Budget Checking Plugin for SLURM

Huub Stoffers  huub.stoffers@surfsara.nl
HPC systems expert, project lead Cartesius Supercomputer

SLURM USER GROUP MEETING 2014, SEP. 23-24TH LUGANO

Replaced IBM P6

In production since June 2013

First, and so far only, system at our site to use SLURM
Ideas on strict budget checking applied right before job dispatching in SLURM, with elements of a site presentation mixed in:

• A brief overview of the compute facilities

• Some aspects of the SLURM configuration on Cartesius:
  - The partitioning applied to the system, and how that fits our needs
  - QOS policies that, in addition to the partition attributes, also act as partition resource usage limits

• Accounting at SURFsara:
  - The basis on which users are granted access and how budgets to use resources are determined
  - The tracking of their resource usage – that is done pretty well by SLURM
  - “Pricing” of resources, or how the resource usage is reduced to “SBU” deductions from project budgets
  - Budget restitution decisions and other events, that are not directly in view of the batch system, that can affect the remaining budget of a project “from the outside”

• ‘live’ budget checking on top of the configuration:
  - What we have in place for that right now
  - why that implementation is not good enough
  - What sort of “logic” would be more efficient and scalable
  - our ideas on how to implement it in the context of the SLURM environment
### (SURF)sara National Supercomputing History

<table>
<thead>
<tr>
<th>Year</th>
<th>Machine</th>
<th>batch</th>
<th>( R_{\text{peak}} ) GFlop/s</th>
<th>kW</th>
<th>GFlop/s / kW</th>
</tr>
</thead>
<tbody>
<tr>
<td>1984</td>
<td>CDC Cyber 205 1-pipe</td>
<td></td>
<td>0.1</td>
<td>250</td>
<td>0.0004</td>
</tr>
<tr>
<td>1988</td>
<td>CDC Cyber 205 2-pipe</td>
<td></td>
<td>0.2</td>
<td>250</td>
<td>0.0008</td>
</tr>
<tr>
<td>1991</td>
<td>Cray Y-MP/4-128</td>
<td>NQS</td>
<td>1.33</td>
<td>200</td>
<td>0.0067</td>
</tr>
<tr>
<td>1994</td>
<td>Cray C98/4-256</td>
<td>NQS</td>
<td>4</td>
<td>300</td>
<td>0.0133</td>
</tr>
<tr>
<td>1997</td>
<td>Cray C916/12-1024</td>
<td>NQS</td>
<td>12</td>
<td>500</td>
<td>0.024</td>
</tr>
<tr>
<td>2000</td>
<td>SGI Origin 3800</td>
<td>LSF</td>
<td>1,024</td>
<td>300</td>
<td>3.4</td>
</tr>
<tr>
<td>2004</td>
<td>SGI Origin 3800 + Altix 3700</td>
<td>LSF</td>
<td>3,200</td>
<td>500</td>
<td>6.4</td>
</tr>
<tr>
<td>2007</td>
<td>IBM p575 Power5+</td>
<td>LL</td>
<td>14,592</td>
<td>375</td>
<td>40</td>
</tr>
<tr>
<td>2008</td>
<td>IBM p575 Power6 (104 nodes)</td>
<td>LL</td>
<td>62,566</td>
<td>540</td>
<td>116</td>
</tr>
<tr>
<td>2009</td>
<td>IBM p575 Power6 (108 nodes)</td>
<td>LL</td>
<td>64,973</td>
<td>560</td>
<td>116</td>
</tr>
<tr>
<td>2013</td>
<td>Bull bullx B710 (DLC) + R428</td>
<td>SLURM</td>
<td>270,950</td>
<td>245</td>
<td>1106</td>
</tr>
<tr>
<td>2014</td>
<td>+ Bull bullx B515 (NVIDIA K40m)</td>
<td>SLURM</td>
<td>210,000</td>
<td>44.4</td>
<td>4729 ( ! )</td>
</tr>
<tr>
<td>2015</td>
<td>Bull bullx ‘complete system’</td>
<td>SLURM</td>
<td>&gt;1,400,000</td>
<td>&gt;700</td>
<td>&gt;2000</td>
</tr>
</tbody>
</table>
(SURF)sara has always hosted and managed other HPC and “Big Data” facilities, besides the Dutch national supercomputer.

- **Systems for specific communities:**
  - LISA ➔ VU + UvA + NWO
  - Grid ➔ National Life Sciences Grid + BigGrid + EGI

- **Systems tuned to a special purpose:**
  - Hadoop cluster
  - Visualization render cluster
  - HPC cloud
  - Multi-petabyte (tape) archive facility

- Some share a common user administration with the national super computer
- Facilities have their own independent scheduling and/or resource reservation systems
- Resource usage records post-processed by the central SURFsara accounting server
- LISA is closest to Cartesius in mode of operation, but uses Torque
Cartesius phase 1
(June 2013 – June 2014)

SLURM User group meeting 2014, Sep. 23-24th Lugano
Cartesius phase 1 + GPU Island (June 2014 - )

SLURM User group meeting 2014, Sep. 23-24th Lugano
We try to keep resource usage limits and job prioritizing simple:

- Basic scheduling, First In First Out, with backfilling
- No preempting and suspending of running jobs
- No fair share rules – we would not know how to define what is “fair”
  - Fair with respect to users, or with respect to accounts?
  - Fair with respect to short term usage or with respect to with respect to the size of a project and what is supposed to do within its lifetime?
- No more resource usage limits than necessary
- Just try to prevent that one account (project) can monopolize the usage of a particular system
- Add more rules and policies when it turns out they are needed
## SLURM configuration on Cartesius (2/2) - Partitions

<table>
<thead>
<tr>
<th>Partition</th>
<th># Nodes</th>
<th>Node usage</th>
<th>MaxNodes</th>
<th>MaxTime (min.)</th>
<th>MaxNodesPU (QOS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>All TCNs -16</td>
<td>Exclusive</td>
<td>360</td>
<td>7200</td>
<td>360</td>
</tr>
<tr>
<td>Short</td>
<td>All TCNs</td>
<td>Exclusive</td>
<td>480</td>
<td>60</td>
<td>480</td>
</tr>
<tr>
<td>Fat</td>
<td>All FCNs</td>
<td>Exclusive</td>
<td>16</td>
<td>7200</td>
<td>48</td>
</tr>
<tr>
<td>Staging</td>
<td>All SRVs</td>
<td>Shared</td>
<td>1</td>
<td>7200</td>
<td>N.A.</td>
</tr>
<tr>
<td>GPU</td>
<td>All GCNs – 2</td>
<td>Exclusive</td>
<td>48</td>
<td>7200</td>
<td>48</td>
</tr>
<tr>
<td>GPU_short</td>
<td>All GCNs</td>
<td>Exclusive</td>
<td>64</td>
<td>60</td>
<td>64</td>
</tr>
</tbody>
</table>

- 16 TCNs and 2 GCNs are in their respective “short” partition to ensure that there are always some nodes available for short test runs within an hour.
- We use a `MaxNodesPerUser` “sacctmgr” limit, via a QOS per partition. We would rather have it `per Account` though.
- Not all users have access to all partitions. We use “sacctmgr” associations to grant/limit access.
Nowadays there are two ways to get access and a budget:
1. Write a proposal and get it approved by the NWO council
2. Since a few years also, via PRACE, write a DECI proposal

- We are expected to take care that projects get what they need, that they can spend the budget granted …
- … but also that they cannot use more than they were granted – the council deems overspending inadmissible
- Putting a price on resource is very site specific:
- Budgets are in terms of abstract core hours or “System Billable Units” (SBUs)
- Core hours of TCNs, FCNs, and SRVs have the same “price” of 1 SBU, but core hours on GCNs cost 3 SBUs:

<table>
<thead>
<tr>
<th>Node type</th>
<th>Node resource “package”</th>
<th>Whole node SBUs / wall clock hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCN</td>
<td>24 cores, 2 GiB/core</td>
<td>24</td>
</tr>
<tr>
<td>FCN</td>
<td>32 cores, 8 GiB/core</td>
<td>32</td>
</tr>
<tr>
<td>SRV</td>
<td>16 cores, 2 GiB/core + high perf. external network connectivity</td>
<td>16</td>
</tr>
<tr>
<td>GCN</td>
<td>16 cores, 6 GiB/core + 2 K40m GPUs</td>
<td>48</td>
</tr>
</tbody>
</table>
Accounting at SURFsara (2/5)

Economic capacity of the machine and project size, given the chosen “pricing”

<table>
<thead>
<tr>
<th>System</th>
<th>Economic capacity in SBUs per day</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current system (Phase 1 + GPUs)</td>
<td>&gt; 410,000</td>
</tr>
<tr>
<td>Complete system (Phase 2)</td>
<td>&gt; 1,030,000</td>
</tr>
</tbody>
</table>

- Pilot projects may get 100,000 – 200,000 SBUs
- Small projects get several 100,000 SBUs
- Large projects get several 1,000,000 of SBUs
- Ultimately also large projects will have little budget left …
- … But it is unwieldy, if doable at all, to dynamically adjust limits per project while budget is being spent …
• Rather keep track of the budget, however SLURM records job resource usage, but is not aware of SBU budgets and “pricing” …

• … But a central accounting server is

• Every 24 hours a “sacct” query is run, and a new batch of job records, that have completed since the last previously sent job record, is sent to the central accounting server

• The central accounting server processes the batch of records, converts resource usage into job cost in terms of “SBUs” and deducts that from budgets accordingly.

• Since jobs continuously spend while running, which can be up to five days, and post-processing is done only after the fact, when the job is done, budget adjustment at the central accounting server might be too late – when gross overspending has already happened.

• Other events, besides post-processing job records, may affect the remaining budget …
The central accounting server has an administrative (web)interface with several options for “bookkeepers”:

- Initialization of new projects, accounts
- Expiration of old projects – reducing the budget to 0
- Budget restitution for jobs for various reasons
- Transfer of budget from one project to another may also be a legitimate action in some cases

Cartesius runs an hourly cron job to retrieve updated account and associated budget state information from the central accounting server:
- To adapt “sacctmgr” accounts and associations of accounts and users with partitions to new projects and to the expiration of old ones
- To make use of in a budget check that is run at job dispatch time, in the SLURMctld prolog
Accounting at SURFsara (5/5)

Central Accounting Server
- administrative events
- post-processing job records
  - Updates to account and budgets
  - Updates To budgets
- job records

Cartesius (SLURM)
- Updates to user administration
- job records

LISA (Torque)
- ldap-sssd

Central User Administration (LDAP)
- ldap-sssd

Database, account and budget state
- Updates to account and budgets

Tape Archive Facility
- Updates to account and budgets
Current budget check implementation (1/2)

- A script that is called by SLURMctld prolog
- Job cost functions for all node types are hardcoded into the script
- The script determines the remaining budget from cached, hourly refreshed, data retrieved from the accounting server
- From these data it also determines the effective timestamp of the remaining budget, i.e.: the latest end time of jobs already post-processed by the accounting server and hence already deducted from the budget
- It calls “sacct” to retrieve all jobs that have finished since the last post-processed job
- It calculates the actual job cost of all these jobs, on the basis of their actual resources and actual runtime, and deducts this amount from the budget
- It calls “squeue” to retrieve all running, still unfinished, jobs of the account including the job in in the process of being dispatched to run
- It calculates the maximum job cost of these unfinished jobs on the basis of their actual resource allocation and their maximum runtime, and deducts this amount from the budget too
- If the resulting budget is zero or negative, the job is cancelled, otherwise it runs
Current budget check implementation (2/2)

- In principle, it works well, correctly.
- But it is not very efficient and hence not very scalable.
- It results in a lot of “squeue” and “sacct” queries.
- Each successive job dispatch retrieves the same data over and over again, that are only slightly incremented and changed with information of meanwhile finished and newly dispatched jobs.
- And it recalculates the same job cost over and over again.
- Towards the moment of send post-processing a new batch of job records by the accounting server the work to be done by the check is ever increasing.
- On “really bad days” it does not work at all and can even get the SLURMctld into trouble:
  - Bad days are:
  - When there are a lot of “farmers”, running many small short jobs.
  - When, in addition, there are some moments at which many such jobs can be dispatched at virtually the same time many squeue and sacct queries retrieving huge record sets will run in parallel.
A better organization (1/3)

- Split the work, cache and keep track of the remaining account budgets
- Do the work that the current script does only *once* for per account to produce something like this:

```c
struct budget_state {
    char  *accountname;
    time_t  timestamp;
    long   base_budget;
    long   remaining_budget;
};
```

- Keep it somewhere were you can do atomic “transactional” updates on the record:
  - Two times per job: viz. at dispatch time, and at completion time
- Originally I thought the SLURM “sacct” database should be extended hold such records, but it could be some file governed with e.g. `ioctl(2)` locking, or any other mechanism that avoids race conditions when updating the remaining budget.
A better organization (2/3)

- long jobcost(job_info_msg_t *jobinfo, int mode);
- Calculates either worst case or actual job cost, depending on mode, on the basis of the jobinfo record and site specific “pricing” rules.
- int init_budget_state(long base_budget, time_t timestamp, char *accountname);
- Do at sacct –S timestamp –A accountname sort of query, to retrieve every job of account accountname that has started since timestamp; In the list retrieved, there may be finished and unfinished jobs.
- Call jobcost with the respective mode for finished and unfinished job to. calculate the remaining budget and update an budget_state record
- int jobdispatch_chk(uint32_t jobID, char *accountname);
- Run at “prolog time”
- int jobcomplete_chk(uint32_t jobID, char *accountname);
- Run at “epilog time”
A better organization (3/3)

• At prolog time
  - Use a `slurm_load_job()` query to get data to calculate the maximum job cost only of the job being dispatched
  - “atomically”:
    - {
      - subtract the **maximum** job cost from the account’s remaining budget
      - If this brings the remaining budget below zero, cancel the job and do not update remaining budget
      - If not, then update the remaining budget with the subtracted maximum job cost
    }

• At epilog time
  - Use a `slurm_load_job()` query to get data to
    - calculate the **actual** job cost of the completing job
    - (re)calculate the **maximum** that was subtracted at dispatch time
    - “atomically” add the difference between maximum and actual job cost to the remaining budget

• Only if an external event changes the base budget, by cronjob getting fresh information from the accounting server, throw away the cached budget_state and start anew by complete recalculation, i.e. by reusing the init_budget_state routine.