Architecture of the Slurm Workload Manager

Morris A. Jette and Tim Wickberg

SchedMD LLC, Lehi, UT 84043, USA
{jette,tim}@schedmd.com

Abstract. Slurm is an open source, fault-tolerant, and highly scalable workload manager used on many of the world's supercomputers and computer clusters. As a cluster workload manager, Slurm has three key functions. First, it allocates exclusive and/or non-exclusive access to resources for some duration of time. Second, it provides a framework for starting, executing, and monitoring work on the allocated resources. Finally, it arbitrates contention for resources by managing queues of pending work and enforcing administrative policies. This paper describes the current design and capabilities of Slurm.

Keywords: scheduling · slurm · hpc

1 Introduction

The development of Slurm began at Lawrence Livermore National Laboratory (LLNL) in 2002. It was originally designed as a simple resource manager [1] capable only of allocating whole nodes to jobs, then dispatching and managing those applications on their allocated resources [1]. Slurm relied upon an external scheduler such as Maui [2] to manage queues of work and schedule resources. Slurm has since evolved into a comprehensive workload scheduler capable of managing the most demanding workflows on many of the largest computers in the world. While Slurm has evolved a great deal over its two decades of existence, its original design goals have largely persisted and proven critical to its success.

- Open Source: Slurm's source code is distributed under the GNU General Public License and is freely available on Github [3]. This openness has resulted in contributions from roughly 300 individuals ranging from minor corrections to the documentation to complex added functionality.
- Portability: Slurm is written in the C language with a GNU autoconf configuration engine. Slurm can be thought of as a highly modular and generic kernel with hundreds of plugins available for customization using a building-block approach in order to support a wide variety of hardware types, software environments, and scheduling capabilities. Site-specific plugins and scripts can easily be integrated for even greater customization. This flexibility allows Slurm to operate effectively in virtually any environment.

[1] Slurm was originally an acronym for "Simple Linux Utility for Resource Management", and stylized as "SLURM". The acronym was dropped in 2012 and the preferred capitalization changed to "Slurm".

- Scalability: Slurm daemons are highly concurrent — with locking on a set of core data-structures — in order to support the largest computers with tens of thousands of jobs. As system sizes have increased through time, only relatively minor changes to Slurm have been required. The first exascale system — Frontier at Oak Ridge National Laboratory (ORNL) — runs Slurm, and required no system- or scale-specific modifications to the code at installation [4]. A full system application on Frontier from job submission to termination of all processes and release of its resources can be completed in under 13 seconds.
- Fault tolerance: Slurm has no single point of failure. Backup daemons and cached information insure effective utilization of even the most failure prone systems. Applications can continue execution through failure of allocated compute nodes.
- Security: Slurm authenticates all communication between processes within the cluster using MUNGE [5], ensuring that users cannot impersonate one another or Slurm's control processes. Optional support for JWT [6] allows authentication to Slurm from the REST interface to permit external systems interactions with the cluster.
- System administrator friendly: Slurm has strived to make the support of large and complex systems as simple as possible for system administrators. This includes use of "hostlists" for node name specifications, centralized configuration management, and simplified dynamic reconfiguration.

Simplicity was one of Slurm's original design goals. While that goal may have been satisfied while Slurm was only performing resource management, complexity has increased greatly with the addition of workload scheduling logic. Slurm's code base is currently over six hundred thousand lines, primarily in the C language.

The original functionality of Slurm was loosely based on Quadrics RMS [7], a closed source resource manager which only supported Quadrics proprietary networks and thus failed to satisfy the broader requirements of LLNL.

The remainder of this paper will present information about Slurm's overall design and capabilities as of the 23.02 release (February 2023). Detailed information about its configuration, daemons, commands and operation currently spans many hundreds of pages [8] and is outside the scope of this paper.

2 Slurm Entities

A **job** in Slurm is a resource allocation request. It can either be in the form of a batch script to be queued for later execution, or an interactive job for which the user awaits resource allocation and then makes real time use of it. Job information includes a numeric ID, name, time limit (minimum and/or maximum), size specifications (a minimum and/or maximum count of nodes, CPUs, sockets, cores and/or threads), names of specific nodes to include or exclude in its allocation, node features — both required (e.g., specific processor type,

this specification can include AND, OR, exclusive OR, and count specifications) and preferred (e.g., faster clock speed desired), required licenses, account name, job dependency specifications (to control the order of job execution), Quality Of Service (QOS), relative priority, and queues/partitions to use. The minimum time limit and size specifications are valuable if the user is willing to sacrifice run time and/or resources in order that the job be initiated as soon as possible. Slurm's backfill scheduler will take advantage of this flexibility to allocate such a job with the maximum run time and resources possible within it's specified range without delaying the initiation time of higher priority jobs. The ability for a job to explicitly exclude specific nodes from its allocation is valuable if the user has doubts as to the integrity of specific nodes.

A **job step** in Slurm is a set of parallel tasks, typically an MPI application. A job can initiate an arbitrary number of steps serially and/or in parallel, with Slurm providing the queuing and resource management for those steps within the job's existing resource allocation. Use cases in which jobs execute thousands of steps are not uncommon, particularly when jobs may need to wait lengthy periods of time for their initial resource allocation request to be satisfied. Job step state information maintained by Slurm include its ID expressed as a job ID followed by a period and step ID (e.g., "123.45"), name, time limit (maximum), size specification (minimum and/or maximum count of nodes, CPUs, sockets, cores and/or threads), specific node names to include or exclude from its allocation, and node features required in its allocation. The job step management is lighter than job management. If currently available resources within a job allocation are insufficient for a step to be initiated then it is queued until resources are available. Slurm does not support dependencies between job steps — that functionality can be provided by the job script managing the job step workflow if necessary.

A **cluster** typically consists of a collection of nodes sharing a common network. A Slurm cluster can be on premises, in the cloud, or spread across both. A Slurm **federation** is a collection of clusters sharing a common configuration database. By default, a job is submitted to the local cluster, and user commands to gather information about jobs and queues report information about the local cluster. However, all clusters in a federation may be configured to operate as a single system from the perspective of the users. Jobs can be submitted to any individual cluster in the federation or any set of clusters (e.g., clusters from the same manufacturer or having the same architecture), and may be eligible for execution on any of the federated systems.

Node configuration includes a wide variety of information, most of which is collected directly from the compute node when Slurm's slurmd daemon is started. Information collected from a compute node and maintained by Slurm includes: count of boards, sockets, cores and threads, a count of CPUs (usually defined as $boards \times sockets \times cores \times threads$, but may vary depending upon configuration), memory size, generic resources (GRES) including names, types and counts (used for GPUs, network bandwidth, scratch disk space, etc.). Information not collected from a compute node but maintained by Slurm include a

scheduling preference weight (used to administratively steer jobs towards specific nodes), features (an arbitrary string typically used to record operating system version, CPU type, etc.), node state (up, down, draining, etc.), and a reason for the current node state including the user ID of the individual modifying the node state and the time when state changed. For ease of use with larger clusters, node names may be specified using comma separated numbers and/or ranges (e.g., "node[00,08]", "node[0-15]", "rack[0-7]_blade[0-63]"). **NodeSets** can also be used as a shorthand for collections of nodes, and defined either by an explicit list of nodes, or automatically established on a set of common features. Nodes can be dynamically added and removed from a system, which is generally required for dynamic cluster configurations such as cloud bursting (supplementing an on premises cluster with resources from the cloud on an as needed basis).

Slurm supports the concept of **specialized cores**, which are one or more specific cores on a node reserved for system use and not allocated to jobs. This ensures an application's access to compute resources are never blocked by system overhead and optimal application parallelism can be achieved [9].

A Slurm **partition** is the name given to a job queue [2]. These queues have a wide range of available limits and access controls. Each partition may be associated with a specific set of compute nodes, although each node can be in more than one partition. Alternatively a "floating" partition may be assigned a size, but no specific nodes. When a floating partition's workload is insufficient to fully utilize it's allocated nodes, those resources are made available to other partitions. If the workload exceeds physical resources, the available resources will be distributed per partition scheduling priority, job scheduling priority and applicable limits. Each partition has an access control list, state information (up, down, draining, etc.), scheduling priority, billing weight (higher priority or otherwise more capable partitions may charge jobs at a higher rate), and rules for preemption, over-subscription, gang scheduling [10] and a wide variety of limits. A partition can include a heterogeneous variety of node types. In this case, a resource allocation request can specify the partition(s) to use along with required node features, memory size, etc. A simple example of Slurm entities is shown in Figure 1.

In order to support a series of closely related jobs, Slurm provides the concept of a **job array**. A job array is a batch job in which a single script is executed multiple times. Typically, input and output files for the batch script differ for each execution. The user can then monitor and manage the entire job array as a single entity. Internally Slurm manages the entire job array as a single record in its job table until execution of an element in the array is ready to begin. At that time, a new job record is created for the element. This minimizes both the memory footprint and management overhead for job arrays. Internally, a single pair of files are used to record the batch script and environment variable for all tasks in the job array. For example:

[2] The "partition" term was inherited from Quadrics RMS, which required a strict partitioning of compute nodes within the environment. The requirement that nodes be in disjoint partitions was discarded very early on, but the terminology has persisted.

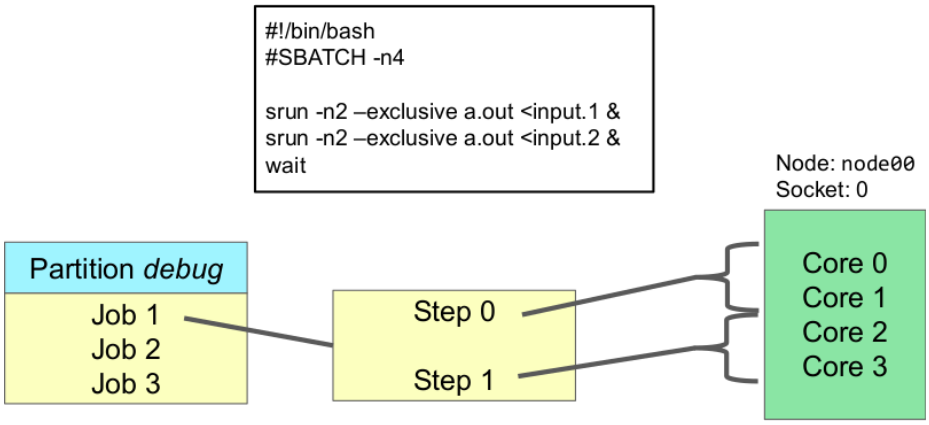

Fig. 1. Example partition, job and job step allocation.

$ sbatch --array=1-9000 -i my_in_%a -o my_out_%a -N1 my.bash
Submits the batch script "my.bash" to be executed 9000 times. The input and output files for each element in the job array will have a prefix of "my_in_" and "my_out_" respectively with the element number appended (e.g., "my_in_123" for the input file used by element number 123). In case some individual elements in the array fail for some reason, those specific job array element IDs may be resubmitted with those IDs identified using the command's "--array" option. For example:

$ sbatch --array=5,28 -i my_in_%a -o my_out_%a -N1 my.bash
The user can optionally specify element IDs using a range with a step count (e.g., from 1 to 50000 by 10). The user can also specify the maximum number of elements in the job array to be executing at any given time, which is valuable in managing their aggregate workload within their various resource limits. By default, each element in the job array is counted as one job for limit enforcement although limits may be configured for the maximum number of elements in any job array. Environment variables are available in the job array applications identifying the job array parameters.

Slurm supports the concept of **heterogeneous jobs**. For example, some elements of a job may require additional memory or other resources. All components of the heterogeneous job will be monitored and managed as a single entity, although each component can also be monitored and managed independently. A heterogeneous job will be allocated resources only when all components of that job can be allocated resources simultaneously, considering both resource availability and applicable limits. Heterogeneous job steps are also supported and options are available to control how applications are launched across the various components of the heterogeneous job.

In order to support complex workflows, Slurm supports a wide variety of **dependencies** as listed below:

- After: job can begin execution after the specified job IDs have begun execution
- AfterOK: job can begin execution after the specified job IDs have completed successfully (run to completion with exit code of zero)
- AfterNotOK: job can begin execution after the specified job IDs have terminated in some failed state
- AfterAny: job can begin execution after the specified job IDs have terminated in any state
- AfterCorr: an element of a job array can begin execution after the corresponding element ID in another job array completes
- Singleton: the job can begin execution after any previously initiated job with the same job name and user ID have completed (i.e., only one job owned by a given user with the same job name can be running at any time)

The system administrator can configure the desired behavior for jobs with dependencies that cannot be satisfied (e.g., a job dependent upon the successful completion of another job, but that job fails). Typically such jobs are configured to be purged.

Users can request to be notified by email when their jobs change state. This can be valuable for batch jobs in environments where long delays are possible. Job state transitions which can be used to trigger email include: begin, end, fail, requeue, and invalid dependency detected.

A Slurm **account** is used to group users into sets, independent of UNIX groups. Accounts are typically organized in a hierarchical fashion (e.g., division, group, project, etc. see Figure 2). A user can have access to multiple accounts with a default value. Each account can have one or more account coordinators who are able to create sub-accounts, add or remove users from their account, modify limits and resource apportioning to the users and accounts under their control. The account coordinator may also be able to view accounting information normally hidden from other users, such as a record of jobs executed by other users in the accounts over which they have control.

A Slurm **association** is a combination of cluster, account, user name, and (optional) partition name. Each association can have a fair share allocation of resources and a multitude of limits. It is worth noting that these limits come in two forms. Many limits apply to individual jobs, such as the maximum time limit. Other limits apply on an aggregate basis, such as the maximum number of running jobs for an individual user or all users in some account.

A **Quality Of Service (QOS)** is used to control a job's limits, priority, and charge multiplier. A QOS may be associated with partition or independent of partitions and selected on a job by job basis. A QOS not explicitly bound to a partition can be used with any partition. The benefit in associating a QOS with a partition is in making a greater number of limits available than otherwise provided in Slurm's configuration file for a partition. A typical use case is to configure "standby", "normal", and "expedite" QOS on a system. Jobs submitted to any partition with a "standby" QOS would have very low priority and a corresponding low charge multiplier, say being charged for resource use at 20% of

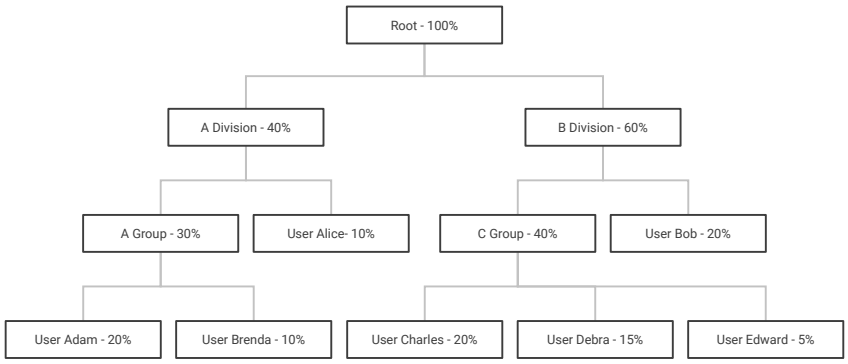

Fig. 2. Example account hierarchy with resource allocations.

the normal change. Similarly, jobs submitted to any partition with a "expedite" QOS would be given a very high priority and high charge multiplier, perhaps being charged 5 times the normal rate. Access control lists can be configured to limit which accounts or users can use each QOS. QOS can also be used to define job preemption rules, so the "expedite" QOS might be configured to preempt (terminate running jobs) from the lower priority "standby" QOS. QOS limits available override the partitions and associations limits. So if a user's association has a maximum number of running jobs set to 10, but the "expedite" QOS has a limit of 20, the higher limit will apply to jobs running in the "expedite" QOS. In order to avoid confusion with the multitude of configurable limits, only a subset of limits are typically configured for associations and QOS.

The order of precedence for limits is as follows:

1. Partition QOS
2. Job QOS
3. User association
4. Account associations (ascending the hierarchy)
5. Root/Cluster association (i.e., the top of the account association hierarchy)
6. Partition configuration

If limits are defined at multiple points in this hierarchy, the point in this list where the limit is first defined will be used. Consider the following example:

- MaxJobs=20 and MaxSubmitJobs is undefined in the partition QOS
- No limits are set in the job QOS and
- MaxJobs=4 and MaxSubmitJobs=50 in the user association

The limits in effect will be MaxJobs=20 and MaxSubmitJobs=50

A **Generic RESource (GRES)** is an arbitrary on-node resource. These can be allocated to jobs, accounted for, and have limits imposed on their use all in a generic fashion. GRES can also be managed in a hierarchical fashion. GPUs are managed in Slurm as GRES. Jobs can request a specific count of GPUs, administrators can configure GPU limits by job or user, etc. If a cluster has more than one variety of GPU, the job can request some count of GPUs using any variety of GPU available (e.g., "srun --gres=gpu:2 ...") or identify the specific varieties and counts of GPU desired (e.g., "srun --gres=gpu:rtx_5000:2,gpu:rtx_6000:1 ..."). The cgroup [11] plugin is used to enforce binding of applications to their allocated GPUs or other GRES. Resource limits for GRES can be configured with similar flexibility and the possibility of different counts for different varieties of GPU.

An **advanced reservation** is a block of resources reserved for current or future use. These resources can include specific nodes, specific cores, licenses, burst buffer space, start time, end time, and access control list (user ID or Slurm account name). The reservation can identify a count of nodes or cores without identifying specific nodes or cores, which can enable more efficient resource usage when the reservation is not regularly used, and permits any failed resources to be replaced in order to maintain the requested reservation size. The reservation can identify a start time relative to whatever the current time is (e.g., 10 minutes in the future), which we refer to as a "floating" reservation. Use of a relative start time can be used to ensure that privileged users or accounts can access compute resources in a timely fashion without preventing their use by the wider user community when otherwise not required. For example, one might create a floating reservation consisting of 16 arbitrary compute nodes that must be available within 30 minutes of the current time. This will ensure that a job requesting 16 nodes in that reservation will be able to secure those resources and begin execution within 30 minutes without preventing other users from making use of those resources with shorter lived jobs or jobs approaching their time limits. Reservations can be recurring (e.g., daily, weekly, etc.). A reservation can be configured so that jobs must fit entirely within its time and space constraints. Alternately a job can use resources in a partition in addition to any other available resources. Reservations can be configured as "magnetic", in which case any job that matches that reservation's access controls will be eligible to run within resources it controls when the reservation is active without explicitly requesting that use of that reservation. Another common use case for advanced reservations is to limit access to resources so that they will be idle at the time of a scheduled maintenance without the need to terminate any active jobs using those resources, but permitting jobs with lower time limits to be initiated on those resources as time permits.

3 Daemons

Slurmdbd is Slurm's database daemon, which can interface either MariaDB [12] or MySQL [13] (see Figure 3). There is typically one slurmdbd daemon per enterprise. The slurmdbd records accounting information and manages some configuration information centrally (many limits, fair share information, QOS, licenses, etc.). When a system administrator modifies this centralized configuration information, those changes are transmitted to other Slurm daemons as needed. If the slurmdbd or its database are unavailable for some reason, the entire Slurm enterprise will continue to operate normally, but use cached configuration information and locally store accounting information until database access is restored. Slurmdbd is typically configured to execute as user "SlurmUser", an unprivileged user.

The **slurmctld** daemon — commonly referred to as the Slurm controller — run as a single active daemon per cluster, although an arbitrary number of backup daemons may be configured. The slurmctld monitors the state of resources, decides when and where to initiate jobs and job steps, and processes almost all user commands (except for accounting and other database related commands). If configured for high-availability, a file system must be configured to store the slurmctld state information such that is accessible on both the primary and all backup controllers. Slurmctld executes as user "SlurmUser", an unprivileged user.

The **slurmd** is Slurm's compute node daemon. There is typically one slurmd daemon per compute node. The slurmd daemon is the only Slurm daemon that needs to execute as user root in order to spawn processes as the appropriate user. Slurmd is quiescent after launching the job step management process (slurmstepd) except for optional accounting and message forwarding. One slurmstepd process is spawned by slurmd for each batch script and application on a compute node. Slurmstepd is responsible for launching the batch script or user application's tasks, handling any cgroup and filesystem namespace management, managing accounting, application I/O, signals, etc. It is possible to configure multiple slurmd daemons to execute on each compute node in order to evaluate Slurm behavior on larger systems. Moderately large clusters can be emulated on just a single compute node or desktop. Each compute node in such an emulated cluster can be configured independently, for example with differing memory or CPU counts, potentially with larger sizes than the compute node actually executing the slurmd daemon.

The **slurmrestd** daemon is an interface to Slurm via REST API that translates JSON/YAML-encoded requests into Slurm Remote Procedure Calls (RPCs). Any user can initiate their own slurmrestd daemon and all remote clients are authenticated using HTTP headers. This is intended to enable use of Slurm's API outside of the cluster to web interfaces, user desktops (with appropriate security considerations), and other third-party applications.

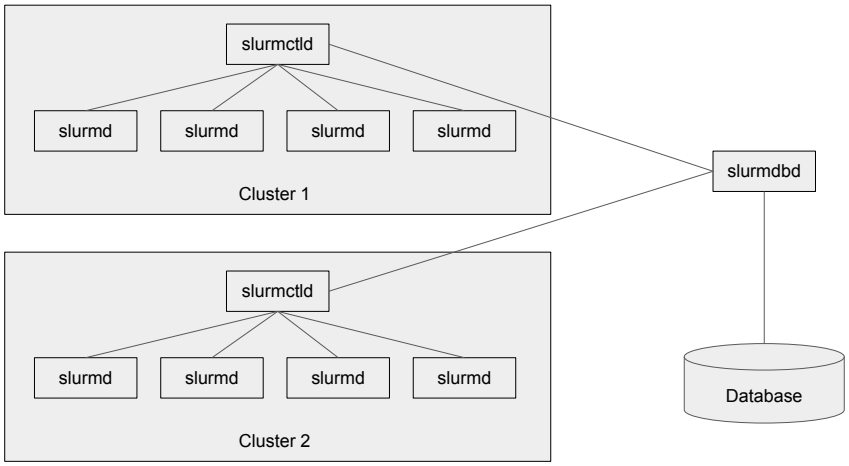

Fig. 3. Typical Slurm federation with two clusters.

4 Plugin Infrastructure

Slurm's extensive use of plugins is particularly valuable in providing portability and flexibility. Roughly 65% of Slurm's code is within its kernel including the primary data structures, system daemons, and user commands. Over 100 plugins form the remainder of the code to support a wide range of differing hardware types, software environments, and scheduling capabilities. The configured plugins are loaded when a daemon or command is started and persists throughout its lifetime. Some plugins are called from multiple daemons and commands. In some cases plugins are also called by other plugins, such as the network topology plugin being called by the scheduling plugin. Plugin infrastructure provides a level of indirection to some configurable underlying functions. One example of the value in plugins was development work performed by Hewlett-Packard (HP) in 2005 [14]. Slurm's original implementation only supported the allocation of whole compute nodes to jobs — implemented through the select/linear scheduling plugin. HP added the ability to allocate resources on a node down to the core level with a new select/cons_res plugin. Later development introduced a new select/cons_tres plugin, now the default, which extended resource scheduling support to GPUs within the nodes as well. Roughly 80% of the changes to Slurm for this enhancement were in the form of a new job scheduling plugin with the remaining changes in the kernel, much of that in the form data structure changes. Given the number of plugins available, only a few will be described here.

Slurm's topology plugin is used to gather network topology and use that information to optimize resource allocations with respect to communication bandwidth. The topology plugins developed to date include 3 dimensional torus, 4

dimensional torus, hypercube, dragonfly, and tree. Slurm's GUI, sview, displays the nodes allocated to jobs, partitions, advanced reservations, etc, so that one can readily observe the network topology they utilize as shown in Figure 4.

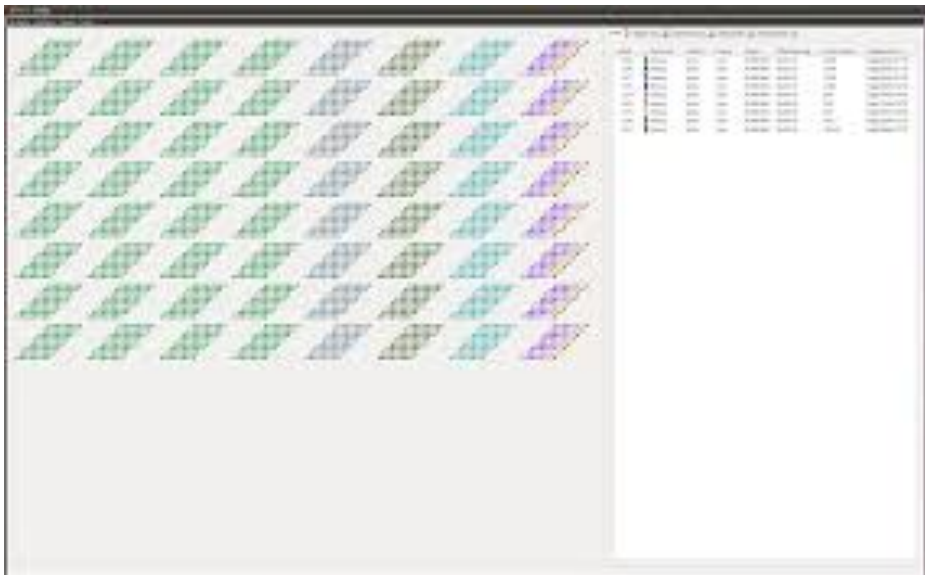

Fig. 4. Example of job allocations on a 4D torus interconnect. A list of jobs is shown on the right and their color coded node allocations on the left.

Slurm's job submit plugin is called from the slurmctld daemon. It is executed for each job submit or job modify RPC. An arbitrary number of job submit plugins may be used with a configurable call sequence. Each of these plugins can modify the arguments passed to slurmctld and return error messages as appropriate to the user. Some of the job submit plugins packaged with Slurm include: throttle (limits the rate at which a user can submit jobs, sleeping as needed to decrease job submission rates for individual users), require_time_limit (rejects jobs without an explicit time limit specification), pbs (adds PBS [15] environment variables for newly submitted jobs and supports the "before" job dependency), cray (sets Cray specific generic resource parameters), and Lua (executes a customer provided Lua script with almost limitless flexibility).

Four plugins are available to gather energy consumption from a node including IPMI, RAPL, and Cray. Should some new mechanism become available to gather a node's energy consumption data, one would need to develop a new Slurm plugin to gather that information and present it to the Slurm kernel in the appropriate format.

SPANK (Slurm Plugin Architecture for Node and job [K]ontrol) is a generic plugin mechanism. The plugins are written in C, but without requiring access to

the Slurm source code. The plugins are executed by the Slurm daemons and the Slurm commands used for job submission. SPANK plugins can be used to add new site-specific job options, including making information about those options visible in the command's help messages. One example of SPANK use was the initial integration of Singularity containers with Slurm [16]. This plugin added new options to the Slurm job submission commands: --singularity-container, --singularity-bind and --singularity-args. It also added support for Singularity specific environment variables. The slurmstepd job step management process made use of this newly added information to initiate the application in an appropriate container environment.

The most common integration point remains through the use of site-specific scripts in the Slurm prolog and epilog interfaces. These are executed in various places (the submit host, the head node by slurmctld, or the compute node by slurmd or slurmstepd), at various times (e.g., job allocation, step startup, and task launch), by various users (SlurmUser, root or the job user). Typical use cases include establishing the environment for the job (boot nodes, node health check, configure temporary storage, etc.) or cleaning up at job completion (deleting temporary files).

5 Configuration

Slurm requires at least one configuration file, although some plugins require their own configuration file. These files can either be in a location readable by all daemons and user commands or the files can be replicated on every node. Alternatively, the files can be placed on the nodes where the slurmctld daemons execute (primary daemon plus backups), and the slurmctld daemon will make that configuration information available to the other daemons and commands upon request with a newer optional feature referred to as "configless" support.

A system administrator may find it difficult to upgrade the Slurm installation simultaneously across all of the enterprise. In order to support rolling upgrades, every daemon and command is able to support RPCs for three major releases, which includes its release plus the previous two major releases of Slurm. Since major releases are currently scheduled every 9 months, that supports a relaxed upgrade schedule. Changes to RPCs are limited to major releases, so upgrades between maintenance releases can be performed on a node-by-node basis.

6 Communications

Slurm uses a fault-tolerant hierarchical communication mechanism with configurable fanout for communications to the compute nodes. This offloads as much work as possible from the slurmctld daemon, which typically has a multitude of active threads. This also minimizes the wall time required for operations involving a large number of nodes, such as application launch and file transfer. It is typically recommended to configure a fanout value so that no more than a five level communication tree can be used to reach all compute nodes. For

example, if the slurmctld needs to kill a job running on 1110 compute nodes and Slurm's fanout is configured at 10. The slurmctld daemon will take the list of 1110 compute nodes and divide it into 10 sets of 111 nodes each. The slurmctld daemon will then launch 10 threads, each communicating with a single slurmd daemon notifying it of the job to be killed along with a list of the additional 110 compute nodes that slurmd should forward the request to. Each of these 10 slurmd daemon launches 10 threads to communicate with additional slurmd daemons on other compute nodes. The process is continued until every slurmd daemon involved in the operation is reached. In this case, the process requires three levels in the communication tree. Note the communication hierarchy is created as needed and destroyed upon completion of the communications. Multiple communication hierarchies may be active at any time using different or even overlapping sets of slurmd daemons. See the example in Figure 5.

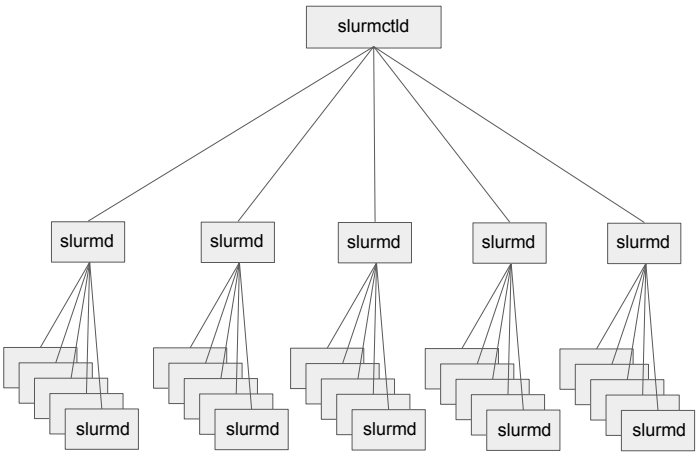

Fig. 5. Example hierarchical communication with fanout of 5.

Slurm's configurable timeout parameter is used to help detect communication failures. In the event of a failure, the daemon originating the message will send the message to another slurmd daemon in the group of compute nodes to be communicated with. Since this retry logic may result in messages being sent more than once to a slurmd daemon, the daemon retains state information to avoid duplicating operations.

The logic to spawn job steps follows a similar hierarchical communication mechanism. The srun command sends its job step launch request to the slurmctld daemon. When the slurmctld daemon satisfies the request (which might be queued pending resource availability), slurmctld returns a digitally signed step launch credential to srun. The srun command then forwards this credential with

an application launch request to a set of slurmd daemons using the same fanout logic as for messages originating from the slurmctld daemon.

7 Job Priority

Slurm assigns a priority to each job based on a multitude of factors including age, fair share, queue/partition, QOS, size, nice value, association, and a site-managed value. The weight of each factor in determining a job's priority is configurable so that a job's priority may be based 60% on age age, 30% on fair share, etc. The age component of job priority is based upon the time when the job first becomes eligible for execution, after dependencies are satisfied, rather than its submission time. It value is proportional to that wait time with a configurable maximum time (i.e., the value could be configured to stop increasing once a job is 7 days old). Fair share is a measure of how over- or under-serviced an association is relative to its resource allocation. The window of time used in this calculation is either fixed with usage data cleared periodically (i.e., at the end of each week, month, quarter, year, etc.) or historic resource usage data is continuously decreased on an exponential basis through time. Different algorithms and parameters are available to control how fair share is computed based upon the association tree, and a plugin interface called site_factor is provided should an administrator wish to develop their own novel prioritization approach. For example, should an individual user be allowed to consume all resources allocated to his group if no other users in that group are active or should some portion of that group's resources be retained for when other users in that group become active and if so how much [17]. Job size requirements can be used to consider a job's resource requirements in computing its priority. Size in this context is configurable to consider a variety of resources with different weights for each resource (e.g., one GPU might be given the same weight as 1TB of memory in computing a job's size component of priority). A system administrator may want to increase the scheduling priority of jobs with large CPU, memory, or license requirements. This can also be reversed to favor jobs with low resource requirements. A user may specify a job's nice value to establish their relative scheduling priority in Slurm and this works similar to a process's Linux nice value, although Slurm's nice value range is much larger with a signed 32-bit value. If necessary, a system administrator may also explicitly set a job's scheduling priority in order to override the default calculated value. This is typically done to force a job to have the highest priority, and ensure it will begin execution as soon as possible.

8 Typical Configurations

Each site and each cluster have unique configurations and scheduling considerations. Before considering Slurm's scheduling algorithms, some typical configurations and their scheduling requirements are described below.

Roughly half of the clusters that we work with are homogeneous: every compute node has the same processors, memory size, GPUs, etc. Homogeneous clusters may be configured with a single partition for the most efficient use of resources. While every job may be in a single queue/partition, a variety of limits are typically used to prevent any single user or group of users from being allocated more resources than desired. Depending on the configuration and use case, it is not uncommon to see dozens or even hundreds of the highest priority jobs have their resource allocation deferred by one or more limit (e.g., maximum running job count by user, maximum allocated GPU count by account). In addition to the compute nodes, global resources such as licenses and burst buffer space must be managed.

The other half of the clusters we work with are heterogeneous. Such clusters may include a small number of unique compute nodes, say with GPUs or a larger memory size. Other clusters may include a dozen or more unique compute node configurations including different processor types and clock speeds. Such clusters are typically assembled over time with different organizations contributing hardware best suited for their workload and budget. For example, the physics department at a university may purchase two racks of nodes with large memory size, the chemistry department another rack of nodes with GPUs, etc. We refer to each set of resources as a "condo" or "condominium" and they are interconnected into a single cluster sharing a high speed network. Typically each condo can be accessed from two or more partitions. One partition will provide priority access to the resources with an access control list identifying the organization financing those resources (e.g., the "physics" partition will have an access control list containing the faculty and students in the physics department). A second partition might span all compute nodes in the cluster with lower priority access for any user, typically with a lower size and/or time limits. Slurm allows jobs to be submitted to multiple partitions simultaneously to take advantage of this configuration. The node "feature" parameter can be used to prevent job allocations from spanning different processor types as shown in Figure 6.

Job throughput rate requirements also vary widely. Some workloads consist primarily of jobs that execute for days, in which case expending considerable time to optimize scheduling may be warranted. Other workloads consist primarily of jobs that execute for a few seconds and a throughput rate in the of hundreds of jobs per second may be required [20]. Slurm can support both workloads, but with different algorithms and scheduling parameters.

9 Scheduling Algorithm

Slurm performs a quick and simple (first-in first-out, FIFO) scheduling attempt on an event driven basis (with a configurable minimum time interval between executions): upon each job submission, job completion, or configuration change. Only the top priority jobs (a configurable count) in each partition will be evaluated for initiation and this can be useful for high throughput computing. Given the appropriate configuration and hardware, Slurm can sustain a throughput

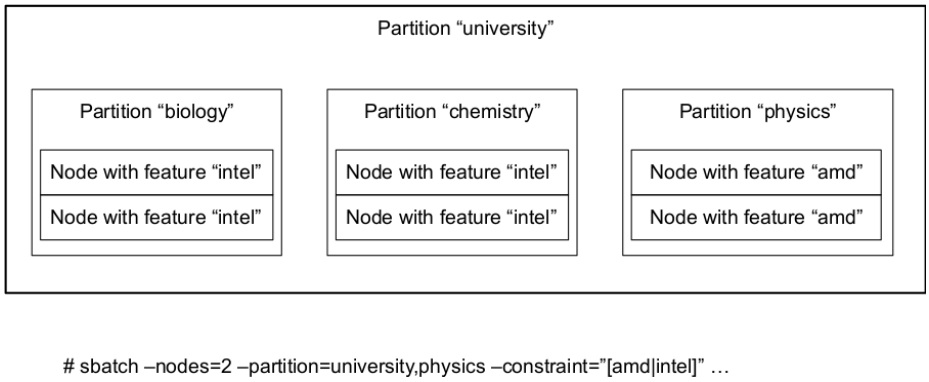

Fig. 6. Example heterogeneous cluster with job submission request for any two nodes with the same architecture. Plugins can be used to set default partitions and node features as appropriate.

rate exceeding 100 jobs per second. Since this scheduling algorithm is FIFO, once any job in a partition is found unable to be initiated, all lower priority jobs in that partition will be left pending without further consideration.

Slurm also performs a more comprehensive FIFO scheduling attempt on a periodic basis with a configurable interval. This algorithm typically executes far less frequently than the event driven scheduling algorithm, but uses the same logic with different configuration parameters. The default configuration parameters will typically enable this algorithm to evaluate jobs in every partition from the highest priority until a job unable to be initiated is identified.

Except for some high throughput computing configurations, backfill scheduling plugin is used to initiate most jobs. Without the backfill plugin, jobs in each queue are scheduled strictly in priority order as described above. The backfill scheduling plugin will initiate lower priority jobs only if doing so does not delay the expected start time of any higher priority job, although reservation of resources for higher priority jobs can be limited to jobs which have been pending for over some configurable period time or the highest priority jobs in each partition within a configurable queue depth. The expected start time of pending jobs is dependent on the expected completion time of running jobs based solely upon the job's time limit. Approximately 20 configuration parameters are available for tuning backfill scheduling such as: how far in the future to consider, what is the time resolution for scheduling, how many jobs to consider for each user, how many jobs to consider from each partition, etc. The backfill scheduler builds a table of expected resource availability through time, tracing the expected initiation and termination time of running and pending jobs. All resource limits are enforced by the backfill scheduler as it builds this table. For example, sufficient compute resources may be available to initiate the highest priority job in some partition one hour in the future, but job initiation prevented by the maximum number of CPU hours of running jobs for the job's association. In this case a

lower priority job may be initiated. Since a cluster may have thousands of executing jobs and tens of thousands of pending jobs, backfill scheduling overhead is kept more manageable by the time resolution configuration parameter cited above. Say the time resolution is configured to 300 seconds. The resources that become available in any 300 second interval are recorded in a single record rather than one record per job completion, which might number in the tens or even hundreds of jobs for any 300 second interval. For example, consider a job expected to end in 600 seconds and another job expected to end in 610 seconds. Rather than creating records of expected system state at those two times and determining which pending jobs can start at each of those times, the records are combined and pending jobs will only be evaluated for initiation at that one time. While this does result in some loss of precision when evaluating when pending jobs are expected to start, that is likely insignificant compared to the inaccuracy in job time limits. The benefit is that computational overhead of the backfill scheduler can be dramatically reduced. The backfill scheduler determines the reason each pending job is currently unable to be initiated (e.g., some specific limit, dependency, waiting for resources, held by administrator) and its expected start time (i.e., when compute resources are available and no limits exceeded). This information is made available to users and is regularly consulted. Since the backfill algorithm can require multiple minutes to complete with large workloads, it is performed in a piecemeal manner. It acquires the appropriate locks, executes for a configurable time interval (typically a couple of seconds), releases locks for a configurable time interval (typically less than one second) in order to perform other outstanding operations (e.g., accept newly submitted jobs, process newly completed jobs, provide users with status information, etc.), and repeats the process until all pending jobs have been considered.

Slurm supports burst buffers via a plugin mechanism. Slurm will allocate burst buffer space for a job when it approaches its expected initiation time and stage-in any required data. The job will not be allocated compute resources until data stage-in has completed. When the job's computation has completed, data will be staged-out and the burst buffer space released. The burst buffers supported by Slurm plugins include Cray's DataWarp [21] and a generic Lua script based plugin.

Slurm has the ability to support a cluster that grows and shrinks on demand, typically relying upon a service such as Amazon Elastic Computing Cloud (Amazon EC2), Google Cloud Platform or Microsoft Azure for resources. These resources can be combined with an existing cluster to process excess workload (cloud bursting) or it can operate as an independent self-contained cluster. Good responsiveness and throughput can be achieved while only paying for the resources needed.

Slurm has dozens of available scheduling parameters available to control the number of jobs considered for scheduling in each partition, maximum scheduling frequency, etc. [18] Slurm also maintains detailed information about scheduling performance and makes that information available to system administrators for tuning purposes [19]. Detailed information about anticipated resource avail-

ability in the future and expected resource allocations and initiation times of pending jobs are also available.

10 License Scheduling

In addition to node-centric resources, Slurm supports scheduling for **licenses**. Licenses can be used to represent any resource available on a global basis such as network bandwidth or global scratch disk space; although, as implied by the name, they are most commonly used to track software licenses.

Licenses are requested as part of the job submission, and will be allocated to the job alongside the compute resources. Backfill scheduling management for these licenses is a recent optional addition, and can be enabled by the administrator. Preemption support, in which lower priority jobs can be preempted to free up sufficient licenses for higher priority ones, has also been recently added.

11 Application Layout

The job has control over its resource allocation with respect to sockets, cores, and threads using options at job submit time, for example threads per core, cores per socket, and sockets per node. Similarly the job step allocation can specify the number of tasks to launch per core, socket and/or core. The job step has complete control over how tasks are distributed over the allocated resources by specifying layout patterns across nodes, sockets, and/or cores. Binding tasks to allocated resources can be performed using CPU affinity and/or Linux cgroups [11]. Linux cgroups are essential for configurations where more than one job can be active at the same time on a compute node. Besides limiting each job to its allocated CPUs, cgroups can also ensure that each job is constrained to its allocated memory and not interfere with another job's memory allocation. Linux cgroups can constrain each job's RAM, kernel memory, swap space and allocated generic resources such as GPUs. There are a variety of options available to control how each task/rank of the application are bound to the job step's allocated CPUs. Cgroups are also used to collect usage data for allocated resources.

Some topology plugins support the ability for a job to specify the maximum number of leaf switches desired in its resource allocation and the maximum time to wait for such an allocation (e.g., wait up to an extra 10 minutes for my job's allocated nodes to be on one leaf switch). The system administrator can configure the maximum wait time for any job to secure its desired leaf count in order to limit resources idled for job layout optimization.

Jobs can increase or decrease their size per administrative controls. In the case of increasing the size of a job, the user must submit a new job to acquire the additional resources desired. Once this second job allocation has been made, the user merges the two job allocations into a single job, a process which generates a shell script the user executes in the original job in order to modify its environment variables as appropriate.

12 Job Profiling

Slurm has the ability to collect detailed performance information about a job step on a periodic basis. This is more information than can reasonably be recorded in Slurm's database for every job, and may involve some additional overhead to collect — therefore it is only collected when requested by the user. The user specifies what types of information are to be collected and at what frequencies (independent frequency can be configured for each data type). The types of information available for collection includes power consumption, file system usage, network interconnect usage, CPU and memory usage, and GPU utilization. At application termination the data is collected and stored into a single HDF5 [22] dataset. We recommend the HDFView [23] tool to graphically view the resulting data, which can easily identify problems such as spikes in memory usage with the timing and offending task ID (rank) identified.

13 Compute Node Management

Alongside traditional workload management capabilities, additional integrations have been developed to solve common HPC systems administration tasks.

Slurm has the ability to limit the power consumption of a cluster [24]. It does this by monitoring each node's power consumption and periodically adjusting its available power. Nodes consuming less than their available power will have their power availability reduced and that power will be redistributed to the other nodes in the cluster. Special mechanisms exist to manage power limits uniformly across all nodes allocated to a job as well as job startup and termination.[3]

Container support for OCI images [25, 26] is supported and allows for the launch of compatible containers without any external tooling on the compute nodes. Management for these container images is a direct extension of existing support for Linux subsystems such as cgroups and filesystem namespaces, and required only minimal changes to the slurmd daemon. A newly added command, scrun, allows for use of Slurm as the underpinning for common tools such as Docker [27] and Podman [28].

slurm_pam_adopt is a PAM [29] module that intercepts user SSH connections, and confines them to the resources that were allocated to their job. For connections initiated from other compute nodes (common with SSH-based MPI launchers), it will interrogate the originating compute node for which job that network connection originated from, and can perfectly match that to the allocated resources on that node. (If the job has not been allocated resources, the connection is usually denied depending on configuration.) For connections initiated from login nodes or other external machines, the connection will usually be permitted under resources allocated to the first job running under that user account. Besides confining these processes to resources already allocated to the

[3] This capability was successfully used at King Abdullah University of Science and Technology (KAUST) for a period when a power availability was limited. We are unaware of any other organization currently using of this capability.

job, these processes have an accounting record created for them, which details their resource usage.

nss_slurm is an NSS [30, 31] module that allows Slurm to centrally propagate the user and group information corresponding to each jobs' owner. This mitigates issues when compute nodes, which are usually connected to LDAP, all simultaneously try to resolve user and group information when the job is initiated. In addition, certain cluster deployment approaches can preclude the need to either synchronize /etc/passwd and /etc/group files to the compute nodes, or connect the compute node to LDAP or NIS.

The **sbcast** command, and the associated --bcast option to the srun command, allow for Slurm to distribute files through its built-in tree hierarchical communication systems. Optional compression can further improve performance. This can be used for large-scale job launches to avoid performance issues from large-scale job launches on common network filesystems by moving executables into local scratch (possibly tmpfs) space. An optional mode, enabled through the --send-libs argument, allows for dynamic libraries to be identified and transmitted alongside an executable image, further improving performance for large-scale job launches.

The **scrontab** command allows for users to register a number of periodic compute jobs with a crontab compatible syntax. This is designed to mitigate a common HPC user request for cron access on login nodes, and instead launches compute jobs on designated intervals (while ensuring that only one copy of the process / Slurm job is launched concurrently, a feature that cron itself lacks), and avoids reliance on specific login nodes remaining online continously.

14 Conclusion

This paper presents an overview of Slurm's design and functionality. Slurm provides an open source tool that can effectively manage a wide variety of workloads on computers of any size. It is also highly modular in order to provide excellent flexibility and extensibility. Motivated researchers can experiment with alternative scheduling algorithms, network topologies, etc. by developing their own plugin while leveraging Slurm's extensive and stable framework.

Slurm continues to evolve with the help of numerous dedicated engineers. Areas under current development include improved integration with external orchestration systems such as Kubernetes, full adoption of an internal extensible HMAC authentication scheme [32], further improvements to the "cloud bursting" modes of operation, performance improvements in job throughput and user command interaction, and integration with external interfaces such as PMIx. Future directions, subject to community interest, may also involve improved support for energy-centric computing, ways to improve job step performance and workflow capabilities by divorcing responsibility from the central slurmctld process, and refactoring the core scheduling system to allow for different compute node hierarchies than the traditional board/socket/core/CPU model that is embedded in the existing scheduler subsystems.

References

1. Jette M., Yoo A., and Grondona M.: SLURM: Simple Linux Utility for Resource Management. In: Feitelson D., Rudolph, L., Schwiegelshohn, U. (eds.) Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), LNCS, vol. 2862, pp 44-62, Springer-Verlag (2003).
2. Jackson D., Snell Q., and Clement M.: Core Algorithms of the Maui Scheduler. In: Feitelson D. and Rudolph, L. (eds.) Proceedings of the 7th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), LNCS, vol. 2221, pp 88-102, Springer-Verlag (2001).
3. Slurm code repository, https://github.com/SchedMD/slurm.git. Last accessed 3 Feb 2023.
4. Frontier User Guide, https://docs.olcf.ornl.gov/systems/frontier_user_guide.html. Last accessed 3 Feb 2023.
5. MUNGE home page, https://dun.github.io/munge/. Last accessed 26 Apr 2023.
6. JWT home page, https://jwt.io/. Last accessed 1 May 2023.
7. Quadrics in Linux Clusters presentation, https://hsi.web.cern.ch/HNF-Europe/sem3_2001/hnf.pdf. Last accessed 3 Feb 2023.
8. Slurm Documentation, https://slurm.schedmd.com/. Last accessed 4 Feb 2023.
9. Pritchard H., Roweth D., Henseler D., and Cassella P.: Leveraging the Cray Linux Environment Core Specialization Feature to Realize MPI Asynchronous Progress on Cray XE Systems. In Proceedings of the Cray User Group (2012).
10. Jette M.: Expanding symmetric multiprocessor capability through gang scheduling. In: Feitelson D. and Rudolph, L. (eds.) Proceedings of the 4th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), LNCS, vol. 1459, pp 199-216, Springer-Verlag (1998).
11. Ondrejka P., Majorsinova E., Prpic M., Landmann R., Silas D.: Resource Management Guide, https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/index. Last accessed 23 Mar 2023.
12. MariaDB Foundation home page, https://mariadb.org/. Last accessed 23 Mar 2023.
13. MySQL corporate home page, https://www.mysql.com/. Last accessed 23 Mar 2023.
14. Balle, S. M. and Palermo, D.: Enhancing an Open Source Resource Manager with Multi-Core/Multi-threaded Support. In: Frachtenberg E. and Scwiegelshohn U. (eds.) Proceedings of the 13th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), LNCS, vol. 4942, pp 37-50, Springer-Verlag (2007).
15. OpenPBS home page, https://www.openpbs.org/. Last accessed 28 Mar 2023.
16. Singularity plugin for Slurm, https://github.com/sol-eng/singularity-rstudio/blob/main/slurm-singularity-exec.md Last accessed 4 Feb 2023.
17. Cox, R. and Morrison, L.: Fair Tree: Fairshare Algorithm for Slurm. Slurm User Group Meeting (2014). https://slurm.schedmd.com/SC14/BYU_Fair_Tree.pdf. Last accessed 28 Mar 2023.
18. Slurm Scheduling Configuration Guide, https://slurm.schedmd.com/sched_config.html Last accessed 31 Mar 2023.
19. Slurm Scheduling Diagnostic Documentation, https://slurm.schedmd.com/sdiag.html Last accessed 31 Mar 2023.
20. High Throughput Computing Administration Guide, https://slurm.schedmd.com/high_throughput.html. Last accessed 30 Mar 2023.

21. Henseler D., Landsteiner B., Petesch D., Wright C., and Wright N.: Architecture and Design of Cray DataWarp. In Proceedings of the Cray User Group (2016). https://cug.org/proceedings/cug2016_proceedings/includes/files/pap105s2-file1.pdf
22. HDF5 download page from The HDF Group, https://www.hdfgroup.org/downloads/hdf5. Last accessed 25 Mar 2023.
23. HDFview download page from The HDF Group, https://www.hdfgroup.org/downloads/hdfview. Last accessed 25 Mar 2023.
24. Jette, M.: Slurm Power Management Support. Slurm User Group Meeting (2015). https://slurm.schedmd.com/SLUG15/Power_mgmt.pdf. Last accessed 26 Mar 2023.
25. Open Container Initiative organization home page, https://opencontainers.org/. Last accessed 28 Mar 2023.
26. Slurm container guide https://slurm.schedmd.com/containers.html Last accessed 4 Feb 2023.
27. Docker home page, https://www.docker.com/. Last accessed 28 Mar 2023.
28. Podman home page, https://podman.io/. Last accessed 28 Mar 2023.
29. Garfinkel S., Spafford G., and Schwartz A.: Pluggable Authentication Modules (PAM). In Practical UNIX and Internet Security, 3rd Edition. pp,. 94-96. O'Reilly(2003).
30. Name Service Switch description, https://guix.gnu.org/manual/en/html_node/Name-Service-Switch.html. Last accessed 3 Feb 2023.
31. Name Service Switch implementation for Slurm, https://slurm.schedmd.com/nss_slurm.html. Last accessed 3 Feb 2023.
32. Wikipedia description of HMAC, https://en.wikipedia.org/wiki/HMAC. Last accessed 1 May 2023.