Scheduling in HPC Resource Management Systems: Queuing vs. Planning

Matthias Hovestadt[1], Odej Kao[1,2], Axel Keller[1], and Achim Streit[1]

[1] Paderborn Center for Parallel Computing
University of Paderborn, 33102 Paderborn, Germany
{maho,kel,streit}@upb.de
[2] Faculty of Computer Science, Electrical Engineering and Mathematics
University of Paderborn, 33102 Paderborn, Germany
okao@upb.de

Abstract. Nearly all existing HPC systems are operated by resource management systems based on the queuing approach. With the increasing acceptance of grid middleware like Globus, new requirements for the underlying local resource management systems arise. Features like advanced reservation or quality of service are needed to implement high level functions like co-allocation. However it is difficult to realize these features with a resource management system based on the queuing concept since it considers only the present resource usage.
In this paper we present an approach which closes this gap. By assigning start times to each resource request, a complete schedule is planned. Advanced reservations are now easily possible. Based on this *planning* approach functions like diffuse requests, automatic duration extension, or service level agreements are described. We think they are useful to increase the usability, acceptance and performance of HPC machines. In the second part of this paper we present a planning based resource management system which already covers some of the mentioned features.

1 Introduction

A modern resource management system (RMS) for high performance computing (HPC) machines consists of many vital components. Assuming that they all work properly the scheduler plays a major role when issues like acceptance, usability, or performance of the machine are considered. Much research was done over the last decade to improve scheduling strategies [8, 9].

Nowadays supercomputers become more and more heterogenous in their architecture and configuration (e. g. special visualization nodes in a cluster). However current resource management systems are often not flexible enough to reflect these changes. Additionally new requirements from the upcoming grid environments [11] arise (e. g. guaranteed resource usage).

The grid idea is similar to the former metacomputing concept [25] but takes a broader approach. More different types of resources are joined besides supercomputers: network connections, data archives, VR-devices, or physical sensors

and actors. The vision is to make them accessible similar to the power grid, regardless where the resources are located or who owns them. Many components are needed to make this vision real. Similar to a resource management system for a single machine a grid scheduler or co-allocator is of major importance when aspects of performance, usability, or acceptance are concerned. Obviously the functionality and performance of a grid scheduler depends on the available features of the underlying local resource management systems.

Currently the work in this area is in a difficult but also challenging situation. On the one hand requirements from the application are specified. Advanced reservations and information about future resource usage are mandatory to guarantee that a multi-site [2, 6] application starts synchronously. Only a minority of the available resource management systems provide these features. On the other hand the specification process and its results are influenced by the currently provided features of queuing based resource management systems like NQE/NQS, Loadleveler, or PBS.

We present an approach that closes this gap between the two levels RMS and grid middleware. This concept provides complete knowledge about start times of all requests in the system. Therefore advanced reservations are implicitly possible.

The paper begins with a classification of resource management systems. In Sect. 3 we present enhancements like diffuse resource requests, resource reclaiming, or service level agreement (SLA) aware scheduling. Sect. 4 covers an existing implementation of a resource management system, which already realizes some of the mentioned functions. A brief conclusion closes the paper.

2 Classification of Resource Management Systems

Before we start with classifying resource management systems we define some terms that are used in the following.

- The term *scheduling* stands for the process of computing a schedule. This may be done by a queuing or planning based scheduler.
- A *resource request* contains two information fields: the number of requested resources and a duration for how long the resources are requested.
- A *job* consists of a resource request as above plus additional information about the associated application. Examples are information about the processing environment (e. g. MPI or PVM), file I/O and redirection of stdout and stderr streams, the path and executable of the application, or startup parameters for the application. We neglect the fact that some of this extra job data may indeed be needed by the scheduler, e. g. to check the number of available licenses.
- A *reservation* is a resource request starting at a specified time for a given duration.

In the following the term *Fix-Time* request denotes a reservation, i. e. it cannot be shifted on the time axis. The term *Var-Time* request stands for a

resource request which can move on the time axis to an earlier or later time (depending on the used scheduling policy). In this paper we focus on space-sharing, i. e. resources are exclusively assigned to jobs.

The criterion for the differentiation of resource management systems is the planned time frame. *Queuing systems* try to utilize currently free resources with waiting resource requests. Future resource planning for all waiting requests is not done. Hence waiting resource requests have no proposed start time. *Planning systems* in contrast plan for the present and future. Planned start times are assigned to all requests and a complete schedule about the future resource usage is computed and made available to the users. A comprehensive overview is given in Tab. 1 at the end of this section.

2.1 Queuing Systems

Today almost all resource management systems fall into the class of queuing systems. Several queues with different limits on the number of requested resources and the duration exist for the submission of resource requests. Jobs within a queue are ordered according to a scheduling policy, e. g. FCFS (first come, first serve). Queues might be activated only for specific times (e. g. prime time, non prime time, or weekend). Examples for queue configurations are found in [30, 7].

The task of a queuing system is to assign free resources to waiting requests. The highest prioritized request is always the queue head. If it is possible to start more than one queue head, further criteria like queue priority or best fit (e. g. leaving least resources idle) are used to choose a request. If not enough resources are available to start any of the queue heads, the system waits until enough resources become available. These idle resources may be utilized with less prioritized requests by backfilling mechanisms. Two backfilling variants are commonly used:

- Conservative backfilling [21]: Requests are chosen so that no other waiting request (including the queue head) is further delayed.
- EASY backfilling [18]: This variant is more aggressive than conservative backfilling since only the waiting queue head must not be delayed.

Note, a queuing system does not necessarily need information about the duration of requests, unless backfilling is applied.

Although queuing systems are commonly used, they also have drawbacks. Due to their design no information is provided that answers questions like “Is tomorrow’s load high or low?” or “When will my request be started?”. Hence advanced reservations are troublesome to implement which in turn makes it difficult to participate in a multi-site grid application run. Of course workarounds with high priority queues and dummy requests were developed in the past. Nevertheless the ‘cost of scheduling’ is low and choosing the next request to start is fast.

2.2 Planning Systems

Planning systems do resource planning for the present and future, which results in an assignment of start times to all requests. Obviously duration estimates are mandatory for this planning. With this knowledge advanced reservations are easily possible. Hence planning systems are well suited to participate in grid environments and multi-site application runs. There are no queues in planning systems. Every incoming request is planned immediately.

Planning systems are not restricted to the mentioned scheduling policies FCFS, SJF (shortest job first), and LJF (longest job first). Each time a new request is submitted or a running request ends before it was estimated to end, a new schedule has to be computed. All non-reservations (i.e. *Var-Time* requests) are deleted from the schedule and sorted according to the scheduling policy. Then they are reinserted in the schedule at the earliest possible start time. We call this process *replanning*. Note, with FCFS the replanning process is not necessary, as new requests are simply placed as soon as possible in the schedule without discarding the current schedule.

Obviously some sort of backfilling is implicitly done during the replanning process. As requests are placed as soon as possible in the current schedule, they might be placed in front of already planned requests. However, these previously placed requests are not delayed (i. e. planned at a later time), as they already have a proposed start time assigned. Of course other more sophisticated backfilling strategies exist (e. g. slack-based backfilling [28]), but in this context we focus on the easier variant of backfilling.

Controlling the usage of the machine as it is done with activating different queues for e. g. prime and non prime time in a queuing system has to be done differently in a planning system. One way is to use time dependent constraints for the planning process (cf. Figure 1), e. g. "during prime time no requests with more than 75% of the machines resources are placed". Also project or user specific limits are possible so that the machine size is virtually decreased.

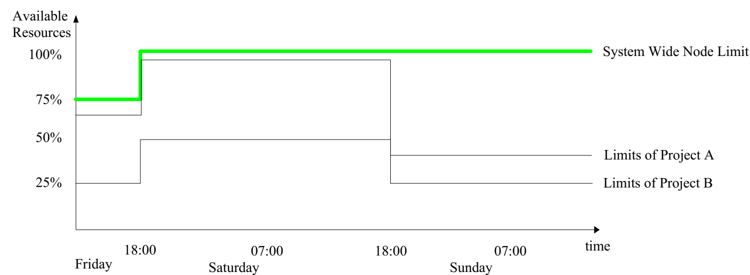

Fig. 1. Time dependent limits in a planning system.

Times for allocating and releasing a partition are vital for computing a valid schedule. Assume two successive requests (A and B) using the same nodes and B

has been planned one second after A. Hence A has to be released in at most one second. Otherwise B will be configured while A is still occupying the nodes. This delay would also affect all subsequent requests, since their planned allocation times depend on the release times of their predecessors.

Planning systems also have drawbacks. The cost of scheduling is higher than in queuing systems. And as users can view the current schedule and know when their requests are planned, questions like "Why is my request not planned earlier? Look, it would fit in here." are most likely to occur.

	queuing system	**planning system**
planned time frame	present	present and future
submission of resource requests	insert in queues	replanning
assignment of proposed start time	no	all requests
runtime estimates	not necessary[1]	mandatory
reservations	not possible	yes, trivial
backfilling	optional	yes, implicit
examples	PBS, NQE/NQS, LL	CCS, Maui Scheduler[2]

[1] exception: backfilling

[2] According to [15] Maui may be configured to operate like a planning system.

Table 1. Differences of queuing and planning systems.

3 Advanced Planning Functions

In this section we present features which benefit from the design of planning systems. Although a planning system is not necessarily needed for implementing these functionalities, its design significantly relieves it.

3.1 Requesting Resources

The resources managed by an RMS are offered to the user by means of a set of attributes (e.g. nodes, duration, amount of main memory, network type, file system, software licenses, etc.). Submitting a job or requesting a resource demands the user to specify the needed resources. This normally has to be done either exactly (e.g. 32 nodes for 2 hours) or by specifying lower bounds (e.g. minimal 128 MByte memory). If an RMS should support grid environments two additional features are helpful: diffuse requests and negotiations.

Diffuse Requests We propose two versions. Either the user requests a range of needed resources, like "Need at least 32 and at most 128 CPUs". Or the RMS "optimizes" one or more of the provided resource attributes itself. Examples are "Need Infiniband or Gigabit-Ethernet on 128 nodes" or "Need as much nodes

as possible for as long as possible as soon as possible". Optimizing requires an objective function. The user may define a job specific one otherwise a system default is taken.

Diffuse requests increase the degree of freedom of the scheduler because the amount of possible placements is larger. The RMS needs an additional component which collaborates with the scheduler. Figure 2 depicts how this *optimizer* is integrated in the planning process. With the numbered arrows representing the control flows several scenarios can be described. For example, placing a reservation with a given start time results in the control flow '1,4'. Planning a diffuse request results in '1,2,3,4'.

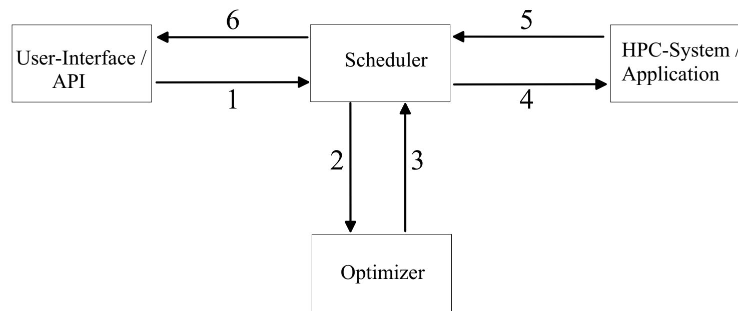

Fig. 2. RMS supporting diffuse requests.

Negotiation One of the advantages of a grid environment is the ability to co-allocate resources (i.e. using several different resources at the same time). However one major problem of co-allocation is how to specify and reserve the resources. Like booking a journey with flights and hotels, this often is an iterative process, since the requested resources are not always available at the needed time or in the desired quality or quantity. Additionally, applications should be able to request the resources directly without human intervention. All this demands for a negotiation protocol to reserve resources. Using diffuse requests eases this procedure. Referring to Fig. 2 negotiating a resource request (with a user or a co-allocation agent) would use the pathes '1,2,3,6,1,...'. Negotiation protocols like SNAP [4] are mandatory to implement service level agreements (cf. Sect. 3.3).

3.2 Dynamic Aspects

HPC systems are getting more and more heterogeneous, both in hardware and software. For example, they may comprise different node types, several communication networks, or special purpose hardware (e. g. FPGA cards). Using a deployment server allows to provide several operating system variants for special

purposes (e.g. real time or support of special hardware). This allows to tailor the operating system to the application to utilize the hardware in the best possible way. All these dynamical aspects should be reflected and supported by an RMS. The following sections illuminate some of these problem areas.

Variable Reservations *Fix-Time* requests are basically specified like *Var-Time* requests (cf. Sect. 2). They come with information about the number of resources, the duration, and the start time. The start time can be specified either explicitly by giving an absolute time or implicitly by giving the end time or keywords like `ASAP` or `NOW`. However the nature of a reservation is that its start time is fixed. Assume the following scenario: if a user wants to make a resource reservation as soon as possible, the request is planned according to the situation when the reservation is submitted. If jobs planned before this reservation end earlier than estimated, the reservation will not move forward.

Hence we introduce *variable reservations*. They are requested like normal reservations as described above, but with an additional parameter (e. g. `-vfix`). This flag causes the RMS to handle the request like a *Var-Time* request. After allocating the resources on the system, the RMS automatically switches the type of the request to a reservation and notifies the user (e. g. by sending an email) that the resources are now accessible. In contrast to a *Var-Time* request a variable reservation is never planned later than its first planned start time.

Resource Reclaiming Space-sharing is commonly applied to schedule HPC applications because the resources are assigned exclusively. Parallel applications (especially from the domain of engineering technology) often traverse several phases (e.g. computation, communication, or checkpointing) requiring different resources. Such applications are called malleable or evolving [10] and should be supported by an RMS. Ideally, the application itself is able to communicate with the RMS via an API to request additional resources (duration, nodes, bandwidth, etc.) or to release resources at runtime. If an HPC system provides multiple communication networks (e.g. Gigabit, Infiniband, and Myrinet) combined with an appropriate software layer (e.g. Direct Access Transport (DAT) [24, 5]) it is possible to switch the network at runtime. For example, assume an application with different communication phases: one phase needs low latency whereas another phase needs large bandwidth. The running application may now request more bandwidth: "Need either 100 MBytes/s for 10 minutes or 500 MBytes/s for 2 minutes". According to Fig. 2 the control flow '5,2,3,4,5...' would be used to negotiate this diffuse request.

Until now DAT techniques are often used to implement a failover mechanism to protect applications against network breakdowns. However, it should also be possible that the RMS causes the application to (temporarily) switch to another network in order to make the high speed network available to another application. This would increase the overall utilization of the system. It also helps to manage jobs with a deadline.

Automatic Duration Extension Estimating the job runtime is a well known problem [21, 26]. It is annoying if a job is aborted shortly before termination because the results are lost and the resources were wasted. Hence users tend to overestimate their jobs by a factor of at least two to three [27] to ensure that their jobs will not be aborted.

A simple approach to help the users is to allow to extend the runtime of jobs while they are running. This might solve the problem, but only if the schedule allows the elongation (i. e. subsequent jobs are not delayed). A more sophisticated approach allows the delay of *Var-Time* requests because delaying these jobs might be more beneficial than killing the running job and processing the resubmitted similar job with a slightly extended runtime. The following constraints have to be considered:

- The length of an extension has to be chosen precisely, as it has a strong influence on the costs of delaying other jobs. For example, extending a one day job by 10 minutes seems to be ok. However if all other waiting jobs are only 1 minute long, they would have to wait for 10 additional minutes. On the other hand these jobs may have already waited for half a day, so 10 minutes extra would not matter. The overall throughput of the machine (measured in useful results per time unit) would be increased substantially.
- The number of granted extensions: is once enough or should it be possible to elongate the duration twice or even three times?

Although many constraints have to be kept in mind in this automatic extension process, we think that in some situations delaying subsequent jobs might be more beneficial than dealing with useless jobs that are killed and generated no result. Of course reservations must not be moved, although policies are thinkable which explicitly allow to move reservations in certain scenarios (cf. Sect. 3.3). In the long run automatic duration extension might also result in a more precise estimation behavior of the users as they need no longer be afraid of losing results due to aborted jobs. In addition to the RMS driven extension an application driven extension is possible [14].

Automatic Restart Many applications need a runtime longer than anything allowed on the machine. Such applications often checkpoint their state and are resubmitted if they have been aborted by the RMS at the end of the requested duration. The checkpointing is done cyclic either driven by a timer or after a specific amount of computing steps.

With a restart functionality it is possible to utilize even short time slots in the schedule to run such applications. Of course the time slot should be longer than a checkpoint interval, so that no results of the computation are lost. If checkpointing is done every hour, additional information provided could be: "the runtime should be x full hours plus n minutes" where n is the time the application needs for checkpointing.

If the application is able to catch signals the user may specify a signal (e. g. USR1) and the time needed to checkpoint. The RMS is now able to send the given

checkpoint signal in time enforcing the application to checkpoint. After waiting the given time the RMS stops the application. This allows to utilize time slots shorter than a regular checkpoint cycle. The RMS automatically resubmits the job until it terminates before its planned duration.

Space Sharing "Cycle Stealing" Space-sharing can result in unused slots since jobs do not always fit together to utilize all available resources. These gaps can be exploited by applications which may be interrupted and restarted arbitrarily. This is useful for users running "endless" production runs but do not need the results with a high priority. Such jobs do not appear in the schedule since they run in the "background" stealing the idle resources in a space sharing system. This is comparable to the well known approach in time-sharing environments (Condor [19]). Prerequisite is that the application is able to checkpoint and restart.

Similar to applying the "automatic restart" functionality a user submits such a job by specifying the needed resources and a special flag causing the RMS to run this job in the background. Optionally the user may declare a signal enforcing a checkpoint.

Specifying the needed resources via a diffuse request would enable the RMS to optimize the overall utilization if planning multiple "cycle stealing" requests. For example, assume two "cycle stealing" requests: A and B. A always needs 8 nodes and B runs on 4 to 32 nodes. If 13 nodes are available the RMS may assign 8 nodes to A and 5 to B.

Deployment Servers Computing centers provide numerous different services (especially commercial centers). Until now they often use spare hardware to cope with peak demands. These spare parts are still often configured manually (operating system, drivers, etc.) to match the desired configuration. It would be more convenient if such a reconfiguration is done automatically. The user should be able to request something like:

- "Need 5 nodes running RedHat $x.y$ with kernel patch z from 7am to 5pm."
- "Need 64 nodes running ABAQUS on SOLARIS including 2 visualization nodes."

Now the task of the RMS is to plan both the requested resources and the time to reconfigure the hardware.

3.3 Service Level Agreements

Emerging network-oriented applications demand for certain resources during their lifetime [11], so that the common best effort approach is no longer sufficient. To reach the desired performance it is essential that a specified amount of processors, network bandwidth or harddisk capacity is available at runtime.

To fulfill the requirements of an application profile the computing environment has to provide a particular quality of service (QoS). This can be achieved

by a reservation and a following allocation of the corresponding resources [12] for a limited period, which is called *advanced reservation* [20]. It is defined as "*[..] the process of negotiating the (possibly limited or restricted) delegation of particular resource capabilities over a defined time interval from the resource owner to the requester*" [13].

Although the advanced reservation permits that the application starts as planned, this is not enough for the demands of the real world. At least the commercial user is primarily interested in an end-to-end service level agreement (SLA) [17], which is not limited to the technical aspects of an advanced reservation. According to [29] an SLA is "*an explicit statement of the expectations and and obligations that exist in a business relationship between two organizations: the service provider and the customer*". It also covers subjects like involved parties, validity period, scope of the agreement, restrictions, service-level objectives, service-level indicators, penalties, or exclusions [22].

Due to the high complexity of analyzing the regulations of an SLA and checking their fulfillment, a manual handling obviously is not practicable. Therefore SLAs have to be unambiguously formalized, so that they can be interpreted automatically [23]. However high flexibility is needed in formulating SLAs, since every SLA describes the particular requirement profile. This may range up to the definition of individual performance metrics.

The following example illustrates how an SLA may look like. Assume that the University of Foo commits, that in the time between 10/18/2003 and 11/18/2003 every request of the user "Custom-Supplies.NET" for a maximum of 4 Linux nodes and 12 hours is fulfilled within 24 hours. Example 1 depicts the related WSLA [23] specification. It is remarkable that this SLA is not a precise reservation of resources, but only the option for such a request. This SLA is quite rudimental and does not consider issues like reservation of network bandwidth, computing costs, contract penalty or the definition of custom performance metrics. However it is far beyond the scope of an advanced reservation.

Service level agreements simplify the collaboration between a service provider and its customers. The customer fulfillment requires that the additional information provided by an SLA has to be considered not only in the scheduling process but also during the runtime of the application.

SLA-aware Scheduler From the schedulers point of view the SLA life cycle starts with the negotiation process. The scheduler is included into this process, since it has to agree to the requirements defined in the SLA. As both sides agree on an SLA the scheduler has to ensure that the resources according to the clauses of the SLA are available. At runtime the scheduler is not responsible for measuring the fulfillment of the SLA, but to provide all granted resources.

Dealing with hardware failures is important for an RMS. For an SLA-aware scheduler this is vital. For example, assume a hardware failure of one or more resources occurs. If there are jobs scheduled to run on the affected resources, these jobs have to be rescheduled to other resources to fulfill the agreed SLAs. If there are not enough free resources available, the scheduler has to cancel or

```
<daytimeConstraint id="workday constraints">
  <day>Mon Tue Wed Thu Fri</day>
  <startTime>0AM</startTime>
  <endTime>12PM</endTime>
</daytimeConstraint>

<SLA ID="Uni-Foo.DE/SLA1">
  <provider>Uni-Foo.DE</provider>
  <consumer>Custom-Supplies.NET</consumer>
  <startDate>Sat Oct 18 00:00:00 CET 2003</startDate>
  <endDate>Tue Nov 18 00:00:00 CET 2003</endDate>

  <SLO ID="SLO1">
    <daytimeConstraint idref="workday constraints">
    <clause id="SLO1Clause1">
      <measuredItem idref="ResponseTime">
      <evalFuncResponseTime operator="LT" threshold="24H"></evalFuncResponseTime>
    </clause>
    <clause id="SLO1Clause2">
      <measuredItem idref="ReqNodeCount">
      <evalFuncReqNodeCount operator="LT" threshold="5"></evalFuncReqNodeCount>
    </clause>
    <clause id="SLO1Clause3">
      <measuredItem idref="ReqNodeOS">
      <evalFuncReqNodeOS operator="EQ" threshold="Linux"></evalFuncReqNodeOS>
    </clause>
    <clause id="SLO1Clause4">
      <measuredItem idref="ReqDuration">
      <evalFuncReqDuration operator="LT" threshold="12H"></evalFuncReqDuration>
    </clause>
  </SLO>
</SLA>
```

Example 1: A Service Level Agreement specified in WSLA.

delay scheduled jobs or to abort running jobs to ensure the fulfillment of the SLA.

As an agreement on the level of service is done, it is not sufficient to use simple policies like FCFS. With SLAs the amount of job specific attributes increases significantly. These attributes have to be considered during the scheduling process.

Job Forwarding Using the Grid Even if the scheduler has the potential to react on unexpected situations like hardware failures by rescheduling jobs, this is not always applicable. If there are no best effort jobs either running or in the schedule, the scheduler has to violate at least one SLA.

However if the system is embedded in a grid computing environment, potentially there are matching resources available. Due to the fact that the scheduler knows each job's SLA, it could search for matching resources in the grid. For instance this could be done by requesting resources from a grid resource broker. By requesting resources with the specifications of the SLA it is assured that the located grid resources can fulfill the SLA. In consequence the scheduler can forward the job to another provider without violating the SLA.

The decision of forwarding does not only depend on finding matching resources in the grid. If the allocation of grid resources is much more expensive

than the revenue achieved by fulfilling the SLA, it can be economically more reasonable to violate the SLA and pay the penalty fee.

The information given in an SLA in combination with job forwarding gives the opportunity to use overbooking in a better way than in the past. Overbooking assumes that users overestimate the durations of their jobs and the related jobs will be released earlier. These resources are used to realize the additional (overbooked) jobs. However if jobs are not released earlier as assumed, the overbooked jobs have to be discarded. With job forwarding these jobs may be realized on other systems in the grid. If this is not possible the information provided in an SLA may be used to determine suitable jobs for cancellation.

4 The Computing Center Software

This section describes a resource management system developed and used at the Paderborn Center for Parallel Computing. It provides some of the features characterized in Sect. 3.

4.1 Architecture

The Computing Center Software [16] has been designed to serve two purposes: For HPC users it provides a uniform access interface to a pool of different HPC systems. For system administrators it provides a means for describing, organizing, and managing HPC systems that are operated in a computing center. Hence the name "Computing Center Software", CCS for short.

A *CCS island* (Fig. 3) comprises five modules which may be executed asynchronously on different hosts to improve the response time of CCS.

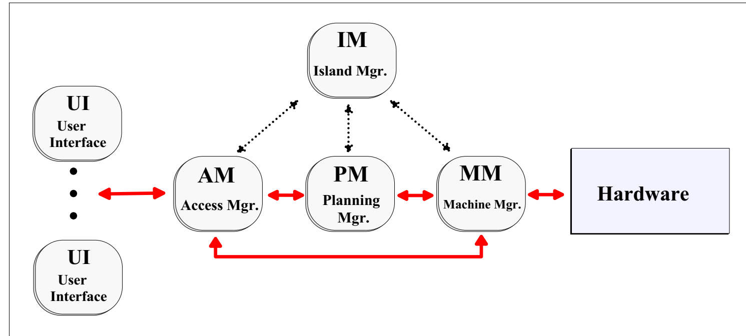

Fig. 3. Interaction between the CCS components.

- The *User Interface (UI)* provides a single access point to one or more systems via an X-window or a command line based interface.

– The *Access Manager (AM)* manages the user interfaces and is responsible for authentication, authorization, and accounting.
– The *Planning Manager (PM)* plans the user requests onto the machine.
– The *Machine Manager (MM)* provides machine specific features like system partitioning, job controlling, etc. The MM consists of three separate modules that execute asynchronously.
– The *Island Manager (IM)* provides CCS internal name services and watchdog facilities to keep the island in a stable condition.

4.2 The Planning Concept

The planning process in CCS is split into two instances, a hardware dependent and a hardware independent part. The *Planning Manager (PM)* is the hardware independent part. It has no information on mapping constraints (e. g. the network topology or location of visualization- or I/O-nodes).

The hardware dependent tasks are performed by the *Machine Manager (MM)*. It maps the schedule received from the PM onto the hardware considering system specific constraints (e. g. network topology). The following sections depict this split planning concept in more detail.

Planning According to Sect. 2 CCS is a planning system and not a queuing system. Hence CCS requires the users to specify the expected duration of their requests. The CCS planner distinguishes between *Fix-Time* and *Var-Time* resource requests. A *Fix-Time* request reserves resources for a given time interval. It cannot be shifted on the time axis. In contrast, *Var-Time* requests can move on the time axis to an earlier or later time slot (depending on the used policy). Such a shift on the time axis might occur when other requests terminate before the specified estimated duration. Figure 4 shows the schedule browser.
The PM manages two "lists" while computing a schedule. The lists are sorted according to the active policy.

1. The *New list(N-list)*: Each incoming request is placed in this list and waits there until the next planning phase begins.
2. The *Planning list(P-list)*: The PM plans the schedule using this list.

CCS comes with three strategies: FCFS, SJF, and LJF. All of them consider project limits, system wide node limits, and Admin-Reservations (all described in Sect. 4.3). The system administrator can change the strategy at runtime. The integration of new strategies is possible, because the PM provides an API to plug in new modules.

Planning an Incoming Job: The PM first checks if the N-list has to be sorted according to the active policy (e. g. SJF or LJF). It then plans all elements of N-list. Depending on the request type (*Fix-Time* or *Var-Time*) the PM calls an associated planning function. For example, if planning a *Var-Time* request, the PM tries to place the request as soon as possible. The PM starts in the present and moves to the future until it finds a suitable place in the schedule.

Backfilling: According to Sect. 2 backfilling is done implicitly during the re-planning process, if SJF or LJF is used. If FCFS is used the following is done: each time a request is stopped, an Admin-Reservation is removed, or the duration of a planned request is decreased, the PM determines the optimal time for starting the backfilling (`backfillStart`) and initiates the backfill procedure. It checks for all *Var-Time* requests in the P-list with a planned time later than `backfillStart` if they could be planned between `backfillStart` and their current schedule.

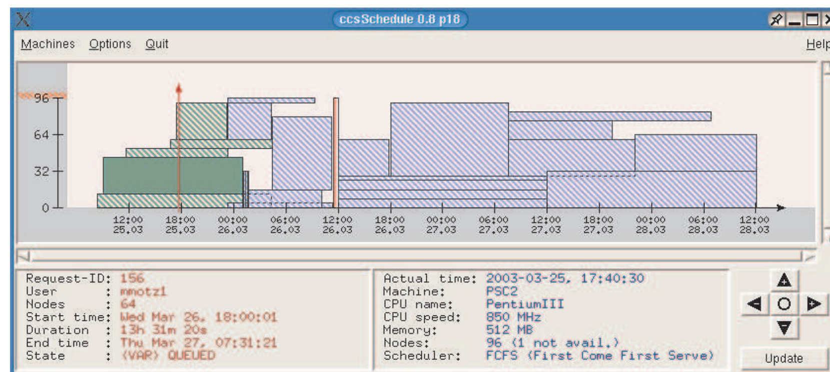

Fig. 4. The CCS schedule browser.

Mapping The separation between the hardware independent PM and the system specific MM allows to encapsulate system specific mapping heuristics in separate modules. With this approach, system specific requests (e. g. for I/O-nodes, specific partition topologies, or memory constraints) may be considered.

One task of the MM is to verify if a schedule received from the PM can be realized with the available hardware. The MM checks this by mapping the user given specification with the static (e. g. topology) and dynamic (e. g. PE availability) information on the system resources. This kind of information is described by means of the Resource and Service Description (RSD, cf. Sect. 4.4).

If the MM is not able to map a request onto the machine at the time given by the PM, the MM tries to find an alternative time. The resulting conflict list is sent back to the PM. The PM now checks this list: If the MM was not able to map a *Fix-Time* request, the PM rejects it. If it was a backfilled request, the PM falls back on the last verified start time. If it was not a backfilled request, the PM checks if the planned time can be accepted: Does it match Admin-Reservations, project limits, or system wide limits?

If managing a homogeneous system verifying is not mandatory. However, if a system comprises different node types or multiple communication networks

or the user is able to request specific partition shapes (e. g. a 4 x 3 x 2 grid) verifying becomes necessary to ensure a deterministic schedule.

Another task of the MM is to monitor the utilization of partitions. If a partition is not used for a certain amount of time, the MM releases the partition and notifies the user via email.

The MM is also able to migrate partitions when they are not active (i. e. no job is running). The user does not notice the migration unless she runs time-critical benchmarks for testing the communication speed of the interconnects. In this case the automatic migration facility may be switched off by the user at submit time.

4.3 Features

Showing Planned Start Times: The CCS user interface shows the estimated start time of interactive requests directly after the submitted request has been planned. This output will be updated whenever the schedule changes. This is shown in Example 2.

Reservations: CCS can be used to reserve resources at a given time. Once CCS has accepted a reservation, the user has guaranteed access to the requested resources. During the reserved time frame a user can start an arbitrary number of interactive or batch jobs.

Deadline Scheduling: Batch jobs can be submitted with a deadline notification. Once a job has been accepted, CCS guarantees that the job is completed at (or before) the specified time.

Limit Based Scheduling: In CCS authorization is project based. One has to specify a project at submit time. CCS knows two different limit time slots: weekdays and weekend. In each slot CCS distinguishes between day and night. All policies consider the project specific node limits (given in percent of the number of available nodes of the machine). This means that the scheduler will sum up the already used resources of a project in a given time slot. If the time dependent limit is reached, the request in question is planned to a later or earlier slot (depending on the request type: interactive, reservation, deadline etc.).
Example: We define a project-limit of 15% for weekdays daytime for the project FOO. Members of this project may now submit a lot of batch jobs and will never get more than 15% of the machine during daytime from Monday until Friday. Requests violating the project limit are planned to the next possible slot (cf. Fig. 1). Only the start time is checked against the limit to allow that a request may have a duration longer than a project limit slot. The AM sends the PM the current project limits at boot time and whenever they change (e. g. due to crashed nodes).

System Wide Node Limit: The administrator may establish a system wide node limit. It consists of a threshold (T), a number of nodes (N), and a time slot [start, stop]. N defines the number of nodes which are not allocatable if a user requests more than T nodes during the interval [start, stop]. This

ensures that small partitions are not blocked by large ones during the given interval.

Admin Reservations: The administrator may reserve parts or the whole system (for a given time) for one or more projects. Only the specified projects are able to allocate and release an arbitrary number of requests during this interval on the reserved number of nodes. Requests of other projects are planned to an earlier or later time. An admin reservation overrides the current project limit and the current system wide node limit. This enables the administrator to establish "virtual machines" with restricted access for a given period of time and a restricted set of users.

Duration Change at Runtime: It is possible to manually change the duration of already waiting or running requests. Increasing the duration may enforce a verify round. The MM checks if the duration of the given request may be increased, without influencing subsequent requests. Decreasing the duration may change the schedule, because requests planned after the request in question may now be planned earlier.

```
%ccsalloc -t 10s -n 7 shell date
ccsalloc: Connecting default machine: PSC
ccsalloc: Using default project     : FOO
ccsalloc: Using default name        : bar%d
ccsalloc: Emailing of CCS messages  : On
ccsalloc: Only user may access      : Off
ccsalloc: Request (51/bar_36): will be authenticated and planned
ccsalloc: Request (51/bar_36): is planned and waits for allocation
ccsalloc: Request (51/bar_36): will be allocated at 14:28 (in 2h11m)
ccsalloc: 12:33: New planned time is at 12:57 (in 24m)
ccsalloc: 12:48: New planned time is at 12:53 (in 5m)
ccsalloc: 12:49: New planned time is at 12:50 (in 1m)
ccsalloc: Request (51/bar_36): is allocated
ccsalloc: Request 51: starting shell date
Wed Mar 12 12:50:03 CET 2003
ccsalloc: Request (51/bar_36): is released
ccsalloc: Bye,Bye (0)
```

Example 2: Showing the planned allocation time.

4.4 Resource and Service Description

The *Resource and Service Description (RSD)* [1, 3] is a tool for specifying irregularly connected, attributed structures. Its hierarchical concept allows different dependency graphs to be grouped for building more complex nodes, i. e. hypernodes. In CCS it is used at the administrator level for describing the type and topology of the available resources, and at the user level for specifying the required system configuration for a given application. This specification is created automatically by the user interface.

In RSD resources and services are described by nodes that are interconnected by edges via communication endpoints. An arbitrary number of attributes may

be assigned to each of this entities. RSD is able to handle dynamic attributes. This is useful in heterogeneous environments, where for example the temporary network load affects the choice of the mapping. Moreover, dynamic attributes may be used by the RMS to support the planning and monitoring of SLAs (cf. Sect. 3.3).

5 Conclusion

In this paper we have presented an approach for classifying resource management systems. According to the planned time frame we distinguish between queuing and planning systems. A queuing system considers only the present and utilizes free resources with requests. Planning systems in contrast plan for the present and future by assigning a start time to all requests.

Queuing systems are well suited to operate single HPC machines. However with grid environments and heterogenous clusters new challenges arise and the concept of scheduling has to follow these changes. Scheduling like a queuing system does not seem to be sufficient to handle the requirements. Especially if advanced reservation and quality of service aspects have to be considered. The named constraints of queuing systems do not exist in planning systems due to their different design.

Besides the classification of resource management systems we additionally presented new ideas on advanced planning functionalities. Diffuse requests ease the process of negotiating the resource usage between the system and users or co-allocation agents. Resource reclaiming and automatic duration extension extend the term of scheduling. The task of the scheduler is no longer restricted to plan the future only, but also to manage the execution of already allocated requests.

Features like diffuse requests and service level agreements in conjunction with job forwarding allow to build a control cycle comprising active applications, resource management systems, and grid middleware. We think this control cycle would help to increase the everyday usability of the grid especially for the commercial users.

The aim of this paper is to show the benefits of planning systems for managing HPC machines. We see this paper as a basis for further discussions.

References

1. M. Brune, J. Gehring, A. Keller, and A. Reinefeld. RSD - Resource and Service Description. In *Proc. of 12th Intl. Symp. on High-Performance Computing Systems and Applications (HPCS'98)*, pages 193–206. Kluwer Academic Press, 1998.
2. M. Brune, J. Gehring, A. Keller, and A. Reinefeld. Managing Clusters of Geographically Distributed High-Performance Computers. *Concurrency - Practice and Experience*, 11(15):887–911, 1999.
3. M. Brune, A. Reinefeld, and J. Varnholt. A Resource Description Environment for Distributed Computing Systems. In *Proceedings of the 8th International Symposium High-Performance Distributed Computing HPDC 1999, Redondo Beach*, Lecture Notes in Computer Science, pages 279–286. IEEE Computer Society, 1999.

4. K. Cjajkowski, I. Foster, C. Kesselman, V. Sander, and S. Tuecke. SNAP: A Protocol for Negotiation of Service Level Agreements and Coordinated Resource Management in Distributed Systems. In *Proceedings of the 8th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, volume 2537 of *Lecture Notes in Computer Science*, pages 153–183. Springer Verlag, 2002.
5. Direct Access Transport (DAT) Specification. `http://www.datcollaborative.org`, April 2003.
6. C. Ernemann, V. Hamscher, A. Streit, and R. Yahyapour. Enhanced Algorithms for Multi-Site Scheduling. In *Proceedings of 3rd IEEE/ACM International Workshop on Grid Computing (Grid 2002) at Supercomputing 2002*, volume 2536 of *Lecture Notes in Computer Science*, pages 219–231, 2002.
7. D. G. Feitelson and M. A. Jette. Improved Utilization and Responsiveness with Gang Scheduling. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 3rd Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 238–262. Springer Verlag, 1997.
8. D. G. Feitelson and L. Rudolph. Towards Convergence in Job Schedulers for Parallel Supercomputers. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 2nd Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 1–26. Springer Verlag, 1996.
9. D. G. Feitelson and L. Rudolph. Metrics and Benchmarking for Parallel Job Scheduling. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 4th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1459, pages 1–24. Springer Verlag, 1998.
10. D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, and K. C. Sevcik. Theory and Practice in Parallel Job Scheduling. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 3rd Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 1–34. Springer Verlag, 1997.
11. I. Foster and C. Kesselman (Eds.). *The Grid: Blueprint for a New Computing.* Morgan Kaufmann Publishers Inc. San Fransisco, 1999.
12. I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. In *Proceedings of the International Workshop on Quality of Service*, 1999.
13. GGF Grid Scheduling Dictionary Working Group. Grid Scheduling Dictionary of Terms and Keywords . `http://www.fz-juelich.de/zam/RD/coop/ggf/sd-wg.html`, April 2003.
14. J. Hungershöfer, J.-M. Wierum, and H.-P. Gänser. Resource Management for Finite Element Codes on Shared Memory Systems. In *Proc. of Intl. Conf. on Computational Science and Its Applications (ICCSA)*, volume 2667 of *LNCS*, pages 927–936. Springer, May 2003.
15. D. Jackson, Q. Snell, and M. Clement. Core Algorithms of the Maui Scheduler. In D. G. Feitelson and L. Rudolph, editor, *Proceddings of 7th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2221 of *Lecture Notes in Computer Science*, pages 87–103. Springer Verlag, 2001.
16. A. Keller and A. Reinefeld. Anatomy of a Resource Management System for HPC Clusters. In *Annual Review of Scalable Computing, vol. 3, Singapore University Press*, pages 1–31, 2001.
17. H. Kishimoto, A. Savva, and D. Snelling. OGSA Fundamental Services: Requirements for Commercial GRID Systems. Technical report, Open Grid Services Architecture Working Group (OGSA WG), `http://www.gridforum.org/Documents/Drafts/default_b.htm`, April 2003.

18. D. A. Lifka. The ANL/IBM SP Scheduling System. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 1st Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 295–303. Springer Verlag, 1995.
19. M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS'88)*, pages 104–111. IEEE Computer Society Press, 1988.
20. J. MacLaren, V. Sander, and W. Ziegler. Advanced Reservations - State of the Art. Technical report, Grid Resource Allocation Agreement Protocol Working Group, Global Grid Forum, `http://www.fz-juelich.de/zam/RD/coop/ggf/graap/sched-graap-2.0.html`, April 2003.
21. A. Mu'alem and D. G. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. In *IEEE Trans. Parallel & Distributed Systems 12(6)*, pages 529–543, June 2001.
22. A. Sahai, A. Durante, and V. Machiraju. Towards Automated SLA Management for Web Services. HPL-2001-310 (R.1), Hewlett-Packard Company, Software Technology Laboratory, HP Laboratories Palo Alto, `http://www.hpl.hp.com/techreports/2001/HPL-2001-310R1.html`, 2002.
23. A. Sahai, V. Machiraju, M. Sayal, L.J. Jin, and F. Casati. Automated SLA Monitoring for Web Services. In *Management Technologies for E-Commerce and E-Business Applications, 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, volume 2506 of *Lecture Notes in Computer Science*, pages 28–41. Springer, 2002.
24. Scali MPI Connect^{TM}. `http://www.scali.com`, April 2003.
25. L. Smarr and C. E. Catlett. Metacomputing. *Communications of the ACM*, 35(6):44–52, June 1992.
26. W. Smith, I. Foster, and V. Taylor. Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance. In D. G. Feitelson and L. Rudolph, editor, *Proc. of 5th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1659 of *Lecture Notes in Computer Science*, pages 202–219. Springer Verlag, 1999.
27. A. Streit. A Self-Tuning Job Scheduler Family with Dynamic Policy Switching. In *Proc. of the 8th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2537 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2002.
28. D. Talby and D. G. Feitelson. Supporting Priorities and Improving Utilization of the IBM SP2 Scheduler Using Slack-Based Backfilling. In *13th Intl. Parallel Processing Symp.*, pages 513–517, April 1999.
29. D. Verma. *Supporting Service Level Agreements on an IP Network*. Macmillan Technology Series. Macmillan Technical Publishing, August 1999.
30. K. Windisch, V. Lo, R. Moore, D. Feitelson, and B. Nitzberg. A Comparison of Workload Traces from Two Production Parallel Machines. In *6th Symposium Frontiers Massively Parallel Computing*, pages 319–326, 1996.