HADM: Integration of Resource Allocations in Slurm

Jeffrey Frey[1]

Author Affiliations

[1] IT Research Cyberinfrastructure, University of Delaware, Newark, DE.

Abstract

The University of Delaware's "DARWIN" HPC cluster[1] required the creation of a system for the management of resource allocations. Each XSEDE (now ACCESS[2]) resource provider (RP) is responsible for implementing this infrastructure locally. Design mandates included scalable, tight integration with the Slurm job scheduler; a containerized client-server implementation of the data storage and access; user-facing allocation query tools; and integration with the XSEDE/ACCESS accounting reporting workflow. The resulting solution has been in continuous operation since June of 2021.

Introduction

Slurm[3] is an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters. Slurm stands apart from many other job-scheduling systems in its being an open source software product. With full access to the source code, consumers at all levels are free to analyze the code and make changes or additions. When presented with site- or system-specific conditions on job scheduling, this flexibility can often set Slurm apart from commercial products.

Prior to the deployment of the University of Delaware's "Caviness" HPC cluster[4] a licensed commercial scheduler was employed. Though both the commercial product and Slurm have advantages and disadvantages in their functionality, the capability to elucidate bugs and apply our own patches to Slurm as well as alter functionality through plugins and source changes proved most lucrative. Eliminating the annual license fees associated with the commercial

[1] https://docs.hpc.udel.edu/abstract/darwin/darwin
[2] https://access-ci.org
[3] https://www.schedmd.com
[4] https://docs.hpc.udel.edu/abstract/caviness/caviness

product and our being able to contribute knowledge and code back to the Slurm developers and community was also a major factor in our adoption of Slurm.

In 2019, the existing HPC systems model matched faculty and departmental hardware investment (stakeholders) with IT infrastructural investment in larger-scale HPC systems. Ownership of specific resources was reflected in preferential access to those resources, and opportunistic access to all idle computational resources — regardless of investment level — was a value-added benefit at scale. The "DARWIN" HPC cluster was purchased with an NSF MRI grant and was to facilitate access to HPC resources locally, regionally, and nationally. Interested parties would submit requests for resource allocations on the system: a major departure from the stakeholder-oriented model previously employed.

An *allocation* was defined to encompass:

- a single category determining resource limit range and purpose/applicability
- a single resource type: traditional CPU, GPGPU coprocessor, storage
- a time period (usually 1 year)

An allocation maintains a balance of some arbitrary unit of computational work with credits and debits made against it. The parity of this with the XSEDE/ACCESS definition of an allocation meant that a single implementation could handle allocations made by both University and XSEDE/ACCESS committees.

Design Goals

End-user and sysadmin experience with the Slurm job scheduler on the "Caviness" cluster was overwhelmingly positive, so the "DARWIN" cluster would carry forward that choice. Any changes in functionality necessary to implement the allocations system would hopefully be realizable using the various plugin APIs provided to extend Slurm: alterations to the base source code could hopefully be avoided.

A foundational design tenet of an allocations system is the granularity at which resource consumption is resolved. The absolute ideal would be for the job scheduler itself to handle the accounting, but if that facility is absent or deficient it must be implemented externally. Perhaps the simplest external solution is to periodically aggregate resource usage over the last unaccounted interval of time and debit it from the allocation balance; an allocation carrying a zero or negative balance cannot execute work in subsequent periods. One major issue with this

mechanism is that an allocation could incur a negative balance during the unaccounted time interval, which equates with using resources not granted. The shorter the interval between accounting runs, the smaller the magnitude of impact possible. As the interval duration approaches zero, though, the computational burden increases significantly. The situation can be improved by making one mandate: in addition to core counts, GPU counts, and memory limits, every job must supply a finite time limit at submission. From these data a maximum allocation *pre-debit* is calculable for each job. If the pre-debit does not yield a negative balance, the job may execute; otherwise, the job is denied. When the job completes execution, the real amount of elapsed time is used to calculate the *actual debit* to the allocation: if the actual debit is less than the pre-debit, the unused quantity is effectively credited back to the allocation.

It is important to define how the system associates a job with the allocation against which it should be billed. The Slurm job-scheduling environment on our HPC clusters requires jobs be submitted from a shell running under a *workgroup* with access to the system. A workgroup is an abstraction on top of a Unix group (with gid name and number) and a Slurm account (of the same name). The gid of the submitting process is associated with the job record in Slurm and processes associated with the job will execute with that gid. If the user does not supply an explicit Slurm account at job submission, the workgroup name is used. Slurm job submission and CLI filter plugins ensure these policies are enforced on each job. Since every job will thus have a workgroup associated with it, tying allocations to workgroups provides an easy link between a job and its billing target. Since an allocation is defined with regard to a specific resource type, the Slurm *partition* for the job determines the resource type: jobs running in GPGPU-oriented partitions bill against a GPGPU allocation, traditional CPU-oriented partitions bill against a CPU allocation. So the workgroup and partition form a complete association with an allocation *as long as a single matching allocation exists during any instant in time*. These qualifications provide a uniqueness constraint on allocations: for any combination of workgroup, resource type, and time period, there can be no other allocation with the same workgroup and resource type with an overlapping time period. Thus, the job's workgroup and partition in concert with the current date and time determine what allocation will be billed. It is important to note that a job whose execution period straddles two distinct allocation periods will incur a pre-debit on the earlier of the two allocations, so the actual debit must be against the same.

Service Units

Each resource type must have a billable unit of work defined in arbitrary *service units*. For CPU-oriented resources an amount of time on a single CPU core is used. The total available SU for a CPU-oriented resource is the total number of CPU cores multiplied by a period of time; for example, annual CPU SU in core•hours is the number of cores times 8760 hours. For GPU-oriented resources, the formula instead uses total GPGPU devices (since the system is not configured to grant access to subsets of those devices' resources). Allocations committees grant credits in terms of these SU definitions and at the granularity of an hour of time.

The job scheduler grants resource usage at the resolution of one second. Consideration must be given with respect to how to resolve, for example, 55 core•seconds of CPU usage against an allocation made in core•hours: is the usage rounded-up or rounded down, or should floating-point values be used rather than integers? For the sake of avoiding issues with imprecise zero, integer values are preferable. The job cited above would be billed either as 1 core•hour or 0 core•hour, which either severely penalizes short jobs or rewards them unfairly. DARWIN is a general-purpose cluster and not specifically designed for high-throughput scenarios, so billing by the core•minute with round-up of the scheduler's tracked core•second usage seemed the most appropriate compromise. While allocations committees will issue credits in an SU defined in terms of hours, internally the system will do its accounting in terms of minutes.

In order to make use of the CPU or GPU resources in any node the job must have RAM available to it, as well. RAM usage is not accounted for in the SU definition, but it does have a strong relationship with availability of the SU resources: a job that requests a single CPU core and all of the available RAM in a node blocks other jobs from running on the unused cores. In Slurm this issue is resolved by switching the billing formula from being the sum of weighted accounted resources to being the maximum weighted resource (by adding MAX_TRES to the PriorityFlags). Billing weights are assigned such that each CPU (or GPU) equates with a fraction of available RAM. For a 64 core node with 128 GiB of available RAM, the weights would be 1.0 and 0.5, respectively, and a job requesting (1 core, 64 GiB) would be billed exactly the same as a job requesting (32 core, 1 GiB) since both effectively consume half of the node's resources. In addition, each GPU must also equate with a fraction of CPU cores: for 64 cores, 128 GiB RAM, and 4 GPGPU devices, the weights would be 0.0625, 0.03125, and 1.0, respectively.

This design does have implications on the partitions that must be present on DARWIN.

Partitions

Billing formulae are set independently for each partition in Slurm. As mentioned above, we already must restrict partitions to encompass a single allocatable resource type (CPU or GPU); to properly equate RAM with CPU core count, all nodes in a partition must also have equal RAM, CPU core, and GPGPU device counts. These criteria led to the creation of eight distinct partitions: 4 CPU partitions of differing RAM footprint and 4 GPU partitions of differing GPGPU device kind.

Implementation

The collection of software associated with implementing the DARWIN allocation system has affectionately been named the Hermes Allocation Data Manager. Hermes is in reference to the accountant of the same name in the Futurama television show. The acronym should be pronounced "hate 'em" in deference to how much sysadmins enjoy doing the tasks it performs.

Database

A PostgreSQL[5] database was used to hold the allocation data. The database server is run inside a Singularity container. An external directory is bind-mounted into the container at the expected PGDATA path.

Each workgroup (project in the XSEDE/ACCESS parlance) is represented by a record in the "projects" table:

KEY	TYPE	DESCRIPTION
project_id	integer, primary key	additional integer id orthogonal to Unix gid number; used for Lustre quotas, foreign key references in this database
gname	text	the name of the workgroup's Unix group
gid	integer	the gid number of the Unix group
account	text	the Slurm account name associated with the workgroup

Schema 1: "projects" table

Projects can have any number of current and historic allocations present in the database:

[5] https://www.postgresql.org

KEY	TYPE	DESCRIPTION
allocation_id	integer, primary key	for foreign key references in this database
project_id	integer -> projects.project_id	referential link to a "projects" tuple
category	text	the allocation category
resource	text	the resource type (CPU or GPU)
start_date	timestamp	the date on which the allocation becomes active
end_date	timestamp	the date on which the allocation expires

Schema 2: "allocations" table

All transactions against an allocation balance share some common properties: the allocation to which they apply, the SU amount, and optional comment text. A base table is used to describe these fields:

KEY	TYPE	DESCRIPTION
activity_id	integer, primary key	for foreign key references in this database
allocation_id	integer -> allocations.allocation_id	referential link to an "allocations" tuple
amount	integer	an SU count
comments	text	optional text describing the transaction

Schema 3: "activity" table

A credit to an allocation requires no additional fields, so the "credits" table simply inherits the "activity" table.

A pre-debit to an allocation builds on the "activity" table it inherits:

KEY	TYPE	DESCRIPTION
job_id	integer	job id assigned by Slurm
status	text	status of Slurm job: executing, completed
owner_uid	integer	Unix uid number associated with the job

Schema 4: "predebits" table

Once a pre-debit is resolved it moves into the "debits" table as an aggregate by owner_uid (with the amount field being the total SU debits associated with that user):

KEY	TYPE	DESCRIPTION
owner_uid	integer	Unix uid number associated with the job

Schema 5: "debits" table

Finally, all submitted jobs that fail due to an insufficient allocation balance are moved from the "predebits" table to the "failures" table:

KEY	TYPE	DESCRIPTION
job_id	integer	job id assigned by Slurm
owner_uid	integer	Unix uid number associated with the job

Schema 6: "failures" table

The schema includes several views, most notably a "balances" view that sums across all debit/pre-debit/credit activities for each distinct allocation.

The full schema makes extensive use of stored procedures to validate tuples when they are inserted and updated (to prevent introduction of overlapping allocations, for example). One particular difficulty that was encountered during implementation and testing was serializing access to the "activity" cluster of tables. Since two separate sessions have shared access to query the table, two jobs requesting 20 SU against an allocation with a balance of +30 SU could both succeed before one of them manages to commit its pre-debit. This could lead to (possibly extensive) overdraft on an allocation. Initially an SQL exclusive access lock was acquired by the procedure prior to querying the "balances" view and dropped on exit from the procedure per the SQL standard. In testing this worked fine, but at scale we observed cyclic contention for the lock among multiple sessions that would effectively deadlock the PostgreSQL server. The procedure was altered to use PostgreSQL *advisory locks* (similar to POSIX locks) which can be acquired/dropped explicitly in the procedure. With advisory locks, no such contention or deadlock has been observed at scale. We eventually extended use of the advisory lock to all stored procedures that modify the distribution of SUs among the tables to avoid under- and over-counting in overlapping calls to those procedures.

Web API

A RESTful interface to the PostgreSQL database was created using Python[6] and the Flask[7] module. URL paths are used to query projects, allocations, and activity records and (in some cases for some remote users) alter those records. There are three levels of privilege present:

- superuser: create, read, update, and delete all records; run as alternative user
- admin: read all records; run as alternative user
- standard: read records associated with projects to which the user is a member

Since the cluster already uses the MUNGE daemon[8] for credentialed signing and authentication of Slurm RPCs, it makes sense to reuse that facility for authentication in this context, as well. Every HTTP request must have a custom header providing a MUNGE signature that the Python code can verify; if decoded and verified with MUNGE, the sender uid on the signature is the user associated with the HTTP request. Membership in the superuser and admin role is configured directly in the Python application.

URL Path Hierarchy

The REST API is documented in the source code, currently posted online at https://gitlab.com/udel-itrci/hadm/. The URL paths used in the web API are constructed with the target entity or information as the first component: /project, /alloc, /job, /failure. A GET request to each path yields a list of objects filtered by the requesting user: all objects are visible to users with the superuser or admin roles, while regular users are limited to objects associated with their user or group ids. Each returned object has a distinct numerical identifier used to interact with it, e.g. /project/129. In most cases, GET requests can include query parameters (e.g. gid=<gid-#>) to limit the scope of the search.

The sproject utility

To simplify use of the WebAPI for users and staff, a Python utility was written that generates REST queries from CLI arguments and displays the resulting JSON in a tabular format. Some filtering, sorting, and collation outside the WebAPI's inherent capabilities is present (e.g. show only currently-active allocations).

---

[6] https://www.python.org
[7] https://flask.palletsprojects.com/en/stable/
[8] https://dun.github.io/munge/

Slurm Plug-ins

A variety of Slurm plug-ins were created to ensure all pre-conditions necessary for allocation processing are met.

CLI filter

A *CLI filter* plug-in[9] validates job parameters on the submission node prior to the job record's being sent to the Slurm controller daemon (slurmctld). The DARWIN plugin:

- sets the "account" parameter to the Unix group name if absent
- checks that the "account" parameter matches the Unix group name
- for short-term allocations, forces the "partition" and "reservation" parameters to match the Unix group name
- ensures one and only one "partition" parameter has been provided
- advise the user that "--gpus" should be used and not "--gres=gpu"
- check that the subtype provided with "--gpus" and the selected partition embody a consistent GPGPU device type
- overwrite any "qos" parameter provided by the user with our desired "qos"

If any errors are present in the job submission, the CLI filter plug-in can notify the user before the job has consumed a job id and any processing time in slurmctld. Error messages produced by the plug-in provide guidance on remedying the issue preventing submission.

Job submission

A *job submission* plugin[10] runs in slurmctld and enforces the same checks as the CLI filter plugin. In addition:

- if no "--time-min" was provided, set it to the job's time limit
- if no memory limit parameter was provided, use a default value per CPU

The backfill scheduler always looks to fit jobs into available resource time slots. This can go awry in some cases if there is no minimum time provided for the job: the requested walltime effectively ranges from 0 up to the job time limit and any amount of time on the required resource footprint is a match. Users who do not explicitly communicate a "--time-min" value in a submission are assumed to want the full time limit and nothing less.

[9] https://slurm.schedmd.com/cli_filter_plugins.html
[10] https://slurm.schedmd.com/job_submit_plugins.html

The job does not yet have a job id assigned at this stage.

PrEp

A *Prolog-Epilog (PrEp)* plug-in[11] runs within those contexts of job dispatch in slurmctld. When slurmctld selects a pending job for execution, the prolog function is executed; when slurmctld is told the job has completed execution, the epilog function is executed.

Web API queries will be used to interact with the database, so the prolog/epilog steps will necessarily wait on a response from the server. A straightforward procedural model for these queries will introduce a bottleneck in the slurmctld job processing pipeline: each job will block while a response from the server is pending. If the response is a timeout or other inconclusive failure, the web API query will need to be repeated, further delaying subsequent job completions. Slurm does build-in an asynchronous completion model for the PrEp plugin, whereby the prolog or epilog can use callbacks to update and progress the job records asynchronously. Rather than spawning each prolog or epilog web API procedure in a separate thread, a pool of threaded query handlers is employed. The pool has a minimum and maximum number of worker threads and can expand/contract based on wait times for queued requests from the PrEp plug-in. On shutdown, any pending queries are serialized to disk, and on startup any queries present on disk are deserialized into the work queue.

In the prolog function, Slurm has already calculated (based on partition and requested resource levels) the projected billable SUs for the job. The job also has a job id assigned to it, so all information necessary to introduce a pre-debit is present:

- job_id
- owner_uid
- amount
- resource type (from partition)
- project (from account/group name)

A web API query for pre-debit is created and submitted asynchronously to the work queue. When the query completes, the job will be set for acceptance or rejection and progressed to its next state.

[11] https://slurm.schedmd.com/prep_plugins.html

In the epilog function, Slurm has updated the job record's billable SUs to match with the actual wall time consumed by the job. If the job state indicates that it ended due to node failure, then we do not bill for the job: the SU usage is set to zero. A web API query for completion is created and submitted asynchronously to the work queue. When the query completes the job is progressed to its next state.

Site factor

A *site factor* plugin[12] introduces a value external to Slurm's own metrics into the calculation of job–scheduling priorities. In the context of an allocation-based system, one biasing factor that could be exploited to improve user experience is the magnitude of remaining credit and the time until expiration of the allocation. For the sake of fulfilling consumption of allocated credit, jobs associated with an allocation nearing its expiration must run sooner than jobs with a later expiration date. In concert with that prioritization factor, the magnitude of remaining credit should influence priority: with similar expiration dates, an allocation with just 5% remaining should receive lower priority than one with 95% remaining.

It is important to note that a site factor plugin has *not* been implemented on DARWIN: it is a useful feature that could be exploited in the future.

Calculation of the two factors for active allocations indexed by i follows the formulae:

$$f_{credit,i} = 1 - \frac{C_{used,i}}{C_{alloc,i}}$$

$$f_{temporal,i} = 1 - \frac{\Delta t_{expire,i}}{max(\Delta t_{expire,1},\ldots,\Delta t_{expire,N})}$$

where $C_{used,i}$ and $C_{alloc,i}$ are the credits consumed and granted to an allocation, respectively; $\Delta t_{expire,i}$ is the time interval until expiration of an allocation; and $max(\Delta t_{expire,1}, \ldots, \Delta t_{expire,N})$ is the longest time interval until expiration over all allocations. These factors are in the closed interval $[0, 1]$ for all allocations. Both formulae are linear; alternative formulations that bias the behavior are permissible. Two examples are presented here.

[12] https://slurm.schedmd.com/site_factor.html

Power rule

Rather than using the fractional factors directly the value can be exponentiated: for example, squaring the value biases higher values of the fraction over lower values. In other words, for $f_{credit,i}$ between one allocation and another with twice the unused credits the latter is given four times the priority over the former. The values still fall within the closed range $[0, 1]$.

In one form, the product of the two fractional factors is raised to the power N to yield the site factor. Two example contour plots appear in Figure 1:

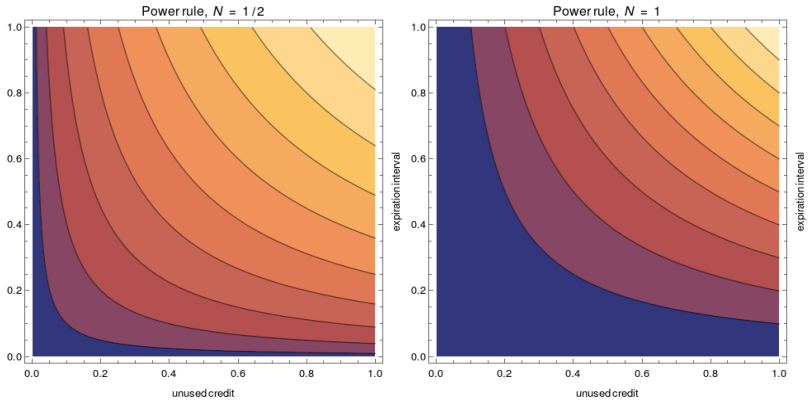

Figure 1: Two example power rule formulae for the site factor.

By construction this site factor will always be symmetric; the same weight is given to differences in temporal factor as is given to differences in credit factor. It is likely to be beneficial to bias the weighting of one factor versus the other; for example, making the site factor grow more quickly with respect to impending expiration. This asymmetry is affected by making the site factor, ς, the product of the fractional factors each raised to a distinct exponent, N_t and N_C.

$$\varsigma(f_{c,i}, f_{t,i})_{N_c,N_t} = f_{c,i}^{N_c} f_{t,i}^{N_t}$$

The choice of parameters in Figure 2 uses the square root to stretch the site factor response to $f_{credit,i}$ and bias the response to $f_{temporal,i}$ to favor allocations closer to their expiration date.

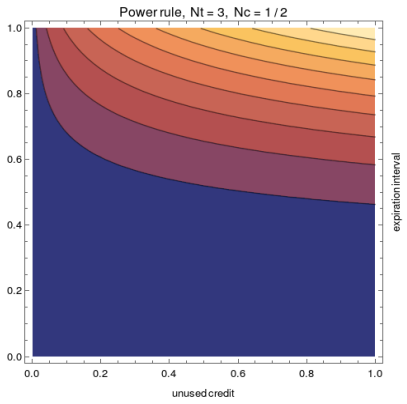

Figure 2: Asymmetric power rule formula.

In terms of implementation in a site factor plug-in, the power rule would require two adjustable parameters. The code could be as simple as two calls to the C pow() function, or as complex as recognizing an exponent's being an integer that could be optimally implemented as a looped multiplication (e.g. a single floating-point multiplication for $N = 2$ or a NOOP for $N = 1$).

Sigmoid

A sigmoid function (or logistics function) can be defined as

$$S(x)_{l,x_0} = \frac{A}{1 + e^{-l\,(x - x_0)}} - B$$

where the parameters l and x_0 determine the transition rate and midpoint of the curve. For our purposes there exist two conditions that must be satisfied

$$S(0)_{l,x_0} = 0\,;\; S(1)_{l,x_0} = 1$$

for the fractional values to map to the allowable range $[0, 1]$. These two equations can be solved for A and B and the function rewritten as:

$$S(x)_{l,x_0} = \frac{(e^{l\,x} - 1)(e^{l} + e^{l\,x_0})}{(e^{l} - 1)(e^{l\,x} + e^{l\,x_0})}$$

The behavior of the function under variation of the two parameters is illustrated in Figures 3 and 4:

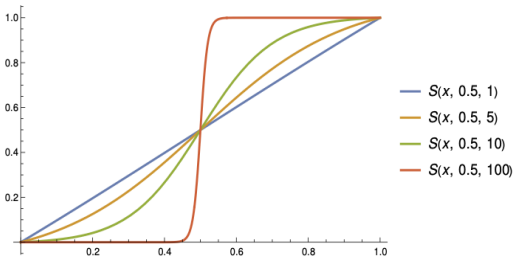

Figure 3: Variation of parameter l

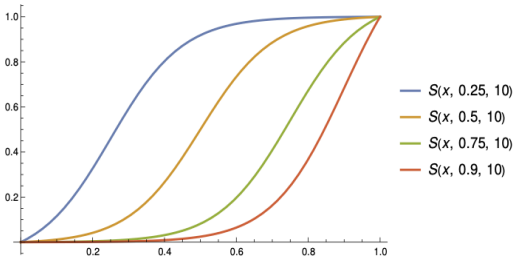

Figure 4: Variation of parameter x_0

A value of $l = 1$ approximates a linear function of unit slope and $l = 0$ a linear function of zero slope. Note that as the magnitude of parameter l increases, the sigmoid more closely mimics a step function that transitions at x_0:

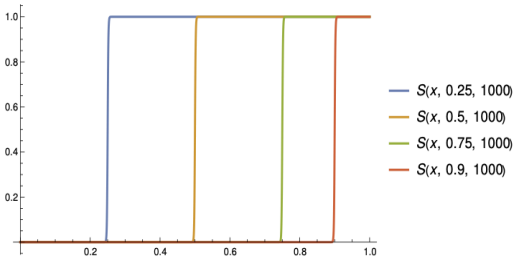

Figure 5: Sigmoid as a step function

A site factor defined as the composition of two sigmoid functions is generally asymmetric and requires four parameters.

$$\varsigma(f_{c,i}, f_{t,i})_{l_c, l_t, x_{0,c}, x_{0,t}} = S(f_{c,i})_{l_c, x_{0,c}} \; S(f_{t,i})_{l_t, x_{0,t}}$$

Symmetry is a special case realized by reduction to only two adjustable parameters ($l_c = l_t$ and $x_{0,c} = x_{0,t}$). Consider a parameterization that heavily favors allocations nearing their expiration in concert with a wide gradient of prioritization with respect to level of unused credit (Figure 6):

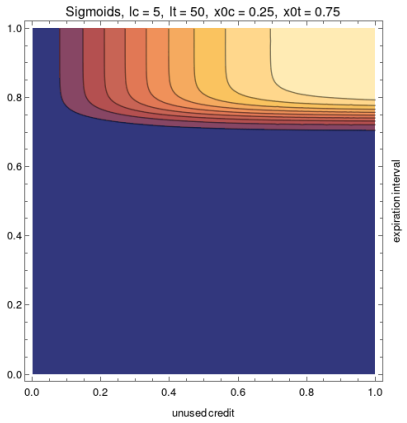

Figure 6: Sigmoid site factor with $l_c = 5$, $l_t = 50$, $x_{0,c} = 0.25$, $x_{0,t} = 0.75$

Until an allocation reaches 75% of its allocation period, the degree of credit remaining plays no role in site factor prioritization.

Summary

DARWIN's HADM facility has been in operation for 1428 days as of the writing of this document. In that time it has encompassed or processed:

- 321 unique projects/workgroups, 216 of which are XSEDE/ACCESS
- 714 unique allocations
- 5,525,365 user jobs
- 45,289,179 CPU•hours (74% avg. utilization)
- 117,214 GPU•hours (57% avg. utilization)

The Slurm PrEp plugin and HADM WebAPI daemon have operated reliably with zero downtime. Staff and users alike have benefitted from the *sproject* utility to easily manage allocations: checking balances, noting usage by user, and explaining job failures due to insufficient credits.

HADM presents a feature-rich, well-tested option in the implementation of future allocations-based systems at the University. With a non-trivial amount of work the database backend could be modified, for example to use a NoSQL system like MongoDB or Redis;

pivoting to MariaDB/MySQL or another SQL system would also be permissible, although they typically lack parity with the flexibility of PostgreSQL. Though very few user complaints have been voiced with regard to the "fairness" of scheduling priorities, the influence of remaining balance and time could effect an improvement via the site factor.