Masarykova univerzita
Fakulta informatiky

Job scheduling with the SLURM resource manager

Bachelor thesis

Michal Novotný

Brno, autumn 2009

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: Hana Rudová

Acknowledgement

Firstly, I would like to thank my supervisor Hana Rudová for directing the work in an admirably patient way. I am also grateful to my family and my mother particularly for their understanding and providing me with a peaceful place where I could finish this long-lasting labour. Finally, special thanks are devoted to Bára Hekerová for being an important key to my motivation.

Abstract

The aim of this bachelor work is to provide SLURM analysis based on which development of a scheduling system employing SLURM can be undertaken. It deeply explores possible ways of cooperation between the resource manager and a scheduler, gathering material required for an evaluation and comparing the ways between each other. The gathered information allow a developer to decide which approach is suitable for his intentions, and it also directly enables implementing it. Most of the knowledge comes directly from SLURM source codes that have been an object of our exploration for more than one and half year.

Keywords

SLURM, controller, interface, schedule plugin, selection plugin, wiki2, API, scheduler, queue, cooperation

Contents

Introduction

Resources in a cluster are usually controlled by one centralized software entity called *Local Resource Management System* (LRMS). Such resource manager monitors resources in the cluster and gives us a standardized way of using them by providing an interface through which we can submit allocation requests. These requests are then assigned a subset of all available resources at a certain time. When our work is being performed, LRMS provides some means of monitoring its state and reporting results after it is finished.

SLURM (Simple Linux Utility For Resource Management) is a representative of such systems created in Lawrence Livermore National Laboratory (LLNL). Letter 'S' in its name stands for Simple, but this is no longer true at over 250k lines of code and support for Linux, AIX, OS X, Open Solaris, and many interconnect architectures (see platforms at [6]). As SLURM requires no kernel modifications and being written in C with GNU autoconf configuration engine, we can expect additional operating system ports to appear. Especially if we consider, it was designed to support large clusters with up to 65,536 nodes [6], and it runs on circa 40% of supercomputers in the Top500 list today, e.g., BlueGene/L directly at LLNL with 106,496 dual-core processors which is ranked fifth on the list [12].

When talking about what SLURM is, it is also important to realize what it is not, or rather, what it was not supposed to be at the beginning. When the project was started, the idea was to create a simple, fast, but also a highly scalable LRMS with only basic queueing capabilities, such as First Come First Served. Advanced scheduling was planned to be left to an external scheduler interfaced with SLURM through its API (the same API that is used by SLURM commands for job submission, monitoring work, etc.). Today's reality is a bit different as SLURM has started to provide reservations, hierarchical bank accounts, topology awareness, etc.; it means, features we can find in today's robust schedulers.

But as it is still just an advanced cluster-centric LRMS, and connecting the cluster to a grid infrastructure might be needed, we can have good reasons to interface SLURM with an external scheduler. A favourite solution is a connection with Maui or Moab suites. These schedulers do not use the SLURM standard API; instead, a special programming interface was developed for them in a form of wiki and wiki2 plugins, which are dynamically linked modules loaded at runtime that enable the integration. The mentioned meta-schedulers provide many advanced features, but if we want to manage just a single disconnected cluster without a need for highly sophisticated scheduling, there might be a more suitable solution. The wiki plugins can be switched for others that substitute the interface for an actual scheduling logic, and such plugins are already included in the distribution, e.g., builtin or backfill. In addition to these schedule plugins, there are also interchangeable resource selection plugins. Together with a central SLURM controlling program, they implement the internal queueing mechanism.

The objectives of this thesis are to explore the queueing mechanism and the role of the plugins in it. Furthermore, to provide a description of the wiki2 and plugin interfaces so that a scheduler developer can make a practical use of it. In the end, using the gathered information to compare the internal scheduling approach against external. The thesis is divided with respect to the goals into six chapters. Chapter 1 presents basic concepts related to scheduling in computing environments. Chapter 2, being also introductory, depicts the SLURM system from a bird's-eye view to prepare ground for a detailed analysis of the scheduling system part in Chapter 3. Gained knowledge is then further used for specification of interfaces enabling internal scheduling in Chapter 4 and external scheduling in Chapter 5. Chapter 6 is a summary that compares both approaches and proposes an efficient way of cooperation between SLURM and a scheduler.

The results of this thesis shall perish as their validity is dependent on an actual SLURM version; however, we would like to see them projected into the real world before it happens. The information is based on **slurm-2.0.2**, and we expect it to be mostly correct at least up to the version 3.

Chapter 1

Scheduling In Computing Environments

1.1 Preliminaries

1.1.1 Grids and Clusters

Back in 1998, Carl Kesselman and Ian Foster defined *grid* to be a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities [3]. Because this definition leaves a lot space in what can be considered as grid, other redefinitions has appeared that try to give the notion sharper boundaries. For instance in 2002, one of the mentioned authors Ian Foster, published an article in which he gives the following definition:

> A grid is a system that coordinates resources that are not subject to centralized control, using standard, open, general purpose protocols and interfaces to deliver non-trivial quality of service. [2]

To be even more specific, we can add that grid is heterogeneous, interconnected environment with a high degree of dynamics.

Thanks to the given definitions, we can now easily differentiate between grid and another *computing environment* called *cluster*. Cluster or computational cluster is a collection of computers connected by a high-speed local area network and designed to be used as an integrated computing or data processing resource [3]. Whereas grids can be world-wide, clusters are located in a small area belonging to one organisation (control domain), such as a university. Thus, they cannot be denoted "pervasive", and they are not under decentralized control; also heterogeneity and high degree of dynamics are uncommon features of a cluster. In fact, clusters are often parts of a grid, which enables sharing of resources among organisations to provide better access to computing power, storage space, and other services.

1.1.2 Scheduling and its objects

Firstly, we provide a clarification of what is precisely meant by *resource* and *work*. This will form our objects of interest, with which we manipulate throughout the whole text hereafter. These basic scheduling objects are going to be *nodes* and *jobs*.

Resource is a very general term that can include one CPU of a machine as well as an array of disks, but we shall be interested only in one type of resource, and that is a whole machine with one or more CPUs and own memory subsystem (e.g., a personal computer). We will call such machine *node* in this text and, for us, it will be the finest resource unit that can be allocated to execute work. Many resource managers support allocating even individual CPUs of a node and SLURM is not an exception, but this is not covered in this thesis not to overcomplicate the problematics. Nodes posses various characteristics, such as the number of CPUs, amount of memory, or available disk space.

Job will be a user allocation request to which one or more nodes can be assigned (allocated). Such request can contain various constraints on an allocation, e.g., the minimum number of nodes, the number of CPUs per one node, or the earliest starting time. When LRMS grants the allocation request and reserves resources for it, the actual work (e.g., user's parallel program) can be performed on the set of allocated nodes. Hence, a job forms kind of a ground for the work itself. In spite of that, a phrase like "a node that executes a job" might be used in the following text because we want to keep the diction as simple as possible. But in fact, "a node that is assigned to a job" is more correct. So, the former phrase should be understood as the latter. In addition, "in a job" or "inside a job" shall mean "on nodes assigned to a job".

Now we can define the notion of *scheduling*, which is allocating available nodes to jobs in time. Actually, we can find this very important process in its other forms in various areas of human activity. As an example for all we can consider scheduling nurses in a hospital.

1.1.3 Schedules

A result of the scheduling process is represented by *schedule*, and we will differentiate two types of it. *Complete schedule* must provide information about time and place of objects we are scheduling (e.g., lectures, appointments, or jobs).

On the contrary, *queue* does not provide such specific information; instead, it gives a time ordering of objects and the actual time and place is determined at the moment *head* of a queue (the first object) is removed and processed after.

A validity of a schedule is not usually defined as to be dependent on satisfying constraints connected with a problem. Instead, we just say that a schedule not satisfying them is *infeasible*. On the other hand, a schedule satisfying constraints is called *feasible*.

1.1.4 Schedulers

A program dedicated for scheduling in (and not only in) clusters is called *scheduler*, and it can be a subpart of a LRMS or a standalone software package, e.g., The Maui scheduler [9]. Schedulers like Moab [9], GRMS [13], or GridWay [4] operate on the grid level and are called *meta-schedulers*. There might be several of them in a grid, cooperating together and submitting work to underlying cluster resource managers. They schedule in grid-wide manner; instead, a LRMS with a built-in or external scheduler then schedules the assigned work inside the cluster.

Generally, creating of a schedule for the cluster scheduling problem (i.e., which nodes are allocated at what time for jobs we have got) depends on: scheduler's decision algorithms, the system state at the moment the decision process was started, and on user specified constraints of jobs. Further, the scheduler might continuously rebuild the schedule to incorporate new incoming jobs and changes in the hardware state (e.g., a resource failure), or it might finish constructing the schedule in accordance to the starting input data and rebuild it after that.

1.1.5 Scheduling taxonomy

In the following paragraphs, we briefly introduce categories into which grid or cluster scheduling may fall and provide comparison of scheduling in these two environments based on it.

Scheduling is *centralized* if only one entity has control of it. The opposite *distributed* scheduling means the decision-making process is shared among several entities [11]. A cluster employing an arbitrary LRMS with a built-in scheduler is a good example of centralized scheduling. For the other class, we can name grid-wide scheduling where several meta-schedulers are used as peers.

We can also distinguish between *local* and *global* scheduling. The local scheduling stands for scheduling processes of a job to CPUs in a single machine and is usually performed by a dispatcher of an operating system. On the contrary, global scheduling is responsible for mapping whole jobs to nodes as we have introduced it.

Static scheduling practically does not occur in the computing environments, as it presumes perfect knowledge of a current and also future system state and resource demands of jobs. If we possessed such knowledge, *optimal* scheduling that reaches optimal schedules would be possible. But since the computing environments are highly dynamic and the perfect information cannot be obtained, scheduling in grids and clusters is *suboptimal*. It is then also classified as *dynamic* just for it works with the imperfect information and must readily react on new events, such as an incoming job or a node failure.

Equipped with the taxonomy, we can know say that scheduling in both environments is dynamic but a degree of dynamics is much higher in grids because of its strong heterogeneity and more resources in use. We are able to achieve only suboptimal results in both but then again, having reliable homogeneous resources in a cluster can allow much better scheduling results than in a grid. Clusters today are usually centralized but there is no hitch in them being distributed. On the contrary, grids are distributed as the matter of fact because being centralized brings serious risks of the infrastructure collapse.

The truth is that they are also not completely distributed with all machines as peers in control; instead, a hierarchical control model with more roots is applied. Finally, scheduling is global in both environments, but some cluster schedulers are able to perform also the local scheduling without any help from an operating system. For instance, SLURM with its integrated schedulers is capable of addressing particular processors in a machine.

1.2 Scheduling methods

The area of scheduling techniques is vast and it is far beyond our scope to be comprehensive here. Hence, we just settle for presenting one method called *queueing* with optional *backfill* modification, which is of the centralized scheduling paradigm. Queueing is also suboptimal even without setting it into a dynamic environment with imperfect information. It can be used for the local scheduling as well as global. We cannot state whether it is static or dynamic, since dynamics is exclusively an attribute of an environment where the technique is employed.

Before we start describing the method, there is one more concept to explain, denoted as *job preemption*. It is not a scheduling technique as queueing but rather an extension to it. This useful functionality enables to suspend an arbitrary job and schedule another on its place. This generally allows constructing more optimal schedules, but job preemption is not without certain issues. Processes of a job being preempted can be involved in communication with some other job's processes. In such case, the whole computation involving these jobs might get stuck or even terminate prematurely if such event is not properly treated in applications running the computation. To prevent from a potential loss of all work that has been done, *checkpointing* can be utilized. In that case, the system periodically saves states of running jobs (chekpoints them) to allow later restart from the latest checkpoint. In SLURM, jobs can be also *requeued*, which means they are terminated on purpose in order to release not only computational power but also memory. Such jobs can be later restarted, continuing execution from their last checkpoint.

1.2.1 Queuing

This method is extensively used in computing systems today, so we can name GridWay, GRMS, and also SLURM as a few picked examples of *queue-based* systems. This technique uses one or more *queues*, as we have defined them, where elements are jobs waiting for their execution. In every iteration of a queueing algorithm, a head of a queue is chosen and the respective job is then allocated nodes—a procedure that is an essence and cannot be taken away from this method. However, there is still space to differentiate by using various policies for the head selection and node allocating after that.

Regarding the first step, these policies, also referred to as dispatching rules, are usually simple heuristics, falling into the group of *static* rules if they are not time dependent or *dynamic* rules if they do depend on the current time. *First Come First Served* (FCFS), or *Longest/Shortest Job Time First (LJF/SJF)* can be named as examples of dispatching rules. They are all static because the selection works in accordance to job arrivals and processing times respectively.

FCFS, as simple as it is, offers several advantages. Mainly, it does not require an estimation of processing times, and it has an undeniable fairness property towards users. Nonetheless, this policy tends to low utilization of resources for obvious reasons. Hence, *backfill* modification, which enables overtaking jobs, has been developed. This modification allows to allocate jobs out of order if these jobs do not delay an estimated starting time of the head. So, resources that otherwise stay idle doing nothing are allocated with the backfilled jobs. The other two rules as well as the backfill modification need to know processing times to perform well, which is often a catch in computing environments.

For the second step—allocating nodes to a head—there are also various heuristics, such as *First Fit* or *Min Load First*. The First Fit policy is very simple as it chooses the first resources it finds available and sufficient. Min Load First attempts to be little more sophisticated by preferring resources with the least load in the system.

Chapter 2

SLURM overview

2.1 Architecture

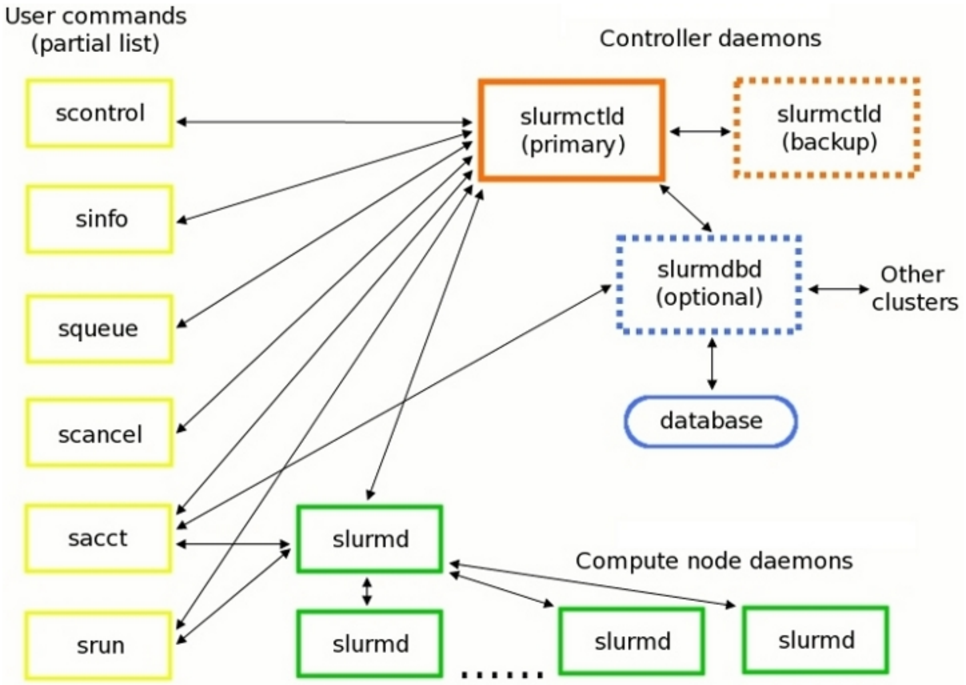

Figure 2.1: SLURM architecture

As depicted in Figure 2.1, SLURM consists of several components. A key element of the system is *slurmctld* daemon, representing a central brain that controls system core functionality. If a failure occurs on a node running *slurmctld*, all system control might be undertaken by its optional fail-over twin. The *slurmctld* controller is interfaced with various commands, such as *scontrol*, *sinfo*, *squeue*, *scancel*, *sacct*, or *srun* (see [6] for the full list). These commands, which can be run anywhere in the cluster, enable a user to submit and monitor work as well as watch and control resources but only if the user is authorized to perform his intended action.

When we call *slurmctld* a brain, we could certainly consider *slurmd* daemons as his hands, since they run on every compute node in the system, carrying out incoming orders. Their duties include launching, monitoring, and signalling job processes, and they are also responsible for gathering information about a host machine and sending it to the controller. As you can see in

Figure 2.1, there are links connecting certain commands with *slurmd*. The reason for it is simply to take away some processing load off the controller. For the same purpose, when *slurmctld* needs to order the *slurmd* daemons, it will communicate only with one of them that will, subsequently, become a root of a distributed communication tree.

Finally, the last and optional unit of the system is *slurmdbd*, possibly connected to more than one SLURM clusters, which provides an interface to a database. This is particularly useful for storing and accessing information about past work. [14]

2.2 Entities

Every complex artificial system has its own set of abstractions called entities, which represent appropriate pieces of reality, and SLURM cannot violate this. One of the most basic system entity is *node* that respects the definition of node given in the previous chapter. Every node must be included in at least one *partition* that aggregates resources into a logical structure. Hence, partitions are usually used for linking nodes having a strong connection in their characteristics, which is, usually, the same set of hardware or software features. While covering all resources in the cluster intended to being employed, they does not need to be disjoint so that a node might be included in more than one partition. Again, there are many different parameters defining a partition, namely, job time limit, users permitted to use it, priority, and so forth.

We have discussed resource entities, but there would be no reason for their presence without other entities giving them a purpose. Not surprisingly, the other entities are *jobs*—basic work units corresponding to the definition in Chapter 1; but additionally, the SLURM jobs cannot be allocated nodes from more that one partition. A job forms a framework for *job steps* that constitute the work itself (e.g., user's application). These work units can be executed in parallel as well as sequentially, but neither they are atomic. The lowest work level is represented by *job tasks* that are parallel computing processes, each running on one node in their job step frame. See Figure 2.2 that depicts the entity relations.

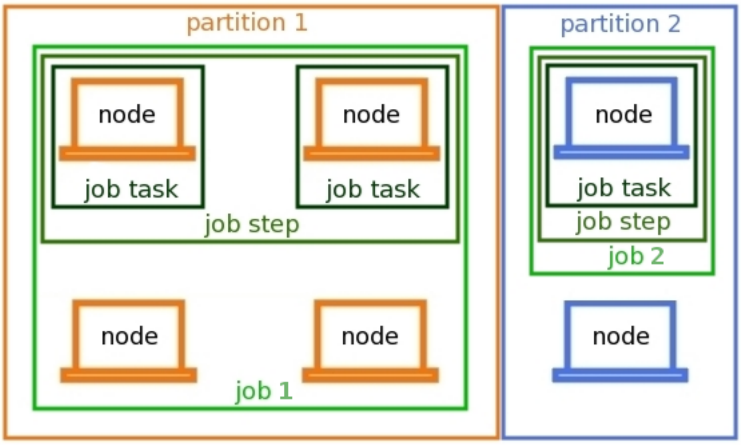

Figure 2.2: SLURM entities

2.3 Configuration

The system functioning extensively depends on SLURM configuration file called `slurm.conf`. Many settings can be configured and adjusted in it: a hostname of a node running *slurmctld*, a debugging output level of the controller, specifications of nodes to be managed, grouping those nodes into partitions, choices of particular plugins, and many others. We concentrate on a few settings relevant to our goal.

2.3.1 Node Configuration

Node configuration entries assign particular characteristics to nodes they are related to. When SLURM is selecting resources for a job, data in the entries are being compared against the job constraints so that unsuitable nodes might be excluded from possible candidates. One entry might contain a configuration for multiple nodes as SLURM uses *node range expressions* that allow this.

In fact, only node names must be supplied in `slurm.conf`. All other parameters are optional although it is advisable to establish *baseline* configurations, especially if the cluster is heterogeneous. Nodes that register to the system with less than the configured resources (e.g., too little memory) will be placed in *down* state to avoid allocating jobs to them. This standard behaviour can be changed by `FastSchedule` setting (see Scheduling Configuration).

The registration process takes place when the *slurmd* daemon is launched on a node in the cluster. It gathers the actual configuration of the node and sends it to the controller. Values of `Procs`, `RealMemory`, and `TmpDisk` are recorded into a controller's node record and might be used in the selection procedure. But primarily, the baselines from `slurm.conf` (or default minimum values) are used. Establishing them will allow SLURM to make the excluding decisions upon a few entries in `slurm.conf` instead of numerous individual node records. The exact behaviour depends again on the value of `FastSchedule`. Node parameters include:

NodeName Name that SLURM uses to refer to a node. Multiple node names can be specified in one entry using the node name expression, e.g., `nymfe[1-10]`.

Feature Comma delimited list of arbitrary strings indicative of some characteristic.

Procs Number of CPUs on the node.

RealMemory Size of real memory on the node.

TmpDisk Size of temporary disk storage.

Weight Priority of the node for selection purposes. Jobs tend to be allocated on the lowest weight nodes which satisfy their requirements.

EXAMPLE `nymfe[1-10] Procs=2 RealMemory=4096 Weight=5 Feature=foo1,foo2`

2.3.2 Partition Configuration

As we have stated, a job cannot span partitions, but in fact, the relation between jobs and partitions is much more complex than that. Partition is internally treated also as a job's mandatory attribute, assigned upon submission, which has a significant impact on the job itself. Limits defined in the partition configuration entries might be propagated into the job's own attributes (constraints), or they might even cause rejecting it from the system.

Of course, partition is not just a common attribute of job. From the other point of view, it is a queue with limits and assigned nodes that accepts jobs and imposes additional constraints on them. We have chosen the following parameters to present:

PartitionName Name by which the partition is referenced.

Nodes Comma separated list of node names which are associated with this partition. Node names may be specified using the node range expressions.

Default Defines if this partition will be assigned to jobs with no partition specified.

MinNodes Minimum count of nodes which might be allocated to any single job.

MaxNodes Maximum count of nodes which might be allocated to any single job.

RootOnly Specifies if only user with id zero (i.e., root) may allocate resources in this partition.

Priority On partition overlaps, jobs submitted to a higher priority partition will be dispatched before jobs in lower priority ones.

Shared Controls the ability of resources to execute more jobs at the same time. Possible values are `YES`, `NO`, `EXCLUSIVE`, `FORCE`.

EXAMPLE `PartitionName=debug Nodes=nymfe[1-10] MaxNodes=10 Priority=1 Default=YES Shared=YES RootOnly=NO`

2.3.3 Scheduling Configuration

This is the place where internal plugins are chosen and several other parameters are set that influence scheduling process in SLURM.

FastSchedule Controls the node excluding processes. Possible values are:

- **0** Set a node down if it does not satisfy configured resources. For a job, check the baselines first. If some is not sufficient, exclude related nodes upon their actual configurations one by one.
- **1** Set a node down if it does not satisfy configured resources. For a job, check the baselines and drop all nodes in not sufficient ones.
- **2** Keep a node available for allocating even if it does not satisfy configured resources. For a job, check the baselines and drop all nodes in not sufficient ones.

SchedulerType Identifies the schedule plugin to be used. Acceptable values include:

- **sched/builtin** For the simplest FCFS scheduler.
- **sched/hold** To hold (not allocate) all newly arriving jobs if file `/etc/slurm.hold` exists. If it does not exist, FCFS is used.
- **sched/backfill** For a backfill scheduler that augments FCFS scheduling.
- **sched/wiki** To enable the *wiki* interface designed for the Maui scheduler.
- **sched/wiki2** To enable the *wiki2* interface designed for the Moab scheduler.
- **sched/gang** For a gang scheduler that time-slices jobs (suspends and reruns them) in predefined time intervals. This plugin also supports preemption of jobs in lower-priority partitions on overlaps with higher-priority ones. See [5] for details.

SchedulerPort The port number on which *wiki/wiki2* plugins should listen for connection requests from an external scheduler.

SelectType Identifies the resource selection algorithm to be used. Acceptable values include:

- **select/linear** For allocation of entire nodes to jobs assuming nodes form a linear connected chain in which sequentially ordered elements are preferable.
- **select/cons_res** The resources (CPUs, sockets, cores, memory) within a node are individually allocated as consumable resources.
- **select/bluegene** For a three-dimensional torus BlueGene system.

SelectTypeParameters The permitted values of SelectTypeParameters depend upon the value of `SelectType`. For us, only options applicable to `select/linear` will be important and then there is just one choice:

- **CR_Memory** Treats memory as a consumable resource and prevents memory oversubscription.

2.4 Commands

For a user to allow his interaction with the system, there must be some sort of entry points and also keys to them. The SLURM application programming interface (API) is one such entry point and examples of the keys to it can be found in the incomplete list below. The commands communicate with the controller by using *RPCs* (Remote Function Calls) that invoke associated handlers on each side. Note that the described behaviour below corresponds to the most common use but it can be modified with various commands' arguments and system environment variables.

2.4.1 Administrative Commands

sinfo Used to view partition and node information for the system.

squeue Used to view information about jobs and job steps managed by SLURM.

scancel Used to cancel a job running or waiting for its execution.

scontrol Used to view or modify SLURM configuration, job states, node states, etc.

2.4.2 Allocation Commands

salloc Attempts to create a resource allocation (job in other words) and blocks the execution until *slurmctld* grants it. Then, *salloc* spawns a shell on the first node in the allocation, for the user to launch job steps by *srun* subsequently. *srun* will recognize it is being run in the existing allocation, so all the job steps will run only on the allocated nodes.

srun This command has two operational modes. Being run in an allocation previously created with *salloc*, it immediately performs job step execution inside its job. Otherwise, it first has to do the work of *salloc*, i.e., create an allocation, and continue with job step execution after that. Basically, *srun* is an extension of *salloc*. From our point of view, only the job allocation is important, which means only the *salloc* part.

sbatch Submits a given batch script to SLURM. It is similar to *salloc*, except *sbatch* does not hang its execution until resources are granted, and the consecutive job steps, which would be run manually after using *salloc*, are predefined in the script and thus automatically processed.

2.4.3 Allocation Parameters

The allocation parameters (constraints), which can be specified by the allocation commands, are important because they significantly influence the selecting process inside SLURM. A range of supported constraints is also related to complexity of a scheduling problem we face; hence, an algorithm design must be done with respect to them. However, we will meet most of these parameters as job attributes in the following chapter (Controller's Job Processing) with their meaning explained. Therefore, here we present just the most complex ones.

--exclusive By this, a user demands the job allocation cannot share nodes with other running jobs (in other words, it forbids overlaps with other resource allocations). The partition `Shared` setting and selection plugin in use have impact on the resulting behaviour. The same applies for the following `--share` option. See 2.1 for details.

--share The job allocation can share nodes with other running jobs. This may result in the allocation being granted sooner and allow higher system utilization, but application performance will likely suffer due to competition for resources within a node.

	`no parameter`	`--exclusive`	`--share`
`EXCLUSIVE`	No	No	No
`NO`	No	No	No
`YES`	No	No	Yes
`FORCE`	Yes	Yes	Yes

Table 2.1: Sharing policy with the linear plugin

--dependency=<dependency_list> Defers the start of this job until the specified dependencies have been satisfied. There are various types of dependency that can be specified:

after This job can begin execution after specified jobs have begun execution.

afterany This job can begin execution after specified jobs have terminated.

afternotok This job can begin execution after specified jobs have terminated in a failed state (not satisfiable constraints, a node failure during execution, etc.).

afterok This job can begin execution after specified jobs have successfully completed.

singleton This job can begin execution after any previously launched jobs sharing the same job name and user have terminated.

--constraint=<feature_expression> A parameter whose meaning is the most complex and problematic of all. Unfortunately, it cannot be left aside as it has also a great impact on the selection procedure. Basically, one `feature_expression` is internally treated as three subexpressions each imposing essentially different constraints on the allocation. The subexpressions can be characterized as:

xor features e.g., `[A|B|C]`. All nodes in the allocation must have the same one feature that occurs in the expression. In fact, the implementation allows allocating also a node that has multiple features occurring in the expression, but the name "xor features" is related to their usage. It is assumed that the features themselves will be mutually exclusive, e.g., different racks.

count features e.g., `A*2&B*3`. Each feature in the expression has assigned a number. For every count feature, the allocation must contain at least the specified number of nodes possessing the feature. But as we will see in the following chapter, it has significant "side effects".

per-node features e.g., `A&B|C`. This is an expression that must hold for every node in the allocation. The vertical bar means the logical OR and the ampersand means AND. Other operators and parentheses cannot be specified.

Count features are not compatible with xor features; thus, the subexpressions in effect are always just two. In fact, a vertical bar must not be present together with count features. Now, we will show by examples how a `feature_expression` can be syntactically divided into the subexpressions of which a meaning is known. Parentheses are added to illustrate a modified order of evaluation.

1. `A|B&[C|D]|E` -> `A|B&(C|D)|E` and `[C|D]`
2. `A&B*1&C*2&D` -> `A&D` and `B*1&C*2`

Chapter 3

Controller's job processing

As we already know, SLURM is a system supporting just the basic queueing scheme. Yet, the scheduling functionality can be substantially modified thanks to its plugin technology that enables switching parts of the scheme. To remind, these parts are two—selecting a queue head and allocating nodes to the head.

It seems natural to assign one plugin (*schedule plugin*) to the first task and another (*selection plugin*) to the second task so that the slurmctld controller just interconnects them to build the scheme. We could imagine that the schedule plugin determines a head by giving the queue an order with respect to some queueing policy. Then, the controller would just pick the head and pass it to the selection plugin to select nodes for it. And after the selection plugin makes its decision, the controller would start the job on the selected nodes, handling all possible resource managing issues. Nevertheless, this model of one scheduling iteration is just partially utilized in SLURM.

The main difference is that the controller does not only pass the head but performs a lot of selecting work itself, and the selection plugin is used as a tool in this procedure. Of course, nothing changes from the programmer's point of view if he is exactly given his input and required output. Then, an algorithm picking the best nodes in accordance to applied criteria might be designed as well as before. But serious issues come when we want to analyze overall system scheduling behaviour in dependence on utilized plugins. In such case, we need to know how exactly the selection plugin is used and what is the piece of scheduling the controller handles itself.

This chapter provides the answers by presenting the logic connecting the "upper" schedule plugin and "lower" selection plugin. It was extracted from slurmctld's source codes and transformed into a natural language. We tried to prune details to leave only the core needed for further analysis. For that purpose, reservations, accounting policy, architecture-dependant pieces, detailed error handling and some other things have stayed left out. There are also places where the controller's processing itself is influenced by the plugins employed, e.g., in `_pick_best_nodes` (see below) with the `select/cons_res` plugin enabled. Therefore, we implicitly suppose the `sched/builtin` and `select/linear` plugins to be employed as they do not have this impact. Only if something is interesting or relevant for the next chapters, we point it out even if it concerns other plugins.

Although the pruning is effective, important principles should be untouched. In the end of the chapter, you can find an application of the logic to interfere an interesting corollary about count features. Before we finally present the logic, objects of its processing are to be introduced.

3.1 The logic objects

3.1.1 Global job table

A central storage for all jobs is `job_list` declared in `slurmctld.h`. Interestingly, it contains also jobs that cannot be executed anymore (e.g., in the `JOB_CANCELLED` state). This is for accounting purposes; however, `job_list` is being continuously purged in predefined time intervals. The table is implemented as List that comes from from LSD-Tools (LLNL Software Development Toolbox).

3.1.2 Global node bitmaps

Global bitmaps, also declared in `slurmctld.h`, store derived node state layout. Each bit in them corresponds to some node in the system. While they are frequently used in the logic, some of them might be of particular interest also for a plugin programmer. For instance, the backfill scheduler uses `up_node_bitmap` and `avail_node_bitmap` for its purposes. We provide their meaning by defining when a bit is set in them.

`up_node_bitmap`	set if the node is not down (i.e., switched off, not responding, etc.)
`avail_node_bitmap`	set if the node is available for allocating to a job
`idle_node_bitmap`	set if the node is not allocated to any job
`share_node_bitmap`	set if the node is not allocated to an exclusive job

Thanks to the bitmaps, we are not forced to slowly go through a global array of node records to find out if a node can be allocated to a job. In addition, there is no need to even touch any actual node state (e.g., `NODE_STATE_IDLE`) while scheduling because the derived content of the bitmaps is all we need. As a result, we do not introduce node states here, but we should know some bits about possible job states.

3.1.3 Job states

`JOB_PENDING`	queued and waiting for initialization
`JOB_RUNNING`	allocated resources and executing
`JOB_SUSPENDED`	allocated resources, but suspended
`JOB_COMPLETE`	completed execution successfully
`JOB_CANCELLED`	cancelled by user
`JOB_FAILED`	completed execution unsuccessfully
`JOB_TIMEOUT`	terminated on reaching time limit
`JOB_NODE_FAIL`	terminated on node failure

3.1.4 Job constraints

There are two categories of job constraints: per-node constraints and allocation constraints. Members of the first group impose limits on every node in an allocation, whereas the second group members concern overall properties of an allocation. We shall be interested mainly in the latter as there is just one point in the logic where the former is considered. This point can be found in `select_nodes` below. So, we give only a few important members.

`job.job_min_procs`	minimum of CPUs per node
`job.job_min_memory`	minimum of memory per node
`job.exc_node_bitmap`	nodes that must not be allocated
`job.req_node_bitmap`	nodes that must be allocated

Note that `job.feature_list` is a member of both groups because xor and count features are related to whole allocation characteristics while ordinary features impose a constraint on each node in the allocation. A list of allocation constraints with their meaning follows.

`job.num_procs`	minimum total number of CPUs in an allocation
`job.time_limit`	maximum duration
`job.min_nodes`	minimum number of nodes
`job.max_nodes`	maximum number of nodes
`job.begin_time`	the earliest starting time
`job.depend_list`	dependencies on other jobs
`job.req_node_layout`	task layout for required nodes
`job.shared`	set if job can share nodes with other jobs (i.e., not exclusive)
`job.feature_list`	node features required (all the three kinds of features included)
`job.contiguous`	set if nodes must form a contiguous chain

3.1.5 Other job properties

Here are additional job properties we will work with. These properties are mostly not constraints, so they do not fall into the previous section. An exception is `job.part` because jobs cannot span partitions. In addition, `job.part` contains some sub-properties also being constraints related to the job itself.

`job.priority`	priority relative to other jobs
`job.job_state`	one of the introduced job states
`job.nodes`	list of nodes allocated to job
`job.start_time`	time when execution started
`job.end_time`	expected time of termination
`job.nice`	requested priority change
`job.part`	assigned partition
`job.part.priority`	partition priority (superior to job.priority)
`job.part.max_nodes`	maximum number of nodes for a job
`job.part.node_bitmap`	nodes included in partition
`job.part.max_share`	maximum of jobs sharing one node

3.1.6 Errors

Finally, we define four basic errors that the logic uses. They correspond to real errors used in the implementation. Their names are followed by where a reason for the job rejection lies, based on its constraints.

`node_error`	node configuration
`part_error`	partition configuration or current state of nodes
`busy_error`	system load
`held_error`	job.priority=0

3.2 The logic

Now, we are ready to proceed by describing individual procedures involved. Please, refer to Figure 3.1 and Figure 3.2 that depict the procedure hierarchy and particularly mind parts marked in red as these are under our scope. You might notice we do not include the job entry into the system that starts in `_slurm_rpc_allocate_resources` and continues through `job_allocate` to connect on select_nodes. While being an important portion, it can be omitted from our scheduling point view. Still, a few things are worth mentioning about it.

Job_records (slurmctld.h) are created out of received specifications in job_allocate by calling _job_create, which also includes appending them to job_list. And after this is done, the initial priority of a job is calculated using slurm_sched_initial_priority but only if it has not been set directly in the specification. This is the place where job.nice might be considered; hereafter, it is not taken into account. The select_nodes function can be invoked in two modes from there. If the job is eligible to start now (i.e., top priority job, satisfied dependencies, job.begin_time not in future, etc.), select_nodes is called in test_only=false mode in attempt to launch it now. Otherwise, test_only=true mode is used to determine if a job is runnable at all. For instance, if node_error is returned, the job is set to JOB_FAILED state in both cases. Soon after, it will be removed from the system.

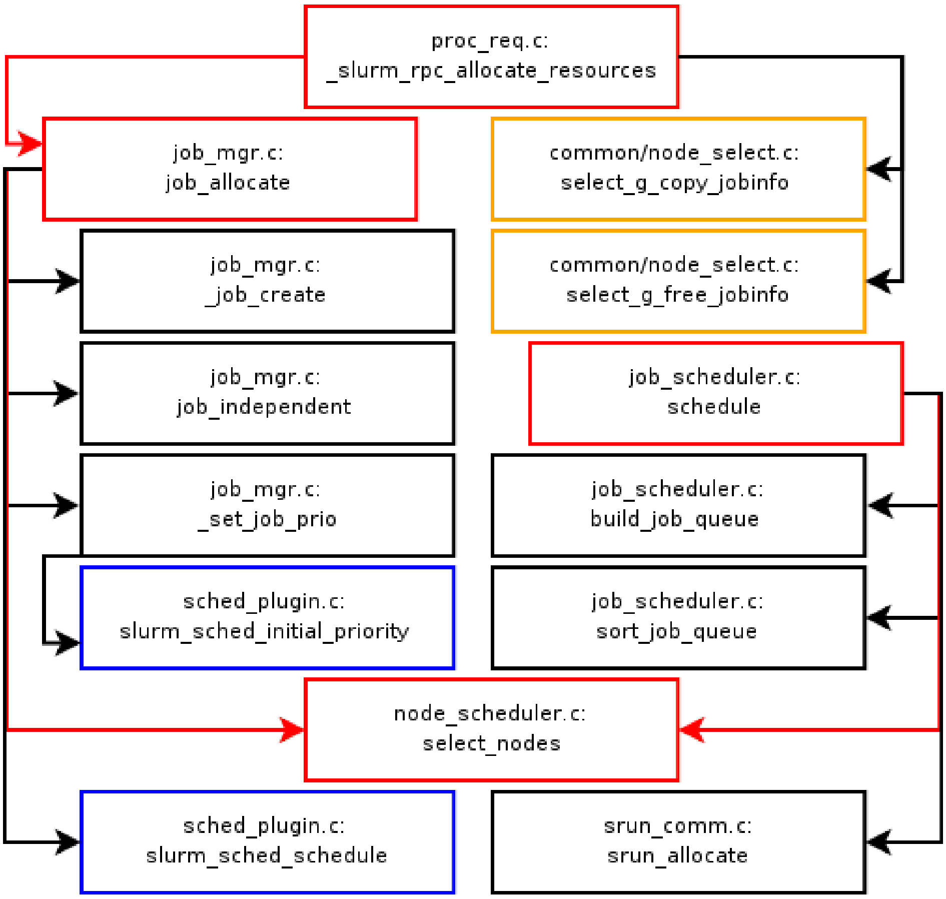

Figure 3.1: The controller's job processing

3.2.1 schedule

Arguments: None

The `schedule` function forms a bridge between the schedule plugin, giving jobs an order of execution, and the node selection procedure that starts in `select_nodes`, going down to reach the selection plugin. Altogether, they form a queuing scheme employed in SLURM. Despite its name, `schedule` is just an extension of the take-head step we can find in any queuing algorithm.

It is an extension because of three main reasons: handling selection errors, responsibility for launching jobs, and processing upon a queue of queues. In fact, it is exactly the way of processing what makes the data structure to be more complex than queue. On the level of programming language, it is just a dynamic one-dimensional array. The matter is, we distinguish between jobs from different partitions that naturally form different inner queues (or groups). When there are not enough resources for a job in its partition, we further skip all other jobs of the same partition, which makes the logical equivalence complete.

The array is not a global variable but it is built every time we enter `schedule`. It involves picking all pending jobs from `job_list`, leaving out those with: `job.priority=0`, `job.begin_time` in future, or unsatisfied dependencies in `job.depend_list`. Although a precedence order of jobs is given, `job_list` is not kept in it. And after the array is built, it is unsorted as well, so we perform selection sort to enable linear passing that respects the order.

The sorting to descending order is performed primarily in accordance to `job.part.priority`. Only if it is the same, `job.priority` is considered. We think, the schedule plugin was principally designed to work only with `job.priority`, but modifying partition priorities is possible as well although not that easily.

After the array (queue of queues) is constructed and sorted, we iterate over its elements, calling `select_nodes` (with `test_only=false`) for jobs if their partition has not been locked yet. A partition becomes locked if `select_nodes` returns `busy_error` for its job. In such case, we also make nodes of this partition unavailable for all the following jobs processed. Obviously, this step, consisting of temporary modifying `avail_node_bitmap` accordingly, is useless if there are no overlapping partitions. If the error returned was either `part_error` or `node_error`, we continue without locking or modifying `avail_node_bitmap`. But in the latter case without wiki or wiki2 enabled, the job is set to the `JOB_FAILED` state as it is non-runnable with a given node configuration. If `_select_nodes` was successful, on the other hand, we call procedures that will start the job on the selected nodes.

The last thing to be mentioned is a fragmentation counter-policy. If there is a completing job in `job_list`, we just exit at the beginning of the function to be invoked a bit later after the job completion. This optimization of resource usage is disabled in case one of the wiki plugins is in charge.

3.2.2 select_nodes

Arguments: `(job, test_only, select_node_bitmap)`

First, we confirm that `job's` partition is up and its node and time limits are compatible with `job's`. If they are not (e.g., `job.min_nodes` is higher than `job.part.max_nodes`), `job.priority` is decreased to one, which makes it be at the end of the job queue so that it does not prevent other jobs from starting. Only jobs with zero priority could be behind those, but these jobs are not even supposed to be executed until their priority is increased. If our `job` is of zero priority, it means this function has been called from `job_allocate` as `schedule` skips them. In both cases (zero priority or broken partition limits), we just exit `select_nodes` immediately, returning an error (`held_error` or `part_error`, respectively).

Next, a very important part follows, since a weight-ordered list of `node_sets` is built by the `_build_node_list` function. This data structure is constructed to keep possible candidates for allocating throughout the following selecting procedure. Every node_set in the list has its construction frame in a node configuration record defined in slurm.conf (e.g., NodeName=nymfe[1-10] Weight=5 Procs=2 Feature=A), but not all the records are necessarily involved. It is because they are immediately dropped if their baseline configuration cannot satisfy job's per-node constraints and FastSchedule is set to 1 or 2. Even if FastSchedule is zero and we, therefore, check the actual node configurations as reported by slurmd daemons, we can end up purging out a whole node set of which the baseline has been broken. See FastSchedule in Slurm Overview for more information about this parameter and how it influences the selecting behaviour. Note that also job.exc_node_bitmap and job.part.node_bitmap are used to remove unwanted and unusable candidates.

After _build_node_list is successfully finished with the construction, we check if all nodes marked in job.req_node_bitmap are contained in node_sets. If any of them is missing, we can immediately state that job is not runnable on our node configuration and return node_error.

The next step is setting the values of min_nodes (lower bound), max_nodes (upper bound), and req_nodes (desired count of nodes), which influence the resulting number of nodes assigned to job. This operation is performed here and not while job creating because a partition of a job can be changed while the job is being queued (e.g., by using the scontrol command). And since the respective partition limits are taken into consideration together with the job limits, the values would probably have become invalid if we had done it in _job_create and then the job's partition changed. If min_nodes is not higher than max_nodes, we proceed by calling _get_req_features that further undertakes the responsibility for selecting.

If _get_req_features returns with success, we can find selected nodes in its output parameter select_bitmap. In case job is test_only, i.e., not intended to be started now, we exit assigning select_bitmap to select_node_bitmap and calling slurm_sched_job_is_pending. This notifies the schedule plugin of a job pending execution, which means it could be started now if there were no other jobs in the system. Here, the internal backfill scheduler would be invoked to process, if enabled.

If job is to be started now, job.start_time is set to the current time, and job.end_time is then derived from job.start_time, job.time_limit, and job.part.max_time. But even at this state, the job start might be spoiled if the call of select_g_job_begin is not successful. Its purpose is to notify the selection plugin of job being initiated so that it can perform changes in its internal tables. Nothing bad should therefore happen if the plugin is well programmed.

We proceed by modifying global idle_node_bitmap and share_node_bitmap accordingly, setting job.state to JOB_RUNNING and job.nodes to nodes marked in select_bitmap. As the matter of fact, share_node_bitmap is modified only if job.shared=0, which means that job needs an exclusive allocation.

Additionally, select_g_job_update_nodeinfo and slurm_sched_newalloc are called. The former is to inform of state changes of job.nodes that are about to happen, and the latter serves as a notification of a new successful allocation for the schedule plugin. After that, select_bitmap is assigned to select_node_bitmap and the success is finally passed above.

Handlers for errors of _get_req_features are also involved, causing _select_nodes to exit. Just two things are worth noticing in them. If busy_error has been encountered, the function slurm_sched_job_is_pending is called. If it has been part_error, job.priority is decreased to 1. For this time, not the success but error is passed to the above layer.

3.2.3 _get_req_features

Arguments: (node_sets, select_bitmap, job, job.part, min_nodes, max_nodes, req_nodes, test_only)

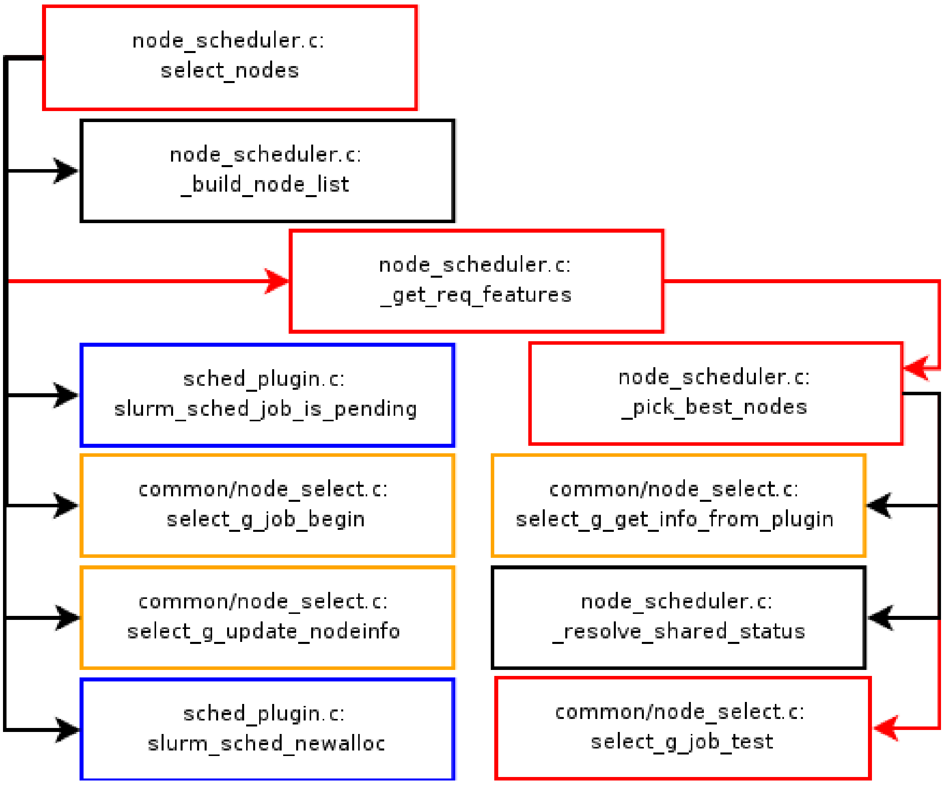

Figure 3.2: The controller's job processing

This function is responsible for satisfying count features using an iterative accumulation scheme. We iterate in a loop over all count features occurring in `job.feature_list`, if any, and call `_pick_best_nodes` to satisfy each of them. But before this process is started, we need to save values of a few variables being a modified input for the following `_pick_best_nodes` calls. These variables include: `min_nodes`, `req_nodes`, `job.num_procs`, and `job.req_node_bitmap`. Then, we clear `job.req_node_bitmap` so that the user request does not influence the procedure.

Interesting thing is, the loop is not started if `job.req_node_layout` is not `NULL`. This attribute is an array of integers that defines the number of tasks for each `job's` required node. It is used by the wiki2 interface to enable scheduling with a finer than node granularity. Nevertheless, the consumable resources plugin must be active to make use of the layout as it is ignored otherwise; particularly with the linear plugin.

The reason why the loop is passed over if the layout is set is unknown to us, but we can guess a logic of the `cons_res` plugin cannot handle just a partial information about task layout for `job's` required nodes. The matter is that nodes accumulated for count features will be set in `job.req_node_bitmap` in the end, but no task layout is created for these.

As a result, if the `cons_res` plugin is not enabled, we could set this attribute from the schedule plugin just to skip the accumulating procedure so that count features will not be respected. It could be desirable because the implementation brings some issues.

Back to the logic flow now, supposing the loop over all count features of job is processed. In each its iteration, we first pass through node_sets, choosing out those in which all nodes satisfy the current feature. These chosen node_sets are stored into local tmp_node_sets. Next, min_nodes and req_nodes parameters are set to the feature's count so that at least this number of nodes will be selected. Also, job.num_procs is set to the count, disabling the constraint on the total sum of required CPUs. Finally _pick_best_nodes is called to select the required number of nodes from tmp_node_sets.

If successful, we set the selected nodes in job.req_node_bitmap in order to pick them in the next iteration as well. And this step, which causes accumulated nodes to be well clustered, is also one of reasons for the problems we will discuss in a dedicated section later on.

Assuming all the iterations has been successful, we shall end up with the accumulated set of nodes, satisfying all the count features, in job.req_node_bitmap. After that, we add the original user request into this bitmap and restore the saved values of min_nodes, req_nodes, and job.num_procs with respect to the modified job.details.req_node_bitmap (e.g., min_nodes will not be lower than the number of set bits in job.req_node_bitmap).

In the end, the final _pick_best_node call is performed to include the user's requested nodes and satisfy overall constraints, such as job.num_procs. After this call, job.req_node_bitmap, min_nodes, req_nodes and job.num_procs are returned to the values they were before calling _get_req_features.

3.2.4 _pick_best_nodes

Arguments: (node_sets, select_bitmap, job, job.part, min_nodes, max_nodes, req_nodes, test_only)

At the beginning, we ask the node selection plugin if it is a consumable resource plugin by select_g_get_info_from_plugin. The answer, being negative in our case, is used together with job.shared and job.part.max_share to determine whether job can share nodes with other jobs or requires an exclusive allocation. The result is assigned to the local variable shared but also to job.shared attribute that is later utilized in the selection plugin and also in select_nodes above.

Furthermore, we perform several checks on job.req_node_bitmap to find out if the set nodes can be assigned to job. In particular, their count cannot be higher than max_nodes and all the nodes must be set in avail_node_bitmap. In addition, job.req_node_bitmap also must form a subset of share_node_bitmap if shared is true or idle_node_bitmap otherwise. If these checks are successful, we proceed to the core loop of the function.

This core is responsible for satisfying xor features in job.feature_list. Let an abstract list containing all the extracted xor features flow in your mind. Then, the loop is processed exactly once even if job has this list empty. We shall think of it as the empty xor list is also a xor feature but implicitly satisfied by all the node_sets. Thus, we can now say the loop iterates over all job's xor features. Note that xor features and count features are incompatible. If count features are used, the xor list must be empty and this loop is processed just once.

Similarly as in _get_req_features, there is an inner loop iterating over node_sets, skipping those that do not satisfy a given feature (xor feature here). A difference is that a call of an underlying selection function can be performed even during the accumulating process. In _get_req_features, the call of _pick_best_nodes can be done only after the accumulation.

In the inner loop, we first add nodes in a current node_set to total_bitmap that will, therefore, keep also unavailable nodes (e.g., down nodes). After that, contents of avail_node_bitmap, share_node_bitmap (if shared=true), and idle_node_bitmap (if shared=false) are propagated into the node_set. Finally, we add nodes left in the node_set to avail_bitmap that stores only by job usable nodes (do not confuse it with the global avail_node_bitmap). The count of bits set in avail_bitmap is stored into avail_nodes.

This process is repeated again if at least one of these conditions is satisfied: shared=true and the following node_set has the same weight as the current one, job.req_node_bitmap is not a subset of avail_bitmap, or avail_nodes is less than max(req_nodes, min_nodes). The latter two conditions should be clear. The reason for the first one is to allow allocation of more lightly loaded nodes by the selection plugin.

If all the three conditions are finally broken, we reach a call of select_g_job_test, which is the main function of the selection plugin. Its task, being further discussed in Internal Scheduling, is to pick the best nodes for job from avail_bitmap, where a definition of "the best" depends on a node selection plugin employed. The called function stores results of its processing into avail_bitmap itself, so we must create a backup in case this call fails.

If it fails, we restore avail_bitmap and continue accumulating in the same manner as before; apparently, with the second and third condition satisfied in all the following iterations. If we were successful, on the other hand, the resulting avail_bitmap is assigned to the input/output variable select_bitmap and _pick_best_nodes successfully returns.

It is possible that the selection plugin has not been invoked while accumulating; for instance, if avail_nodes was less than max(req_nodes, min_nodes) during the whole loop processing. But if req_nodes is higher than min_nodes, we might be able to pick nodes so as to satisfy the min_nodes constraint. The fact that req_nodes will be broken is not so important, since it is just a desired count of nodes. Therefore, an optional select_g_job_test call for the given xor feature is present after the accumulating loop.

If this fails or is not performed, a piece of logic responsible for revealing a reason for the selection failure follows. This reason can be one out of three:

A job is not runnable with a given node configuration (e.g., when min_nodes is higher than the number of set bits in total_bitmap)

B job is not runnable because some nodes are not available (e.g., when min_nodes is higher than the number of set bits in avail_node_bitmap)

C job is not runnable because of too high system load (i.e., job is not runnable and the cases A, B are false)

The logic is processed for each xor feature until we reach the C-case; then, it is ignored hereafter. If we reach the B-case, we cannot degrade back to the default A-case, but we can upgrade to the C-case. A case of failure is remembered through the iterations and a final error is returned in the end if we have not managed to succeed before. The respective errors to return are node_error, part_error, and busy_error.

The interesting matter here is that select_g_job_test is used also for this failure logic. But as opposed to the previous mentioned calls, we utilize it in a test mode you shall meet also in the Internal Scheduling chapter. This mode makes the selection plugin ignore all jobs in its internal tables in order to determine if job could be executed assuming the nodes provided as an input are unloaded. By a successful call of select_g_job_test on total_bitmap in the test mode, we can exclude the A-case from the options. If also calling it on the intersection of total_bitmap and avail_node_bitmap is successful, we know the C-case has happened. That is how the logic works.

3.3 Count features issues

The code support for this constraint on the resulting allocation is just provisional. It can be recognized from source code comments and some other marks that `_get_req_features`, being responsible for count features, has been added between `select_nodes` and `_pick_best_nodes` additionally, without having it planned before. Regrettably, the system behaviour has got some not very nice properties because of this. Namely, if multiple count features are used, a job might be incorrectly rejected as non-runnable, depending on an order of the `node_sets` array.

Here are steps with an impact on the behaviour:

1 In the accumulating inner loop of `_get_req_features`, we build a new `tmp_node_sets` structure. This structure contains only original `node_set` records that satisfy a given feature. The feature is given by an iteration of the outer loop passing through all job's count features.

2 We call `_pick_best_nodes` on the `tmp_node_sets` structure that keeps an order of the original `node_sets`. The `_pick_best_nodes` function prefers lower `node_set` records while selecting.

3 The selected nodes are stored into `job.req_node_bitmap`. That means, they must be present in `tmp_node_sets` for the next count feature; otherwise, the required nodes cannot be picked and the job is stated to be non-runnable.

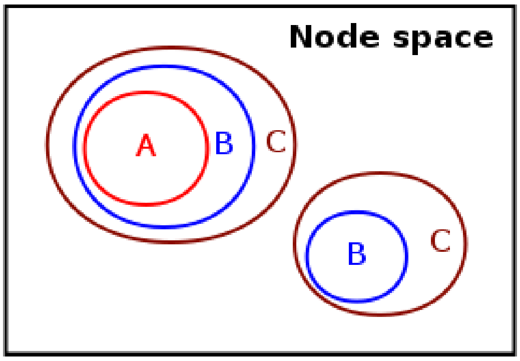

Figure 3.3: Count features: Satisfying A*3&B*5&C*7

See Figure 3.3 depicting that nodes selected to satisfy a current count feature must always form a superset of nodes that have been selected before. Although only the first and third point are needed to infer the statement, we have added the second so that we can construct a problem instance. But for that, we also need to know how the order of `node_sets` is defined:

1 `node_set` records of the same weight (the default is 1) are sorted in order of the respective entries in `slurm.conf`

2 `node_set` records of different weights are sorted in accordance to their weight from the lowest to highest.

The order is created in `read_config.c:read_slurm_conf` and then passed to `node_sets` in `_build_node_list`. Note we do not consider weight updates in the definition (e.g., by using `scontrol update`). Now, we are ready to construct an instance of the problem. We show that job rejection/acceptance might depend on the order of node configuration entries in `slurm.conf`.

```
Preliminaries 1:
   slurm.conf:
      NodeName=nymfe[71-72] Feature=A
      NodeName=nymfe[73-74] Feature=A,B
      PartitionName=debug Nodes=nymfe[71-74] Default=YES
   $ sinfo
      PARTITION AVAIL TIMELIMIT NODES STATE NODELIST
      debug*       up  infinite     4  idle nymfe[71-74]

Action 1 :
   $ srun -C "A*1&B*2" sleep 10 &

Result 1:
   srun: error: Unable to allocate resources:
      Requested node configuration is not available

Preliminaries 2:
   slurm.conf:
      NodeName=nymfe[73-74] Feature=A,B   # just the entries switched
      NodeName=nymfe[71-72] Feature=A
      PartitionName=debug Nodes=nymfe[71-74] Default=YES
   $ sinfo
      PARTITION AVAIL TIMELIMIT NODES STATE NODELIST
      debug*       up  infinite     4  idle nymfe[71-74]

Action 2:
   $ srun -C "A*1&B*2" sleep 10 &

Result 2:
   $ squeue
      JOBID PARTITION   NAME     USER ST TIME NODES NODELIST(REASON)
        216     debug  sleep xnovot19  R 0:02     2 nymfe[73-74]
```

Yet another but less important issue is their counter-intuitive meaning arising from the implementation. For instance, it can be seen that `"A*4&B*3&C*2"` requests four nodes and all of them must possess `A`, `B`, and `C` features. To conclude, count features are an area of SLURM that certainly needs to be improved.

Chapter 4

Internal Scheduling

We have already met some plugin function hooks present in the controller's job processing flow (e.g., `slurm_sched_initial_priority`). But the truth is there are many more of them, which enables modifying and augmenting SLURM's basic queueing capabilities. Some of those hooks inform of plugin relevant events in the system, passing the affected data if needed; others are used to elicit an information influencing the processing flow afterwards. These hooks have direct access to all global system resources, which brings along great power and great responsibility of a plugin programmer.

Since the slurmctld daemon is a multithreaded application, it is also necessary for a plugin itself to be thread-safe if it uses any own global data. This can be useful to keep a particular piece of world state between invocations so that it needs not to be rebuilt every time again. The internal data are kept synchronised with the controller's state thanks to the functions called upon event. However, as we will see in the case of schedule plugin, not all relevant events are passed down, which is one of things that impose additional limits on its constrained *passive* functioning.

In the passive mode, only interface functions (the hooks) can invoke a plugin to process; otherwise, it is sleeping. But then, there is also an option in spawning own threads, which will set the plugin equal to any controller's thread in processing power. This mode (or a plugin in it), which enables much greater influence on the system, will be called *active*, and, for instance, the backfill scheduler is active as it could not operate correctly otherwise. There is also the bluegene selection plugin employing the approach, but threading has been explored just in the context of schedule plugins.

The main content of this chapter is a description of interfaces to each of schedule and selection plugins. Bluegene specific and trivial functions stay left out so that there is enough space to show the essence. Please, refer to SLURM developers' section at [5] for the full but otherwise stripped lists. We also present our sample implementations to point out peculiarities of the areas. Before we start with that, there is again a brief technical introduction.

4.1 Preliminaries

Every plugin in SLURM has `init` and `fini` functions that are called when loading them into memory takes place. If a programmer wants to create/destroy a thread or read a configuration file, this the right place to do it. Loading into memory and using plugins generally is provided by a middle layer through which all calls are directed. In the previous chapter, we prefixed plugin functions with `select_g_` and `slurm_sched_`, as these are calls into the middle layer. Now, it is going to be `select_p_` and `slurm_sched_plugin_`, respectively, to work with actual function names inside plugins. But often, we will leave out the prefixes completely if it is not misleading. The same applies also for job's attributes we will be using, so you might encounter `req_node_bitmap` as well as `job.req_node_bitmap`.

Some data structures the controller handles have been already declared but for purposes of this chapter, we need to present additional ones. A plugin programmer should be aware not only of `job_list` but also of:

`part_list`	a list of all partition records in SLURM
`node_record_table`	an array of individual node records

All the three are declared in `slurmctld.h` together with definitions of structures they contain (`job_record`, `part_record`, and `node_record`, respectively). As for single job attributes, we mentioned `node_bitmap` that identifies nodes allocated to the job. However, if any node in the allocation goes down during the job processing, it is unset in `node_bitmap` even if the job allocation remains (`--no-kill` option). Another bitmap `select_job.node_bitmap` has the same purpose but with down nodes kept. This attribute and the whole `select_job` allocation structure is of great importance, because `select_p_job_test` is required to put its processing output there.

4.2 Schedule plugin

4.2.1 Interface

STATE SYNCHRONIZATION FUNCTIONS

» int slurm_sched_plugin_reconfig (void)
Arguments: none
When:
- slurmctld is being reconfigured (e.g., by `scontrol reconfigure`)

From: `read_config.c:read_slurm_conf`
Purpose: Notify of slurmctld reconfiguration. This is relevant if the plugin uses any configuration files. If any internal data are kept, we should not synchronize them here but in the `partition_change` function that is called after the process and is more general.

» void slurm_sched_plugin_partition_change (void)
Arguments: none
When:
- slurmctld has been reconfigured (e.g., `scontrol reconfigure`)
- a partition has been updated (e.g., `scontrol update`)
- a partition has been deleted (e.g., `scontrol delete`)

From: `proc_req.c:_slurm_rpc_reconfigure_controller`, `partition_mgr.c:update_part`, `partition_mgr.c:delete_partition`, `controller.c:slurm_reconfigure`
Purpose: Notify of possible configuration changes. This is the right place to synchronize any internal data.

» void slurm_sched_plugin_job_is_pending (void)
Arguments: none
When:
- `job_list` contains a job that could be executed if there were no other jobs in the system (i.e., it is waiting because of load, priority, etc.)

From: `node_scheduler.c:select_nodes`
Purpose: Notify of a job suitable to be executed. For instance, a backfill scheduler is invoked by this notification because such jobs can be backfilled. Note that this function is not called for every job in the `JOB_PENDING` state. A job might be in that state also if it has, e.g., required nodes that are unavailable.

» int slurm_sched_plugin_schedule (void)

Arguments: none

When:

- a job has completed and all its nodes has been deallocated
- a non-batch job has been submitted

From: `job_mgr.c:kill_running_job_by_node_name`, `node_mgr.c:make_node_idle`, `node_scheduler.c:deallocate_nodes`, `node_scheduler.c:re_kill_job`, `job_mgr.c:job_allocate`

Purpose: Hard to say here. Perhaps, to invoke a scheduling process of the plugin because a relevant event has occurred, or to notify of changes in `job_list`. Regrettably, it does not do its work right in neither of cases. It is not called (and no other function as well) when a pending job (i.e., `JOB_PENDING`) is cancelled or some job has been updated by `scontrol update`. Also, it omits batch job submissions. The reasons for these flaws are unknown, if any. Perhaps, it might be that this function was specifically designed for an external scheduler and is not well suited for internal scheduling.

As opposed to the `freealloc` function, this is called at the end of the deallocating process. As for the submission event, the call is performed immediately after `select_nodes` returns. It happens even if `node_error` has occurred, entailing the job rejection (i.e., `JOB_FAILED` state).

JOB SPECIFIC FUNCTIONS

» int slurm_sched_plugin_newalloc (job_record *job_ptr)

Arguments:

- input `job`: a job that has been allocated resources

When:

- `job` has been allocated resources and is going to be started

From: `node_scheduler.c:select_nodes`

Purpose: Notify of `job` having allocated resources and pass it to the plugin. Changes in any internal data might be performed upon this event.

» int slurm_sched_plugin_freealloc (job_record *job_ptr)

Arguments:

- input `job`: a job is going to be deallocated

When:

- `job` is going to be terminated and deallocated resources

From: `node_scheduler.c:deallocate_nodes`

Purpose: Notify of `job` releasing resources and pass it to the plugin. Deallocating takes time and goes successively node by node. We are called at the beginning of the process. `Job's` allocation bitmaps (`job.node_bitmap` and `job.select_job.node_bitmap`) are still set, so we can modify our internal structures according to one of them.

» u_int32_t slurm_sched_plugin_initial_priority (u_int32_t last_prio, job_record *job_ptr)

Arguments:

- input `last_prio`: the last returned initial priority that was greater than one, for a job without set `nice` attribute; or `TOP_PRIORITY` (defined in `job_mgr.c`) if we have not been invoked yet
- input `job`: a job that requires the initial priority

When:

- `job` has just been submitted or is being requeued
- a node has become available for allocating; called for all not finished jobs with `priority=1`

- in some cases, after updates of `job.nice` and `job.priority` attributes (e.g., by using `scontrol update`)

From: `job_mgr.c:_set_job_prio`

Purpose: Get the (re)initial priority for `job`. The `last_prio` argument can be used to easily implement FCFS policy by returning `last_prio-1`. This function can also return zero for all inputted jobs to hold them in the job queue.

» void slurm_sched_plugin_requeue (job_record *job_ptr, char *reason)

Arguments:

- input `job`: a batch job being requeued
- input `reason`: e.g., `"Job requeued due to failure of node nymfe72"`

When:

- `job` is being requeued (e.g., `scontrol requeue`)

From: `job_mgr.c:job_requeue`, `job_mgr.c:kill_running_job_by_node_name`

Purpose: Pass batch `job` being requeued. Running or suspended batch jobs with requeue attribute set might be requeued, which involves deallocating their nodes and returning them back to the `JOB_PENDING` state. Later, they might be again allocated nodes and restarted. This cannot be accomplished with non-batch jobs.

4.2.2 A sample schedule plugin

We attempted to build a passive schedule plugin that dynamically keeps all pending jobs in `job_list` in the held state (i.e., `job.priority=0`) except the shortest and longest one, considering `time_limits`. These two are set to a fixed non-zero priority (e.g. `job.priority=10`) to be available for allocating and starting by `schedule`. If a job in the `JOB_PENDING` state is added to or removed from the list, priorities of pending jobs need to be reset to reflect the new state and hold the specification.

The task was also to influence job placing. We demanded that generally two different SLURM partitions were dedicated to the shortest jobs and longest jobs separately. That entails a pending job might be switching between the two partitions if it still alternating between being the shortest and longest. The requested partition assignment must be sustained so that any time `schedule` is invoked, jobs are allocated nodes in the right partitions.

Implementing the combined SJF/LJF dispatching rule was seemingly straightforward. A single procedure was developed for it that passes through `job_list` skipping not pending or dependent jobs (see `job_mgr.c:job_independent`). It zeros priority of all not skipped jobs, compares `time_limits`, and remembers the shortest and longest job. After that, priorities of the two remembered jobs are raised. We placed a call of this procedure into the `initial_priority`, `requeue`, and `newalloc` functions. So, when a pending job is added into or removed from `job_list`, the procedure is called and priorities are reestablished.

The problem is that another way of removing a pending job exists—cancelling it. This is obviously not covered by `newalloc` and also not by any other plugin function. If we use `scancel` to remove the only jobs with non-zero priority, all other pending jobs will be held in the queue until a new job is submitted or allocated is requeued. In addition, the situation is not optimal even without cancelling because of rare `schedule` calls.

When the two jobs are allocated nodes, `slurm_sched_newalloc` is called for each of them at the end of `select_nodes`. That will pick other two jobs to be started. But if there is no RPC activity, `schedule` is usually not called until an internal time counter reaches 60 seconds. Consequently, resources might be kept idle while they should be allocated.

While there are some flaws concerning the dispatching rule implementation, we failed to realize the placing requirement as it is not possible.

When we raise priorities of the shortest and longest job, configured partitions are also assigned to them. More precisely, we assign partition names to their `char *partition` attributes, relying on a code block in `select_nodes` that looks up partition records in `part_list`, based on their names. If successful for a job, `part_record *part_ptr` is set to the found record pointer.

However, we missed that the block is processed only if `job.part_ptr=NULL`. Also, it is placed before the initial error checks that can exit `select_nodes` if `job.priority=0` with `held_error`. As a result, the record pointer is inferred from the partition name when a job is submitted and never after. Therefore, our name alternating has no impact and if a job has been once assigned to a SJF-partition, it cannot be reassigned to LJF-partition and vice versa. Note that in the chapter about the job processing, we abstracted from the starting look-up in the `select_nodes` description.

The algorithm works but without the dynamic placement policy. Of course, attempts to fix it were undertaken. The first idea was to null the partition pointers as we zero priorities passing through `job_list` so that the block in `select_nodes` is processed every time again. That took down the slurmctld daemon by a segmentation fault because we nulled a pointer while it was working with it in another thread. This directly showed we could not simply place the look-up block into the plugin (without the NULL check) because this would have resulted in an inconsistent system at least.

The second idea was to add a controller's locking system into the algorithm implementation in order to null the pointers safely. We included `locks.h`, locked job data at the beginning and unlocked it at the end. For change, we accomplished to deadlock the controller, since the same locking is done also in above layers that call the interface functions.

Very likely the only problem solution is unaccessible for a passive scheduler—parallelism. Indeed, if we spawn a thread from the plugin's `init` function, we might synchronize access to the critical data without the deadlock threat. Additionally, we can also call `select_nodes` on our own, instead of waiting for `schedule`, to solve the resource utilizing problem. And we might be periodically awaken to ensure no jobs are pending in the queue without a reason.

The backfill scheduler uses this approach as well to determine nodes to be selected for a job by modifying bitmaps of required and excluded nodes. After that, it calls `select_nodes` for the job to confirm the selection and fill out a job's allocation data (see the next section). Finally, it starts it on the chosen nodes by using the start procedures called also from `schedule`. The scheduler is locked at the beginning of its processing and unlocked at the end so that no racing and synchronization problems occurs. But such lock-in-thread solution has probably not a great impact on the system processing performance.

4.3 Node selection plugin

4.3.1 Interface

STATE INITIALIZATION FUNCTIONS

» int select_p_node_init (node_record *node_ptr, int node_cnt)

Arguments:

- input `node_ptr`: points to `node_record_table`
- input `node_cnt`: count of records in `node_record_table`

When:

- slurmctld has just started

- slurmctld is being reconfigured (e.g., by `scontrol reconfigure`)

From: `read_config.c:read_slurm_conf`

Purpose: Pass global node records that just have been initialized to the plugin so that it can build its own node structure. Note that the configuration reported by slurmd daemons (e.g., the actual number of CPUs) is not yet involved. We have got what is stated in `slurm.conf` or default values. Data of particular interest might be `node_ptr[index].part_pptr` (an array of pointers to node's partitions), and `node_ptr[index].part_cnt` (count of node's partitions) for `0<=index<node_cnt`.

» int select_p_block_init (List part_list)

Arguments:

- input `part_list`: reference to the global `part_list`

When:

- slurmctld has just started
- slurmctld is being reconfigured (e.g., by `kill -1 slurmctld`)

From: `read_config.c:read_slurm_conf`

Purpose: Pass global partition records that just have been initialized. Data of particular interest might be partition `node_bitmaps`.

» int select_p_job_init (List job_list)

Arguments:

- input `job_list`: reference to the global `job_list`

When:

- slurmctld has just started and loaded saved jobs
- slurmctld is being reconfigured

From: `read_config.c:read_slurm_conf`

Purpose: Pass global job records that just have been (partially) initialized. Mind that their bitmaps have not been refreshed yet. Thus, this function should not be the place to create an initial layout of jobs in an internal node structure. For this purpose, `select_p_update_nodeinfo` is the right function (see below). This SLURM's catch will probably stay no-operational in most selection plugins.

State Synchronization Functions

» int select_p_update_nodeinfo (job_record *job_ptr)

Arguments:

- input `job`: a job of which `node_bitmap` and `select_job.node_bitmap` have been updated.

When:

- `job` has been allocated resources
- slurmctld has just started and completely restored saved jobs, including `job`
- slurmctld is being reconfigured

From: `node_scheduler.c:select_nodes`, `job_mgr.c:reset_job_bitmaps`

Purpose: Pass `job` with updated node layouts so that the plugin can modify its internal structures. This is the right function for initial filling the structures with job data if we want to utilize `job's` bitmaps. It is also called after the successful allocation of resources for `job` a few steps after `select_p_job_begin`. As `select_p_job_begin` is invoked only in `select_nodes`, we can leave it no-operational because this function can cover it.

» int select_p_update_node_state (int index, uint16_t state)

Arguments:
- input `index`: points to a node record in `node_record_table`
- input `state`: new state of the node

When:
- state of the node pointed by `index` has been updated

From: `node_mgr.c:validate_nodes_via_front_end`, `node_mgr.c:_make_node_down`, `node_mgr.c:update_node`, `node_mgr.c:drain_nodes`, `node_mgr.c:validate_node_specs`

Purpose: Inform the plugin of the state change. Node states are defined in `slurm.h`.

» int select_p_reconfigure (void)

Arguments: none

When:
- slurmctld has been reconfigured
- a partition has been updated
- a partition has been deleted

From: `controller.c:slurm_reconfigure`, `partition_mgr.c:update_part`, `partition_mgr.c:delete_partition`, `proc_req.c:_slurm_rpc_reconfigure_controller`

Purpose: Notify of reconfiguring the controller or a change in partitions data. If we maintain internal data about partitions, it might be better to clear it and rebuild from a scratch here; especially, if we keep pointers to system partition records. Note that `scontrol reconfigure` will spawn `select_p_block_init` but `select_p_reconfigure` as well.

JOB SPECIFIC FUNCTIONS

» int select_p_job_test (struct job_record *job_ptr, bitstr_t *bitmap, uint32_t min_nodes, uint32_t max_nodes, uint32_t req_nodes, int mode)

Arguments:
- input/output `job`: a job for which resources are being selected. `Job's` constraints to respect are `num_procs`, `req_node_bitmap`, `contiguous`, `part.max_share`, and `shared`
- input/output `bitmap`: on input, possible candidates for allocation are set; the rest is cleared. On output, only nodes selected by plugin are left set.
- input `min_nodes`: minimum number of nodes that must be selected for `job`
- input `max_nodes`: maximum number of nodes that can be selected
- input `req_nodes`: desired number of nodes that should be selected if possible
- input `mode`: determines the operational mode; one of: `SELECT_MODE_RUN_NOW`, `SELECT_MODE_TEST_ONLY`, `SELECT_MODE_WILL_RUN`

When:
- we need to finish selecting resources for allocation to `job` (`RUN_NOW`)
- we need to determine if `job` could run on the set nodes without any load (`TEST_ONLY`)
- we need to find out if, when, and where `job` can run on the set nodes (`WILL_RUN`)

From: `node_scheduler.c:_pick_best_nodes`, `job_scheduler.c:job_start_data`, `backfill/backfill.c:_try_sched`

Purpose: Depends on the operational mode (see When), but the main purpose is to choose the best nodes that satisfy the constraints, out of those set in `bitmap`. "The best" is defined by plugin specific selection criteria. Their evaluation might involve utilizing also other `job's` attributes than only the mentioned constraints. For instance, the `select/linear` plugin utilizes `job_min_memory` not to overcommit memory of a shared node (`CR_MEMORY` mode). Obviously, the considered constraints determine node characteristics we are interested in. To continue in the example, a plugin that does not take memory into account, will need not to touch

real_memory of nodes. As for node configurations, we should respect the FastSchedule parameter and look into node.config.[attribute] instead of node.[attribute] for an arbitrary node from node_record_table, if FastSchedule>0.
Involving load criteria will require to maintain an internal node structure for a better performance. Otherwise, we would need to draw load information from node_record_table (or worse from job_list) every time we are invoked, which would be immensely inefficient. There are functions that enable such maintaining, such as job_begin. Note we will probably not use update_node_state because clearing out unavailable nodes from bitmap is one of the things the controller is responsible for.
We can also make use of the global topology information from a topology plugin. This information is stored in switch_record_table declared in controller.c. If interested, see the switch_record structure in slurmctld.h and the linear plugin for an exemplary usage.

The operational modes:
SELECT_MODE_RUN_NOW: In this mode, if selection is successful, job will be started. To support starting procedures, the plugin must fill out job.select_job allocation structure. Please, see common/select_job_res.h for the structure definition and description and the linear plugin for an example again. This mode is utilized in _pick_best_nodes when the test_only parameter is false.

SELECT_MODE_TEST_ONLY: This mode allows the controller to be user-friendly because it helps revealing a cause of a job allocation failure. We could see how in the Controller's job processing chapter. It permits to reject jobs, not runnable because of node configuration, at submit time rather than after they have spent hours queued. The essence of this mode is not considering any running or suspended jobs on the cluster. It simulates the system without load. It is used in _pick_best_nodes with test_only=true as well as test_only=false.

SELECT_MODE_WILL_RUN: A mode specifically designed for backfilling jobs. It is responsible for determining if, when, and where job can be started so that it does not collide any currently executed jobs. If it decides job can be started now, an above layer considers such job suitable for backfilling. The backfill scheduler is a good example of such above layer, but it can be also used by an external scheduler if it leaves placement of jobs on an internal selection plugin. The function in this mode should additionally fill job.start_time with the expected initiation time. "If" is determined by a return code and "where" by bitmap on output.

In each mode, job.total_procs should be set to the sum of CPUs on the selected nodes if selection was successful. In theory, the output bitmap does not need to be a subset of input bitmap, but benefits of this are highly arguable. The plugin might even break some job's constraints (e.g., allocating less nodes than min_nodes) and not only those under its direct responsibility. It will work somehow but badly; hence, a plugin programmer should be system-friendly.
To sum up, the constraints under the plugin's responsibility are: num_procs, part.max_share, contiguous, req_node_bitmap, shared, and the parameters min_nodes, max_nodes. Also other job attributes are used to get count of available CPUs on a node, such as cpus_per_task or ntasks_per_node, but only for this and no other purpose in the exemplary select/linear plugin (see the _get_avail_cpus function).

» int select_p_job_list_test (List req_list)

Arguments:
- input/output `req_list`: list of `select_will_run_t` requests (see `common/node_select.h`)

When:
- multiple requests in JOBWILLRUN query of wiki2

From: `wiki2/job_will_run.c:_will_run_test`

Purpose: Find out if, when, and where job requests in `req_list` can be satisfied. It is required that no job in `req_list` with not filled `start_time` will delay any job located lower in the list. The attributes `job.start_time`, `job.total_procs`, and `avail_bitmap` contained in a one request are the places to store the results into. This function is an extension of `select_g_job_test` in the `SELECT_JOB_WILL_RUN` mode. But it is not implemented in neither of the `select/linear` and `select/cons_res` plugins. As a result, an external scheduler should send only one request in the JOBWILLRUN query (see External Scheduling) with the plugins enabled so that `select_g_job_test` will be used.

» int select_p_job_begin (job_record *job_ptr)

Arguments:
- input `job`: a job going to be started; for which resources has been selected

When:
- the resource selection procedure for `job` in the `test_only=false` mode has been successful

From: `node_scheduler.c:select_nodes`

Purpose: Pass `job` with accordingly modified `node_bitmap` and `select_job.node_bitmap` so that we can update our internal structures. It is note-worthy that the nodes has not been allocated yet in `node_record_table` and the global bitmaps, although it should not be important as we will probably not need to look there. Still, the function `update_nodeinfo` is called after the allocation, so it is a better place for updating upon this event anyway because it covers a more general case.

» int select_p_job_fini (job_record *job_ptr)

Arguments:
- input `job`: a job going to be deallocated

When:
- `job` is going to be terminated and deallocated resources

From: `node_scheduler.c:deallocate_nodes`

Purpose: Pass `job` with still filled `node_bitmap` and `select_job.node_bitmap` so that we can deallocate the nodes in our structures first.

» int select_p_job_suspend (job_record *job_ptr)

Arguments:
- input `job`: a job going to be suspended

When:
- `job` is going to be suspended

From: `job_mgr.c:_suspend_job_nodes`

Purpose: Pass `job` for the plugin to update its structures according to (one of) the bitmaps. It might be useful to know that `job's` nodes are going to be globally deallocated in the process.

» int select_p_job_resume (job_record *job_ptr)

Arguments:
- input `job`: a job going to be resumed

When:

- `job` is going to be resumed

From: `job_mgr.c:_resume_job_nodes`

Purpose: Pass `job` for the plugin to update its structures. The job's nodes are going to be reallocated in the global bitmaps and `node_record_table`.

Get information function

» int select_p_get_info_from_plugin (enum select_data_info info, void *data)

Arguments:

- input `info`: identifies data to be returned; one of entries in `select_data_info` defined in `slurmctld.h`
- output `data`: the requested data

When:

- the controller considers a selection plugin in charge and certain data provided by it

From: `node_scheduler.c:_pick_best_nodes`, `node_mgr.c:validate_node_specs`, `job_mgr.c:reset_job_bitmaps`, `node_mgr.c:pack_all_node`, `job_mgr.c:_top_priority`

Purpose: Find out an information from the plugin that has influence on controller's processing. As an example, `info=SELECT_CR_PLUGIN` will cause returning `data=1` in case of the `cons_res` plugin. The linear plugin does not touch `data` at all but it could also return `data=0`, informing of not being a consumable resources plugin by both means.

4.3.2 A sample node selection plugin

Here, we achieved a success as opposed to the previous case, but some troubles were still encountered. And there might have been more of them if a selection scheme in the linear plugin had not been followed. In fact, we implemented a lot simplified version of it, being suitable to present basics of this problematics. But that does not mean the code is similar; the core constructions significantly differ to efficiently support a skeleton of `select/linear` functionality.

The objective is to select nodes that have least number of jobs allocated to them (i.e., Min Load First), counting only running jobs (not suspended). The linear plugin also tries to minimize the number of contiguous sets formed by selected nodes (e.g., `nymfe[1-2]`, `nymfe4` are two contiguous sets). Our plugin puts no effort into this and, in fact, it ignores also the contiguous user request. The support for memory as a consumable resource is omitted as well, which can result in frequent memory overcommitting in case of shared jobs and high system load. The linear plugin in `CR_MEMORY` mode would prevent from it by rejecting jobs in such situations. Finally, we also do not make use of the topology plugin that allows to minimize the number of net switches. As a result, our plugin might often select nodes scattered all over the cluster.

Even with the limitations, the plugin is quite complex and there are three subareas of its functioning, being separate as different interface functions are used in each. These subareas are: initialization of internal data structures, maintaining (refreshing) the internal data, and the core selection procedure. You can see the data definition in Figure 4.1.

Data initialization

It happens just after the controller has started and then upon every reconfiguration RPC. The `node_sel` records are constructed in `node_init`, but job entries are filled later in `update_nodeinfo`. As we stated, `job_init` is not suitable for this purpose as job bitmaps have not been set yet. We do

```
struct part_sel_rec {
   struct part_record *sys_part;    /* pointer to the record in part_list */
   uint16_t njobs;         /* jobs running on the node for this partition */
   struct part_sel_rec *next;     /* pointer to next part_sel_rec or NULL */
};

struct node_sel_rec {
   uint16_t total_jobs;            /* number of jobs running on this node */
   uint16_t exclusive;        /* nonzero if dedicated to an exclusive job */
   struct part_sel_rec *parts; /* pointer to linked list of part_sel_recs */
};

struct node_sel_rec *node_sel = NULL;              /* plugin node records */
```

Figure 4.1: The plugin's knowledge of the world

not use `block_init` at all because it immediately follows the `node_init` call in `read_slurm_conf`. Making it in one procedure is more efficient (although less pure).

Note we do not keep node states although we could by employing `update_node_state`. It is not necessary because the `job_test` function gets always only available nodes on the input. And if configuration data are needed, such as number of CPUs, we can read it out from `node_record_table` easily, since node positions in `node_sel` and `node_record_table` correspond.

It is interesting that `select/linear` does not use the init procedures at all; instead, it has one own initialization procedure for node, partition, and also job entries. In each function where it uses its node structure, a test whether it is NULL is performed at the beginning. If true, the initialization procedure is called.

Data maintaining

The `node_sel` structure is updated upon allocating/resuming and deallocating/suspending a job. The `total_jobs` and `njobs` counters are incremented or decremented accordingly. We also maintain the exclusive flag and you might wonder why as this is considered in `_pick_best_nodes` by applying the idle or share node bitmap. There are two reasons and the first one is that the bitmaps are applied only if `sched/gang` is disabled; otherwise input `bitmap` of `job_test` generally contains also taken nodes. It is so because the interpretation of exclusivity differs with `sched/gang`. Thanks to the flag, compatibility with the gang scheduler might be later added if desired. But more important is that we need it to support the `WILL_RUN` mode. We shall explain why in the Selection part.

The counters are incremented in `job_resume` but also in `job_begin` because at the time the plugin was developed, our knowledge of the system did not include the fact that `job_begin` might be eventually left no-operational. We keep a global pointer to the last job allocated not to increment the counters twice upon a job allocation (in `update_nodeinfo` too).

Admittedly, there is another matter missed concerning the `reconfigure` function. We believed the init functions filled in for this, but they are not called when an update or delete partition RPC is issued. In our plugin, it is not a problem because we always access only the `sys_part` pointers, not their contents. Still, this is a dangerous flaw and the delete RPC could cause serious problems if we modified the code to utilize `sys_part` entries in a certain affected way. The best thing to do in reconfigure is probably to dump node_sel and build it again; select/linear does it too with its structure.

SELECTION

The key idea of the selection process implemented in the job_test core and auxiliary static functions is to build an ordered list of nodes. The ordering is given by the following rules:

1 Required nodes are ahead of non-required.

2 Nodes with less jobs running on them are ahead of nodes with more jobs on them.

3 Rule 1 is precedential.

In the TEST_ONLY mode, we add all nodes set in bitmap into list. On the other hand, the WILL_RUN and RUN_NOW modes require considering the layout of running jobs not to surpass max_share limits of partitions.

When list is built, we take the first N nodes where N=min(length(list),req_nodes). To recall, req_nodes is a desired number of nodes that should be selected. If N is less than min_nodes or the sum of CPUs on the nodes is less than job.num_procs, N+1 nodes will be taken from list in the next iteration and N is incremented. This is repeated until the success is reached or N becomes higher than min(length(list),max_nodes). If we succeed, only the first N nodes in list are left set in bitmap; others are cleared and the success is returned. If mode=WILL_RUN, job.start_time is set to current_time before exit. If mode=RUN_NOW, job.select_job and job.total_procs are set. And if the limit was surpassed and the mode is different from WILL_RUN, an error is returned.

If we are in the WILL_RUN mode, the algorithm continues despite surpassing the limit. We sort all running jobs in the system with respect to their end_times in ascending order and simulate terminating one after another in a loop. In each its iteration, we add the released nodes set in bitmap into list if they have not been added there before. Then we again execute the selection procedure described in the previous paragraph but with current_time shifted to end_time of the job of iteration. We proceed until the success is reached or there are no more jobs that could be assumed to be terminated, in which case an error is returned.

Note the algorithm above does not mention that all the required nodes must be picked. It does not need to because there is a precondition at the beginning of job_test. It says that the number of required nodes must be at most min_nodes. Also, the other reason for exclusive flag in node_sel can be seen now. When job_test is called in the WILL_RUN mode, bitmap is not cleared from allocated (shared=0) or just exclusively allocated nodes (shared=1) so that the simulating procedure can release them. But if the global bitmaps are not applied, we need to respect exclusivity of jobs on our own in the WILL_RUN procedure.

Chapter 5

External Scheduling

External scheduling means that responsibility for scheduling decisions is handed over to a standalone software package (e.g., The Moab scheduler), mostly running on a different machine than slurmctld. Such package can provide more extensive scheduling features than SLURM does; thus, it might be advantageous to let them cooperate for better cluster performance. It is also a step towards higher flexibility as various external schedulers with different features could be used to mastermind SLURM. Regrettably, there do not seem to exist any standard protocols for such coworking. As a result, not every scheduler is compatible with SLURM.

If we want to build a compatible scheduler outside SLURM to instruct it when and where to run jobs, we can choose out of three ways of doing it. First, there is the API used also by SLURM commands; second, we can use one of gateways into the system created specifically for The Maui and Moab schedulers—wiki and wiki2 respectively. Finally, linking it upon both these entry points is possible and might be advantageous.

In this chapter, we shall concentrate especially on the wiki2 interface as it is probably the most promising option. The API is very well documented in the manual pages and the wiki interface is the same in principle as wiki2 but offers less functionality. These are reasons to leave these options unanalysed in this thesis.

5.1 The wiki2 interface

From controller's point of view, the wiki2 interface is the same as any schedule plugin we have met in previous chapters, because it uses exactly the same plugin mechanism and the controller calls exactly the same functions to invoke a scheduling process. The difference, visible from a larger scope, is that these functions provide almost no scheduling logic. Instead, they are mainly used so that the scheduler can take appropriate actions, which might also include using wiki2 to acquire additional information about a system state and inform the controller of its scheduling decisions. This bidirectional communication is implemented via Internet stream sockets that enable transmitting data in order and on a reliable basis.

The controller certainly cannot just stop execution and wait for optional messages (e.g., a query about all jobs in the system) from the scheduler. Therefore, a thread dedicated to receive and process messages asynchronously is spawned in the plugin initialization function. The similar mechanism should be used on the scheduler's side to receive events if we demand continuous processing. The messages sent to slurmctld from the scheduler are of two types: commands demanding a system change and queries to acquire more information about the world.

When a message from the scheduler is received, the processing thread calls an appropriate handler in accordance to the message type (e.g., GETJOBS). This handler then uses the central lock system for enforcing limits on access to an entity it will read data from or write data into. In the end, appropriate actions are performed and a reply is sent back the scheduler. Such reply contains a status code and comment in case the controller replies to a command, or in detail structured, requested data if the message was a query. We will talk about the possible message types in much more details later on.

5.1.1 Scheduling takeover

The controller in each its scheduling cycle attempts to launch jobs with the highest priority for each partition as described in Chapter 3. For the scheduler to have a full control over the scheduling process, there exists a way of preventing the controller from launching jobs spontaneously, which we call *hold strategy*. By setting the job's priority to zero, we instruct the controller to leave this job queued and bypass it during the scheduling cycle. If we set the initial priority of all jobs to zero using `slurm_sched_initial_priority`, none of them is going to be allocated until their priority is raised to a non-zero value.

The external scheduler can make use of it by sending STARTJOB command for a job intended to start execution at the moment, with respect to its internal schedule. After the wiki2 message thread receives the command, a handler parses its content consisting of the job id, list of assigned nodes for it, and an optional comment field which is not important now. Next, the job's required nodes are set to the assigned nodes and all the others are excluded, using the job's bitmaps; that assures the job will be allocated the nodes chosen for it by the scheduler. Finally, the job's priority is raised to a non-zero value and the `schedule` function is called from the handler to launch the job on the specified resources.

The allocating process in slurmctld after calling `schedule` is mostly the same as if scheduling would be performed internally, not externally. That is, we perform all the node selection steps down to the invocations of `select_g_job_test` as described in Chapter 3. Only if we do use a consumable resource plugin for resource selection; that means a caller of the function `select_g_get_info_from_plugin` is informed the plugin is of that type; satisfying count features is disabled.

Hence, a resource selection plugin is still active but has no other choice than to select all the nodes picked by the external scheduler because of the respectively modified required/excluded nodes of the job. Another consequence is that validity of every STARTJOB request is checked and if the chosen nodes does not hold all the job's constraints, the request is rejected. Along with constraints such as the maximum of nodes, also features must be respected and that brings a problem. Because as we have showed, satisfying multiple count features is not very well solved in SLURM, so it could be desirable to replace the implementation of the controller's with a better one provided by an external scheduler. Regrettably, this is generally not possible as it strangely depends on the type of selection plugin in use. As we said, only with a consumable resource plugin, the controller's accumulation of nodes to satisfy count features is left out in case wiki2 is being used. As a result, only with this plugin type we could modify the behaviour successfully.

Another related problem is caused by the fact that not all job constraints (e.g., the minimum of CPUs per node) are reported through wiki2. Obviously, the scheduler might try to start a job by using STARTJOB without success if its internal data about the world state are not up to date (i.e., a node has just gone down). But this issue implies, an unsuccessful attempt to start a job might happen even if all the scheduler's data are correct; just because they are not complete. Such event should happen rarely, since unreported attributes are few and not critical, but if occurring, it might be a trouble to determine what to do with a job that fails executing for that reason. We propose to use a JOBWILLRUN query (see below) in such case, setting available nodes in the message to all nodes in the system. The reply will contain either an error report, in which case we know the job is not runnable at all, or the nodes on which the job can be executed. Then, we can try to use these nodes at a desirable time when, again, our data admit it, and the request should succeed this time. Admittedly, this solution is a kludge, so we draft yet another neater way of untangling the problem—implementing an interface for submitting jobs in the scheduler itself. Such approach allows us to have full information about jobs and their constraints specifically and it also brings other advantages. Particularly, the information cannot be inconsistent, and we can add job constraints to respect without any changes in SLURM itself. Moab with its command for job submission is an instance of this approach (see [10]).

The last thing to be said for now is that the list of assigned nodes (`TASKLIST`) in the STARTJOB command might be empty, which will pass a decision about the job placement back on the controller. By using it, we can follow the intended SLURM's scheduling scheme just with the internal scheduler plugin switched for an external scheduler choosing jobs to start. Efficiency is decreased, but thanks to the clearer wiki2 interface, we are offered a simpler implementation.

5.1.2 Wiki2 setup

To enable wiki2, we need to set `SchedulerType=sched/wiki2` in `slurm.conf` and, in the same directory, make a few settings in `wiki.conf`, which is a configuration file for this interface. The following keywords are recognized:

AuthKey defines a DES encryption key that is used to sign messages sent between SLURM and the scheduler. More precisely, this key is used to calculate a checksum value of a message, sent as a part of it unless `AuthKey=""`. In that case, the checksum is not a mandatory field and the external scheduler can leave it out since it is ignored anyway.

EPort is an event notification port in the scheduler. For instance, when a job is submitted to or terminates in SLURM, a message is sent to the port to inform about it.

EHost is the event notification host. This identifies the machine on which the scheduler executes and where the events should be sent. By default `EHost` will be identical in value to the `ControlAddr` configured in `slurm.conf`.

EHostBackup is the event notification backup host for the scheduler. It names the computer on which a backup scheduler executes. By default `EHostBackup` will be identical in value to the `BackupAddr` configured in `slurm.conf`.

ExcludePartitions is a comma delimited list of partitions whose jobs are to be scheduled directly by wiki2 rather than the scheduler. For that purpose, a simple FCFS policy is implemented in the `initial_priority` function. When such job is submitted, its initial priority is set to a non-zero value that is lower by one than an initial priority for a previous job. Note that the nice attribute of jobs is not honored in the excluded partitions. Because these jobs are not subject to the hold strategy, they might be immediately allocated nodes. This can be used to provide a faster system response; nevertheless, if the excluded partitions overlap with the ones under the scheduler's control, this can potentially collide the scheduler's decisions. All jobs in the excluded partitions are still reported in replies to the scheduler's GETJOBS query unless the following keyword is used.

HidePartitions identifies partitions whose jobs are not to be reported to the external scheduler. Any partitions listed here must also be listed in `ExcludePartitions`, which means they are absolutely out of the scheduler's control and sight. If more than one partition is to have its jobs hidden, use a comma separator between their names.

HostFormat controls the format of node lists built by wiki2 and reported to the scheduler in GETJOBS and GETNODES messages. Possible values, which can be later changed by INITIALIZE command, are 0, 1, 2.

JobAggregationTime is used to avoid notifying the scheduler of large numbers of events occurring about the same time. If an event occurs within this number of seconds since the scheduler has been last notified of an event, another notification is not sent. This should be an integer number of seconds. The default value is 10 seconds.

JobPriority is used to enable/disable the hold strategy by values `hold/run` respectively. If the hold strategy, being the default, is disabled, the priorities of incoming jobs are set to non-zero values with respect to FCFS policy. Setting this to `run` is almost equivalent to listing all partitions in the system in `ExcludePartitions`, except the nice attribute is honored this time. The documentation at [5] says that the scheduler can modify priorities of jobs and re-order the job queue, but there is no support for this at least up to slurm-2.0.7. The only thing the scheduler can do is to start a job by STARTJOB command, which raises the priority of the job to 100,000,000, so the job will be probably allocated first. Since this offers just a very weak control of the queue, we do not recommend using this option.

5.1.3 Wiki2 messages

In the following section, we list and comment possible types of messages that can be sent between wiki2 and an external scheduler. We have also developed the full communication protocol, but it is not included here because of its technical essence. You can find it on the attached CD-ROM and an actual version at [7].

Just to mention basic aspects of that protocol, data in every message (except events) must be encapsulated in message headers. This header includes the size of the message, checksum used for check and also security reasons, time stamp denoting the time when the message was sent, an authorization field that identifies a system user whom the message is sent on behalf, and finally a data part that differentiates message types. A description of possible contents of the data part follows; divided into queries, commands, and events as introduced earlier. Note that even the commands are replied, but the command responses are not so comprehensive as query responses. They contain only a status code that indicates success or identifies a type of error, and a simple string description of what happened.

Queries

GETJOBS is used to get information about jobs in the system. One could think only pending jobs are included in the reply as only these are subject to the scheduling process, but that could work only in a model with known job durations. In the real systems where nodes fails and durations are highly unpredictable, we do not know when a job we have launched will actually terminate; as a result, we need to be informed when it happens. And because the wiki2 events are not at all sufficient for this, the GETJOBS reply must include at least all running jobs in addition. In fact, it includes jobs in all states for user reports and accounting purposes with the exception of jobs from hidden partitions. This is feasible because slurmctld leaves even failed jobs in its records for some time before it purges them. Note that not all job attributes are reported, which can entail problems as discussed earlier. Examples of those involve the minimum number of CPUs per node and features if they include an operator different from AND.

GETNODES queries for nodes in the system and their configuration. There are, again, unreported attributes, such as the number of sockets or cores on a node, which limits the resource granularity of an external scheduling process. Nevertheless, the number of CPUs (the lowest level logical processors) is included for each node in the reply so that the CPU constraint of a job can be satisfied.

JOBWILLRUN is especially useful for backfill scheduling as it employs the `WILL_RUN` mode of `select_g_job_test` (see chapter Internal Scheduling), specifically designed for this purpose. Moab uses this query for the exactly same purpose, which speaks for itself. But due to using placement algorithms of a node selection plugin when answering the query, we should hand over all placement decisions to the plugin to enable the backfilling with JOBWILLRUN. Otherwise, we could end up mixing two different placement strategies. Handing over is achieved by empty `TASKLIST` in STARTJOB.

COMMANDS

CANCELJOB cleans a specified job from the system. In case of a running job, it forces all its job steps to terminate immediately. Note that even cancelled jobs stay for some time in the controller's job queue before they are really purged, so they might be reported in a GETJOBS response. This command might be useful for enforcing hard limits on resource usage.

JOBMODIFY modifies specified job attributes. Only a limited number of them can be changed; yet, it can be useful if the controller is responsible for placing jobs and we want to impose a certain constraint on it, e.g., the minimum of nodes that can be allocated.

NOTIFYJOB notifies a user that submitted a job using srun (only) with a certain message. This message, which appears on the user's console, might contain, for instance, the expected initiation time.

STARTJOB attempts to start a job on specified resources. The resources are specified using the `TASKLIST` field that might be empty. If so, the selection process is performed by the controller. No matter if the scheduler works with a queue or complete schedule, it will use this command for starting a certain job when its time comes with respect to the schedule. Regrettably, we cannot tell the controller to start some job in future by any means. It is important to respect that all nodes in `TASKLIST` must be in the same partition; otherwise, the request will fail.

INITIALIZE redefines a port on which events are sent and `HostFormat` set in `wiki.conf`.

REQUEUEJOB requeues a running batch job (only), which means all the job steps are terminated and the job is returned into the job queue as it would have just been submitted. It will be then eligible for execution again. SLURM has a checkpointing system that allows to continue in the work that has been done. This system together with the capability of returning a job into the queue is useful for job preemption, when a job releases memory and computing power for another job to make use of it.

SUSPENDJOB is used to release computing power employed by a job to enable softer form of job preemption. "Softer" because tasks of the job remain in memory, which might negatively influence performance of the higher priority job if swapping occurs. In addition, suspended jobs cannot change assigned resources as requeued jobs. This command is applicable also to non-batch jobs.

RESUMEJOB resumes a suspended (preempted) job which might involve loading from secondary storage. The job then continues on nodes it had been executing before it was suspended.

SIGNALJOB signals the job, sending the specified signal to all tasks of the job. Many signals but SIGKILL are supported. For killing a job the CANCELJOB should be used. SIGNALJOB command is particularly helpful if the scheduler implements its own interface for submitting jobs, in which case, forwarding signals spawned by a user might be desirable.

EVENTS

Events are simple messages consisting only of an event code, without the header, sent by wiki2 when a system change relevant to scheduling occurs. These changes that spawn an event are of two types: a change in job data and change in partition data. The event codes are also just two (1234 and 1235, respectively), which entails the scheduler is informed only of the type of change. Any additional information, such as id of the job to which the change is related, is not provided. For this reason, we are made to query the controller in a very general manner (i.e., get all jobs in the system). Not every change in job or partition data will generate an event. We extracted specific system events spawning event messages from controller's source codes:

1234 event is generated when:

A non-batch job has been submitted

B any job has been submitted

C job has been completed

D job has been requeued

Obviously, the system event A happens always when B. They are introduced as two different entries because event messages (1234) for them are not spawned by the same function in the wiki2 plugin. The source for A and C is `slurm_sched_schedule`; on the other hand, B and D are related to `slurm_sched_initial_priority`. The fact we are informed twice about a submission of a non-batch job is a redundancy that can cause processing overhead in the scheduler if `JobAggregationTime` is set to zero. The appropriate action to take after receiving this event is to update all scheduler's data about jobs.

1235 event is generated when:

A slurmctld has been started

B slurmctld has been reconfigured (i.e., `scontrol reconfigure`)

C partition data has been modified through the SLURM API (e.g., by using `scontrol update`)

Generally, the case B does not mean a change in partition or node data has happened; yet, the event is generated. The functions `slurm_sched_reconfig` and `slurm_sched_partition_change` are responsible for sending 1235 events. The right action to take here is to update all scheduler's data about nodes and jobs. The jobs as well because possibly modified partition node limits (`MaxNodes` and `MinNodes`) are projected into job node limits that are reported in a GETJOBS response.

As we can see, the scheduler is not notified of all systems changes that can be important for its processing. For instance, no event is generated when a node goes down, which means it is no longer available for allocating. Also, cancelling a pending job does not produce any message, so the scheduler might be operating with this job for nothing.

These (and other) omissions, possibly non-zero `JobAggregationTime`, and a communication delay are the reasons for the scheduler cannot be purely event-based. It is not sufficient to update its internal data only upon incoming events. As an example, they should be also updated after a job start error or certain time without any received event.

5.1.4 A sample external scheduler

As for internal scheduling, we have built a sample scheduler that illustrates essential elements of scheduling externally by utilizing the wiki2 interface. The scheduler is queue-based and implements two simple policies: SJF and LJF (the Shortest and Longest Job First). A placing policy is also implemented inside the scheduler, so the responsibility is not passed to the controller. Yet, the algorithm for it is highly unsophisticated, since idle nodes are assigned to jobs based on the alphabetic order, supposing picked nodes will satisfy constraints of jobs. If they do not, the scheduler may try to start a job forever as it always fails for breaking some of user's requests, while other jobs are perpetually denied to be executed. The minimum number of nodes is the only request being considered, and in fact, the scheduler always assigns exactly this number of nodes, allocated exclusively to one job (i.e., no sharing).

The realized dispatching rules are applied in the round-robin fashion going from one to another, leaving out those that failed to start a job when applied. The algorithm, designed to be easily extendible with other arbitrary dispatching rules, uses only a subset of nodes in the system for a certain dispatching rule. In other words, we are limited on resources we can choose, e.g., for the shortest job in the queue. This can be beneficial if we pursue different objectives across nodes in the cluster. Nevertheless, the possibly overlapping subsets, being specified for each dispatching rule in a configuration file, should include nodes from one partition only. In the other case, the scheduler will end up trying to start a job on nodes from multiple partitions, which will fail. It is so because a job might be executed in one partition at most, and the wiki2 starting procedure sets job's required nodes to the chosen ones.

In fact, our scheduler is also linked against SLURM libraries that provide the API functions. Although we use only a small piece of the functionality, allowing us to determine if a job has been successfully started, it is sufficient to briefly demonstrate basics of the API usage. In reality, we would rather look into the STARTJOB response that contains a number code indicating success or error. Since we have not chosen this way, the response is dropped, producing an error in controller's logs.

The last property to mention before introducing a pseudocode, depicting the main logic flow, is the scheduler being event-based. Yet, it is not a pure event-based system because, as we stated, events are not always sent when a relevant change in the controller's state occur. Therefore, we put a backup mechanism into it, which checks the world state on its own if nothing has happened for a certain time.

A sample external scheduler

```
DEFINITIONS:
   NODE_LIST: LIST(NODE) always kept in the alphabetic order
   RULE: STRING   # denoting a dispatching rule
```

```
STRUCTURES:
   NODE:
      STRING nodeName
   JOB:
      INT jobId
      INT timeLimit   # interpreted as an expected processing time
      INT minNodes   # also count of nodes we allocate
   CONFIG:
      LIST(RULE, SET(NODE)) allNodes
      COMMSETUP commSetup
GLOBALS:
   LIST(JOB) jobQueue   # for PENDING jobs only
   LIST(RULE, NODE_LIST) idleNodes   # for IDLE nodes only
   CONFIG globalCfg

func INT PICK_NODES(IN: RULE schedRule,
      IN: JOB job, OUT: LIST(NODES) pickedNodes):
   if job.minNodes > globalCfg.allNodes[schedRule].size:
      return -1   # job never runnable
   if job.minNodes > idleNodes[schedRule].size:
      return 1   # job not now runnable
   pickedNodes <- first job.minNodes nodes from idleNodes[schedRule]
   return 0   # job now runnable

func BOOL START_JOB(IN: RULE schedRule, IN: JOB job):
   LIST(NODES) pickedNodes   # pickedNodes is filled below
   INT result <- PICK_NODES(schedRule, job, pickedNodes)
   if result == 0:
      start job on pickedNodes using STARTJOB command
      if job has started:   # use the API to evaluate the condition
         remove pickedNodes from idleNodes
         remove job from jobQueue
         return True   # job was removed from jobQueue
   else if result == -1:
      remove job from jobQueue
      return True   # job was removed from jobQueue
   # result == 1 at this point
   return False

func BOOL SJF_SCHED():
   JOB job <- the shortest job from jobQueue
   return START_JOB("sjf", job)

func BOOL LJF_SCHED():
   JOB job <- the longest job from jobQueue
   return START_JOB("ljf", job)

proc SCHEDULE():
   BOOL sjfCont <- True
   BOOL ljfCont <- True
   while sjfCont OR ljfCont:
      if sjfCont == True: sjfCont <- SJF_SCHED()
      if ljfCont == True: ljfCont <- LJF_SCHED()
```

```
57
58  proc UPDATE_DATA():
59     clear idleNodes
60     clear jobQueue
61     send GETNODES query
62     idleNodes <- data from GETNODES response
63     send GETJOBS query
64     jobQueue <- data from GETJOBS response
65
66  proc MAIN():
67     globalCfg <- data from ./config
68     send INITIALIZE cmd
69     UPDATE_DATA()   # initialize our knowledge of the world
70     SCHEDULE()
71     while True:
72        block until event comes or timeout
73        UPDATE_DATA()
74        SCHEDULE()
```

The real implementation is coded in C++ programming language that provides convenient data structures for keeping the world state (std::map and std::set were used). As you can see at lines 59 and 60, we clear these structures every time before we update them. Obviously, it is very inefficient to build the world state from a scratch every time an event is received, but we are of the opinion that no better approach exist. `JobAggregationTime` should be non-zero to at least partially overcome this further consequence of a low informational value of the events. Also note we always update nodes as well as jobs after receiving any event type. It is so for the code to be simpler. To be more effective, we should update only jobs after receiving 1234 event as stated in the Wiki2 messages section. Below, you can find a sample configuration file for easier understanding to the scheduler's logic.

A sample configuration file

```
ljf nymfe70 nymfe71 nymfe72
sjf nymfe73 nymfe74
Wiki2Host nymfe74
Wiki2Port 7321
EventPort 15017
```

Chapter 6

Summary

Data that have been gathered in the previous chapters will be now used to compare the scheduling approaches discussed. For this purpose, we have formulated several criteria influencing a scheduler implementation and processing. The criteria go as follows:

A **Continuous processing**
Whether a scheduler owns at least one processing thread.

B **Complete schedule**
Whether united decisions about time and place can be made and realized.

C **Data consistency**
Whether internal job and node data might become inconsistent with SLURM's.

D **Data completeness**
Whether complete information about SLURM's jobs and nodes is available.

E **Programming overhead**
What programming labour is added to enable cooperation with SLURM.

F **Processing overhead**
What processing is added by cooperation with SLURM.

G **The same machine**
Whether a scheduler must run on the same machine as SLURM.

H **Multiple clusters**
Whether a scheduler is able to operate multiple clusters.

According to these criteria, we will be comparing three types of schedulers: passive internal, active internal, and external. The reason for the two schedulers of the internal type is that there is a large gap between them in available capabilities. The schedulers' descriptions follow.

Our active scheduler processes in a thread spawned from the schedule plugin, and all the time&place (scheduling) decisions are performed upon a complete schedule in there. It uses the hold strategy for incoming jobs, and the time decisions are realized by initiating the controller's procedures for job starting at a desired time. The place is determined before it by locking global job data (might not be necessary) and filling out required information for the procedures (e.g., `job.select_job`, `job.nodes`). The controller's selecting operations and the selection plugin are completely bypassed. This probably the most capable internal mode has not been tested but should be operational according to our knowledge. It is based on the backfill scheduler that is active as well but utilizes `select_g_job_test` in the `WILL_RUN` mode and also `select_nodes` after that.

However, it is not possible to keep scheduler's job and node data, which are critical for correct decisions, in absolute consistency with SLURM's because of important unreported system events. If a user updates any job by `scontrol update` or cancels a job in the `JOB_PENDING` state by `scancel` , no interface function of the schedule plugin is called. Not even node state changes are reported, being a source of additional inconsistency. As a result, the scheduler must periodically and often check validity of its data against the SLURM's global structures. Another solution could be to keep no own data and directly access the SLURM's structures, read-locking them. Viability of this approach depends on the access rate that subsequently depends on a specific scheduler's algorithms.

The passive scheduler follows the default controller's queueing scheme—its time&place decisions are split between non-cooperative schedule and selection plugin. The time decisions concern only execution order, and they are realized by setting job priorities according to an applied queueing policy. No internal job data are kept in the schedule plugin because, being invoked relatively seldom, it can directly access `job_list`. The place decisions are performed upon an internal node structure that keeps the data in a form suitable for an applied placing policy. However, the selection plugin has got only a partial control over the process because its decisions are utilized in the controller's weight-driven algorithm that iteratively extends a set of nodes available for selection.

In addition, this procedure is repeated for every xor feature until selecting is successful, always only with nodes that satisfy a given feature. This does not limit the scheduler's node selecting scope (if xor features are to be respected), but we could be able to find more efficient way of satisfying them in the selection plugin. On the other hand, if count features are a case, the place decisions are composed in `_get_req_features`, and this significantly weakens the control. Excluding on per-node constraints in `_build_node_list` is for the sake of the scheduler because without it, the same procedure would be very probably done in the selection plugin before any other processing.

Finally, we consider an external scheduler upon the wiki2 interface, acquiring the job and node data from the wiki2 queries. Because the job events are of very low informational value, the scheduler needs to frequently ask SLURM about a current world state if the cluster is highly utilized. While parsing the responses, of which size is linearly dependent on the `job_list` length, the acquired data for each job are compared to scheduler's.

Supposing the query is made upon every received job event, `JobAggregationTime` should probably be non-zero, or the scheduler might become overwhelmed by need for processing the incoming messages. On the other hand, increasing `JobAggregationTime` also means causing inconsistencies between scheduler's and SLURM's job data. However, even if `JobAggregationTime` was zero, message processing fast, and communication delay minimal, job data would become invalid at the moment any job is updated or a pending job cancelled, which is not reported as well as any node state changes. It cannot be because even wiki2 as a schedule plugin is not informed of such system events. Finally, the scheduler does not receive complete information about SLURM jobs as several per-node constraints (e.g., `job_min_procs`) and features (if they contain an operator different from '&') are omitted.

The time and also place decisions are realized by sending the STARTJOB commands in accordance to an inner complete schedule. The starting procedure in wiki2 modifies job's required and excluded nodes accordingly and calls `schedule` for job launching, which involves all the selection steps described in Chapter 3. As we could see, there are several reasons for possibly invalid data of the scheduler, which raise invalid STARTJOB commands. Very likely, the `schedule` call is mainly there to ensure that STARTJOB requests are valid, but this approach has also flaws.

Firstly, in large clusters the selecting operations might be slow even with the predetermined result because node bitmaps, being still frequently used in `_pick_best_nodes`, contain one bit for every node in the cluster. Moreover, the linear plugin, which is likely a common companion of wiki2 over the world, performs a lot of processing as well. These are just informally formulated observations, but they will prove to be sufficient for us. The important thing to realize is that the selecting steps would be unnecessary if the scheduler had an access to the correct and complete data. Then the `schedule` call could be replaced for the job starting procedures used by the same manner as in our active scheduler. Additionally, the `schedule` call employs `_get_req_features` and the count features satisfaction if `select/linear` is in charge. That entails the scheduler should respect the problematic implementation. The following evaluation table brings together the considered properties.

	Passive internal	**Active internal**	**External**
Complete schedule	No	Yes	Yes
Continuous processing	No. Invoked by functions.	Yes	Yes
Data consistency	Yes	No. Need for periodical checking.	No. Serious inconsistencies.
Data completeness	Yes	Yes	No. Missing job data.
Programming overhead	Yes. Familiarity with slurmctld.	Yes. Familiarity with slurmctld.	No. Just message processing.
Processing overhead	Yes. The feature and weight algorithm.	Yes. World state checks.	Yes. Messages and the `schedule` call.
The same machine	Yes	Yes	No
Multiple clusters	No. SLURM is cluster centric.	No	Yes

On these results, we can now propose probably the best approach. The programming overhead when developing an internal scheduler is very significant and unpleasant. Furthermore, the scheduler would have to run on the same machine as the controller, which can negatively influence its performance, and the way to schedule on multiple clusters would be closed. But there is also a strong argument against the external approach—serious data inconsistency.

However, a solution exists and was already addressed in Chapter 5. If a job submitting interface was developed for the external scheduler and SLURM was not used for that purpose, the job inconsistencies would be immediately eliminated. It is even feasible to prevent from using SLURM's allocation commands by setting `RootOnly=YES` for every partition in `slurm.conf` and running the scheduler under user root. By this way, we can achieve of full job control. Source codes for the allocating interface can be extracted from one of the SLURM's commands (`sbatch` likely) as they are under the GNU/GPL licence.

But we still need to deliver our jobs to SLURM somehow so that it can actually launch them. For that purpose we can include `slurm.h` and link against the slurm library (`libslurm.la` or `libslurm.so`) to use `_slurm_rpc_submit_batch_job`—the same API call that `sbatch` utilizes. Another possibility is to directly create the respective RPC message (`REQUEST_SUBMIT_BATCH_JOB`) with the well-structured data and send it to the controller.

The scheduler can also easily manage its own internal partitions if desirable. We just modify `slurm.conf` so that it contains only one partition covering all nodes in the cluster. Obviously, this can be also used to just remove any partitions from consideration. Next, the node inconsistency can be solved by creating a selection plugin that informs the scheduler of node state changes, being otherwise no-operational. The selection plugin is informed of that by the interface function `select_p_update_node_state`; hence, it is easily realizable. Not even Moab is provided with such functionality.

Now, we possess a scheduler with valid data, so the `schedule` call in wiki2 is just bothering. Thus, the immediate starting method presented in our active internal scheduler can be used to avoid the unnecessary processing without any side-effects. Note what remained of SLURM—a resource manager. If the solution will be applied, our future work can be creating the selection plugin for sending node events, developing the modification of wiki2 that starts jobs immediately, and finally extracting and optionally modifying the allocating interface.

Bibliography

[1] Peter Brucker. *Scheduling Algorithms*. Springer, second edition, 1998.

[2] Ian Foster. What is the Grid? A Three Point Checklist. *GRIDtoday*, 1(6), July 2002.

[3] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a new computing infrastructure*. Morgan-Kaufmann, first edition, 1998.

[4] Distributed Systems Architecture Research Group. gridway.org. `http://www.gridway.org/doku.php?id=documentation`, 2009. [accessed 31-December-2009].

[5] Lawrence Livermore National Laboratory. SLURM documentation. `https://computing.llnl.gov/linux/slurm/documentation.html`, 2009. [accessed 31-December-2009].

[6] Lawrence Livermore National Laboratory. SLURM introduction. `https://computing.llnl.gov/linux/slurm`, 2009. [accessed 31-December-2009].

[7] Michal Novotny. The wiki2 interface description. `http://fi.muni.cz/~xnovot19/wiki2`, 2009. [accessed 31-December-2009].

[8] Michael L. Pinedo. *Planning and Scheduling in Manufacturing and Services*. Springer, 2005.

[9] Cluster Resources. clusterresources.com. `http://www.clusterresources.com/resources/documentation.php`, 2009. [accessed 31-December-2009].

[10] Cluster Resources. Moab job submit. `http://www.clusterresources.com/products/mwm/docs/commands/msub.shtml`, 2009. [accessed 1-January-2010].

[11] Behrooz A. Shirazi, Ali R. Hurson, and Krishna M. Kavi. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, 1995.

[12] Top500 supercomputer sites. top500.org. `http://www.top500.org/`, 2009. [accessed 31-December-2009].

[13] Supercomputing and Networking Center. gridge.org. `http://www.gridge.org/content/view/18/99/1/3/`, 2006. [accessed 31-December-2009].

[14] A. Yoo, M. Jette, and M. Grondona. SLURM: Simple linux utility for resource management. *Job Scheduling Strategies for Parallel Processing*, 2003. [accessed 31-December-2009].

Appendix A

Content of the attached CD-ROM

The CD-ROM contains a working directory used for the thesis development. Not all information in there is correct, yet almost everything has been included because it partially documents the way we have worked. The following is important and up to date:

`thesis/thesis.pdf`
The thesis in pdf format

`thesis/thesis.tex`
The main .tex file

`thesis/chapters/`
All the thesis chapters

`thesis/figures/`
Figures used in the thesis

`thesis/materials/`
SLURM related documents

`sample_ext/`
Sources of the sample external scheduler

`sample_int_select/`
Sources of the sample selection plugin

`sample_int_sched/`
Sources of the sample schedule plugin

`tests/system_features/`
System tests that document some of our statements

`wiki2_msg/`
Developed specification of the wiki2 interface