Budget Checking Plugin for SLURM

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

Huub Stoffers huub.stoffers@surfsara.nl

HPC systems expert, project lead Cartesius Supercomputer

SURF SARA

Outline of this presentation

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

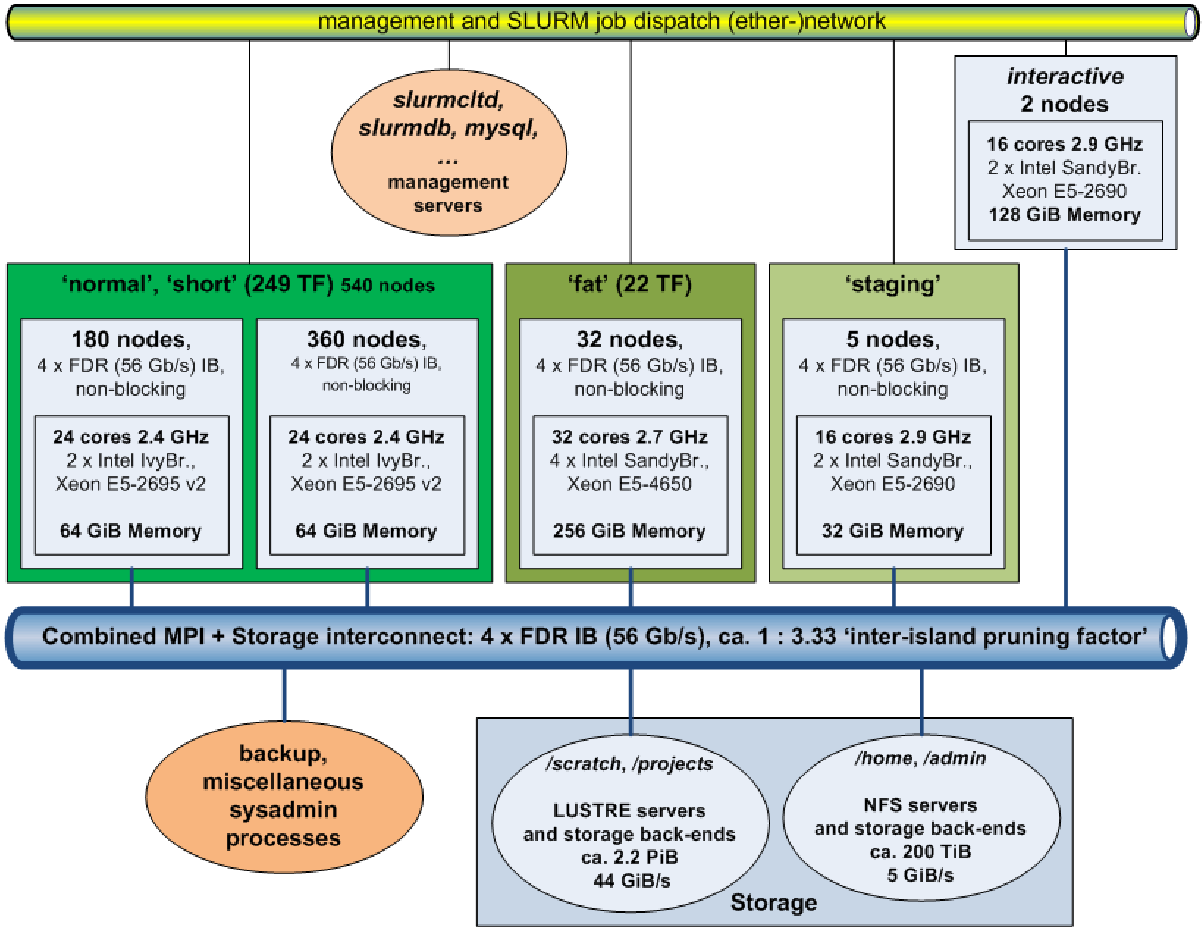
Year	Machine	batch	R_{peak} GFlop/s	kW	GFlop/s / kW
1984	CDC Cyber 205 1-pipe		0.1	250	0.0004
1988	CDC Cyber 205 2-pipe		0.2	250	0.0008
1991	Cray Y-MP/4-128	NQS	1.33	200	0.0067
1994	Cray C98/4-256	NQS	4	300	0.0133
1997	Cray C916/12-1024	NQS	12	500	0.024
2000	SGI Origin 3800	LSF	1,024	300	3.4
2004	SGI Origin 3800 + Altix 3700	LSF	3,200	500	6.4
2007	IBM p575 Power5+	LL	14,592	375	40
2008	IBM p575 Power6 (104 nodes)	LL	62,566	540	116
2009	IBM p575 Power6 (108 nodes)	LL	64,973	560	116
2013	***Bull bullx B710 (DLC) + R428***	***SLURM***	***270,950***	***245***	***1106***
2014	***+ Bull bullx B515 (NVIDIA K40m)***	***SLURM***	***210,000***	***44.4***	***4729*** (!)
2015	***Bull bullx 'complete system'***	***SLURM***	***>1,400,000***	***>700***	***>2000***

Other HPC systems at SURFsara

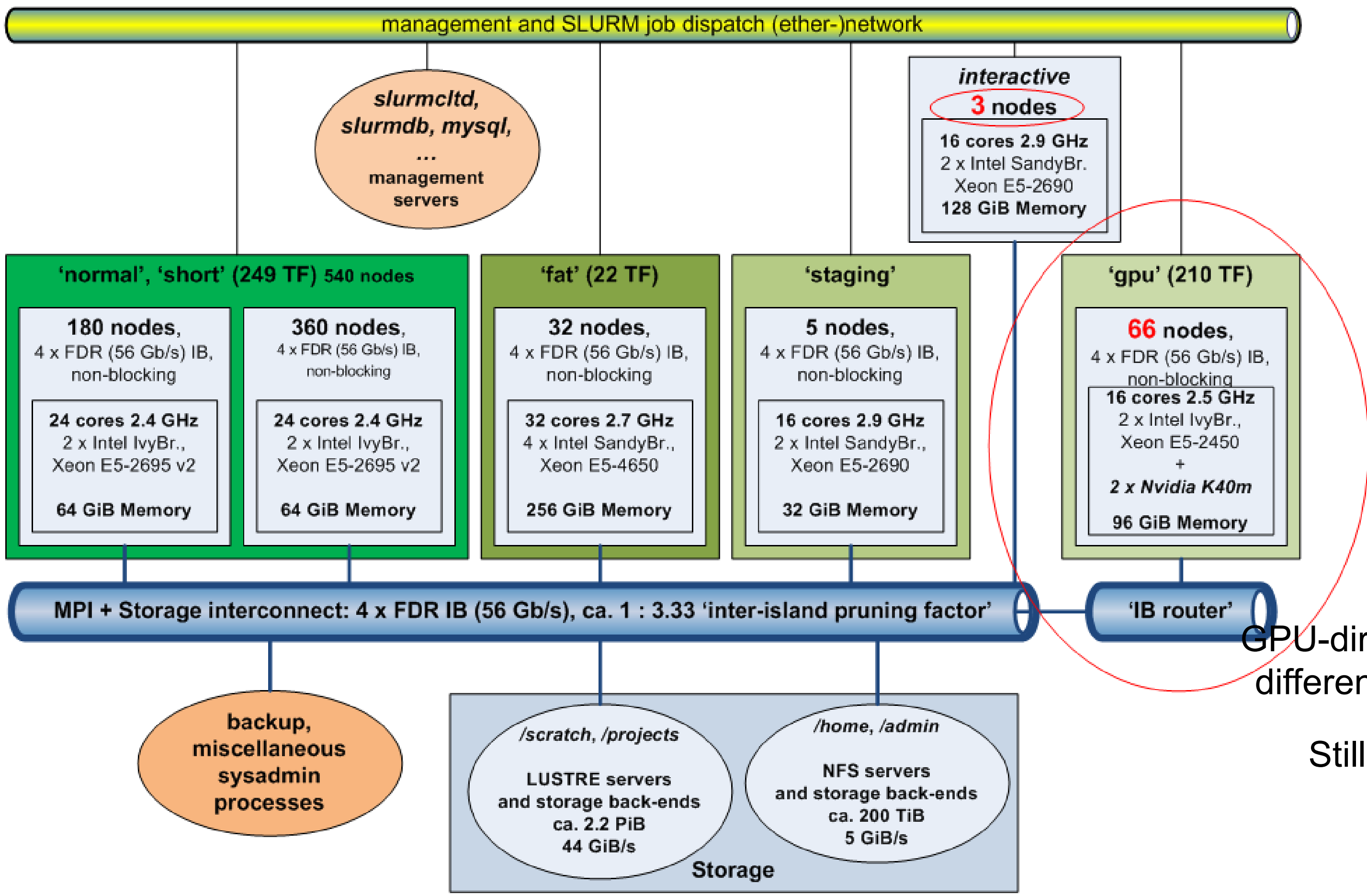
(SURF)sara has always hosted and managed other HPC and "Big Data" facilities, besides the Dutch national supercomputer

- **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)



Cartesius phase 1 + GPU Island (June 2014 -)

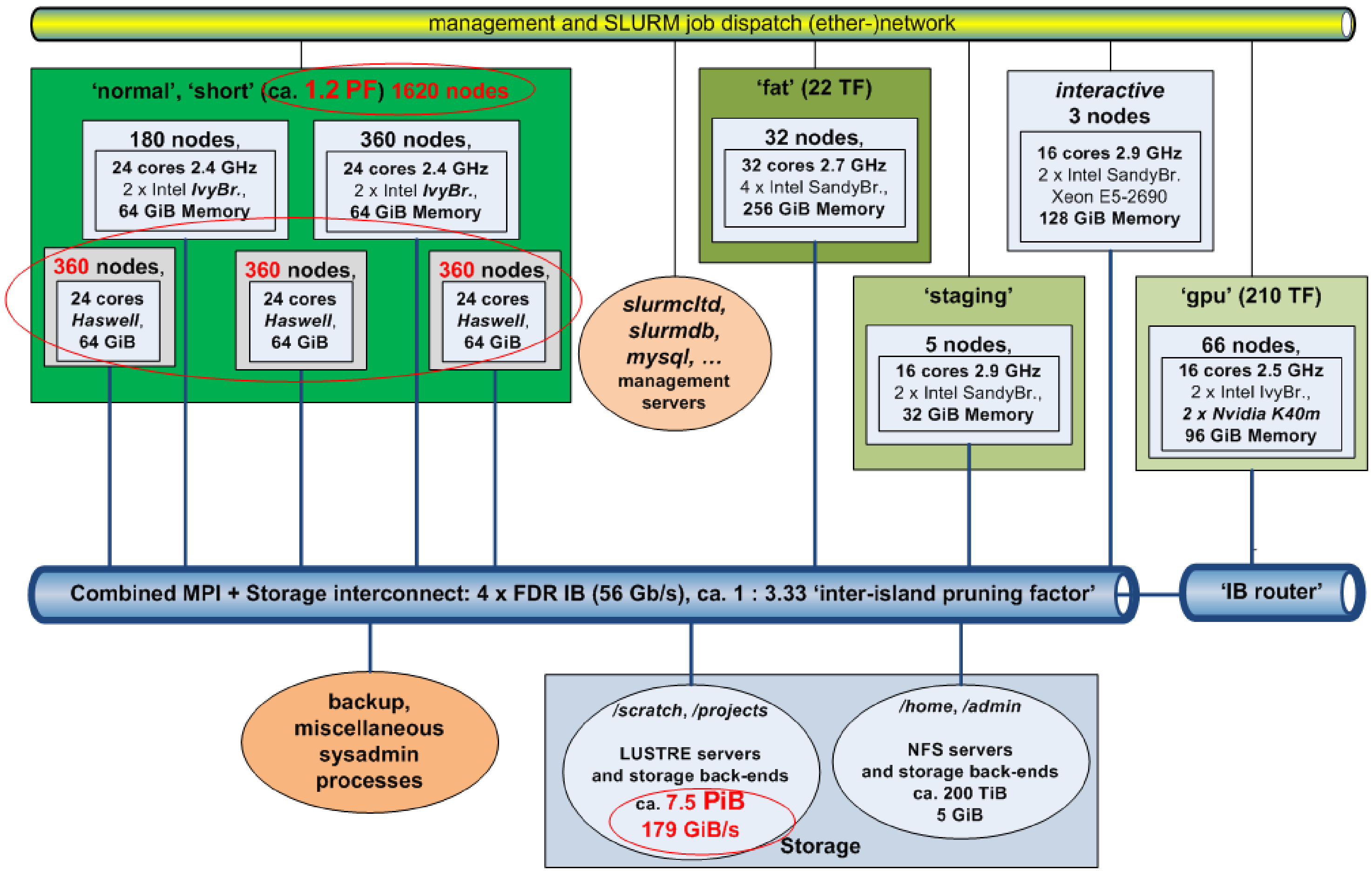


GPU-direct support, different ofed stack

Still one cluster for SLURM

SURF SARA

Cartesius phase 2 + GPU island (November / December 2014 -)



SLURM configuration on Cartesius (1/2)

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

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

- 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

Accounting at SURFsara (1/5)

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:

Node type	Node resource "package"	Whole nodeSBUs / wall clock hour
TCN	24 cores, 2 GiB/core	24
FCN	32 cores, 8 GiB/core	32
SRV	16 cores, 2 GiB/core + high perf. external network connectivity	16
GCN	16 cores, 6 GiB/core + 2 K40m GPUs	48

Accounting at SURFsara (2/5)

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

System	Economic capacity in SBUs per day
Current system (Phase 1 + GPUs)	> 410,000
Complete system (Phase 2)	> 1,030,000

- **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 …**

Accounting at SURFsara (3/5)

- 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 …

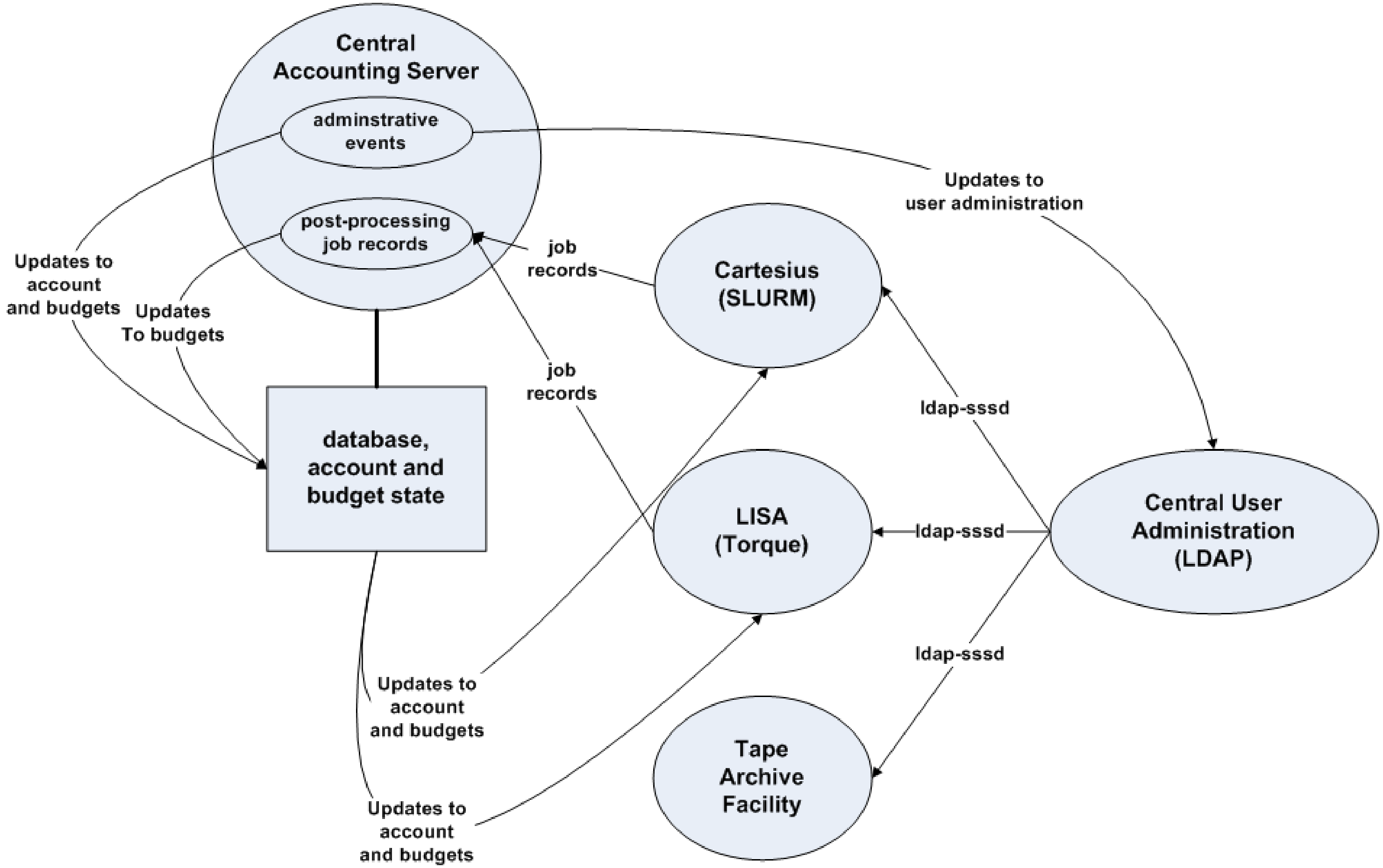
Accounting at SURFsara (4/5)

- 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)



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:

```
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.