SCOJO – Share Based Job Coscheduling With Integrated Dynamic Resource Directory in Support or Grid Scheduling

Angela C. Sodan[a] and Xuemin Huang[a]

[a]University of Windsor, Computer Science, 401 Sunset Ave., Windsor, ON N8Y 1B1, Canada

Grid environments currently use mere batch scheduling or assume jobs to be started immediately. We apply a combination of batch scheduling and time sharing and propose an approach which assigns certain start times and certain shares based on integrated resource management. Employing loosely coordinated coscheduling, provides good chances for latency hiding and high flexibility in executing jobs across multiple sites. Our approach takes potential slowdowns or speedups from coscheduling into consideration. We support such scheduling by a dynamic directory service. Current directory services for grid computing provide static machine descriptions plus some dynamic information like CPU load as obtained via NWS. However, in other work, we have shown that the detailed application characteristics matter in coscheduling. Furthermore, predictability is currently limited to short-term extrapolations from the recent history. We develop scheduling plans and consider site workload and application characteristics, while not disclosing such information to other users or sites. This approach enables higher-quality decisions and longer-term predictability. We demonstrate our approach on the basis of simulations and simplified real coscheduling tests, run on an SMP server.

Les environnements de type grille utilisent généralement un système d'ordonnancement (files d'attente) simple, présupposant un démarrage immédiat des travaux soumis. Nous proposons ici une combinaison de l'approche de traitement par lots et de celle des systèmes à exploitation partagée. Nous déterminons le délais avant démarrage et les ressources qui seront allouées grâce à une analyse des ressources disponibles. Avec l'utilisation du traitement par lots latéral à coordination souple, nous diminuons les effets de latence et augmentons la flexibilité dans la distribution des tâches de calcul sur un plus grand nombre de sites. Notre approche tient compte des possibles ralentissments ou accélérations du traitement par lots latéral. Pour ce faire, nous utilisons un service d'annuaire dynamique. Les systèmes d'annuaire actuellement utilisés sur grille de calcul sont basés sur une description statique des noeuds de calcul, à laquelle s'ajoute des informations dynamique obtenues via NWS, tel le taux d'utilisation des processeurs. Nos précédentes recherches démontrent que les caractéristiques propres à une application influent sur son traitement par lots latéral. On ne peut actuellement faire que des prédictions à court terme basées sur l'historique récente du traitement par lots. Nous développons la méthode de traitement par lots en lui adjoignant une analyse de l'état de charge du site visé, ainsi que des caractéristiques de l'application, le tout en évitant la diffusion de ces informations aux usagers ou sites du réseau. Cette approche améliore le processus de prise de décisions par le système ainsi que la prévisibilité à long terme. Nous illustrons notre approche à l'aide de simulations et de tests de traitement par lots latéral, le tout effectué sur architecture à mémoire partagée.

1 Introduction

The Globus grid system [5] provides services for request submission to local schedulers and for resource reservation (GARA) but does not provide an own scheduler. The assumption is that either batch space sharing (assigning CPUs/nodes exclusively) or immediate execution via time-sharing (sharing CPU resources) are applied. In the latter case, certain CPU shares may be maintained, though, through reservation with e.g. the GARA service. We focus on time-sharing aspects here but combine time sharing with a batch job scheduler which selects the jobs that are run together at the same time. We assume a task per site needing all CPU resources (i.e. we currently ignore space sharing). Furthermore, we focus on loosely coordinated coscheduling (the alternative would be gang scheduling) on shared-memory systems. The combination of admission control and time sharing has the following benefits:

- control of the multiprogramming level while not rejecting jobs which cannot currently be scheduled
- choice between time-shared and exclusive execution according to the predicted slowdown or speedup
- increase of flexibility in scheduling cross-site jobs

The further specific goals of our approach are to support:

- reservation of stable CPU rates which are not changed by coscheduling
- job start-time reservations for cross-site jobs
- estimation of coscheduling cost and predictability
- protocol which is suitable for efficient and flexible multi-site reservations
- support of combined execution of short and long jobs
- secure maintenance of detailed application characteristics and site status

We present the following solutions to meet these goals:

- using effective shares (taking slow-downs or speed-ups into account) for reservations of CPU shares

- running a two-level protocol for cross-site reservations (rough and fine), offering multiple choices via alternate scheduling plans
- keeping application characteristics and differentiating between private and public information
- providing a coscheduling-effect predictor based on integrated site-status and application characteristics
- employing priorities (set dependent on execution time)

In this paper, we present the core scheduling algorithm and investigate the detailed problems in calculating schedules. We demonstrate the scheduler based on a simulation with different workloads and some concrete coscheduling runs (to demonstrate coscheduling effects) on an SMP server.

2 Related Work

Grid scheduling imposes certain requirements. To properly partition and assign parallel programs, the allocation per node needs to be predictable. Thus, reservation of network bandwidth and CPU time are essential. Globus GARA [5] supports the assignment of certain CPU shares, assuming that a scheduler like DSRT [2] is being used. DSRT is originally designed for Multimedia and can assign certain CPU ratios per application. However, coscheduling could only be enforced with prior global synchronization to accomplish kind of gang scheduling. Furthermore, frequent flexible switching is not possible. Share-based scheduling is a currently emerging approach also for OS schedulers, e.g. in Solaris, QLinux or Linux/RT. The Solaris Resource-Manager scheduler [9] can combine flexible priority-based scheduling with share assignment. Furthermore, SUN MPI meets the requirements to run in such an environment on the basis of loose coordination [10]. This means, MPI should not arbitrarily poll for incoming communication but block after some time. Furthermore, receiving processes should get a priority boost to accomplish coscheduling. The limitation of the SUN environment is that the resource manager works per user and not per application. Qlinux allows us to define specific CPU shares per application and assign certain ratios of communication and disk bandwidth. Thus, it is more promising. However, it has not yet been employed in parallel processing and does not work in SMP systems.

Grid schedulers so far typically consider only static machine characteristics and general system status like available CPU rates or free memory space vs. the requirements of the application. The mutual effects of different jobs running together are not yet investigated. In our approach, we consider and keep the characteristics of all jobs and can (re-)evaluate them at any time. A number of grid scheduling approaches are under development but so far not really integrated with local job schedulers. The Working Group on a Grid Resource Allocation Agreement Protocol is currently addressing this problem [6].

3 The Scheduler

We assume distributed grid-job schedulers. The grid-job scheduler for the site with the job request negotiates with the grid-job schedulers of remote sites to find a suitable slot (i.e. per job the negotiation is carried out centrally). There are basically two cases:

- the job is to be run on one site and
- the job is to be run across multiple sites

The first case is relatively easy and only requires to ask different sites, find proper slots and finally select the optimal one. The second case requires to find simultaneous time slots which can be difficult to accomplish. We assume to run the application as one contiguous task per node and search for slots to schedule the application tasks at different sites at the same time. There exist other approaches (promising for loosely dependent computations) which structure the application in a series-parallel manner and run the individual tasks of the application arbitrarily as dependencies permit [8]. We discuss here local scheduling in support of global scheduling but not how the global scheduling plan is calculated (approaches like list scheduling may be applied to find globally simultaneous scheduling slots). We assume that the number of sites to be asked and the work to be possibly scheduled on them are already determined. We ignore site-to-site communication cost, which, if being different for different site choices also plays a role in the decision procedure. Possible solutions would be ranked by finish time.

3.1 The Local Scheduler

We ignore systems implementation consideration as to how existing job schedulers like LSF or PBS can be extended to support time sharing and how the grid-job layer can interact with such a scheduler. Instead, we focus on the scheduling algorithm and refer to it as the local scheduler. This scheduler combines job entry control with time-shared CPU scheduling and can allocate/reserve certain time shares. A job gets a guarantee only for a certain share. If no other job is scheduled at the same time, the share is increased up to 100%. If new jobs are submitted which are eligible to run at this time (e.g. have higher priority) and can run with the remaining un-reserved share, the shares of the currently running jobs are decreased to their reserved ones. Note that an increased share may not necessarily help a cross-site job as it is typically dependent on the computations run at other sites and the whole workload distribution is adapted to the reserved shares.

We apply backfilling [4], i.e. same-priority jobs can

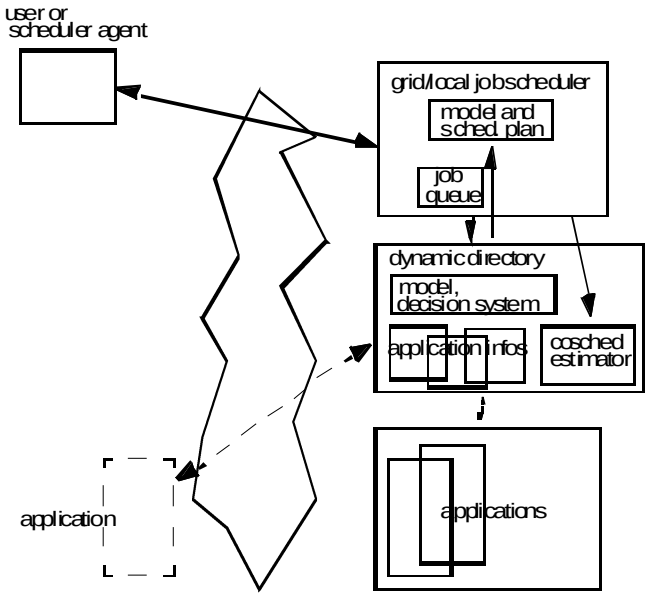

Figure 1. Scheduler and dynamic directory.

be scheduled earlier than their normal position if they can fill schedule holes without delaying other jobs (if priority is assigned other than by runtime, lower-priority jobs may be included as backfilling candidates - otherwise they are too large to fill holes anyway). Note that higher-priority jobs would anyway push other jobs backward in the job queue and therefore never occur as backfilling candidates. Note that whereas backfilling in space sharing means to fill empty space gaps, we here fill time-share space. We never give medium and large jobs a reservation for 100% of the CPU time but only for up to a certain fraction R_{max}. This is done to make it possible to get short jobs through quickly. The time share assignment may also take basic kernel activities into proper consideration (i.e. so to say, reserve a minimum share for background activities).

If a job completes earlier than its runtime estimate had predicted, the jobs behind in the queue can be moved ahead. However, fixed reservations for cross-site jobs are always kept at their scheduled time. We currently do not consider preemption which, however, would be beneficial as additional measure to increase the flexibility in scheduling and the resource utilization by filling holes in front of fixed reservations and dealing with unreliable runtime estimates.

3.2 The Basic Algotithm

SCOJO finds different slots for coscheduling and rates the quality, i.e. the performance available in these slots. Currently, we consider only the available CPU rate. However, for calculating it, we include coscheduling effects, i.e. the benefit or slowdown obtained when executing multiple applications together via time sharing. To keep guarantees for reservations despite such effects, we guarantee an *effective share* as the reservation. The actual share assigned then considers both the requested share and the slowdown/speedup. Furthermore, we decide whether time-coscheduling is meaningful at all or applications better run with exclusive resource allocation. To make these decisions, we provide a coscheduling performance model and a decision system which are further discussed in Sections 3.6 and 3.7. The different components are shown in Figure 1.

We currently mostly simulate our approach, on the basis of having implemented the core scheduling algorithm. We can also start and join real applications on the local machine (without any handling of errors etc.) to demonstrate slowdown effects in coscheduling and that the scheduling algorithm works under practical runtime conditions. The scheduler keeps a scheduling plan (which job is started when, including all reservations) in addition to a job queue. The scheduler basically considers priorities, and requested shares. Jobs with same priority are currently ordered according to their requested share (though other policies are possible like considering submission order). To prevent starvation, priorities are combined with an aging scheme (jobs are aged every T_{age} minutes). Different policies can be applied to assign priorities. We currently roughly classify job size (runtime) and assign different priorities for the classes *short*, *medium*, *large* (as some practical job schedulers do); class *special* is very short of otherwise explicitly assigned.

Cross-site jobs are not shifted even if a higher-priority job arrives or previous jobs finish earlier than expected. For cases where the discrepancy to the reserved start time becomes very high and/or there are large time slots which cannot be filled, later a protocol may be added to try a global cross-site rescheduling of the job. Or the cross-site application may be informed if more CPU time becomes temporarily available and allow it to schedule more work on the remote site.

The scheduler considers coscheduling only if one of the following applies:

- a job is scheduled cross-site and cannot take benefit of getting a higher share than it has reserved
- a job arrives which has higher priority than the currently running one(s)
- jobs have same priority, are classified as coschedulable and slowdown is below a certain limit $CoSl_{Max}$ which may be weighted by the job runtime
- there is a benefit (speed-up) from coscheduling

Furthermore, of course, in any case, the request must be compliant with the available shares. The rationale for the restricitons is that in most cases (i.e. unless long

latencies such as from I/O are involved) coscheduling increases the overall execution time for the individual job. Then, overall throughput is better if running the jobs one after the other - except if a job has made a reservation. Slowdown is less critical for short and more critical for long jobs - thus relating $CoSl_{Max}$ to job runtime makes sense. Jobs may be classified as not coschedulable due to their characteristics - e.g. if needing all memory.

For all jobs in the scheduling plan, we keep references to the application descriptions, their shares etc. If remote requests consider to schedule a job, preliminary reservations are made until the remote site decides whether to use or not to use the site and which slot is preferred. We run one remote protocol after the other (i.e. we serialize the requests and they cannot interfere with each other). To protect against failure and malicious sites, the preliminary reservations expire after a certain timeout.

We never give 100% of CPU time to any job and always keep a share for urgent short tasks (for cases where already long jobs are running) and for overruns. obs that excede their reserved runtime (as a matter of the difficulty to make proper estimates) get their share reduced but can continue to run in an *overrun share* (this idea is taken from DSRT [2]). In practical usage, jobs might be preempted or terminated after a certain grace period (proportional to their runtime). Using an overrun share helps to deal with fixed cross-site reservations which cannot be shifted. Other jobs are moved to later start positions (as we do if higher-priority jobs arrive). Closely related, jobs may finish earlier, either again because estimates were not met or because a higher share became available than was reserved. This changes the start time of the following applications and requires to recalculate the schedule.

Another problem is that jobs have different runtimes and thus shares along the runtime axis can fragment time shares. Thus, we avoid this situation and first try backfilling. For all remaining time, we let the application(s) consume all resources to get a clear cut for the next job. Figure 2a) illustrates the problem. Solution c) would be o.k. but not b). Note that this clean cut is easily possible in time sharing whereas in space sharing jobs would have to be reconfigured. For a concrete scheduling example, see Figure 3.

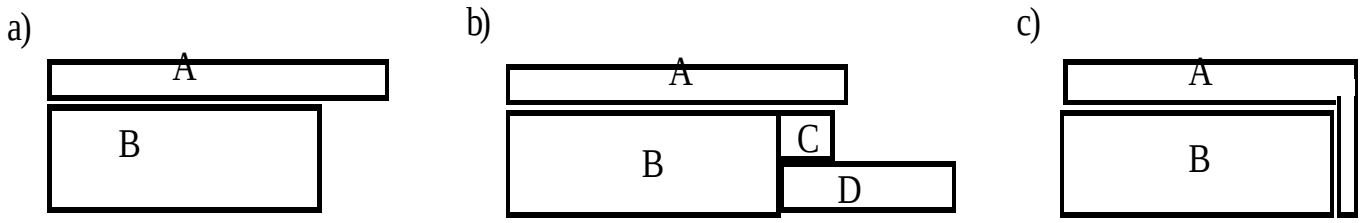

Figure 2. Basic scheduling plan (a) and two possible solutions (b and c).

3.3 The Global Reservation Protocol

We assume that global searches for proper sites are performed prior to the request as anyway accounts have to be established. This also implies that the number of sites under consideration is typically limited to a small number - likely less than 10. Furthermore, we assume that the suitability of the static machine characteristics is already checked in advance and that the runtime is estimated on the basis of a full time-share performance per site.

We propose a two-level protocol for reservation. First, remote sites can query rough load information to decide whether the site is chosen as a candidate for selection. At this stage the query has not yet any impact on the status of the site. In the next step, possible scheduling slots are identified and preliminarily reserved. This is the most costly part of the calculation as multiple plans have to be generated. At this stage, the remote site has to provide detailed information about the application characteristics and about other resource requirements than the CPU (memory, I/O etc.) The rough/fine protocol makes the selection faster if a larger number of sites is involved (otherwise the rough step can be skipped). Security and authentication would be especially important for the second step as it discloses details about the local schedule and the application.

As already mentioned, the assumption is that the requesting site collects all information about the individual different slots and decides about the final schedule. At this point a contract is made with all the involved sites about a specific scheduling slot. In detail, the steps are the following:

- *RoughLoadQuery* (*input:* estimated CPU time and requested share; *output:* earliest possible time slot)
- *FineSchedulingRequest* (*input*: detailed application characteristics, requested share, runtime estimate; *output:* possible scheduling slots and their share and start/ finish time; *effect:* preliminary scheduling plans and application characteristics kept until timeout or confirmation).

JOB_ID	PRIORITY	RUNTIME	SHARE	JOB_TYPE	SUBMIT_TIME	RES_TIME	START_TIME	FINISH_TIME	RESPONSE_TIME
1	0	300	40%	1	11:42:39		11:42:40	11:57:07	868
2	5	60	40%	1	11:43:09		11:43:10	11:46:23	193
3	5	60	20%-40%	1	11:43:39		11:43:40	11:50:02	382
4	15	10	20%	1	11:44:09		11:46:23	11:47:29	199
5	15	10	20%	1	11:44:39		11:47:29	11:48:34	234
6	15	10	40%	1	11:45:09		11:50:02	11:50:35	325
7	10	30	20%-40%	1	11:45:39		11:50:51	11:52:42	422
8	15	10	20%-40%	1	11:46:09		11:50:02	11:50:51	281
9	10	30	20%-40%	1	11:46:39		11:51:16	11:55:05	505
10	15	10	20%-40%	1	11:47:09		11:50:35	11:51:16	246
11	0	300	40%	2	11:56:49	11:57:00	11:57:07	12:12:56	966
12	5	60	40%	1	11:57:19		11:57:20	12:00:34	194

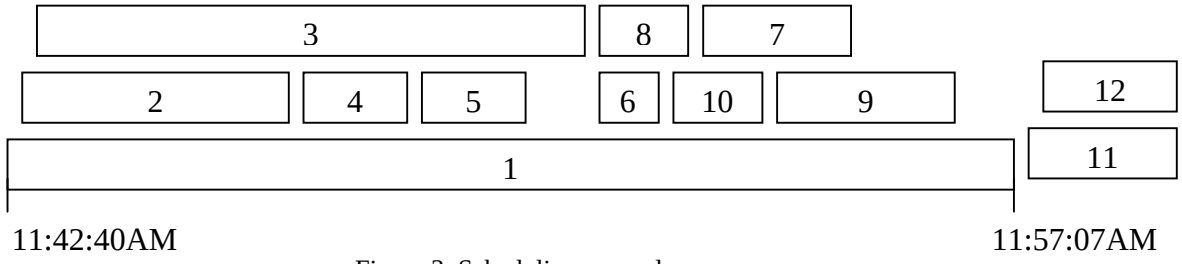

Figure 3. Scheduling example.

- *ConfirmSchedule* (*input:* slot to be confirmed; *effect:* contract agreement, certain slot/plan accepted/reserved and other preliminary plans released, application characteristics stored)
- *TerminateQuery* (*effect:* all slots released, application information removed)

Note that there is always at least one time slot available at the end of the job queue.

3.4 The Dynamic Directory

The dynamic directory keeps information about all currently running and all scheduled applications (jobs in the scheduling plan). Our dynamic directory is a significantly extended version of the rough idea described in [11]. The following information is maintained:

- owner (user)
- remote request yes/no, single-site/cross-site request
- requested share, runtime estimate (and potentially maximum usable share, notification of change yes/no)
- communication pattern (like “all-to-all”, “point-to-point random”, “central”)
- communication frequencies/sizes
- memory, I/O, and other requirements

In the future, we intend to optionally keep distribution functions for certain parameters and to keep structured information for multiple patterns (different phases, multi-level). This includes local/remote patterns with e.g. more intensive communication locally and less intensive remotely.

Furthermore, we assume that information per user (accounting) is maintained (in a database): permitted and left overall resource usage per time interval (e.g. month); any restrictions (like access only during certain hours, e.g. night, etc.), maximum runtime per application; general priority or for certain types of user’s applications; potentially performance information from previous runs (to support estimates) - as much of it is used in Maui [7].

Our design envisions a second dynamic part for the directory which can keep information during the run of the applications (like application-internal workload), see Figure 4. Standard and user-specified information may be transfered, pull and push versions be provided, and standard tools be used or specific ones be chose. The integrated service then provides the chance to combine measurements needed for several applications. Furthermore, certain information - like overall workload - may be summarized and made available to all applications vs. other more specific information be private.

We assume that general authentication would be provided by grid middleware like Globus. In addition, a special more lightweight authentication would be required to control which information is accessible to whom.

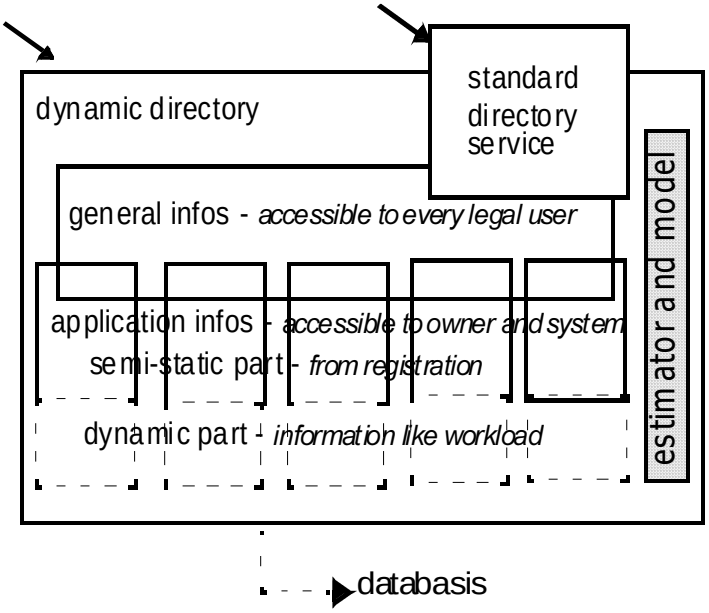

Figure 4. Dynamic resource directory.

3.5 The Coscheduling Estimator

SCOJO uses the provided characteristics of the new job and of the already scheduled jobs. For possible scheduling time slots (slots that fit the application including runtime and time share), we then estimate the slowdown or benefit, taking the characteristics of the other potentially concurrent applications into account. Slowdowns are considered acceptable only under certain circumstances and up to a certain extent. Note that a slowdown of 1.5 means that the runtime increases by a factor of 1.5. Thus, 2 applications which each need 1 hour runtime would be finished after 1 or 2 hours with average response time of 1.5 hours. With a slowdown factor of 1.5 each would have mere runtime of 1.5 hours and they would both the completed after 3 hours, i.e. average response time be 3 hours. But a slowdown of 1.5 would be well acceptable if a short application needing 1 minute can be scheduled quickly while running 1.5 minutes instead of having to wait 1 hour before being started. Benefits can be obtained from coscheduling if I/O or long-distance communication delays are involved.

For cross-site applications, the scheduler provides a list of times when jobs may be scheduled, with which effective CPU share and with which runtime. Otherwise, the best-possible slot is returned (shortest finish time). Important is that we calculate the effective share (the time contributing to the real progress of the application, i.e. not counting slowdown effects) and make sure that each application that got a share granted gets computation time of at least that amount. Thus, if an application got a share of 30% CPU time granted and due to coscheduling there would be a slowdown of 1.33, the application would be granted 40% of the CPU time to keep its effective share. The other application(s)' effective share needs to fit into the remaining share of CPU time.

Table 2. Slowdowns in different application combinations. All runs are on 9 CPUs, 80-85% idle (either directly or 93-95% and via 2 extra processes as background load from other users varies). This was done since there is always minor background load with a few percent of CPU time and 1 or 2 CPUs busy even during the night (measurements were done on our common department server). Numbers in parenthesis are granularities. Left value represents slowdown for row, right value represents slowdown for column application.

	grid-300 (1.3 msec)	**grid-1200** (27.9 msec)	**grid-2400** (116 msec)	**central** (29.3 msec)	**random** (4 msec)	**stream** (3.5 msec)	**seq**
grid-300	1.2	1.4 / 1.1	2.2 / 0.9	0.9 / 1.6	1 / 1.3	1.8 / 0.8	2.5
grid-1200		1.2	1.4 / 0.8	1.1 / 1.7	1 / 1.4	1.3 / 0.8	1.2
grid-2400			1.1	1 / 3.1	1.5 / 2.3	0.8 / 0.8	1
central				1.3	1.3 / 0.9	2.5 / 0.8	3.3
random					2	1.8 / 0.9	3.5

Modeling and cost estimation include both the application and the machine characteristics as coscheduling effects depend on the operating and the communication system. First, a rough classification is done whether coscheduling is possible or not. Next, the slowdown/speedup is estimated.

In previous work [12], we found the following application information to be relevant:

- communication pattern
- communication frequency and size
- asynchronous communication yes/no, probing yes/no
- combination of jobs being scheduled

Important cost contributions are the switching cost, the spinning cost (both in the application and in the spin daemon used in SUN MPI), and potential waiting time because of dependencies. If one process/thread of the application stays behind, none of the others may be able to make progress. Thus, a central master task cannot send out new requests before the previous results are received. Therefore, we also consider the average maximum delay. Furthermore, the probability for each cost to occur has to be taken into account. See Table 1 for a summary of cost terms. We propose the following performance model, describing the extra cost per step of the application:

Table 1. List of cost terms.

Cost Term	Explanation
T_{delay} (Env)	idle time (unused resources because of wait, dependent on other processes)
$P_{ncosched}$ (Env)	probability not to be coscheduled
T_{switch}	process switching cost
T_{spin}	spin time before descheduling
T_{slice} (T_{gran})	time slice, dependent on granularity
T_{gran}	own granularity

$$T_{cost}=P_{ncosched}(Env)*T_{delay}(Env) + P_{ncosched}(Env)*(\min\{T_{spin},T_{gran},T_{slice}(T_{gran})\}+T_{switch})$$

In our measurements with SUN MPI under Solaris with Resource Manager, we found that almost a whole CPU is occupied by the spin daemon (serving all MPI applications) and that therefore one CPU should be "reserved" for it. Furthermore, we could not take advantage of the Resource Manager as it assigns shares to users and our user-level scheduler starts all applications. We found that runtime then is not distributed fairly, i.e. equally (which is consistent with observations in [10]) but that jobs with larger granularities get a larger amount of runtime, i.e. there is an additional effect of $R_{share}* Shift_{share}(Env)$. This demonstrates that true share assignment would be important for predictable performance.

4 Experimental Results

4.1 Coscheduling Effects

Table 2 shows results from coscheduling measurements. The application used are Grid (real simple heat distribution calculation, 4-neighbor communication) with different problem sizes / granularities, Central (synthetic, iterative master-slave), Random (synthetic, random point-to-point with probing). As can be seen, the combination plays a significant role. Observations are that larger granularities of one application (leading to larger time slices) have a negative effect on the coscheduled application, especially if it has strong dependencies.

In tests, we found that T_{switch} = 0.09 msec and $T_{spin} \approx 5$ msec (measured by finding a granularity in a server process from which on there is a difference between limited spinning and unlimited spinnning). T_{slice} is between 20 and 200 msec in Solaris. A rough estimate is that $P_{ncosched}$ = 1 for Grid (except to 300 which accomplishes coscheduling) and the server of Central. For Central, this does not yet explain the observed slowdown. We found a higher runtime share for the applications with larger granularities which explains why in some cases they have a speedup. (This even applied when starting the corresponding applications manually from 2 different accounts which should make the resource management effective.) The reason is probably that the waiting states of the finer-grain processes let the CPU be given to the larger-grain processes which then run for a while and get increased CPU usage. Dependency delays should not produce a slowdown of e.g. larger than 2 for Central as spinning time is relatively small vs. granularity and at worst a single task of each application is finished from the previous iteration while leaving the other CPUs idle. However, it was impossible to extract the details of this behavior (CPU scheduler) to obtain the values of our parameters, and we therefore could not verify the estimations. For our scheduler tests, we have taken the slowdowns as measured and shown in Table 2. These results are slightly better than reported in [12] because we now leave some CPU share for background load.

4.2 Scheduler Benefits

We have done several simulations to demonstrate that our scheduling algorithm works and show which benefits it has. We have oriented our tests slightly (i.e.

with respect to percentages in the different job classes though not with respect to actual runtimes which are too high for our test conditions) to the workloads measured in [3] on a distributed-memory machine (iPSC/860 at NASA AMES), Case 1, and in [1] on a DSM machine (Origin 2000 at NSCA), Case 2. We compare FCFS, mere priority scheduling (Pri), priority + coscheduling (PriCo), and priority + coscheduling + backfilling (PriCoB). Our maximum multiprogramming level is 2 (i.e. 2 jobs run at the same time). For Case 1, we have extended the simulation toward using our sample programs and running them on our SUN SMP server (Enterprise 6500 running Solaris) to check actual coscheduling behavior. However, we had to confine our experiments to equal shares for the resons explained above. Slowdowns are taken as shown in Table 2). We have simulated 36 jobs: 11% long jobs (30 min, Grid-2400), 11% medium jobs (8 to 10 min, Random and Grid-300), 16% short jobs (3 to 5 min, Grid-300 and Central), and 60% very short jobs (1 to 1.5 min, Grid-1200 and Stream). Two of the long jobs are cross-site jobs. Job submission is such that the interarrival time of long jobs is 40 min, whereas the others are equally spread. As can be seen in Figure 5, PriCoB provides the best average response time and the best relative average response time (response time in relation to job runtime). The former reduces from 45 to 25 min vs. FCFS. The latter from about 14 to about 3.

For Case 2, we have not actually run programs but fully simulated the workload, using 40 jobs and assuming that all applications have a slowdown of 1.2. We have chosen 15% long jobs (5 min), 20% medium jobs (1 min), 30% short jobs (30 sec), and 35% very short jobs (10 sec). The long jobs are submitted every 10 min followed by various mixtures of other jobs. The average response time reduces from about 5.5 min to about 3. The average relative response time from about 12.5 to about 1.8. See Figure 6. However, worth to note, the overall

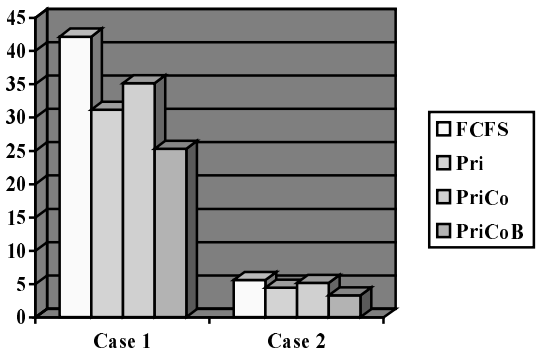

Figure 5. Average response time for Case1 and Case2.

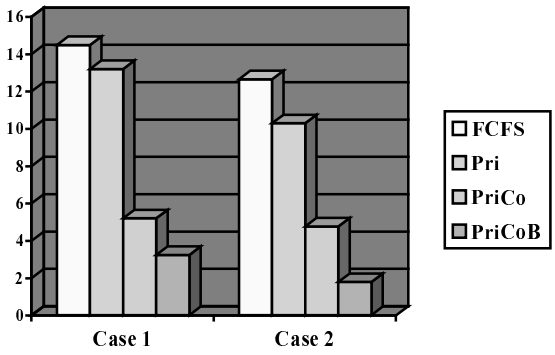

Figure 6. Average relative response time for Case 1&2.

execution time for the workloads increased due to the slowdown effects. In Case 1, the execution times are 3:34h for FCFS and for Pri, 4:08h for PriCo and 4:29h for PriCoB. In Case 2 the numbers are 45min for FCFS and for Pri, 47min for PriCo, and 55min for PriCoB. Case 1 demonstrates that better coscheduling approaches are needed which keep the slowdown low.

Finally, we have simulated different share assignment and considered multiprogramming levels of 2 and 3 (Case 3). The guaranteed shares for the first 2 coscheduled jobs is 40%, leaving potentially 20% for a third job. Slowdown is set to 1.2 for all jobs if coscheduling 2 jobs (C2), and 1.3 if coscheduling 3 jobs (C3). We have simulated 40 jobs: 10% long (5 min), 20% medium (1 min), 20% short (30 sec), and 50% very short (10 sec). The long jobs were submitted every 14 min, immediately followed by medium jobs. Short and very short jobs arrived arbitrarily. Only PriCoB was tested. Figure 7 shows that the average response time drops from 6 to 3.7 min if coscheduling 3 jobs. Average relative response time is almost the same.

Thus, overall we have shown that coscheduling provides benefits, especially if the percentage of short and very short jobs in the workload is high. Furthermore, we have demonstrated that the algorithm works properly for cross-site and local reservation submissions, priorities, backfilling, shares, and slowdowns from coscheduling.

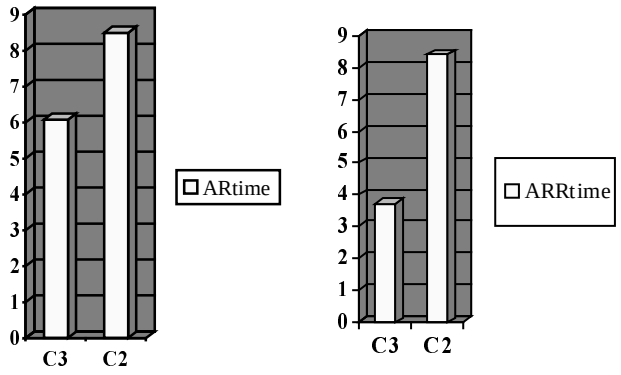

Figure 7. Average and average relative response time for Case3 with varying time shares.

We also have checked that remote requests get the proper lists of possible time slots and that the overflow share works properly (needed in Case 1 as real runs cannot be 100% accurate in time vs. estimate).

5 Summary and Future Work

We have presented a scheduling approach which is capable of making starttime and share reservations for cross-site jobs. Shares are used to coschedule applications in a predictable manner. The main contributions are that coscheduling effects are considered on a per-application and per-combination basis and that guaranteed shares are effective shares. The latter take slowdowns from coscheduling into proper consideration to keep the share reservations. We have incorporated the well-known backfilling approach from space sharing to our share assignment.

Furthermore, we base our estimations on an integrated dynamic directory which keeps information about the applications in the scheduling plan and can later be used to keep dynamic-execution information. We have implemented the core scheduling algorithm and shown that the approach works and that we can obtain better response time than with FIFO or pure priority scheduling. This applies under the assumption that workloads have many short applications.

We have demonstrated coscheduling effects between different communication patterns on sample applications under Solaris and SUN MPI. Flexible share allocation has been demonstrated via simulations. We expect share-based allocation (on the basis of systems like QLinux) to gain larger importance in the future, indicated by the fact that QoS and DSRT have found strong interest in the grid community.

As future work, the goal is to integrate our scheduling approach with Globus or Condor and a local scheduler like PBS, LSF or Maui. This means to extend the local scheduler, put the grid scheduler as a layer on top of the local scheduler and let it interact with the local scheduler.

Acknowledgements

We thank the reviewers for their valuable comments and Feng Yang for an initial directory implementation.

References

[1] S.-H. Chiang and M.K. Vernon. Characteristics of a Large Shared Memory Production Workload. Proc. JSSPP, 2001.

[2] H. Chu and K. Nahrstedt. CPU Service Classes for Multimedia Applications. Proc. IEEE Multimedia Systems, June 1999.

[3] D.G. Feitelson and B. Nitzberg. Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860. Proc. JSSP, 1995.

[4] D.G. Feitelson, L. Rudolph, U. Schwiegelsohn, K.C. Sevcik and W. Parkson. Theory and Practice in Parallel Job Scheduling. In Proc. JSSSP, 1997.

[5] I. Foster and C. Kesselman, The Globus Toolkit. In I. Foster and C. Kesselman (eds), The Grid - Blueprint for a New Computing Infrastructure, Morgan Kaufmann Publishers, San Francisco/CA, USA, 1999.

[6] GRAAP-WG: http://people.man.ac.uk/~zzcgujm/GGF/graap-wg.html, Nov. 2002.

[7] D. Jackson, Q. Snell, and M. Clement. Core Algorithms of the Maui Scheduler. Proc. JSSPP, 2001.

[8] C. Pinchak, P. Lu and M. Goldenberg. Practical Heterogeneous Placeholder Scheduling in Overlay Metacomputers: Early Experiences, Proc. JSSPP, 2002.

[9] Solaris Resource Manager, http://wwws.sun.com/software/resourcemg, 2002.

[10] P.G. Sobalvarro, S. Pakin, W.E. Weihl and A.A. Chien. Dynamic Coscheduling on Workstation Clusters. Proc. JSSPP, 1998.

[11] A.C. Sodan, Towards Asynchronous Metacomputing in MPI, Proc. HPCS, 2002.

[12] A.C. Sodan and M. Riyadh. Coscheduling of MPI and Adaptive Thread Applications in a Solaris Environment. Proc. IASTED PDCS,2002