A Data Structure for Planning Based Workload Management of Heterogeneous HPC Systems

Axel Keller

Paderborn Center for Parallel Computing
Paderborn University, 33098 Paderborn, Germany
axel.keller@uni-paderborn.de

Abstract. This paper describes a data structure and a heuristic to plan and map arbitrary resources in complex combinations while applying time dependent constraints. The approach is used in the planning based workload manager OpenCCS at the Paderborn Center for Parallel Computing (PC^2) to operate heterogeneous clusters with up to 10000 cores. We also show performance results derived from four years of operation.

Keywords: Scheduling, planning, mapping, workload management

1 Introduction

Todays HPC systems are heterogeneous, they consist of different node types and accelerators (e.g., GPUs or FPGAs) are used to increase the application performance. Disk storage, software licenses, virtual machines, or software containers are additional resources to be scheduled by a workload management system (WLM). In the past Grid and Cloud computing brought challenges in form of inter system applications, running on more than one system at the same time or consecutively steered by a workflow-manager. The merge of HPC and Big-Data already started and more complex workflows will arise and enhance the complexity of scheduling. Keywords are for example: data aware scheduling, co-allocations, provisional reservations, or SLAs.

Planning based WLMs are well prepared for such environments. In 2003, we published a paper[6] which compared queueing and planning based WLMs on a high level and introduced OpenCCS as a completely planning based WLM. Since then, OpenCCS implemented some features mentioned in [6], like for example, job-migration using Globus, or SLA negotiation and compliance. This work was primarily done in the EU funded projects HPC4U[1] and Assessgrid[1].

However, all work done there was based on scheduling only entire nodes all of the same type. At that time, we used a generic scheduler and a system-specific mapping instance which verified the schedule. We learned, that planning and mapping has to be done in the scheduler because the mapper had no info about limits or fairness and the scheduler did not consider the requested topology in a satisfying manner (e.g., a 2x4 grid on a system with a 2D-torus topology). Additionally, the scheduler performance collapsed if managing several hundreds of nodes and thousands of jobs.

In 2009, we redesigned OpenCCS. We aimed on supporting time shared operation (i.e., more than one job on a node) on large heterogeneous clusters with thousands of jobs, that is fast online planning of an arbitrary number of resources. It should be easy to integrate commercial applications. Users should be able to reserve resources, submit jobs with a deadline, and steer the mapping. Users and groups should be automatically (un)locked by the system. This paper reports the results of this redesign. Its central contribution is the basic data structure and the planning and mapping heuristic based on this.

We start with a brief comparison of the queueing and planning approach and name challenges of planning based WLMs. In Sect. 3 we introduce OpenCCS focusing on terms which are related to the scope of this paper. Section 4 explains the central data structure used in the OpenCCS scheduler and its basic operations. Based on this, Sect. 5 focuses on the principle process of planning and mapping and describes some resulting aspects in more detail. Section 6 is devoted to performance results derived from real operation over four years. In Sect. 7 we compare the introduced method with other approaches and Sect. 8 summarizes the paper.

2 Queueing vs. Planning

The major criterion for the differentiation of WLMs is the planned time frame. *Queueing systems* try to utilize currently free resources with waiting resource requests and future resource planning for all waiting requests is not done. Hence, waiting resource requests have no assigned start time. *Planning systems* in contrast plan for the present and future. Start times are assigned to all requests and a complete schedule about the future resource usage is computed and made available to the users.

Queueing. In principle there are several queues with different limits on the number of requested resources and the duration (e.g., min, max, defaults, etc.) exist for the job submission. Jobs within a queue are ordered according to a scheduling policy (e.g., FCFS (first come, first serve)) and users may also order their jobs. Queues might be activated only for specific times (e.g., prime time or weekend).

The task of a queueing system is to assign free resources to waiting requests. The job with the highest priority job is always the queue head. If it is possible to start more than one queue head, further criteria like queue priority 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: (1) Conservative backfilling [11]: Requests are chosen so that no other waiting request (including the queue head) is further delayed. (2) EASY backfilling [10]: This variant is more aggressive than conservative backfilling since only the waiting queue head must not be delayed.

Although, it is not mandatory for queueing systems to know the maximum duration of requests, it is often required by the administration, to decrease job waiting times. The "cost of scheduling" is low and choosing the next request to start is fast.

Planning. Planning systems assign start times to all requests. Obviously, duration estimates are mandatory for planning. With this knowledge reservations are easily possible and planning systems are well suited to participate in multi-site application runs.

Fair share[8, 9] is often used in queueing systems for prioritizing jobs on the basis of a share of the machine and past and current usage. In planning systems, controlling the usage of the machine is often done differently. One way is to use time dependent constraints for the planning process. For example, during prime time 25% of the system is kept free for "small" jobs. Also project or user specific limits are possible, so that the system is virtually partitioned. Job priorities and even more job dependencies (e.g., job B may start only after job A has terminated with an error) have a stronger impact on the complexity of the planning process than in queueing systems.

Planning based WLMs are real time systems. Assume two successive requests (A and B) using the same nodes and B has been planned one-second after A. Then, A has to be released in at most one-second. Otherwise B will be started 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. Hence, timeouts are necessary for such operations and the WLM has to concern them while planning and adhere to the planned slots while executing jobs.

Planning often implies mapping, because although the number of requested resources (e.g., cores) may be free in the requested time interval, we cannot be sure that always the same resources are free. Additionally, if planning complex resource sets, comprising several resource types, we have to ensure that the whole set can be mapped to a host and of course using placing directives directly enforces mapping. Mapping is not mandatory while planning a start time, if the requested resource set is provided by all hosts of the system and can be mapped to a single host (e.g., requesting one core).

Changes in the resource configuration implies replanning all affected jobs. Possible reasons are: a node fails or is set offline, or the amount of available nodes resources changes (e.g., a memory DIMM or a network card fails).

The "cost of scheduling" is higher than in queueing 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 likely to occur. In the next section we briefly describe OpenCCS focusing on the terms, which are necessary to understand the following sections. A more detailed description can be found in the OpenCCS manual[3].

3 The Computing Center Software

OpenCCS has a long history starting in the 1990s at the Paderborn Center for Parallel Computing (PC^2)[13]. Today, OpenCCS consists of several modules, which may run on multiple hosts to improve the response time. OpenCCS is based on events (e.g., timers, messages, signals), and the communication is stateless and asynchronous. The modules are multi-threaded but single-tasked. The submission syntax is strongly PBSPro[12] compatible to ease the integration of commercial applications. Figure 1 depicts the OpenCCS modules (described below) and the event handling.

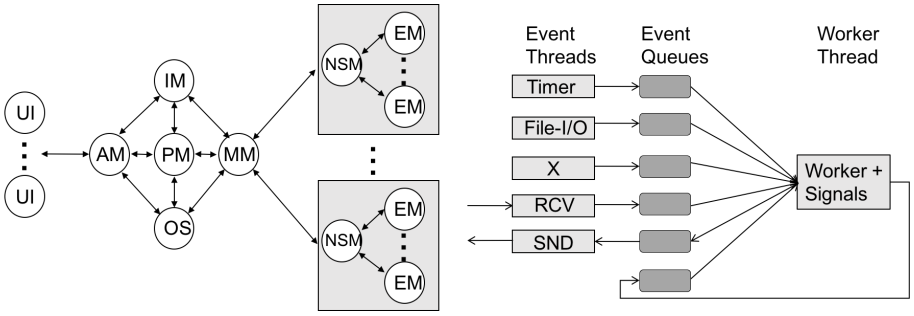

Fig. 1. The OpenCCS modules (left) and event type handling (right)

UI (User Interface): Provides a single access point to one or more systems via command line interfaces.
AM (Access Manager): Manages the user interfaces and is responsible for authentication, authorization, and accounting.
PM (Planning Manager): Schedules and maps the user requests onto the machine.
MM (Machine Manager): Provides machine specific features like node management or job controlling.
IM (Island Manager): Provides OpenCCS internal name services and watchdog facilities to keep OpenCCS in a stable condition.
OS (Operator Shell): The main interface for system administrators to control OpenCCS.
NSM (Node Session Manager): Runs with root privileges on each node managed by OpenCCS. The NSM is responsible for node access and job controlling. At allocation time, the NSM starts an EM for each job.
EM (Execution Manager): Establishes the user environment (UID, shell settings, environment variables, etc.) and starts the application.

OpenCCS uses the Resource and Service Description (RSD) [2] language to specify all system specific hardware and software attributes like node properties, network topology, timeouts, or custom resources.

The planning based approach, implemented in OpenCCS, has some implications. There are no explicit queues in OpenCCS. The "waiting room" is the only equivalent to a queue. A request is moved to this queue if OpenCCS was not able to assign a start time to an already accepted request. Possible reasons are, for example, that resources become unavailable while a request is in state PLANNED or ALLOCATING and there are no comparable resources available. If the resources become available again, OpenCCS automatically tries to replan waiting requests.

Users are supposed to specify the expected runtime of their requests. If no duration is specified, OpenCCS assigns a site specific one.

Privileges, default values, and limitations are attached to groups and users. Entities like user, group, resource, or limit may have a validity period. If the validity is exceeded, the entity is disabled. The rest of the section explains some of this implications in more detail.

Validity. Planning provides an explicit notion of time, and this is also reflected in limits, resource availability, etc. Hence in OpenCCS entities like resources, users, groups, or limits all may have a validity period. It can be given as an absolute end date, an absolute start and end date, or a cron string, specifying repeated intervals. Validities are mandatory to map time dependent constraints to the data structure described in Sect. 4.

Limit. Limits are assigned to a consumer (i.e., a user or a group) and there may be a different limit for each resource. If a consumer has no limits assigned this means all resources are available forever. A limit consists of the following items:

Validity: The validity period of a limit.

Items: The maximum number of allocatable items.
Syntax: `<min[/max]>`
`min` is a integer and `max` specifies the percent of currently available items. If both given, OpenCCS takes the maximum of `min,max`. Example: `30/45%` denotes a limit 30 items or 45% of the available items.

Duration: The maximum timespan the resource may be used.

Area: The maximum area.
For example, the area limit `1024h` for the resource cores, allows a consumer to request one core for 1024 hours, 1024 cores for one hour, or any matching combination in between.

If a time dependent limit is exceeded, the affected request will be scheduled to a later or earlier slot (depending on the request type). In Example 1, the `ncpus` and `tesla` limits override the (`*`) limit (meaning all resources).

```
Resource     Items     Duration Area  Validity
=====================================================
*            unlimited 7d       none  always
ncpus        640       4d3h     none  01.08.17-31.08.17
tesla        unlimited none     500h  always
arrayjobs    1000      none     none  always
jobs         5000      none     none  always
```

Example 1: Some possible limits

FreePool. FreePools are like limits, but describe the conditions for resources to be kept free (i.e., they constrain the access to resources). A FreePool consists of the following items: The validity period, the resource to be kept free, how many of the resource should be kept free, and conditions to get access to the resources. FreePools may be used to:

- Keep free 20% of the available cores but at minimum 10 cores for jobs which request less than four cores for less than one hour.
- Keep all GPUs free for the groups *G1*, *G2* and user *alice.* All others may use the GPUs only for a maximum of two hours.
- Reserve all nodes hosting GPUs for maintenance each two months on Monday from 8am to 6pm.

Requesting Resources In OpenCCS, users specify the resources needed by a job by using chunks and job-wide resources (e.g., licenses or disk space). A chunk specifies a set of resources that have to be allocated as a unit on a single node. Chunks cannot be split across nodes. Syntax: `rset=[N:]chunk[+[N:]chunk...]` A chunk comprises one or more `res=value` statements separated by a colon. `res` is one of the OpenCCS built-in resources (e.g., cores, memory, or ompthreads) or one of the customized specified via the RSD language. Chunks may be combined with a placement specification to control how the chunks should be placed on the nodes. Example 2 may illustrate what is possible.

Table 1. Possible placement specifications

Modifier	Meaning
free	no restriction
pack	all chunks must be placed on one node
scatter	only one chunk per node
exclusive	only this job may use the node
shared	this chunk may share the node with other chunks

```
rset=8:ncpus=2:mem=10g:rack=8
rset=ncpus=27:vmem=20g:arch=linux+4:acc=fpga
rset=5:ncpus=16:mem=12g:net=IB+ncpus=1:mem=4g,sw=g03,place=scatter:excl
```

Example 2: Resource requests using chunks

4 The Resource Usage Vector

In Sect. 2, we outlined challenges of a planning based WLM and in Sect. 3 the way OpenCCS is realizing the goals drafted in the introduction. The data structure introduced here, is our central approach to tackle these issues. It is used to represent time dependent limits, FreePools, reservations, and the available resources in the whole managed system and on its nodes.

We store slots of used or free items sorted by time for each used resource. We call this a resource usage vector (*RUSV*).
A *slot* comprises three components. The start time, the stop time, and the number of items, which are used or free within the interval $[start, stop]$. If stop time is 0, this means $[start, \infty[$.
A *RUSV* additionally has the following components:

maxAvl: the maximum available number of items,
avl: the currently available number of items (i.e., maxAvl - defect),
minDist: the minimal time distance between two slots, normally 1 s.

In the following $R_i[j]$ denotes the $slot_j$ in $RUSV_i$. We do not store slots which are completely "free" (i.e., if storing used items and *slot*.items ≤ 0 or, if storing free items and *slot*.items $\geq RUSV$.avl). Figure 2 depicts a simple example. Please note, that in all intervals, except the specified ones, the number of used items is 0, since this *RUSV* stores used items.

RUSV.avl = 45
RUSV.minDist = 60s

Index	Start	Stop	Items
0	10:00	18:00	30
1	20:00	23:00	15

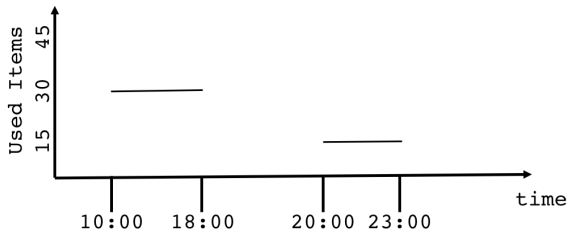

Fig. 2. A simple example of a *RUSV*

4.1 Basic Operations

On *RUSV*s we apply the following basic operations:

Increment ($\oplus$) Adds a slot to a *RUSV*.
Notation: $RUSV \oplus slot$
Increments the number of items of *RUSV* in the interval [*slot*.start, *slot*.stop] by *slot*.items. Missing slots are added.

Decrement ($\ominus$) Subtracts a slot from a *RUSV*.
Notation: $RUSV \ominus slot$
Decrements the number of items of *RUSV* in the interval [*slot*.start, *slot*.stop] by *slot*.items. "Free" slots are removed.

Addition (+) Adds two *RUSV*s.
Notation: $R_3 = R_1 + R_2$
This is done by $\forall i \in R_2$: $R_1 \oplus R_2[i]$.

Subtraction (−) Subtracts two *RUSV*s.
Notation: $R_3 = R_1$ - R_2
This is done by $\forall i \in R_2$: $R_1 \ominus R_2[i]$.

minFree (mf) Minimum of free slots in R_1 and R_2.
Notation: $R_3 = \text{mf}(R_1, R_2)$
This is done by: $\forall i \in R_2$: $R_3[i] = \max(R_2[i].\text{items}, (R_1.\text{avl} - R_2[i].\text{items}))$.
We do not add new slots, and gaps in R_1 are processed. We use this operation to integrate a resource limit into a *RUSV*. For example, R_1 is the number of available resources in the system and R_2 is the limit for this resource.

Intersection ($\cap$) Intersects R_1 and R_2. R_1 and R_2 store free items.
Notation: $R_3 = R_1 \cap R_2$
This is done by: $\forall i \in R_2$: $R_3[i].\text{items} = min(R_1[i].\text{items}, R_2[i].\text{items})$.
As a result R_3 holds all slots for which at least $R_3[i]$.items are free in R_1 and R_2 at the same time.

getFreeSlots (R_1, F, D, T_1, T_2) Search in R_1 for slots with at least F free items with a duration $\geq$ D in the interval [T_1, T_2].
Notation: $R_2 = \text{getFreeSlots}(R_1, F, D, T_1, T_2)$

For all operations, the following is valid: If the *RUSV* stores free items, then slots with *slot*.items $\geq$ *RUSV*.avl are removed. If the *RUSV* stores used items, then slots with *slot*.items ≤ 0 are removed. Consecutive slots are joined if their items are equal and their distance is $\leq$ rus.minDist. Figure 3 depicts the possible overlaps, we have to handle. The actions done are of course specific to the combination of operation and case.

For example, assume a *RUSV* storing used items. Then operation $\ominus$ and case (1) leads to `act`.items -= `new`.items and if `act`.items ≤ 0, slot `b` will be removed. Operation $\oplus$ and case (1) leads to new slots `a` and `c` and `b`.items += new.items. Before we explain the principle planning and mapping process, we introduce the following terms.

***sRS* (System Resource Set)** For each known resource (e.g., cores, memory, GPUs, licenses, etc.), we have one *RUSV* to reflect the usage of the whole system. The *sRS RUSV*s hold used items.
Notations: sRS_r is the RUSV of resource r and $sRS_r[i]$ is *slot* i in sRS_r.

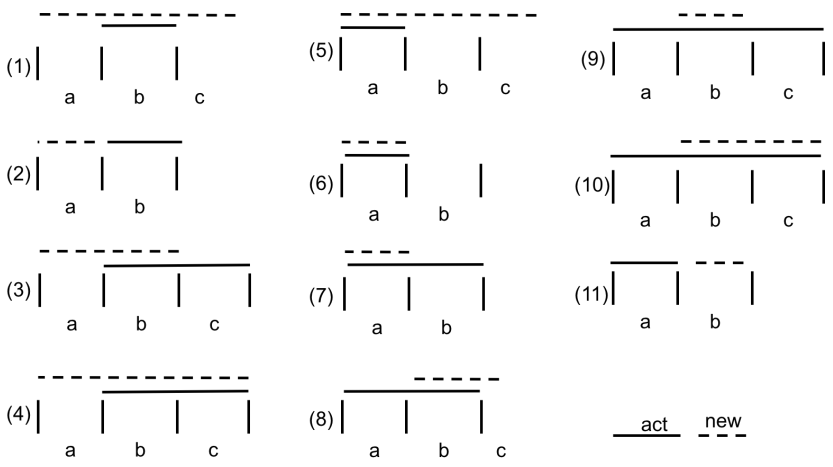

Fig. 3. Possible slot overlap cases

***nRS* (Node Resource Set)** For each known resource, a node has one RUSV to reflect its usage. The *nRS* has the same structure as the *sRS*. If a resource of the *sRS* is not available on a node the related *RUSV* is empty (i.e., $nRS_r.\text{avl} = 0$).

***rRS* (Reservation Resource Set)** It is a subset of the *sRS* depending on what resources are reserved. All planning and mapping routines, described in Sect. 5, are the same for normal jobs and for jobs running in a reservation.

***reqRS* (Requested Resource Set)** The user requested resources (chunks and job wide) in an internal format. Please note, there are no *RUSV*s in a *reqRS*.

***jRS* (Job Resource Set)** For each requested resource of a job, we summarize all requested chunk and job wide resources.
E.g., requesting `8:ncpus=2:mem=10g`, results in a *jRS* of `ncpus=16:mem=80g`.

***usdRS* (Used Resource Set)** The resources which are already assigned to a consumer. The *usdRS* has the same structure as the *sRS* and is used while processing consumer specific limits.

***uRS* (User Resource Set)** For each requested resource we have a *RUSV* reflecting the users view on the system related to limits, FreePools, and already assigned resources. The *uRS* has the same structure as the *sRS* and is built in the planning process.

5 Planning and Mapping

Here, we describe the principle process of planning and mapping requests using *RUSV*s. At submit time the user specifies the resources which should be used (i.e., chunks and job wide resources) and when and how long the resources will be used (e.g., provisional, best-effort, deadline, fixed start time, slot-aware start time, SLAs, duration, etc.). Additionally, the user may specify how the chunks

should be placed (e.g., pack, scatter, free, shared, exclusive) and how the job should be processed (e.g., checkpointing, re-start, etc.). Based on this specifications, the *PM* starts the planning which is divided into three phases.
Phase 1 checks if and when enough resources are free concerning all constraints like limits, FreePools, or already assigned resources.
Phase 2 does the mapping. If the resources can be mapped to all nodes, mapping is postponed to allocation time. Mapping is a separate layer to allow different mapping policies.
Phase 3 updates the *usdRS*, the *sRS*, and the *nRS* of all affected nodes. In the following, we describe these steps in more detail.

5.1 Planning

When a new job comes in, we first scan the resource request and build internal data structures. Thereafter, we add missing default and force values (overwrite user given values), and check if all requested resources are known and available (requested $\leq$ maxAvl). Default and force values may be assigned to the system, the group, or the user.As a result, we get the *reqRS* and the *jRS*.

We then determine the search interval $[T_1, T_2]$ which depends on the job type (e.g., best effort, reservation, deadline). For example, the search interval of a reservation is of course given by the user, whereas the search interval of a best effort job starts at submit time and ends never. All subsequent operations are working in this search interval.

After determining FreePools and limits matching the resource request and $[T_1, T_2]$, we create the *uRS* by computing: $\forall r \in jRS$:

$$uRS_r = sRS_r + FreePool_r \text{ and then}$$
$$uRS_r = \text{mf}(uRS_r, limit_r - usdRS_r) \ .$$

Processing a resource set (e.g., the *sRS*), means that for all resources in question the related *RUSV* operations, introduced in Sect. 4, are performed.

The *uRS* reflects now the user's view on the amount of available resources in the search interval. Hence, we are able to search for slots where all requested resources are available at the same time in the requested amount for the requested duration. This is done by computing:

$\forall r \in jRS$: $R_{freeSlots} = R_{freeSlots} \cap getFreeSlots_r(uRS_r, F_r, D, T_1, T_2)$.
If $R_{freeSlots}$ is not empty, we try to find a valid mapping.

The complexity of the planning process without mapping is independent of the number of jobs in the system, since we process only *RUSV*s.

5.2 Mapping

The input is the job's duration D, the *reqRS*, the *jRS*, and $R_{freeSlots}$ as a result of the planning process described in the last section. Mapping is also done in two phases. Phase 1 determines a candidate list comprising nodes on which at least one chunk $\in$ *reqRS* is unused for the duration D, in the search interval. Phase 2 then uses this list to select nodes according to a policy (e.g., greedy, energy

efficiency, etc.). For this purpose, a weight (i.e., a scalar value) is computed for all nodes. The weight is used to rank the nodes from "cheap" to "expensive". It is computed by: $\forall r \in$ consumable resources provided by the node:

$$W_{node} = max\big(W_{node}, \frac{r}{r_{system}}\big)$$

and then $W_{node} = W_{node} * cores_{system} + prio_{node} * cores_{system}$.
$cores_{system}$ is the number of available cores in the system.
$prio_{node}$ is an integer value and may be specified by the administrator via RSD.
The basic steps of phase 1 are:

1. Build N_{cand}: A list of all usable nodes providing the required chunks in principle. N_{cand} is then sorted by the node's weight.
2. Build $FCN_{n,c}$: $\forall n \in N_{cand}$ and for each requested chunk c build a $RUSV$ where the chunk is free for at least duration D on the node:
   $$\forall r \in chunk_c\text{: } FCN_{n,c} \cap getFreeSlots(nRS_r, F_r, D, T_1, T_2) \ .$$
   If $FCN_{n,c}$ for a node n is empty, this node is removed from N_{cand}. To get a good node utilization, we first compute a weight for each requested chunk related to a node, similar to the node's weight, and sort the chunks by their weight in descending order. As a result we get X $RUSV$s per node. X is the number of requested chunks and $FCN_{n,c}$[i].items holds the number of free chunks.
3. Build FCJ_c: For each requested chunk build a $RUSV$ holding the sum of all related $FCN_{n,c}$ by computing:
   $$\forall n \in N_{cand} \text{ and } \forall c \in reqRS\text{: } FCJ_c = \textstyle\sum_{c,n} FCN_{n,c}$$
   and removing all intervals with less than the required number of chunks or a duration $<$ D. We then check if enough chunks are available. If not then the job cannot be mapped within the search interval.
4. Build FS: The intersection of all $FCJ_c$by computing:
   $$\forall c \in reqRS\text{: } FS \cap FCJ_c \ .$$
   FS then holds all slots with a duration $\geq$ D, where all chunks are available at the same time.

The result of phase 1 is: A $RUSV$ (FS) with available time slots $\geq$ D and a list of nodes (N_{cand}) and for each node $FCN_{n,c}$ (a RUSV for each chunk with free time slots $\geq$ D).

If FS is not empty, we have found a set of nodes which provide the resources. We then enter phase 2. Until now, we do a greedy mapping. The Greedy mapper tries to map expensive chunks on cheap nodes first. If mapping was not possible for all slots in FS, we try another slot of $R_{freeSlots}$ else, we build the mapping-data *njRS*. For each mapped host, it holds information which resources in what amount the host provides. The *njRS* can be seen as a node specific *jRS*.

The complexity of the mapping process is independent of the number of jobs in the system. It depends on the number of nodes and the job's chunk complexity. For example mapping a chunk with requesting one core can be done in more ways than mapping a chunk comprising 32 cores and a GPU.

5.3 Booking

The last step is to commit the planned resources for the planned interval in *usdRS*, *sRS*, and *nRS* by computing:

$$\forall r \in jRS \text{ and } \forall s \in \{\text{sRS, usdRS}\}\text{: } s_r \oplus jRS_r \ .$$

For *nRS*, we book on all mapped nodes:

$$\forall r \in \text{node's } njRS\text{: } nRS_r \oplus njRS_r \ .$$

The inverse operation (i.e., revoke) is done by computing $\ominus$ instead of $\oplus$. Revoke is used if a job is removed or while scanning for a better plan.

5.4 Notable Aspects

Of course, there are a lot of pitfalls and exceptions to cope with while applying the heuristic outlined above. In the following, we describe some aspects.

Backfilling. Backfilling is invoked whenever a job has been removed from the plan. It affects all jobs with a planned start time after the removed job. To avoid long answer times (e.g., 100,000 planned jobs), backfilling is done in the background controlled by a special backfilling thread. The basic two steps are:

1. Sample affected jobs and sort them (by job-priority, submit-time, etc.) A job's priority is computed automatically at submit time. Criteria are for example: job type, requesting expensive or "special" resources, or node parallelism.
2. For each job try to find a "better" place in the plan. Following a First-Fit strategy, we first unbook the job's resources and plan it again. If the planned start time is earlier than the previous one, we book the new time interval, else the old one. Other strategies are possible but not yet implemented. Jobs are started immediately if possible.

Replanning. Replanning is invoked if a job cannot be allocated due to an allocation error or a timeout. We then displace jobs with a lower priority to ensure that the job in question can be allocated at the planned start time. Replanning is also necessary if a user altered the job specification, an already assigned resource becomes unavailable (e.g., if a node is set offline or a node monitors a resource change), or a node is available again. In the latter case all waiting and matching jobs are then replanned.

Estimation of job runtime. Overstimation is handled by backfilling. Underestimation results normally in aborting the running job. However, users may increase the runtime via altering the job. There exists also a limit which we normally set to 10% of the initial runtime. Additionally, users may specify that the running job is notified by OpenCCS X minutes before the maximum duration ends. This is done by running a script or sending a signal to the job. The job then can react.

Reservations. Users may reserve resources in advance and submit then jobs to the reservation. If planning a job for a reservation, we use *rRS* instead of *sRS* and the maximum search interval is the duration of the reservation. Hence, for the scheduler, a reservation is an own system.

Exclusive node access. To be able to compute $R_{freeSlots}$ for exclusive node access, we need to know the number of free cores to be searched for. For example, assume a cluster with nodes having 16, 32, and 240 cores and the user requests `10:npcus=12`, `place=free:excl`. How many cores should be free?

Since arrangement is `free`, we could map more than one chunk on a node with 32 or 240 cores. Here, the planning phase needs mapping data to be accurate which is not possible because we cannot map before planning. To circumvent this, we compute an average number of cores for all nodes.

Timeouts. Since a planning based WLM is a real time system, we have to use timeouts for nearly all operations. For example, the administrator specifies how long allocating or releasing a job may last. If a timeout is exceeded, the node in question is set "down" in the scheduler and all related requests are replanned.

"Expensive" resources. Jobs using a GPU often also need at least one CPU core on the host. To avoid that a job which does not need the GPU blocks all CPU cores on the host, it may be specified in RSD that X cores are kept free for jobs requesting GPUs. Jobs not requesting GPUs, only get $min(availableCores - X, requestedCores)$. While building $FCN_{n,c}$, we also ensure that exclusive node access is not possible for jobs not using a GPU.

To avoid that expensive nodes (e.g., an SMP node with 32 cores and 1 TB RAM) get blocked by long running cheap chunks, but still are usable, the administrator my define node specific limits. For example jobs may only be mapped to an SMP node if they request at least 80 g virtual memory, or 66 g memory, or 17 cores, or their duration is ≤ 12 h. This is applied while building N_{cand}.

Accelerating planning and mapping. If we have FreePools and limits with a cron based validity, we accelerate the planning process, by building a template *RUSV* for the validity and then transpose it to the respective needed time interval. For this purpose, we use the routine: $R_2 = \texttt{cronToRUSV}(R_1, I, T_1, T_2, P)$. R_1 is the template *RUSV* starting at 1.1.1970 (i.e., `(time_t) 0`) and holding one period. P is the cron's period length (hour, day, or week). To get a *RUSV* with absolute times, we add a time offset (derived by T_1, T_2, and P) to each $R_1[i]$.start and stop and set $R_1[i]$.items to I. We do this in a loop with step size P until $[T_1, T_2]$ is filled. If we assume a validity of "every Monday and Friday from 7am to 11am" and T_1 is 1.1.2018 and T_2 is 31.12.2025. P is then a week.

To accelerate the mapping the N_{cand} list is built only once. It is rebuilt, whenever a node becomes available again. The node specific $FCN_{n,c}$ in the mapping process are built in parallel because they are independent. For this purpose the scheduler module *PM* uses a dynamic thread pool. The nodes to process are put in a queue and each of the threads takes a node and computes $FCN_{n,c}$ for this node until the queue is empty.

6 Performance Results

All numbers in this section are derived from real operation over four years on our OCuLUS and ARMINIUS clusters [13].

OCuLUS is running Scientific Linux and consists of 616 compute nodes with in total 9.920 CPU cores, 8 Xeon-Phis, and 32 GPUs. The nodes have 64 GB, 256 GB, or 1 TB RAM. All nodes are connected by Infiniband and Ethernet. The Xeon-Phis may be used in offload or native mode. The scheduler module *PM* is running on a host equipped with two Intel Xeon CPUs E5-2670, 2.60 GHz and 64 GB RAM. The *PM* is configured to pin on the cores 8-15, the thread-pool maximum size is 8.

ARMINIUS is running Scientific Linux and consists of 62 compute nodes with in total 660 CPU cores. All nodes are connected by Infiniband and Ethernet. The *PM* is running on a host equipped with two Intel Xeon CPUs X5650, 2.67 GHz and 36 GB RAM. The *PM* is configured to pin on the cores 8-10 (ARMINIUS has only 62 nodes) and the thread-pool size is limited to 8.

OpenCCS on OCuLUS processed about about 4.5 million jobs for about 200 different users in about 70 groups. The job sizes ranged from one to 4,096 cores. The runtimes ranged from seconds to 60 days. The initial duration limit for a new project at PC^2 is set to 7 days. The average number of processed jobs per day was 3,082 and the maximum was 196,217. 94% of the submitted jobs completed, 6% were removed by the users before they started. The average runtime was 184 m, the average waiting time 191 m. The average accuracy of the job's duration estimation was 22%. The plan normally comprises a time interval of 8 to 10 weeks and the *sRS* holds about 300 to 400 slots. The bad accuracy is mainly driven by the large number of one core jobs, submitted as job arrays. Example 3 from the OpenCCS reporting tool gives an overview of the job distribution related to requested cores. The data was sampled in 2016.

Rank	Req. cores	Jobs	Avg. Walltime	Avg. Accuracy	Avg. Waiting	%Total Occupied	Sum Occ.
1	16	57,516	10h	41.67%	7h	17.75%	17.75%
2	32	9,701	1d50m	39.68%	20h	14.19%	31.93%
3	1,024	277	19h	65.30%	8d18h	10.34%	42.27%
4	768	468	10h	55.91%	12d10h	6.63%	48.90%
5	64	3,883	13h	46.00%	11h	6.30%	55.20%
6	128	14,002	1h	26.43%	5h	4.74%	59.94%
7	256	684	13h	39.30%	14h	4.40%	64.34%
8	1,536	77	16h	64.01%	22d22h	3.62%	67.96%
9	1	1,480,565	1h	21.39%	2h	3.55%	71.51%
10	512	151	16h	62.61%	17h	2.33%	73.83%

Example 3: OCuLUS job distribution related to requested cores, ranked by occupied core hours.

```
----------RUSV INFO -------------------------
Number of RUSV-create calls     : 95,766,132
Number of RUSV-free calls       : 169,898,098
Number of RUSV-slot-new calls   : 250,851,576
Number of RUSV-slot-free calls  : 233,838,317

----------VECTOR INFO -------------------------
Vectors (used/avail)            : 5,780,091 / 6,291,456
Elements(used/avail/filling)    : 11,014,670/ 61,465,796 / 17.92%
Memory  (sum/payload/overhead) : 663.87MB / 153.44MB / 510.43MB
Allocs/Reallocs/Frees           : 11,518,320/ 16,711,065 / 0
```

Example 4: Memory consumption of the OCuLUS *PM*

The *PM* on OCuLUS uses about 6 GB RAM if 15 k jobs are in the system. Based on the data structure introduced in Sec. 4, the *PM* uses a large number of *RUSV*s. Example 4 gives an overview of the memory usage logged by the *PM*. There are 10 k jobs in the plan, the first backfill has been processed and *sRS* holds 345 slots.

On OCuLUS, we measured up to about 100 processed job submissions per second by running 30 clients on the two access nodes. Each client submitted best effort jobs in a loop. The jobs requested chunks with two cores and a maximum runtime of 2 m. Since the jobs were also running on the cluster, this is the OpenCCS performance.

The performance of the *PM* itself is higher. As described in Sec. 5.2, the runtime of the mapping process depends on the number of nodes and the complexity of the requested chunks. This is reflected if we look on the number of backfills per second which is continuously measured by the *PM*. On OCuLUS, we see numbers up to 500 and sorting of 60 k jobs by job priority takes about 20 ms. The time to plan a job array with 10 k jobs takes about 30 s. Job array planning is done in chunks of 500 sub-jobs. On ARMINIUS, we see up to 1500 backfills per second.

Core pinning is essential for the performance of the *PM*. The number of backfills per second increases by a factor of about two if pinning is activated. This is related to the large amount of *RUSV* accesses.

OpenCCS modules may be restarted at any time and if an OpenCCS module crashed, it will be automatically restarted by the *IM*. At restart a module reads its status data and synchronizes the job states with its partners. The time the *PM* needs to recover 5000 jobs takes about 20 s.

7 Related Work

There are a lot of papers related to the planning based approach. Since this paper is more a result of practical work, this section does not cover the whole area of planning based scheduling. We only relate to similar work.

Cluster and Grid. In [4], Chlumsky et al. propose a similar approach as presented here. They extend the Torque[15] scheduler to allow planning jobs to different clusters. Their approach uses job lists, holding start and completion time and gap lists, representing unused periods of CPU time and the amount of free RAM across nodes within a cluster. Both list types are sorted by time. The gap list may be seen as a kind of *RUSV*. A gap list entry points to the appropriate node, the node's free RAM, and to the nearest following job. Planning an incoming job is done by finding a place in the gap list, backfilling is done by shifting jobs into earlier "slots". The authors use a Tabu Search heuristic to optimize the current schedule. Compared to the work presented here, the approach of [4]. is restricted to plan only two resources (cores and memory). Requesting complex resource sets comprising different chunk types or job wide resources is not possible. They also neglect limits, reservations, and placing directives.

In [14], Schneider et al. propose a list based data structure to support advance reservations in Grid environments and local WLMs for HPC systems. Lists hold information about the summed up booked capacity and for each node mapping information. The list entries represent a range of free resources. Such a list may be organized in three ways:

1. As time exclusive list. For each point in time there is only one item, representing the current available capacity. The list is ordered by the start time of the blocks, that is, adjacent blocks follow each other in the free list.
2. As capacity list. Each item spans the whole time span where at least the given capacity is free. During this time span, there may be other sub time spans with more capacity available; these time spans are managed as sub lists of the longer block. Hence, a hierarchical data structure is used.
3. As mixed list. The splitting of the list items does not follow any rule. The items may be ordered by the start time and the available capacity. The list items should have references to all adjacent free blocks.

For their evaluations, the authors simulate a cluster with 128 CPUs and use the time exclusive list type. They compare three ways of organizing the lists: slotted time, list based, and AVL tree.

Schneider et al. use an approach which is very close to the one introduced here. The information about the summed up booked capacity corresponds to our *sRS*, the mapping information to the *nRS*, and the list entries are structured similar to a *slot* in a *RUSV*. The time exclusive list is nearly the same as our resource sets, except that we handle slots of used instead of free resources and do not store slots where all resources are in use.

Schneider et al. support only exclusive booking of nodes, and, just as in [4], complex resource sets comprising different chunk types or job wide resources, limits, and placing directives are not available. Additionally, they do not describe how planning and mapping should work if more than one resource type is requested, like for example `ncpus=5:gpus=3`.

Both, the authors of [4] and [14], compared their approach with other papers and assessed them all weaker, related to their approaches. Hence, and for the lack of space, we do not consider them here.

Big-Data. The following WLM examples are, in principle, all based on the MapReduce model and schedule jobs on a Hadoop platform focusing on the need for locality and elasticity of MapReduce jobs. Such jobs often consist of multiple tasks (e.g., map or reduce) that are run on different cluster nodes, where the unit of per-task resource allocation is a container (i.e, a bundle of resources such as CPU, RAM and disk I/O). An OpenCCS chunk is like a container related to scheduling. Due to the MapReduce model, tasks are often loosely coupled, malleable and may be preempted. MapReduce jobs are mainly characterized by a start time, a deadline, and a collection of stages. Each stage has a total demand of containers and may also have a minimum parallelism constraint (or gang size) of containers. The most important SLO is the job deadline.

YARN [17] schedules jobs on a Hadoop platform and comprises three basic blocks. The Resource Manager (RM), the Application Manager (AM), and, on each node, a Node Manager (NM). The RM is scheduling containers bound to a particular node.
There is on AM for each job. The AM is the head of a job, managing all lifecycle aspects including dynamically increasing and decreasing resource consumption, managing the flow of execution, handling faults and computation skew, and performing other local optimizations. Hence, the AM can be seen as a kind of workflow engine dividing a job into tasks and mapping tasks to containers. An AM is requesting containers from the RM and then starting job-tasks on such containers by using the NMs which are responsible for establishing, observing, and removing containers on a node. A container request to the RM includes: the number of containers, the resources per container, locality preferences, and priority of requests within the application. An AM may request containers to be killed when the corresponding work is not needed any more.
In contrast to OpenCCS, YARN does not plan to the future and it does not know maximum runtimes of a container. To our best knowlegde YARN can only handle containers consisting of CPUs and memory and is not able to schedule job-wide resources like licenses. However, YARN supports preemption of containers which is not supported by OpenCCS.

In [16] the authors describe TetriSched, a scheduler integrated in the YARN reservation system. It considers both, job-specific preferences and estimated job runtimes in its allocation of resources. Job-specific preferences are provided by tenants as composable utility functions, They allow TetriSched to understand which resources are preferred, and by how much, over other acceptable options. Estimated job runtimes and constraints on job execution times (e.g., deadlines or reservations) allow TetriSched to plan ahead in deciding whether to wait for a busy preferred resource to become free or to assign a less preferred resource. TetriSched translates the given requirements into a Mixed Integer Linear Problem (MILP) that is solved by an external solver to maximize the overall utility. The main advantage of TetriSched over OpenCCS is its ability to compute a global schedule by simultaneously considering the placement and temporal preferences of all the jobs in each compute cycle, and to support user given utility functions (e.g., the job needs two time units on GPUs and three on CPUs) which

allow a greater scheduling flexibility. OpenCCS does a greedy job-by-job planning. However, it is not quite clear how long it takes to solve a MILP, if there are tens of thousands of jobs in the system. In [5] the authors use heuristics due to the very long runtime of the MILP solver. TetriSched seems to support only space-sharing (i.e., a job is occupying a node exclusively).

Rayon [5] is another extension to YARN. It provides reservation-based scheduling which leverages explicit information about the deadline and time-varying resource needs of a job. Rayon comes with a declarative reservation definition language (RDL), that allows users to express a rich class of constraints, including deadlines, malleable and gang parallelism requirements, and inter-job dependencies.
The scheduler itself comprises a framework for planning SLA jobs by using fast, greedy heuristics, and a component for the dynamically assignment of cluster resources to the planned and best-effort jobs, which also adapts to changing cluster conditions. Rayon makes use of planning in two ways: online, to accept/reject jobs on arrival, and offline, to reorganize sets of accepted jobs.
Rayon immediately plans incoming SLA jobs and assigns a start time. Best effort jobs are filled in the remaining gaps by the adaptive scheduler component. The Rayon RDL is automatically transfered to a MILP formulation like in [16]. However, for the authors solving MILPs is not practical for online scenarios, and cannot scale to large problem sizes (solver runtime ranged from 80s to 3200s). Hence, Rayon, just like OpenCCS does, plans one job at a time, and never reconsider placement decisions for previously jobs. A job is divided in containers with a minimum runtime of X time-units. In case of under-reservation, an SLA job will run with guaranteed resources up to a point, and then continue as a best-effort job until completion.
OpenCCS also supports reservations and deadline scheduling but does not support neither malleable jobs nor preemption. Jobs in the HPC world are still mainly rigid. Best-efforts jobs in OpenCCS are not starving which may happen in Rayon. OpenCCS also allows renegotiation of accepted jobs.
It seems that Rayon, like YARN does, mainly supports CPUs and memory as container parts. Customizable resources are not possible. It is also not clear if Rayon supports time dependent limitations and heterogeneous clusters.

Morpheus [7], which is integrated in YARN, aims on lowering the number of deadline violations while retaining cluster-utilization. It builds on three key ideas: (1) automatically deriving SLOs and job resource models from historical data, (2) relying on recurrent reservations and packing algorithms to enforce SLOs, and (3) dynamic reprovisioning to mitigate inherent execution variance. The job resource model is a time-varying skyline of resource demands. It employs a MILP formulation, that explicitly controls the penalty of over / under provisioning and balances predictability and utilization. As in [5] Morpheus does not use an external MILP solver due to the long runtimes.
Morpheus continuously observes and learns as periodic jobs (scheduled runs of the same job on newly arriving data) execute over time. The findings are used to

reserve resources for the job ahead of job execution, and dynamically adapt to changing conditions at runtime. Periodic jobs are supported by recurring reservations, a scheduling construct that isolates jobs from the noisiness of sharing induced performance variability by assigning dedicated resources. Morpheus can only enforce container-level resources, but lacks control over globally-shared resources. When Morpheus needs to allocate resources to a new periodic job, it ignores most of the scheduled non-periodic jobs, and then attempts to reallocate resources for non-periodic jobs in case they need more resources. Morpheus assumes a homogeneous cluster. Extra resources are granted for up to T seconds and then are reevaluated. This allows an elastic job to use extra parallelism to make up for lost time. OpenCCS does not displace already planned jobs while planning new ones.
Mapping of containers is done by a cost-based approach that takes into account current cluster allocation and the resource demand of each job. Each time slot in the plan is associated with a cost and the mapper allocates incoming jobs in a way that is cost-efficient with respect to the overall costs. This is analogous to limits and FreePools in OpenCCS. However OpenCCS then reduces the number of available resources for the job in the related time slots.

8 Conclusion

We presented a data structure and a heuristic to plan and map arbitrary resources in complex combinations while applying time dependent constraints. We implemented the heuristic in the planning based WLM OpenCCS. Our approach has stand up to the reality check during four years of real operation on two heterogeneous HPC clusters (one of them with about 10,000 cores) and proved its stability, flexibility, and performance. Of course, there are drawbacks inherent to our approach and there are plenty of additional features to be added. Backfilling, for example, is a time consuming operation, especially if using a more complex policy than First-Fit. A planning horizon (e.g., 4 weeks) could reduce the amount of jobs to be planned. Fair share is part of our limit design, and we see a good and fair system utilization. However, we learned that this approach does not work sufficiently if the jobs do not fully utilize the system. Therefore, we will include dynamic soft limits which depend on the utilization of the system, allowing consumers to extend their hard limits. Additionally, we plan to extend the mapping layer to allow topology aware mapping (e.g., group chunks by switches). Also, releasing and requesting resources while a job is running is part of our future work. There is a reason why queueing based WLMs dominate the market. Queues are fast and very flexible. However, we think the planning based approach is advantageous if time dependent constraints have to be considered.

Acknowledgements

I would like to thank Christoph Kleineweber, Dr. Lars Schäfers, and Jörn Schumacher for their valuable contribution to the current OpenCCS implementation.

References

[1] Battre, D., Hovestadt, M., Kao, O., Keller, A., Voss, K.: Planning-based Scheduling for SLA-Awareness and Grid Integration. In: Proc. of the 26th Workshop of the UK PLANNING AND SCHEDULING Special Interest Group (PlansSIG2007) (2007)

[2] Brune, M., Gehring, J., Keller, A., Reinefeld, A.: RSD - Resource and Service Description. In: Proc. of 12th Intl. Symp. on High-Performance Computing Systems and Applications (HPCS'98). pp. 193–206. Kluwer Academic Press (1998)

[3] OpenCCS Manual. `https://www.openccs.eu` (January 2017)

[4] Chlumský, V., Klusáček, D., Ruda, M.: The Extension of Torque Scheduler Allowing the Use of Planning and Optimization in Grids. Computer Science 13(2), 5–19 (2012), `http://dx.doi.org/10.7494/csci.2012.13.2.5`

[5] Curino, C., Difallah, D.E., Douglas, C., et al.: Reservation-based Scheduling:If You're Late Dont Blame Us! Tech-report msr-tr-2013-108, Microsoft (2013)

[6] Hovestadt, M., Kao, O., Keller, A., Streit, A.: Scheduling in HPC Resource Management Systems: Queuing vs. Planning. In: Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP). pp. 1–20. Lecture Notes in Computer Science, Springer Verlag (2003)

[7] Jyothi, S.A., Curino, C., Menache, I., et al.: Morpheus: Towards Automated SLOs for Enterprise Clusters. In: Proc. of the 12th USENIX Symposium on Operating Systems Desing and Implementation (OSDI'16 (November 2016)

[8] Kay, J., Lauder, P.: A Fair Share Scheduler. Communications of the ACM 31, 44–55 (1998)

[9] Kleban, S.D., Clearwater, S.: Fair share on high performance computing systems: What does fair really mean? In: Proc. of 3rd IEEE International Symposium on Cluster Computing and the Grid (CCGrid03). pp. 145–153. IEEE Computer Society (2003)

[10] Lifka, D.A.: The ANL/IBM SP Scheduling System. In: D. G. Feitelson and L. Rudolph (ed.) Proc. of 1st Workshop on Job Scheduling Strategies for Parallel Processing. Lecture Notes in Computer Science, vol. 949, pp. 295–303. Springer Verlag (1995)

[11] Mu'alem, A., Feitelson, D.G.: Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. In: IEEE Trans. Parallel & Distributed Systems 12(6). pp. 529–543 (June 2001)

[12] PBSPro Open Source. `http://www.pbspro.org` (January 2017)

[13] PC^2: Paderborn Center for Parallel Computing. `https://pc2.uni-paderborn.de` (January 2017)

[14] Schneider, J., Linnert, B.: List-based Data Structures for Efficient Management of Advance Reservations. Int J of Parallel Prog 42, 77–93 (2014), `http://dx.doi.org/10.1007/s10766-012-0219-4`

[15] Torque. `http://www.adaptivecomputing.com/products/open-source/torque/` (January 2017)

[16] Tumanov, A., Zhu, T., Park, J.W., et al.: TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In: Proc. of the 11th European Conference on Computer Systems (EuroSys'16) (April 2016), `http://dx.doi.org/10.1145/2901318.2901355`

[17] Vavilapalli, V.K., Murthy, A.C., Douglas, C., et al.: Apache Hadoop YARN: Yet Another Resource Negotiator. In: Proc. of the 4th Annual Symposium on Cloud Computing (SOCC'13) (October 2013), `http://dx.doi.org/10.1145/2523616.2523633`