credit (additional tickets) for the time that they are asleep, in order to receive their proportional share of the CPU over a longer time-interval. The basic mechanism is exhaustible tickets, which are simply tickets with an expiration time. After a process sleeps for time S and awakens, the scheduler calculates the number of exhaustible tickets for the process to receive its proportional share over some longer time interval C. The process can get this additional tickets in the following C interval. If the process does not use its bonus credits,

exhaustible tickets, during the interval C, those tickets will not be valid afterwards. As in [10], we can set C = S to simplify computations. The ideal value of C can be determined via experiments.

(2) Active Borrowing & Repayment:
A process can borrow a certain fixed amount of tickets from the local ticket server, in a corresponding time interval (stride). This fixed amount can vary and the best value can be determined from experiments (e.g. one solution is to limit the fixed amount of tickets that the process can borrow to the same amount of tickets that the process was granted in this stride).

(3) Preemption Credit:
If a process A is preempted by another process B, due to the need for co-scheduling, then process A should get credit from the system as it is relinquishing its share of CPU time. The credit given to process A is proportional to the duration of time for which it was preempted. This leads to process A having a proportional share of resources in the longer run, and thus not getting penalized for being preempted. This is very important as the case often occurs when need for co-scheduling of a process requires preemption of a currently running process.

(4) Ticket Tracking:
Since in a workstation cluster, the set of resources includes the entire cluster, the same job can be run on different workstations. Thus, if we restrict fairness to each individual workstation, then a job running on many workstations may end up getting a higher share of resources. This situation is depicted in [10]. Therefore, in order to maintain fairness across the cluster, each local ticket server has to somehow know how many tickets have been allocated to the same job on different workstations. This can be achieved by a periodic broadcast done by each workstation. The broadcast contains information related to the local ticket server's state (i.e. the processes running on it & how many tickets are allocated to each process). The broadcast is done periodically and is only performed if there are changes in the local ticket server's state from its last broadcast. Doing this broadcast periodically, and not every time a process is created or deleted, removes the effect of short lived jobs and thus decreases the amount of traffic on the network.

Each workstation uses the information from the broadcast to automatically compute the number of tickets to be given to each of its processes in the next stride. Thus, if a job is running on multiple workstations, it will receive a lower share per workstation, but will have a fair amount of resources, as compared to other jobs, from the perspective of the entire cluster resources. For more details regarding this issue refer to Dusseau and Culler [10].

7.3.2 Message driven process scheduling

When a message arrives, and the destination process is not the current running process, the scheduler will check if the destination process has enough tickets left(maybe it can loan some tickets from the system). If so, the scheduler will preempt the current process and schedule the destination process.

For details, the destination process can either be sleeping/blocked, running or in the ready queue:

(a) Sleeping:
If the destination process is sleeping, then two prerequisites have to be met by the sleeping process to be scheduled immediately:

(i) The scheduler ensures that the process is sleeping on a message arrival and not on an I/O block or other events.

(ii) The process can borrow tickets from the system via Active Borrowing. Note that even if the process is sleeping and is receiving credit for it, it could be that it has already borrowed a lot in the previous stride and thus is still ‘in the red'(i.e. has borrowed more than it can give back) with the local ticket server.

(b) Running:
If the destination process is running, then there is no need for preemption. The only effect will be that the spin-time will increase according to the two-phase spin-block mechanism.

(c) Ready Queue:
If the destination process is in the ready queue, then it will be scheduled immediately provided that it can borrow tickets from the local ticket server via Active Borrowing.

7.3.3 Two-phase spin-blocking synchrony

Like Implicit Coscheduling, we use the two-phase spin-blocking technique when a communicating process is waiting for a message from other parallel processes. It will spin a variable time before blocking. The spin time is decided by the information collected from the response time, messages arriving and scheduling progress.

Combining the Implicit & Dynamic Co-scheduling, and integrating this with the modified stride scheduler, we can expect a good performance and fairness from the modified co-scheduling algorithm. Of course, we need many experiments to claim this, but it seems promising.

8 Conclusion

Co-scheduling is necessary for scheduling processes, constituting parallel jobs, in order to enhance performance and prevent processor thrashing. Although there are many co-scheduling techniques, not all of them are suitable for a cluster of workstations environment. In clusters, several workstations are connected together on a low-latency network, and the shared resources allow a mix of workloads such as parallel, interactive, sequential and client/server jobs to coexist. In such an environment, factors such as (a) dynamically co-scheduling processes, (b) localizing scheduling decisions to avoid multi-context switching performed by a centralized server, and (c) fairness among the mix of jobs, play a very important role. The Dynamic and Implicit co-scheduling techniques thus seem more suited to such an environment. However, after critically analyzing these two techniques we think that there is room for possible improvements in terms of performance and fairness. We thus present a modified co-scheduling algorithm integrating the features of Implicit & Dynamic co-scheduling, to improve performance, and introduce a variation of the Stride scheduling technique to provide better fairness.

References:

[1] T. Anderson, D. Culler, D. Patterson, and the NOW team. "A Case for NOW (Networks of Workstations)". IEEE Micro, February 1995

[2] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc and E. Markatos. **“**Multiprogramming on multiprocessors”. Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing, pages 590-597, Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 1991

[3] A. Gupta, A. Tucker and S. Urushibara. "The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications". Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 120-132, May 1991.

[4] D.G. Feitelson and L. Rudolph. “Distributed hierarchical control for parallel processing”. IEEE Computer, vol.23, no.5, Pages 65-77, May 1990.

[5] A. Tucker and A. Gupta. ”Process Control and Scheduling Issues for Multiprogrammed Shared-Memory multiprocessors”. Proceedings of the 12th ACM Symposium on Operating Systems Principles, pages 159-186, 1989.

[6] Fang Wang, H. Franke, M. Papaefthymiou, P. Pattnaik, L. Rudolph and M.S. Squillante. “A gang scheduling design for multiprogrammed parallel computing environments”. Job Scheduling Strategies for Parallel Processing. IPPS '96 Workshop Proceedings, pages 111-125, Berlin, Germany: Springer-Verlag, 1996.

[7] D.G. Feitelson and L. Rudolph. **“**Coscheduling based on runtime identification of activity working sets”. International Journal of Parallel Programming, vol.23, (no.2), pages 135-160, April 1995.

[8] A.C. Arpaci-Dusseau, D.E. Culler and A.M. Mainwaring. **“**Scheduling with implicit information in distributed systems”. Performance Evaluation Review, vol.26, (no.1), pages 233-243 (SIGMETRICS '98/PERFORMANCE'98. Joint International Conference on Measurement and Modeling of Computer Systems, Madison, WI, USA, 22-26 June 1998.) ACM, June 1998.

[9] P.G. Sobalvarro, S. Pakin, W.E. Weihl and A.A. Chien. “Dynamic coscheduling on workstation clusters”. Job Scheduling Strategies for Parallel Processing. IPPS/SPDP'98 Workshop Proceedings, pages 231-256, Berlin, Germany: Springer-Verlag, 1998.

[10] A.C. Arpaci-Dusseau and D.E. Culler. "Extending Proportional-Share Scheduling to a Network of Workstations." International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), Las Vegas, Nevada, June 1997.