Scheduling in Distributed Systems

Dongning Liang, Pei-Jung Ho, Bao Liu
Department of Computer Science and Engineering
University of California, San Diego

Abstract
This paper presents several scheduling/coscheduling techniques employed in some recent research projects. Two types of local scheduling, proportional-sharing scheduling and predictive scheduling are introduced here. With proportional-share scheduling, the resource consumption rights of each active process are proportional to the relative shares that it is allocated. While the system implementing predictive scheduling can adapt to new architectures and/or algorithms and/or environmental changes automatically. Three types of coscheduling are discussed in this paper. Gang scheduling is a simple coscheduling mechanism that is widely used in distributed systems. While more sophisticated implicit coscheduling and dynamic coscheduling allow each local scheduler in the system to make independent decisions that dynamically coordinate the scheduling of cooperating processes across processors. Finally, this paper will give some discussion among these scheduling mechanisms and their combinations.

1. Introduction
Scheduling parallel applications in a distributed environment, such as a cluster of workstations, remains an important and unsolved problem. One of the main research issues is effectively exploiting idle resources and to time-share the system fairly among the processes.

Local scheduling, where each workstation independently schedules its processes, is an attractive time-sharing option for its ease of construction, scalability, fault-tolerance, etc. Meanwhile, coordinated scheduling of parallel jobs across the nodes of a multiprocessor (coscheduling) is also indispensable in a distributed system. Without coscheduling, the processes constituting a parallel job might suffer high communication latencies because of processor thrashing [1]. By coordinated scheduling across cooperating processes, each local scheduler is able to make independent decisions that tend to schedule the processes of a parallel application in a coordinated manner across processors, in order to fully exploit the computing resource of a distributed system.

There are several on-going research projects in this area, such as the NOW (Network of Workstations) [10] project at UC Berkeley, the HPVM (High Performance Virtual Machine) [11] project at UCSD/UIUC, and Hector at Mississippi State University [2]. In this paper, we will discuss the main scheduling and coscheduling techniques in these projects, comment on their pros and cons, and then make our conclusion. The remainder of this paper is organized as follows: Section 2 gives an overview of the scheduling mechanism. The requirements of a scheduling approach are represented in Section 3. Section 4 describes the ideas on local scheduling, while Section 5 touches on coscheduling. A discussion on scheduling is included in section 6, along with some promising directions for future work. Finally, Section 7 gives a brief conclusion.

2. Overview of the Scheduling Mechanism
Before going into these new specified approaches, let us see how the distributed system runs a job across the whole system, and what role a scheduler plays here.

In general, job scheduling is composed of at least two inter-dependent steps: the allocation of processes to workstations (space-sharing) and the scheduling of the processes over time (time-sharing), while there exist several optional complementary steps to further improve the performance.

When a job is submitted to the system, job placement will be done, i.e., to decide which workstations to run the job cooperatively (space-sharing). Along with the job submission, a description of the attributes of the job is also submitted to the system in order to specify the resource requirement, such as memory size requirement, expected CPU time, deadline time, etc. In the meantime, the system always maintains an information table, either distributed or centralized, to record the current resource status of each workstation, e.g., CPU load, free memory size, etc. Then, a
matchmaking frame will do matching work to find the most suitable set of workstations to meet the requirement of the job. This job is then decomposed into small components, i.e., processes, which are distributed to those assigned workstations. On each individual workstation, the local scheduler allocates some time-slices to the process based on some policies so as to achieve the scheduling requirement, such as response time and fairness.

These decomposed components may require synchronization among themselves. For example, process A requires input from process B to proceed; then, A blocks until the input from B arrives. Therefore, a coordinated scheduling is needed to minimize the time waiting for messages from other processes. We will discuss this in detail in Section 6.

Besides the mechanisms mentioned above, process migration is introduced to improve load sharing, which allows the process to run on the most suitable workstation. During the lifetime of a job, the resource status of the system is always changing, recommending that the process run on another more suitable workstation. For example, a workstation may become light-loaded when finishing its assigned job, so, processes on a heavy-loaded workstation can be migrated onto such a light-loaded workstation, which may let the job finish earlier and improve the overall performance of the system.

Space-sharing approaches will achieve a less interactive response time but probably also smaller throughput; on the contrary, time-sharing approaches have a higher throughput but also lengthen the response time. Therefore, a good approach should be a mixed approach, utilizing both space-sharing and time-sharing, with the complementary coscheduling and process migration. In this paper, we will only discuss the local scheduling and coscheduling, i.e., how to get the best performance after the set of workstations is assigned to a job.

3. Properties of a Good Scheduler

Many research activities are being conducted to develop a good scheduling approach among a set of distributed hosts. The activities vary widely in a number of dimensions, e.g. support for heterogeneous resources, placement objective function(s), scalability, coscheduling methods, and assumptions about system configuration. Based on the experience accumulated during these activities, it is believed that a good scheduler should have the following properties:

*General purpose*: a scheduling approach should make few assumptions about and have few restrictions to the types of applications that can be executed. Interactive jobs, distributed and parallel applications, as well as non-interactive batch jobs, should all be supported with good performance. This property is a straightforward one, but to some extent difficult to achieve. Because different kinds of jobs have different attributes, their requirements to the scheduler may contradict. For example, a real-time job, requiring short-time response, prefers space-sharing scheduling; a non-interactive batch job, requiring high-throughput, may prefer time-sharing scheduling. To achieve the general purpose, a tradeoff may have to be made. As mentioned above, in this paper, we will discuss the scheduling method focused on parallel jobs, while providing an acceptable performance to other kinds of jobs.

*Efficiency*: it has two meanings: one is that it should improve the performance of scheduled jobs as much as possible; the other is that the scheduling should incur reasonably low overhead so that it won’t counterattack the benefits.

*Fairness*: sharing resources among users raises new challenges in guaranteeing that each user obtains his/her fair share when demand is heavy. In a distributed system, this problem could be exacerbated such that one user consumes the entire system. There are many mature strategies to achieve fairness on a single node; we will describe how to achieve it on a distributed system in Section 5.1.

*Dynamic*: the algorithms employed to decide where to process a task should respond to load changes, and exploit the full extent of the resources available.

*Transparency*: the behavior and result of a task’s execution should not be affected by the host(s) on which it executes. In particular, there should be no difference between local and remote execution. No user effort should be required in deciding where to execute a task or in initiating remote execution; a user should not even be aware of remote processing, except maybe better...
performance. Further, the applications should not be changed greatly. It is undesirable to have to modify the application programs in order to execute them in the system.

4. Local Scheduling

In a distributed system, local scheduling means how an individual workstation should schedule those processes assigned to it in order to maximize the overall performance [3]. It seems that local scheduling is the same as the scheduling approach on a stand-alone workstation. However, they are different in many aspects. In a distributed system, the local scheduler may need global information from other workstations to achieve the optimal overall performance of the entire system. For example, in the extended stride scheduling of clusters, the local schedulers need global ticket information in order to achieve fairness across all the processes in the system.

In recent years, there have been many scheduling techniques developed in different models. Here, we introduce two of them: one is a proportional-sharing scheduling approach, in which the resource consumption rights of each active process are proportional to the relative shares that it is allocated. The other is predictive scheduling [4], which is adaptive to the CPU load and resource distribution of the distributed system.

4.1 Proportional-Sharing Schedule

The traditional priority-based schedulers are difficult to understand and give more processing time to users with many jobs, which leads to unfairness among users. Numerous researches have been trying to find a scheduler that is easy to implement and can solve the problem of allocating resources to users fairly over time. In this environment, proportional-share scheduling was brought out to effectively solve this problem. With proportional-share scheduling, the resource consumption rights of each active process are proportional to the relative shares that it is allocated.

In section 4.1.1, we introduce stride scheduling as an example of proportional-sharing schedule, in order to show how to solve a relatively simple problem: fairly allocating a single processor among competing users on a single-node. In section 4.1.2, Two extensions of stride scheduling are presented to provide better response-times for interactive jobs [5]. Finally, we argue that fairness can also be guaranteed when stride scheduling is used in a distributed cluster.

4.1.1 Stride Scheduling

As a kind of proportional-share scheduling strategies, stride scheduling allocates resources to competing users in proportion to the number of tickets they hold. Each user has a time interval, or stride, inversely proportional to his/her ticket allocation, which determines how frequently it is used. A pass is associated with each user. The user with a minimum pass is scheduled at each interval; a pass is then incremented by the job's stride. Figure 1 is an example of stride scheduling.

![Stride Scheduling](image)

Figure 1. Stride Scheduling: Three compete with a 1:2:3 ticket ratio. (In the example here and figure 2, we refer to numbers of tickets after they have been translated into the base currency.)

_Currencies_ allow clients to distribute tickets in a modular way. Besides a global based currency, each user has his/her own currency. By assigning one currency per user, a proportional-share of resources can be allocated to each user, who in turn can allocate a proportional-share of his/her resources to his/her processes.

4.1.2 Extensions to Stride Scheduling

The original stride scheduling only deals with CPU-bound jobs. If the proportional-share schedulers are to handle the interactive and I/O intensive job workloads, they must be extended to improve the responsive time and I/O throughput, while not penalizing competing users. Here we discuss two extensions to stride scheduling that give credits to jobs not competing for resources. In this way, jobs are given incentive to relinquish the processor when not in use and will receive their share of resources over a longer time-interval. Thus, because interactive jobs are scheduled more frequently when they awaken, they can receive better response time. The first approach is _loan & borrow_, and
the second approach is system credit. Both approaches are built upon exhaustible tickets, which are simple tickets with expiration time.

- **Loan & Borrow:**
  In this approach, exhausted tickets are traded among competing clients. When a user temporarily exits the system, other users can borrow these otherwise inactive tickets. The borrowed tickets expire when the user rejoins the system. When the sleeping user wakes up, it stops loaning tickets and is paid back in exhaustible tickets by the borrowing users. In general, the lifetime of the exhaustible tickets is equal to the length the original tickets were borrowed. This policy can keep the total number of tickets in the system constant over time; thus, users can accurately determine the amount of resources they receive. However, it also introduces an excessive amount of computation into the scheduler on every sleep and wake-up event, which we don’t expect.

- **System Credit:**
  This second approach is an approximation of the first one. With system credits, clients are given exhaustible tickets from the system when they awaken. The idea behind this policy is that after a client sleeps and awakens, the scheduler calculates the number of exhaustible tickets for the clients to receive its proportional share over some longer interval. The system credit policy is easy to implement and does not add significant overhead to the scheduler on sleep and wakeup events. Figure 2 shows an example of both approaches.

![Figure 2. Load & Borrow versus System Credit: Three jobs with equal ticket allocations are competing for resources. Job A desires a constant service rate, job B is computer-intensive and willing to borrow tickets, and job C is an interactive job. Job C temporarily exits the system, and sleeps for an interval \( S \); in the time-interval \( C \), job C catches up for the time it missed. In both cases, all jobs receive their proportional-share of 6 allocations over the entire interval of 18 time-units; however, only with the loan & borrow policy is job A always scheduled 1 out of every 3 time units.]

We have discussed allocating a proportional-share of resources to both compute-intensive and interactive jobs on a single workstation. Now we will move our attention to a distributed environment and show that a proportional-share of resources can be allocated to clients running sequential jobs in a cluster. In the cluster, users are guaranteed a proportional-share of resources if (1) each local stride-scheduler is aware of the number of tickets issued in its currency across the cluster and if (2) the total number of base tickets allocated on each workstation is balanced. The solution for the first assumption is simple: each local scheduler is informed of the number of tickets issued in each currency, and then correctly calculates the base funding of each local job. The solution for distributing tickets to the stride-schedulers is to run a user-level tickets-sever on each of the nodes in the cluster. Each stride-scheduler periodically contacts the local ticket server to update and determine the value of currencies.

Further, for parallel jobs in a distributed cluster, proportional-share resources can be provided through a combination of stride-scheduling and implicit coscheduling. Preliminary simulations of implicit coscheduling for a range of communication patterns and computation granularity indicate that the stride-scheduler with system credit performs similarly to the Solaris time-sharing scheduler which is used in the Berkeley NOW environment [5]. We will describe implicit coscheduling in Section 5.2.

### 4.2 Predictive Scheduling

Predictive scheduling differs from other scheduling approaches in that it provides intelligence, adaptivity and proactivity so that the system implementing predictive scheduling can adapt to new architectures and/or algorithms and/or environmental changes automatically.
Predictive scheduling can learn new architectures, algorithms and methods that are embedded into the system. They provide some guarantees of service. Furthermore, they are able to anticipate significant changes to its environment and avoid those changes to become the system performance bottleneck.

Predictive scheduling can be roughly decomposed into three components: H-cell, S-cell and allocator. The H-cell receives information of hardware resource changes such as disk traffic, CPU usage, memory availability, etc., and provides near-real-time control. Meanwhile, S-cell provides long-term control of computational demands--such as what the deadline of a task is and what its real-time requirement is--by interrogating the parallel program code. H-cell and S-cell respectively collect information about computational supply and computational demand, and provide to the allocator the raw data or some intelligent recommendations. The allocator reconciles the recommendations sent by the H-cells and S-cells and schedules jobs according to their deadline, while guaranteeing constraints and enforcing the deadline.

In the allocator, the previous inputs, in the form of a vector of performance information (such as memory, CPU, disk usage etc.), are aggregated into sets. Each set corresponds to a scheduling decision. The allocator re-organizes the sets dynamically to keep a limited memory demand by splitting or merging sets. If a new input matches one of the pattern categories, a decision will be made due to the corresponding decision of that pattern set, otherwise a new pattern category is built to associate this new input pattern with corresponding scheduling decision.

Most of the scheduling policies are used either when a process blocks or at the end of a time slice, which may reduce the performance because there can be a considerable lapse of time before scheduling is done. Predictive scheduling solves this problem by predicting when a scheduling decision is necessary, or predicting the parameters needed by the scheduling decision when not known in advance. Based on the collected static information (machine type, CPU power, etc.) and dynamic information (memory free space, CPU load, etc.), predictive scheduling tries to make an educated guess about the future behavior, such as CPU idle time slot, which can be used to make scheduling decisions in advance. Predicting the future performance based on past information is a common strategy, and it can achieve a satisfactory performance in practical work.

Predictive scheduling is very effective in performance and reliability enhancement, even with the simplest methods, but at the cost of design complexity and management overhead. Furthermore, it is observed that the more complicated method is used, the more design complexity and management overhead, and the less performance and reliability enhancement.

5. Coscheduling

In 1982, Outsterhout introduced the idea of coscheduling [9], which schedules the interacting activities (i.e., processes) in a job so that all the activities execute simultaneously on distinct workstations. It can produce benefits in both system and individual job efficiency. Without coordinated scheduling, the processor thrashing may lead to high communication latencies and consequently degraded overall performance. With systems connected by high-performance networks that already achieve latencies within tens microseconds, the success of coscheduling becomes a more important factor in deciding the performance.

5.1 Gang Scheduling

Gang scheduling is a typical coscheduling approach, which has already been introduced for a long time but still plays a fundamental role. Moreover, there are still many research projects in progress to improve gang scheduling.

The approach identifies a job as a gang and its components as gang members. Further, each job is assigned to a class that has the minimum number of workstations that meet the requirement of its gang members based on a one-process-one-workstation policy. The class has a local scheduler, which can have its own scheduling policy. When a job is scheduled, each of its gang members is allocated to a distinct workstation, and thus, the job executes in parallel. When a time-slice finishes, all running gang members are preempted simultaneously, and all processes from a second job are scheduled for the next time-slice. When a job is rescheduled, effort is also made to run the same processes on the same processors.

The strategy bypasses the busy-waiting problem by scheduling all processes at the same time. According to the experience, it works well for parallel jobs that have a lot of inter-process communications. However, it also has several disadvantages. First, it is a centralized scheduling strategy, with a single scheduler making decisions for all jobs and all workstations. This centralized nature can easily become the bottleneck when the load is heavy. Second, although this scheduler can achieve high system efficiency on regular parallel applications, it has difficulty in selecting alternate jobs to run when processes block, requiring simultaneous multi-context switches across the nodes. Third, to achieve good
performance requires long scheduling quanta, which can interfere with interactive response, making them a less attractive choice for use in a distributed system. These limitations motivate the integrated approaches.

The requirement of centralized control and the poor timesharing response of previous scheduling approaches have motivated new, integrated coscheduling approaches. Such approaches extend local timesharing schedulers, preserving their interactive response and autonomy. Further, such approaches do not need explicitly identified sets of processes to be coscheduled, but rather integrate the detection of a coscheduling requirement with actions to produce effective coscheduling. In Section 6.2 and 6.3, we will introduce two representatives of this new approach.

5.2 Implicit Coscheduling

Implicit coscheduling is a distributed algorithm for time-sharing communicating processes in a cluster of workstations [3]. By observing and reacting to implicit information, local schedulers in the system make independent decisions that dynamically coordinate the scheduling of communicating processes. The principal mechanism involved is two-phase spin-blocking: a process waiting for a message response spins for some amount of time, and then relinquishes the processor if the response does not arrive.

The spin time before a process relinquishes the processor at each communication event consists of three components. First, a process should spin for the baseline time for the communication operation to complete; this component keeps coordinated jobs in synchrony. Second, the process should increase the spin time according to a local cost-benefit analysis of spinning versus blocking. Third, the pairwise cost-benefit, i.e., the process, should spin longer when receiving messages from other processes, thus considering the impact of this process on others in the parallel job.

- The baseline time comprises the round-trip time of the network, the overhead of sending and receiving messages, and the time to awake the destination process when the request arrives.
- The local cost-benefit is the point at which the expected benefit of relinquishing the processor exceeds the cost of being scheduled again. For example, if the destination process will be scheduled later, it may be beneficial to spin longer and avoid the cost of losing coordination and being rescheduled later. On the other hand, when a large load-imbalance exists across processes in the parallel job, it may be wasteful to spin for the entire load-imbalance even when all the processes are coscheduled.
- The pairwise spin-time only occurs when other processes are sending to the currently spinning process, and is therefore conditional. Consider a pair of processes: the receiver who is performing a two-phase spin-block 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 base and local amount, while recording the number of incoming messages. If the average interval between requests is sufficiently small, the process assumes that it will remain beneficial in the future to be scheduled and continues to spins for an additional spin time. The process continues conditionally spinning for intervals of spin time until no messages are received in an interval.

5.3 Dynamic Coscheduling

Dynamic coscheduling makes scheduling decisions driven directly by the message arrivals. When an arriving message is directed to a process that isn’t running, a schedule decision is made. The idea derives from the observation that only those communicating processes need to be coscheduled. Therefore, it doesn’t require explicit identification to specify the processes need coscheduling.

A simple illustration of the dynamic coscheduling implementation schematic is show in Figure 3. Further detail of the implementation can be found in [13].

The implementation consists three parts:
- Monitoring Communication/Thread Activity
  A firmware, which is on the network interface card, monitors the thread activities by periodically reading the host's kernel memory. If the incoming message is sent to the process currently running, the scheduler should do nothing.
- Causing Scheduling Decisions
  If a message received is not sent to the process currently running, an interrupt will be produced and invoke the interrupt routine. When the routine finds that it would be fair to preempt the process currently running, the process receiving the message has its priority raised to the maximum allowable priority for user mode timesharing processes, and is placed at the front of the dispatcher queue. Flags are set to cause a scheduling decision based on the new priorities. This will cause the process receiving the message to be scheduled unless the process currently running has a higher priority than the maximum allowable priority for user mode.
- Making a Decision Whether to Preempt

In dynamic coscheduling, the process receiving the message is scheduled only if doing so would not cause unfair CPU allocation. The fairness is implemented by limiting the frequency of priority boosts that therefore limits the frequency of preemption.

In jobs with fine-grained communication, the sender and receiver are scheduled together and run until one of them blocks or is preempted. Larger collections of communicating processes are coscheduled by transitivity. The experiments taken in HPVM project indicate that dynamic coscheduling can provide good performance for a parallel process running on a cluster of workstations in competition with serial processes. Performance was able to close to ideal: CPU times were nearly the same as for batch processing, and reduced job response times by up to 20% over implicit scheduling while maintaining near-perfect fairness. Further, it claims that dynamic-coscheduling-like approaches can be used to implement coordinated resource management in a much broader range of cases, although most of which are still to be explored.

6. Discussion

After studying various kinds of scheduling approaches with different focus, we suggest some ideas of improving the performance of scheduling by promoting the strength of those approaches while avoiding their drawbacks.

6.1 stride and predictive scheduling

These are two different approaches emphasizing on different properties of scheduling. Stride scheduling aims at fairness, while predictive scheduling emphasizes more on adaptivity to the system. We found it beneficial to build a hybrid mechanism out of them.

Stride scheduling will schedule processes according to their passes that corresponds to their allocated ticket. Usually, the allocated tickets are static. When tickets are allocated for a process, it won't change during the lifetime of the process. This makes it simple, but sometimes it will lead to performance degradation. Consider such scenario: the I/O bandwidth of a workstation is low, and the process to be scheduled running next is I/O intensive. When this process is running, its performance could be restricted by the low I/O bandwidth. We can now use predictive scheduling as a complementary. Stride scheduling will propose the candidate to be run, then predictive scheduling matches the computational demand of this candidate and the computational supply. If the supply satisfies the demand, the candidate will proceed to run; otherwise, this process should be delayed a time-slice and the scheduler will pick up the next process as candidate. In this way, we can avoid the performance degradation mentioned above, without harming
fairness of the system. If a process is delayed, its pass won't change. Since stride scheduling always picks up candidate according its pass, the delayed process will be picked up again as candidate when the current time-slice ends.

6.2 implicit and dynamic coscheduling

Implicit scheduling uses spin-block synchronization primitives and the priority boost provided by the SVR4 scheduler. Since awakened processes can obtain a higher priority, they are likely to run when their communication peer has sent a message (and is therefore running). Implicit scheduling can modify the spin-time in spin-block synchronization to further improve performance.

In contrast, dynamic scheduling achieves coscheduling by explicitly treating all message arrivals (not just those sent to blocked processes) as a demand for coscheduling, and explicitly schedules the destination processes when it would be fair to do so through the explicit control of scheduler priorities. While this is similar to implicit scheduling for the particular case of bulk synchronous jobs using spin-block synchronization, it claims that dynamic coscheduling can be used to achieve coordinated scheduling in a broader range of cases.

We think that using message arrival to invoke coscheduling, just like what is done in dynamic coscheduling, combined with dynamic spin-time technique used in implicit scheduling, can improve the performance and will be suitable for broader range of cases.

6.3 local scheduling and coscheduling

In distributed systems, the overall performance highly depends on how well the local scheduling can cooperate with coscheduling. It is possible that a good coscheduling mechanism will not work well with a good local scheduler. It is also possible that a simple local scheduler may achieve good overall performance in cooperation with a crude coscheduling mechanism.

In section 5, when we talk about implicit and dynamic coscheduling, both of them are implemented on simple local time-sharing schedulers, which are not inherently fair. We believe that a fair local and adaptive scheduler, e.g. extended stride scheduler or predictive scheduler, could provide a better support for coscheduling, by separating the concepts of execution order and processor share.

7. Conclusion

The shared resources in a distributed system have enabled a new set of workloads to coexist: sequential, interactive, and parallel jobs. This new workload and this environment require new approaches for fairly and efficiently allocating resources to competing users.

In this paper, several approaches on scheduling and coscheduling are presented. Besides performance, fairness is a very important requirement for the scheduling approaches. In Section 4.1, we showed an approach extent from stride-scheduling, which fairly allocates resources to a mix of multi-type jobs with improving response time. To a scalable distributed system, it is also beneficial if the scheduler always can make decisions according to the latest system change. Predictive scheduling makes an attempt on this problem, as well as reduces the scheduling overhead by overlapping scheduling decisions with other operations.

In addition to a good scheduling policy, coscheduling is also critical for scheduling processes on distributed systems in order to enhance performance and prevent processor thrashing. Gang scheduling is an approach that plays well on parallel computer. However, its explicit defining processes need coscheduling, and poor fairness makes it unsuitable to the distributed system, which implicitly recommends cooperated scheduling and an acceptable fairness. The referred dynamic coscheduling and implicit coscheduling techniques seem more suitable to such a distributed environment. Combined with a scheduling policy, these two can achieve a fairly good performance on a mixed workload.

Generally speaking, a good scheduling approach requires a good balance between achieving fairness across users and optimizing throughput in a distributed system. The system must often choose between balancing fairness and balancing load when placing jobs. More work needs to be performed to better understand the trade-off.
8. Reference


