Imprecise Calendars: an Approach to Scheduling Computational Grids

Jeffrey K. Hollingsworth Songrit Maneewongvatana
Computer Science Department
University of Maryland
College Park, MD 20742
{hollings,songrit}@cs.umd.edu

Abstract

We describe imprecise calendars, a way to organize and schedule clusters of nodes in a computation grid. Imprecise calendars permit the easy and efficient sharing of resources between different clusters of computers that are part of a computational grid. In addition, they can be used to provide specific time reservations for applications. We describe the algorithms and policies for manipulation of imprecise calendars. We also include a series of simulation studies that compare our approach to previous batch scheduling systems for both a single cluster and collection of clusters up to over 3,000 nodes.

1. Introduction

The performance of individual computers continues to increase, yet there remain important classes of application that have computational demands that far exceed the available capacity of the fastest micro-processors. Traditionally, many of these applications have been run on parallel vector supercomputers, or more recently, parallel machines built from standard micro-processors. However, as the number of vendors of these machines, not to mention their fraction of the overall computational market, has shrunk, an alternative computational structure has started to emerge. Computational Grids[3], the coordinated use of multiple, often geographically distributed, clusters of computers, provide a way to meet the needs of these demanding applications. However, one of the major challenges in creating computational grids is to provide a way to coordinate the use of the combined nodes of multiple semi-autonomous sites.

In addition to coordinating access, computational grids need to support the use of nodes at a specific time. Traditionally, large-scale computational resources have been managed as batch systems in order to maximize the utilization of scarce and expensive machines. However, for a variety of tasks that can utilize the processing power of grids, it is critical that the nodes be made available at a specific time. For example, if the data to be processed on the grid is coming from a large scientific instrument, such as a telescope or particle accelerator, time on the instrument must be reserved in advance. For grids to be able to process this type of data, they too must support this type of reservation. Likewise, for applications that require one or more people to be involved, such as real-time virtual reality visualization of scientific simulation, or interactive steering and debugging, the traditional batch queue model used by large-scale computing centers is not sufficient.

To meet these scheduling challenges, we have been working to develop a new approach to scheduling applications based on an extension of the way many people schedule their daily lives. We call this model an imprecise calendar. Imprecise calendars are a way to organize and schedule jobs of various durations and needs. In addition, imprecise calendars permit the scheduling of jobs at specific times in the future, often called reservations. Our technique works at the level of a single cluster or global confederation of clusters. Section two describes imprecise calendars, Section three describes the results of a simulation study of imprecise calendars and a comparison with previous scheduling disciplines, and Section four reviews related work.

2. Imprecise Calendars

Imprecise calendars are a hierarchical scheduling system. They are designed to mirror the process many people use to allocate their time. When planing events far in the future, a general (somewhat vague) idea about what events are pending and when they will happen is maintained. The larger the task, the more exactly we allocate time for it. For example, a weeklong trip has very specific planning in advance. However, a minor event is only vaguely planned until it is fairly close to the event. One reason people use this type of schedule is that it allows for last minute minor events (minor in terms of the time required) to be added to fill in the items on a calendar. This approach also permits people to plan imprecise meetings, and then refine the time of the meeting, as the specific time becomes closer. For example, two colleagues might suggest getting together to discuss a project they are working on “late next week”, and then as the time of the

meeting got closer, a specific day and eventually a specific hour can be agreed upon.

We propose to use a similar type of scheduling approach to manage the resources of a computational grid. The basic idea is to use a multi-resolution temporal data structure to enable efficient (and coordinated) use of a collection of computers. Like individuals using imprecise temporal planing, a supervisor or manager might employ imprecise calendars to their workers. For example, a manager in a factory might plan when tasks are to be performed but wait to assign them to specific workers until just before the tasks begin. We propose to use a similar approach to management of groups of nodes. One advantage of imprecise calendars is that they provide a compact representation of the data to be managed, and permit "manager" nodes to see the global picture without worrying about the tasks assigned to individual workers.

In this environment, compute servers are just a machine where a task is processed. Servers can be a single machine or a group of computers. Servers can be dedicated machine room computers, or workstations on peoples desks. Users submit their job to one of the job managers closest to their machine. Tasks in this environment can be of any size, from a few seconds on a handful of nodes to hours on hundreds of nodes. Tasks that need only a few nodes or are short in duration can be effectively executed locally and should not be included in the higher levels of our scheduling system. On the other hand, big tasks normally require some cooperation between machines or set of machines, therefore planing and coordination between collections of clusters is required.

2.1 System Components

In this section, we describe the components of our scheduling system. We first present our abstraction and then describe how we manage imprecise calendars. The key abstractions are:

Nodes and managers: A node is a place where tasks are executed. A node is the finest unit of scheduling visible to the grid-aware scheduler. A node could be a single processor, a Symmetric Multi-Processor (SMP) system, or a cluster of workstations. Within a node, a local scheduler makes decisions about when processes are scheduled, or for SMP nodes, which processors run which jobs.

Tasks frequently need more resources than any single node can provide, and so we group a set of nodes into a cluster controlled by a manager. We treat a cluster of nodes as a single entity that can be viewed as a powerful logical machine. Several server clusters can be grouped to form an even more powerful cluster. This structure forms a management tree where the root of the tree represents the whole computing system that users can access.

Manager nodes can be organized based on a variety of criteria. One reason for assigning manager nodes is to group nodes that are part of a single administrative domain (i.e., administered by a single organization) into a cluster. A second reason to assign manager nodes is based on the effective distance between nodes (i.e., the quality of the communication links). A third criterion for organizing manager nodes are to group systems with comparable capabilities together. For example, single processor workstations might be one cluster, and a group of 32-way SMPs would be a separate cluster even if they are in the same building and have identical network connections.

Slot: A slot is the fundamental unit of resource allocation in our system. The duration of all slots are integer multiples of a fundamental slot. A fundamental slot is a system wide parameter that represents the finest unit of temporal scheduling employed on the grid. We anticipate that the duration of a fundamental slot will be on the order of tens of seconds. We use S to denote a time slot. S_0 is the fundamental slot. S_1 denotes the first level multiple of the fundamental slot and so on.

Calendar: A calendar is a data structure to store information about allocation of nodes. Each manager maintains a calendar about the individual resources (or lower level managers) it manages. Each manager has to know when to run each task, and information about the predicted resource requirements of each task. Managers use their calendars to determine the slot allocations.

Calendars consist of an array of time slots. The status of a slot can be one of three values: free (A node will be idle at that time), partially reserved, or reserved (there is no more room for any new tasks). The start time of a slot is the time that the node or nodes begins to schedule the tasks in that slot. In calendar-based scheduling, the size of the slots increases as they are further away from the present time. However, it is not necessary that the slot size increase with every adjacent slot, instead several adjacent slots can have the same time scale. We explore a variety of different slot durations in Section 3.1.

Tasks: Tasks can be one of two types. They can be independent tasks that do not require synchronization with other tasks[1]. Alternatively, tasks can be part of group that communicates with other tasks on other nodes. We have to co-schedule tasks from the same task group such that the start time of the slots they reserved are the same.

2.2 Calendar information

At all times the first few slots in a calendar will be at the finest temporal resolution. As we look out along the

[1] Individual tasks might be sequential or small-scale parallel jobs depending on if nodes are uni-processors, or multi-processors.

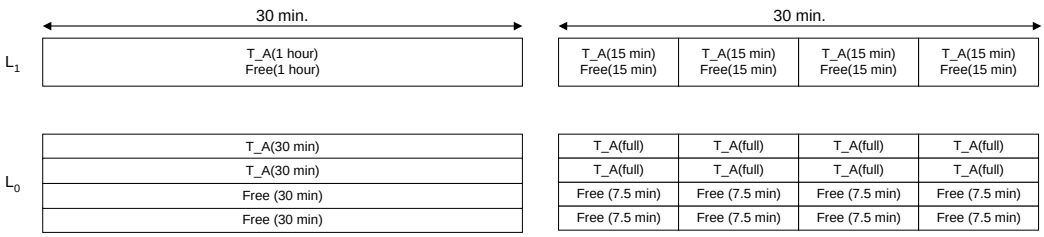

Figure 1: Calendar Data Structure at Four Different Resolutions.

calendar, the start time of slots is farther away, and the temporal granularity of slots becomes coarser. The last few slots in calendar are at the coarsest temporal resolution, S_n. As time progresses, the number of S_0 slots decreases because the time period they cover ends. When the number of slots of size S_0 reaches a low-water threshold, the node partitions a S_1 slot into several S_0 slots, and a similar process repeats up the temporal hierarchy to S_n. Nodes also maintain the size of their calendars (i.e., the summation of their slot sizes). When a slot is consumed, the calendar size decreases. If the total calendar duration falls below the required scheduling horizon, a new S_n slot is added to the end of the calendar. The overall size of the calendar is defined by the needs of the jobs being scheduled. For example, if many jobs are submitted, the calendar duration will grow so that all jobs have an assignment in the calendar.

For each slot, our scheduling policy maintains a variable, *total resources*, the amount of resource available in that slot. Usually, the resource is CPU time. Therefore, the total resource is a product of number of computers under the manager's control and the CPU time of each computer. However, for nodes of different computational speed, a per processor normalization constant is applied.

On a per calendar basis, we have a parameter, *Resource threshold*. This is a per-slot size limit on the amount of a resource that we will commit in advance. The reason for this limit is two fold. First, we want to allow some amount of the resource to be available for last minute tasks. Second, we limit commitments to ensure we can meet our scheduling obligations as we firm up obligations on our calendar. If more than one task is assigned to a single slot, we should limit the amount of resources used since some of these tasks might be parts of larger jobs that need to be co-scheduled on the grid at the same time.

Figure 1 shows an example of the process of dividing slots. The upper left representation of a calendar shows a time-schedule for a 30-minute interval on four different nodes. It is represented as a single slot since it has been compressed both spatially and temporally. There is one task that requires only 1/2 an hour on two nodes in the slot, and the rest of the slot is free. This is the granularity that would typically be maintained by a manager node. The top right figure shows the same period, but the temporal resolution has been refined to show four slots of 7.5 minutes each. This would correspond to a refinement of the temporal schedule of a manager node as the current time approaches the 30-minute interval depicted. The lower left representation of the calendar shows the view each node has of its schedule at the coarser temporal resolution. Finally, the lower right calendar shows all four slots for each node.

A final aspect of our scheduling policy is the tracking of the trade deficits (or surpluses) accumulated by each node. Each node and manager maintains a balance of trade variable. The account balance is the balance of CPU time it borrows from other nodes or lends to other nodes. This account balance helps the system to monitor which nodes (or which users) use excessive amount of CPU time and to limit the requests from such nodes.

2.3 Assigning tasks

When a request for a resource arrives, the system must determine when to allocate the task. We use several policies in making these choices. The key issues are that when a manager receives a request, it must determine a set of slots where that job will be executed. When selecting a set of slots, the manager node determines if its worker nodes can accommodate the request in a timely manner.

To ensure that when slots are split all commitments can be satisfied, we reserve a full slot for any request. To see why it is possible to have an allocation that can not be scheduled when a slot is divided, consider the example shown in Figure 1. If we had two requests, one for one node for 30 min. and a second for four nodes for 7.5 minutes, we would be able to accommodate this reservation at the highest level. However, when we try to accommodate this request at the finest granularity, we discover that the request can not be scheduled since one job needs one node for the entire duration and another needs all the nodes for ¼ of the time.

While restricting the system to one job per slot may seem wasteful, three factors make this policy acceptable.

First, short duration events tend to get submitted into the system near their execution time, and will likely get assigned to S_0 sized slots. Second, slots that are fully occupied by a single job can be 100% allocated. Third, we employ a variation of backfilling to move forward the execution time of jobs when slots are divided and when a job exits the system before its reserved time has ended. In fact, the use of a slotted time in our scheduler often improves the ability to backfill jobs since the length of all jobs is an integer multiple of the fundamental slot size. To backfill, our algorithm starts with the highest-level manager node that has this slot type, and once it has backfilled its jobs, its sub-mangers backfill any local jobs. Our backfill algorithm is based on Feitelson and Weil's [2].

For multi-node tasks, we have an additional requirement: the number of nodes must be at least as many as the job requires. If a job submitted to a manager requires more resources than it manages, it forwards the request up to its manager. This process repeats until a manager with a sufficient number of nodes is located.

Our approach to finding a free set of slots by a manager node is based on a first fit allocation scheme. However, if the first available calendar slot for an application is too far in the future, the manager node sends the request up to its manager to see if the job can be located on a cluster under the supervision of a peer-manager.

Requests to run jobs at specific times are easily accommodated using imprecise calendars. Each start of slot implicitly encodes a specific time when it will start. Jobs with a reserved starting time are flagged as reservation jobs. This flag precludes backfilling the job and thus moving it before its reservation. The reservation flag also ensures that when the slot is divided that the job is assigned to the first new slot to maintain the job start time.

2.4 Calendars and Job Managers

To provide scalability, we need to ensure that managers and workers don't have to keep their calendars in lock-step synchronization. To do this, job managers keep shadow calendars of nodes and sub-mangers in the system. However, a job manager's copy of a calendar may not reflect the master one at the first-level nodes, and a request for a reservation might fail. Managers' calendars are simply hints about the available resources of the worker nodes, and the worker nodes must be consulted before making a specific allocation.

There are two types of requests from job managers: inquiry and reserve. Inquiry requests ask whether the node has any slots that have enough resources at the specified time. The node checks if the needed time slot is still available. If sufficient resources are no longer available, the node declines the request and informs the manager when the specified request can be satisfied. Reserve requests ask nodes to allocate a time slot for it. This is a variation on a two-phase commit protocol and is used to ensure that multiple requests to different sub-managers either commit or abort atomically.

In summary, the salient properties of imprecise calendars are:

Time slots scale: Nearer slots have finer granularity, further away slots have coarser resolution and less precise start time information. Thus, we do not have to fix the absolute starting time of jobs when they are submitted

Resources scale: We group nodes into clusters that also can be viewed as a logical machine. The number of levels can be arbitrary. Also, smaller tasks do not have to go very far up the hierarchy to find a desired resource. Larger tasks can gain the benefit of using the larger collections of nodes available on the grid. By compressing spatially and temporally, we reduce the storage requirement of management nodes.

Job start time guaranteed: By using an assignment scheme based on one job per slot, we can ensure that we have slots for all the tasks after partitioning. This is useful because we do not have to select a victim task to miss its scheduled time, nor must we try to squeeze the victim into a later part of the schedule.

Fairness between big and small tasks: By using slotted time and limiting the maximum slot occupancy that can be reserved at specific temporal granularities, we help to ensure that small and large tasks can be accommodated.

3. Simulation Study

To evaluate the ability of our scheduling system to meet our objectives, we simulated a variety of types of clusters. In particular, we tried to demonstrate the ability of imprecise calendars to link together separate domains, and do so efficiently. In addition, we wanted to compare our approach to other approaches on a single cluster.

3.1 Slot Size and Duration

To compare the different slot sizes and assess their impact on the performance of our system, we simulated the imprecise calendars using a variety of different slot sizes and number of slots at each size. We used as our workload, the trace of jobs submitted to the Swedish Royal Institute of technology (ETH) SP-2 system during the months of October and November 1996. The distribution of jobs of various sizes in each workload is shown in Figure 2.

Since the size and number of slots are such a critical parameter for our system, we investigated several different combinations of each parameter. To evaluate imprecise calendars, we simulated six different combinations of slot sizes. The different slot sizes are shown in Figure 3.

In the first configuration we used a fixed slot size of 100,000 forty-second slots to provide a comparison with a traditional scheduling policy that didn't use temporal compression. Due to the size of the calendar data structure, this would not be a practical configuration for a real system. The second configuration had two temporal levels, 40 seconds and 2,560 seconds. This was designed to determine the penalty incurred by only having a few items at the finest temporal resolution. The fourth and fifth configurations vary the number of slots and their durations. The sixth configuration has 32 ten second slots, eight 320 second slots and as many 2,560 second slots as required. By creating many small and medium sized slots, we hope to achieve similar performance to configuration one, but with a smaller size.

Job Size	Number of Jobs	
	Oct. 1996	Nov. 1996
Total	2,404	1,982
< 10 seconds	156	194
10 to 100 seconds	1,005	580
100 to 1,000 seconds	496	376
Over 1,000 seconds	747	832

Figure 2: Distribution of Job Durations.

For each configuration, we computed the normalized mean queuing delay for all jobs in the system. The normalized mean queuing delay is the time a job spent from when it was submitted into the system until it completed execution, divided by it execution time. The idea of this metric is to allow us to compare the delays seen by both long and short jobs in a unified way. To access the impact of scheduling policies on different types of jobs, we also computed the normalized mean queuing delay for each of four different jobs lengths: less than ten seconds, ten to one hundred seconds, and one hundred to one thousand seconds. These cases are shown in the 2nd through 5th bars for each configuration in Figure 4.

Config.	<number of slots, size>
1	<100 000, 40 sec>
2	<64, 40 sec>, <*, 2560 sec>
3	<8, 40 sec>, <8, 320 sec>, <*, 2560 sec>
4	<2, 40sec.>, <2, 1280 sec.>, <*, 2560 sec.>
5	<2, 10 sec>, <2, 1280 sec>, <*, 2560 sec>
6	<32, 10 sec>, <8, 320 sec>, <*, 2560 sec>

Figure 3: Summary of Slot Sizes Used.

As Figure 4 shows, the mean weighted job queuing time increases from 5.6 to 7.7 for the October data, and from 4.8 to 5.5 for the November data when we switch from using 100,000 fine grained slots to only 64. The third configuration introduces a medium-scale resolution, however this actually decreases the performance of the scheduling system for the November workload since it creates unnecessary fragmentation that slows down small duration jobs (10 to 100 seconds). For both the fourth and fifth configurations, the lack of a large number of slots at the medium temporal resolution hinders the performance of jobs in the 10-100 second class. In the sixth configuration, imprecise calendars with a three level temporal hierarchy can represent the same time interval as 100,000 fixed-size slots yet requires only 1,511 slots, a reduction in size by a factor of sixty.

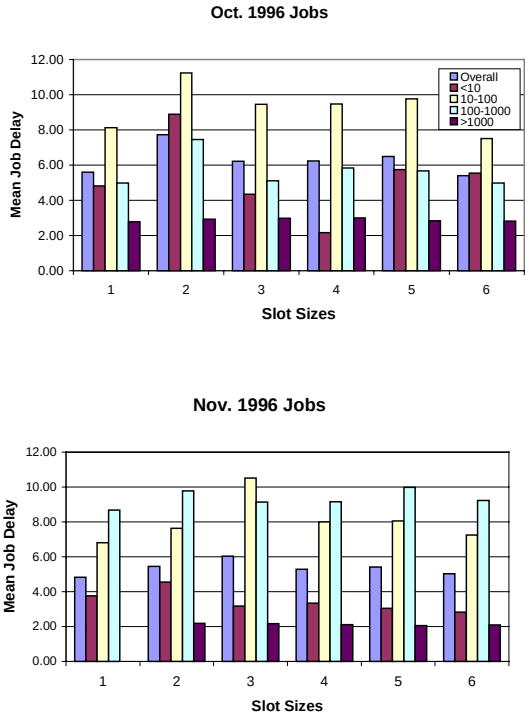

Figure 4: Varying Slot Size.

We also investigated how our scheduling discipline compared with existing batch scheduling techniques. For our comparison, we simulated a variation on a traditional backfilling scheduler that used the algorithm described in [2]. The results of that simulation for the same two months as the previous study are shown in Figure 5. The results show that for the October workload, imprecise calendars achieve approximately the same mean job queuing delay as the backfill-based scheduler. However, for the November workload, imprecise calendars were able to reduce the mean job queuing delay by a factor of two (from 10 to 5). These results indicate that imprecise calendars are roughly comparable to existing techniques for single clusters (and in fact better in some cases). However, the real value of imprecise calendars is to permit combining separate clusters into larger ones.

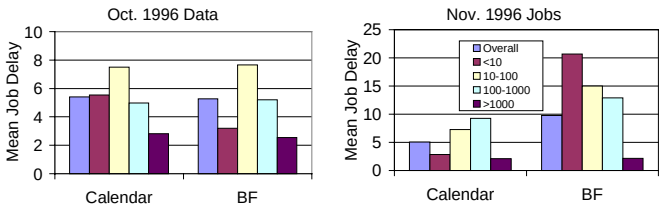

Figure 5: Imprecise Calendars vs. Backfiling.

3.2 Comparison with Partitioned Scheduler

A previously proposed strategy to schedule jobs of different sizes and to allow big and little parallel jobs to co-exist is splitting the cluster into sub-clusters of different size and using a First Come First Served (FCFS) scheduling policy in each sub-cluster. This approach, used on the TMC CM-5, ensures that small jobs don't get stuck behind big ones in the queue, however it can lead to inefficient allocation of nodes. To compare our technique to a partitioned scheduler, we ran our imprecise calendar simulator on data from the Los Alamos CM-5. The Los Alamos CM-5 used fixed partitions of two clusters of 32 nodes, and one each of 64, 128, 256, and 512 nodes. We compared our confederation of schedulers to running each queue separately with FCFS and traditional backfilling[2].

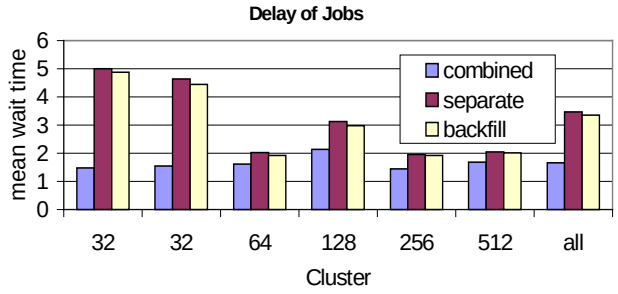

Figure 6: Partitioned Cluster Results.

The results, pictured in Figure 6, show that running the system as a combined resource can greatly reduce the queuing delays for small clusters. For both of the 32 node queues, the delay was reduced by over a factor of three. This set of simulations shows the potential of having small clusters participate in a computational grid even though they may not have that many nodes. By participating, they are able to gain resources from other larger clusters in the system when they are overloaded, and do so without causing any significant delay in the larger jobs on the bigger clusters. In fact, as Figure 7 shows, the overall utilization of individual clusters remained almost unchanged despite the reduction in mean queuing delay.

Queue	Size	Utilization		Trading		
		Separate	Comb.	Supply	Use	Balance
1	32	18.0%	18.1%	170.5	88.4	-82.0
2	32	21.5%	21.7%	162.3	85.4	-77.0
3	64	24.7%	24.9%	281.0	54.2	-226.9
4	128	36.4%	36.3%	64.3	456.1	391.8
5	256	38.9%	38.9%	136.2	84.6	-51.6
6	512	38.8%	38.8%	52.7	98.3	45.6

Figure 7: Queue Statistics for LANL Simulations.

We also computed the balance of trade for each of the queues in the cluster. The last three columns of Figure 7 show the number of node hours that jobs submitted elsewhere in the system ran on that queue, the number of node hours that jobs submitted to it were run elsewhere, and the net balance of trade. The results show that four of the six queues have a negative balance of trade and that two of them have a positive balance of trade. Interestingly, the two queues that had the greatest reduction in their mean queuing delay also had negative balance of trade. However, they had a larger percentage of their jobs run remotely than all other queues. So even though they gave more cycles than they consumes, the load balancing features of grid participation were significant.

3.3 Benefits of Clustering Large Machines

We also wished to investigate how well our system would work for a cluster of clusters. To do this, we need data from a collection of clusters. However, our available trace data only covers a limited number of computer centers, and is often for different types of clusters (e.g., SP-2 vs. CM-5 nodes). To provide a comparison of several similar clusters working together, we decided to use job submission data from the Cornell Theory Center (an IBM SP with over 300 nodes), and use traces from different months to represent different clusters. To do this we selected ten one-month intervals of data, and then simulated a ten-node cluster of clusters for the first eight days of the trace.

We treated the system as a two-level calendar scheduling system with the first level manager running 336 nodes, and the second-level manger running a cluster with a total of 3,360 nodes. Any jobs that requests more than 112 nodes (1/3 of a single cluster) or whose duration was at least 12 hours, we promoted to the second-level manager to be placed in any one of the ten clusters. We constrained those jobs sent to the second level scheduler to be run on a single cluster to ensure that the performance of a job would not be altered by running part of the job on

[2] We used slot sizes of 16x10sec, 16x40sec, 16x160sec, 16x640sec, and 16x2560sec for this study.

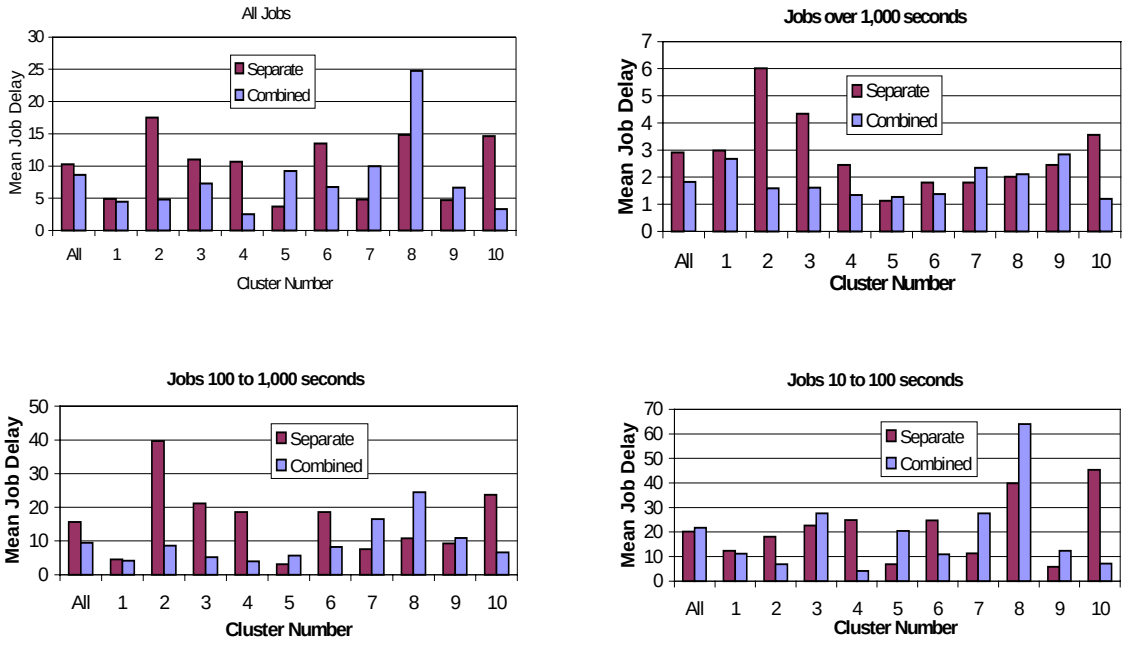

Figure 8: Cluster of Clusters.

different clusters with unknown bandwidth between them. The mean weighted job delay when using two-level management is shown in Figure 8. In addition, we also simulated the behavior of the cluster as if they were independently scheduled using our imprecise calendar scheduler. The first pair of bars shows the results for the entire cluster. The mean delay is reduced from 10.6 to 8.6, a tangible improvement of a 19%. The remaining ten pairs of bars show the queuing delay for each cluster. For six of the clusters the mean job delay is reduced. For two of the clusters, the mean queuing delay increased modestly. However, for two clusters, five and seven, the mean queuing delay more than doubled.

To understand the reason that nodes five and seven experienced increased queuing delay under a confederation of clusters compared with remaining isolated, we looked at the characteristics of the jobs for that cluster compared to the other ones. The summary of job characteristics is shown in Figure 9. Node seven had the highest overall utilization when jobs we run just on their submitting node. Also, notice that the biggest slowdown comes for those jobs that consume between 10 and 100 seconds of time (as shown in the lower right corner of Figure 8). These are jobs that are too short to be considered for submission to the upper levels of the cluster.

We also looked at the breakdown of queuing delay by job size. This data is shown in the three other bar charts in Figure 8. For medium and large jobs (100 to 1,000 seconds and over 1,000 second) there is relatively little variation between mean queuing delays of the clusters. However, for the 10-100 second jobs we see a fairly high variation in the mean queuing time. This is to be expected since, the load balancing achieved by scheduling the cluster of clusters as a unit only applies to larger jobs that get processed by the second level manager.

It is well known that having a single queue with multiple servers reduces queuing delays. However, it is interesting to see that in the simulation of the confederation of ten 336 node clusters where the mean number of nodes per job is less than eleven, that we were able to achieve a significant reduction in the mean queuing delay. This indicates that even among large clusters there is efficiency to be gained by being part of a computational grid.

We also looked at the balance of trade between the clusters. The right half of Figure 9 shows these results. For each cluster, we report the number of CPU hours supplied to other clusters' jobs, CPU hours of jobs for that cluster that were executed on other clusters, the net balance, number of hours spent executing local jobs, and the utilization of the cluster. It is interesting that although, only just over 2,000 jobs were eligible for execution on other clusters (out of a total job pool of over 17,000), these jobs represent 58% of the node hours in the workload. As a result, the majority of time for five of the ten clusters was spent running jobs from other clusters. The simulation studies have demonstrated the potential of imprecise calendars across a collection of clusters to im-

prove the performance of parallel jobs. Both large and small clusters can benefit from this approach.

#	Avg. Util	Two Level Combined Queues				
		Supply	Use	Balance	Local	Util.
1	34.5%	32,001	10,605	21,395	15,698	67.7%
2	72.9%	28,168	31,493	(3,326)	24,162	74.2%
3	79.3%	25,588	34,713	(9,125)	25,816	72.9%
4	70.8%	27,098	32,713	(5,615)	21,319	68.7%
5	55.2%	22,493	16,054	6,439	26,082	68.9%
6	65.4%	26,152	28,778	(2,626)	21,162	67.1%
7	63.6%	25,882	20,516	5,366	28,026	76.5%
8	72.9%	27,489	32,055	(4,566)	23,562	72.4%
9	61.2%	22,881	21,111	1,770	25,570	68.7%
10	77.3%	26,949	36,662	(9,713)	22,306	69.9%
All	65.3%	264,701	264,701	0	25,813	70.7%

Figure 9: Cluster of Cluster Job Results.

4. Related Work

Many non-FCFS policies have been proposed to allow small jobs to be accommodated without having to wait for large jobs to clear the queue[4, 7, 8]. However, these schedulers can result in arbitrary delays due to jobs moving ahead of each other. Feitelson and Weil [2] provide an alternative way to backfill jobs that does not have this limitation. Our system differs in that we used slotted time and allow clusters of clusters to support a general grid environment.

Another system that matches idle resources with requests is the Condor system[5]. Condor uses Classified Ads[6] to match resource suppliers and consumers. Their approach provides a flexible way to describe node attributes such as memory and processing speed. However, it is designed for sequential programs whereas imprecise calendars are designed for parallel programs.

Proportional-share scheduling[1] has been proposed to fairly allocate resources on a cluster of computers. It uses a variation on lottery scheduling to assign tickets to both parallel and sequential tasks. Our imprecise calendars are complementary to this approach. Imprecise calendars provide a batch scheduler that is designed to move jobs around the system, and to help control the number of active jobs at one time, while proportional-share scheduling can be used to decide which active processes to execute.

5. Conclusions

We have presented imprecise calendars, a new way to organize and manage the queues for clusters of high performance distributed systems. Imprecise calendars permit the easy and efficient sharing of resources between different clusters of computers that are part of a computational grid. In addition, we showed how they can be used to provide specific time reservations for applications. By using temporal and spatial compression of data, we are able to efficient represent the schedules of large clusters. We also described the algorithms and policies for manipulation of imprecise calendars.

We presented a series of simulation studies that compare our approach to previous batch scheduling systems for both a single cluster and collection of clusters up to over 3,000 nodes. For these large clusters, imprecise calendars provide an effective way to load balance the collection of clusters.

Acknowledgements

We thank Dror Feitelson for supplying the job workload data used in this paper. This work was supported in part by NSF awards ASC-9703212 & ASC-9711364, and DOE Grant DE-FG02-93ER25176.

References

1. 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'97)*. June, 1997, Las Vegas, Nevada.
2. D. G. Feitelson and A. M. a. Weil, "Utilization and Predictability in Scheduling the IBM SP2 with Backfilling," *2th Intl. Parallel Processing Symposium*. April 1998, Orlando, Florida, pp. 542-546.
3. I. Foster and C. Kesselman, eds. *The Grid: Blueprint for a New Computing Infrastructure*. 1998, Morgan-Kaufmann: San Francisco.
4. D. Lifka, "The ANL/IBM SP Scheduling System," *Job Scheduling Strategies for Parallel Processing*. 1995, Springer-Verlag (LNCS 949), pp. 295-303.
5. M. Litzkow, M. Livny, and M. Mutka, "Condor - A Hunter of Idle Workstations," *International Conference on Distributed Computing Systems*. June 1988, pp. 104-111.
6. R. Raman, M. Livny, and M. Solomon, "Matchmaking: Distributed Resource Management for High Throughput Computing," *Seventh IEEE International Symposium on High Performance Distributed Computing*. July 1998, Chicago, pp. 140-146.
7. D. D. Sharma and D. K. Pradham, "Job Scheduling in Mesh Multicomputers," *ICPP*. April 1994, Boca Raton, Fl,.vol.II, pp. 251-258.
8. J. Skovira, W. Chan, H. Zhou, and D. Lifka, "The EASY - LoadLeveler API Project," *Job Scheduling Strategies for Parallel Processing*. 1996, Springer-Verlag (LNCS 1162), pp. 41-47.