Transitioning from LSF to SLURM

Slides available at: edu.sib.swiss

Robin Engler, SIB
Sébastien Moretti, SIB
Vassilios Ioannidis, SIB
Gregoire Rossier, SIB

Swiss Institute of Bioinformatics

With the invaluable help from the SIB Core-IT team!

Overview

- What is slurm.
- Slurm basics: nodes, partitions and tasks.
- Submitting jobs with *sbatch* and *srun*.
- sbatch / srun basic options.
- Requesting additional resources.
- Other useful slurm commands.
- Job arrays.
- LSF to slurm, script conversion examples.

Slurm - Simple **L**inux **U**tility for **R**esource **M**anagement

- Free and open source, unlike LSF which is proprietary (and expensive) software from IBM.
- Development started in 2002, currently > 500k lines of code.
- Used in many of the world's largest HPC centers.
- Concepts and commands similar to LSF ==> easy transition.

Job scheduler / resource management system / workload manager

The role of a job scheduler is to automate and optimize resource allocation when there is more work than resources.

Specifically, a scheduler is designed to:

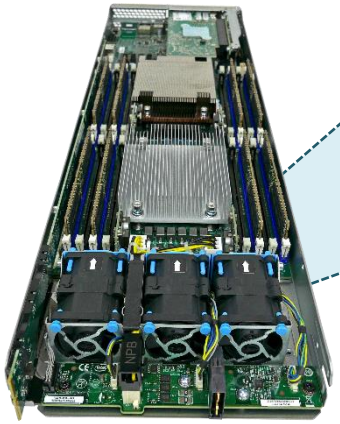
- **Optimize the assignment of jobs to individual compute nodes**, based on job hardware requirements and current hardware availability.
- **Management of job priorities**, based on user profiles and user job-specific parameters. For instance users that already have jobs running will see their priority decreased compared to those with no job running.
- **Monitoring of job execution and enforcement of resource limits**. Jobs that exceed certain set limits, e.g. taking too much time or using too much memory, are terminated. Users are notified when a job completed.
- **Accounting of resource usage** of individual users – used to generate your PI's monthly bill.

Slurm basics:
nodes, partitions and tasks

Compute nodes

Physical unit containing several CPUs, memory (RAM) and local storage.

- Nodes are the actual location where you job is executed.
- All CPUs of a same node have access to the same memory.



Single compute node with multiple CPUs

LSF equivalent:

LSF generally refers to **nodes** as **hosts**.

Slurm partitions

LSF equivalence

LSF refers to **partitions** as **queues**.

- Group of compute nodes to which a job is assigned by the user when submitting it to the cluster. A node can belong to more than one partition.

- Partitions differ in terms of:
  - **Job priority value**, influences how quickly the job starts running on the cluster.
  - **How long** a job can run (runtime limit).
  - **How much resources** can be requested (e.g. CPUs, memory).
  - **How many jobs** a user can run simultaneously.
  - **Who can access** the partition.

- The number and specifications of partitions is cluster specific, but systems are typically setup to have partitions with:
  - higher priority and shorter maximum runtime.
  - Lower priority and longer maximum runtime.

Slurm partitions (queues) on Wally and Axiom

- Currently 2 partitions exist: **normal** [max 24h] and **long** [max 10 days].
- Job priority: **normal** > **long**.
- **Warning**: default runtime < max runtime. This was not the case previously.
- The default partition is **normal**.

Default is half the max time !

Wally cluster:

Partition name (queue)	**CPU per node** [x* = unlimited within a node]	**RAM [per node]**	**Job run time**	**Concurrent job limit**
normal [default partition]	**1 node, 1 cpu [def]** **32 nodes, x* cpu [max]**	**2 GB [def]** **64 GB [max]**	**12h [def]** **24h [max]**	No limit [might change in the future]
long [lower priority]	**1 node, 1 cpu [def]** **16 nodes, x* cpu [max]**		**5 days [def]** **10 days [max]**	

[def] = default, [max] = maximum

Axiom cluster:

To be determined. Most likely the same as on Wally, but with higher memory limits.

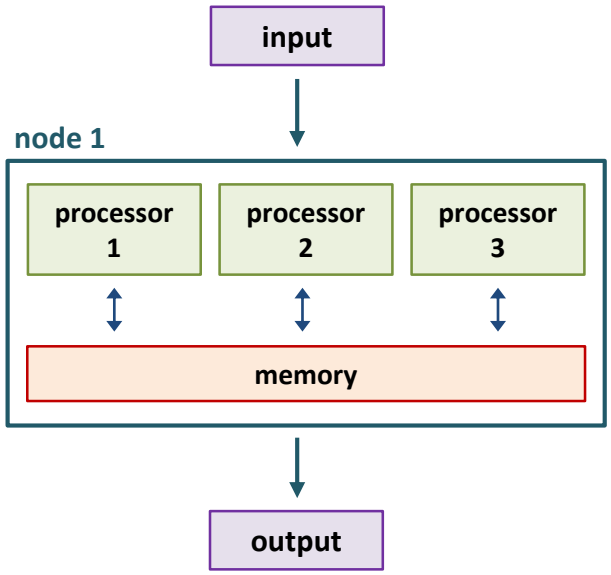
Slurm global allocation (jobs) and tasks

A quick reminder…

Shared memory computing
processors are all located on the same physical node. Memory is shared among them.

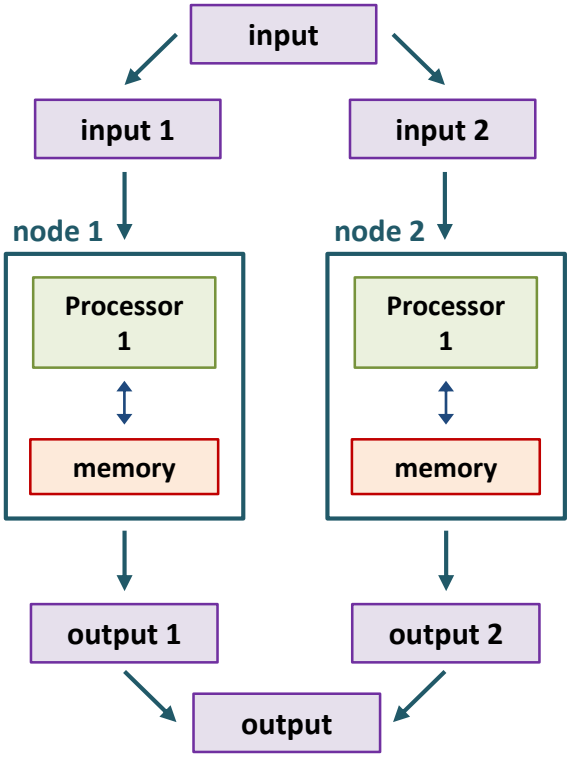
Distributed computing
processors are located in different machines that communicate through a network. Each processor has its own memory, and executes a different subtask.

single node job

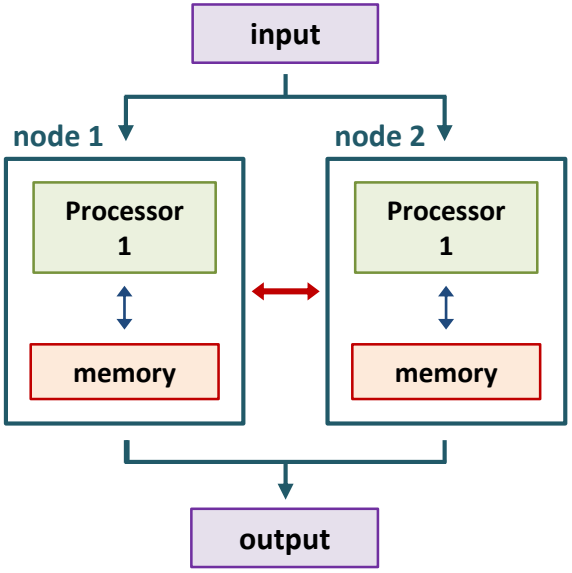


"Embarrassingly" parallel

array job
(e.g. N samples, same pipeline)



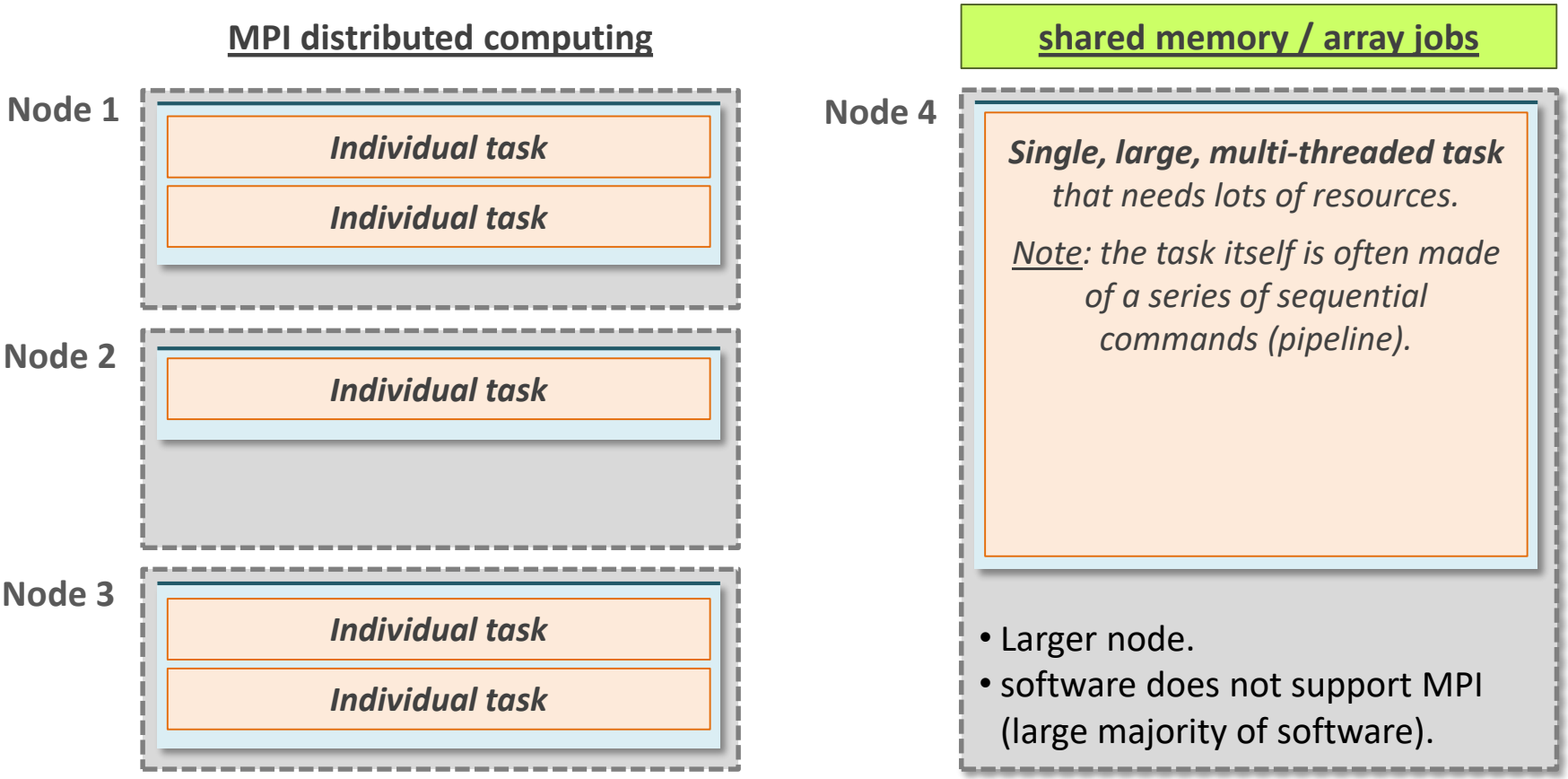
MPI (message passing interface)



Slurm global allocation (jobs) and tasks

Global allocation: ensemble of all resources available for the job (nodes, CPUs, memory, time). Slurm refers to this as a “job”.

Task: sub-allocation ***on a single node*** within a global allocation. Slurm refers to this as a “job step”.



Slurm / LSF command correspondence

Command overview (all commands will be explained in the coming slides)

	Slurm command	LSF command
Submit job to cluster	**sbatch script.sh**	**bsub < script.sh**
Interactive job	**srun**	**bsub -i**
Inquire about job status	**squeue / sacct**	**bjobs**
Force termination of running job	**scancel**	**bkill**
Display information about queues/partitions	**sinfo**	**bqueues**
Display information about cluster nodes	**sinfo --Node**	**bhosts**
Move job to a different queue/partition	**scontrol**	**bswitch**
Display stdout and stderr of unfinished job	**N/A****	**bpeek**

** Slurm prints the stdout/stderr to an output files in real-time, allowing the user to read it from that file as the job is running (e.g. with “tail -f <job_stdoutFile_.out>”).

submitting jobs with *sbatch* and *srun*

LSF vs Slurm equivalency:

bsub < jobScript.sh → ***sbatch*** **jobScript.sh**

bsub <options> command → ***srun*** **<options> command**

sbatch and *srun* – different commands…

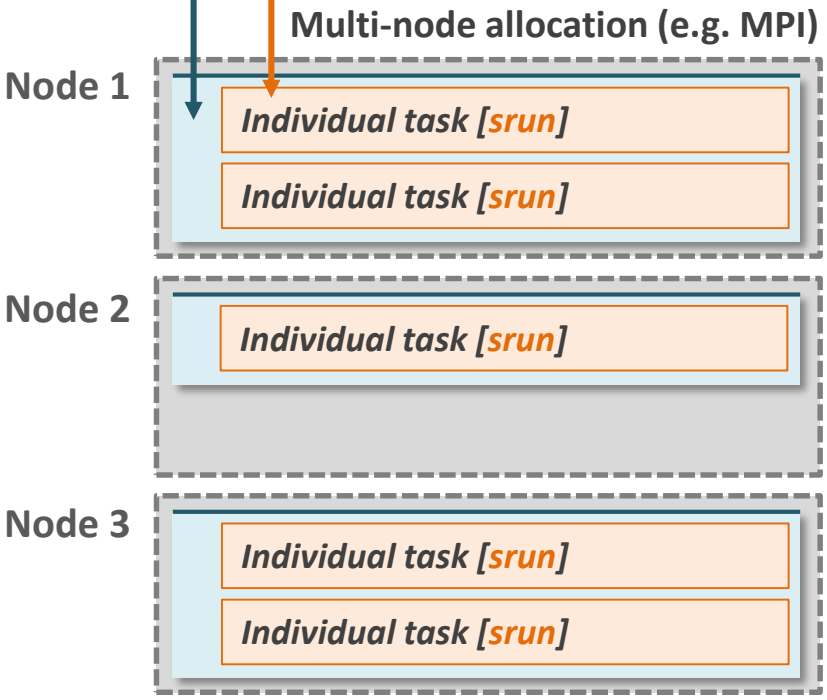
LSF equivalent:
bsub < jobSscript.sh

- ***sbatch*** and ***srun*** are the two commands used to submit jobs with slurm.

sbatch: create a global allocation for a job.

srun: start a new task within a job.

```
#!/bin/bash

# Slurm options:
#SBATCH ...

# Setup environment:
cd ...
module add ...

# Start tasks:
srun ...
srun ...
srun ...
```



Each **srun** command creates a new task and a new “job step”.

Having **srun** commands within a batch script **can be useful, but is not necessary**.

… that can achieve a similar result …

In practice, **for single node applications and array jobs**, both ***srun*** and ***sbatch*** can achieve similar results. This is because:

sbatch and ***srun*** accept (mostly) the same parameters.

- Processes can run directly within a ***sbatch*** global allocation.

Job submitted with only *sbatch*.

The commands in the script have access to the entire global allocation.

compute node

- a ***srun*** command given outside a global allocation creates its own allocation.

Job submitted with only *srun*.

An allocation matching the task's requirements is created by slurm.

…but we still recommend to keep them for separate use cases:

- Both ***sbatch*** and ***srun*** can be used to run a job with slurm, but there are a few important differences.
- We suggest to stick to the following use cases for ***sbatch*** and ***srun***.

sbatch: for regular job script submission.

- parameters can be passed in script with **#SBATCH** keyword. This improves reproducibility as commands + options are in the same file.
- standard output/error streams are always saved to a file.

```
#!/bin/bash
# Slurm options:
#SBATCH ...
#SBATCH ...
#SBATCH ...
# Run commands:
...
```

srun: for interactive jobs or small single command tests.

- standard output/error streams are shown directly in the user's shell.
- job runs in the foreground (unless "&" is passed).
- **Warning**: #SBATCH instructions in scripts are not recognized.

sbatch + srun: for special/advanced use cases.

- If accounting at the job task level (i.e. job steps) is needed.
- Jobs with parallel tasks requiring specific resource allocation (e.g. MPI jobs.)

```
#!/bin/bash
# Slurm options:
#SBATCH ...
#SBATCH ...
# Run tasks:
srun <options> command
srun <options> command
```

srun

- interactive jobs.
- single commands (for tests).

srun – running a single command

General syntax:

srun <options> command to run

With ***srun***, stdout/stderr are shown directly in the terminal.

Examples:

```
[user@login1 ~]$ srun hostname
cpt005.wally.unil.ch
```

```
[user@login1 ~]$ srun -p normal --time 01:00:00 --cpus-per-task 2 --mem 4GB
singularity run /software/singularity/containers/R-3.1.1-1.centos7.simg
################################################################################
### List of packages available R 3.1.1:
### ********************************
 [1] "acepack"       "annotate"      "AnnotationDbi" "base"
 [5] "base64enc"     "BatchJobs"     "BBmisc"        "Biobase"
 ...
[93] "tools"         "utils"         "XML"           "xtable"
[97] "XVector"
################################################################################
[user@login1 ~]$
```

srun – running a single command

Things to keep in mind with ***srun***:

- ***srun*** will **ignore any #SBATCH options passed in a script**, considering them simply as bash comments. Therefore it's best to use ***srun*** **only for single commands (tests)** and use ***sbatch*** **when submitting scripts**.

- with ***srun***, if you disconnect from the front-end machine or otherwise kill your terminal, the job will also get killed (whereas a job submitted with ***sbatch*** will continue to run). **

- ***srun*** automatically does X-forwarding (for graphical interfaces), there is no special option needed. It however requires that you connected to the front-end machine with X-forwarding (i.e. " ssh -X ").

** unless you run your srun inside a "screen" or "nohup" command.

srun – interactive jobs

LSF equivalent:
bsub -i command

- Passing the option **--pty bash** tells ***srun*** to start an interactive job.
- Useful to test commands / debug your scripts.
  (instead of running rests directly on the front-end machine)
- Type "**exit**" or "**Ctrl + D**" to exit the interactive job.

General syntax:

srun <options> --pty bash

Example:

```
[user@login1 ~]$
[user@login1 ~]$ hostname
login1.wally.unil.ch
[user@login1~ ]$ srun -p normal -t 1:00:00 --cpus-per-task=2 --mem=4GB --pty bash
[user@cpt002 ~]$ hostname
cpt002.wally.unil.ch
[user@cpt002 ~]$ which samtools
/usr/bin/which: no samtools
[user@cpt002 ~]$
[user@cpt002 ~]$ module load Bioinformatics/Software/vital-it
[user@cpt002 ~]$ module add /software/module/UHTS/Analysis/samtools/1.8
[user@cpt002 ~]$ which samtools
/software/UHTS/Analysis/samtools/1.8/bin/samtools
[user@cpt002 ~]$
[user@cpt002 ~]$ exit
exit
```

sbatch

submit job scripts

sbatch – submit job scripts

- ***sbatch*** is the recommended way to submit regular jobs to the cluster.

General syntax:

```
sbatch  jobScript.sh
sbatch ./jobScript.sh
```

```
#!/bin/bash
# Slurm options:
#SBATCH <option>
#SBATCH <option>
#SBATCH <option>
# Run commands:
...
```

- Best practice is to put all your options and commands into a single script.
- Options in script must be prefixed with **#SBATCH**.

LSF equivalence

sbatch is the equivalent of LSF's

bsub < jobSscript.sh

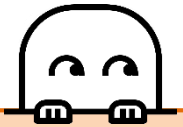
Note that **with *sbatch* the redirection sign "<" is no longer needed.**

Example:

```
[user@login1 ~]$ sbatch ./variant_calling.sh
Submitted batch job 28873
[user@login1 ~]$
[user@login1 ~]$ squeue
JOBID  PARTITION     NAME  USER  ST  TIME  NODES  NODELIST(REASON)
28873     normal  testJob  user  PD  0:10      1  cpt007
```

sbatch – job scripts structure

LSF comparison point



With slurm, **the interpreter line is compulsory**. This was not the case with LSF.

1. Interpreter line.

- Specifies the shell that will be interpreting the commands in the script.
- This line **is compulsory**: all job scripts must start with: **#!/bin/bash**

2. Option sections.

- One option per line.
- All options lines must start with **#SBATCH**.

3. Commands to run, written in bash.

- A number of slurm environment variables can be used (e.g. $SLURM_JOB_ID).

```
#!/bin/bash

#SBATCH --account=324517
#SBATCH --partition=normal
#SBATCH --time=02:30:00
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
#SBATCH --mem=32G
#SBATCH --job-name=rJob
#SBATCH --export=NONE


# Commands start here
module add R/351
cd /scratch/wally/<Institution>/
   <faculty>/<department>/<PI>/
   <project_short_name>
mkdir run_${SLURM_JOB_ID}
cd run_${SLURM_JOB_ID}
Rscript doSomething.R
```

sbatch / srun

basic options

Long and short option names in slurm

As you will see, many of the slurm options have a corresponding “single dash + single letter” abbreviated form:

For instance:

--account	**-A**
--partition	**-p**
--output	**-o**
--error	**-e**
--job-name	**-J**

- Using the long option, the separator between the option and its value can be either an “=“ or a space “ ”.
- Using the short option, the separator can only be a space “ ”.

Example:

```
#!/bin/bash

#SBATCH --partition=long
#SBATCH --partition long
#SBATCH -p long
#SBATCH -p=long
```

Nodes and tasks

Specify the number of nodes and tasks assigned to a slurm job.

Important: if all tasks should be executed on the same node, it is important to specify **--nodes=1**, otherwise slurm will likely distribute tasks across different nodes (depending on resource availability).

Reminder: tasks are sub-allocations that run in parallel.
If you have several sequential tasks requesting **--ntasks=1** is sufficient.

long option	short	description
--nodes	-N	Number of nodes requested for the global allocation.
--ntasks	-n	number of tasks across all requested nodes. Note that unless you specify the **--ntasks-per-node** option, tasks are distributed on requested nodes based on resource availability.
--ntasks-per-node		Specifically assign n tasks per node rather than having slurm assign them based solely on resource availability.

--ntasks and **--ntasks-per-node** are mutually exclusive.

Most frequent case

Examples:

Single task job (most cases).

```
#!/bin/bash

#SBATCH --nodes=1
#SBATCH --ntasks=1
```

10 tasks, “randomly” distributed.

```
#!/bin/bash

#SBATCH --nodes=5
#SBATCH --ntasks=10
```

10 tasks, but exactly 2 per node.

```
#!/bin/bash

#SBATCH --nodes=5
#SBATCH --ntasks-per-node=2
```

--account: resource usage tracking for billing!

LSF equivalent:

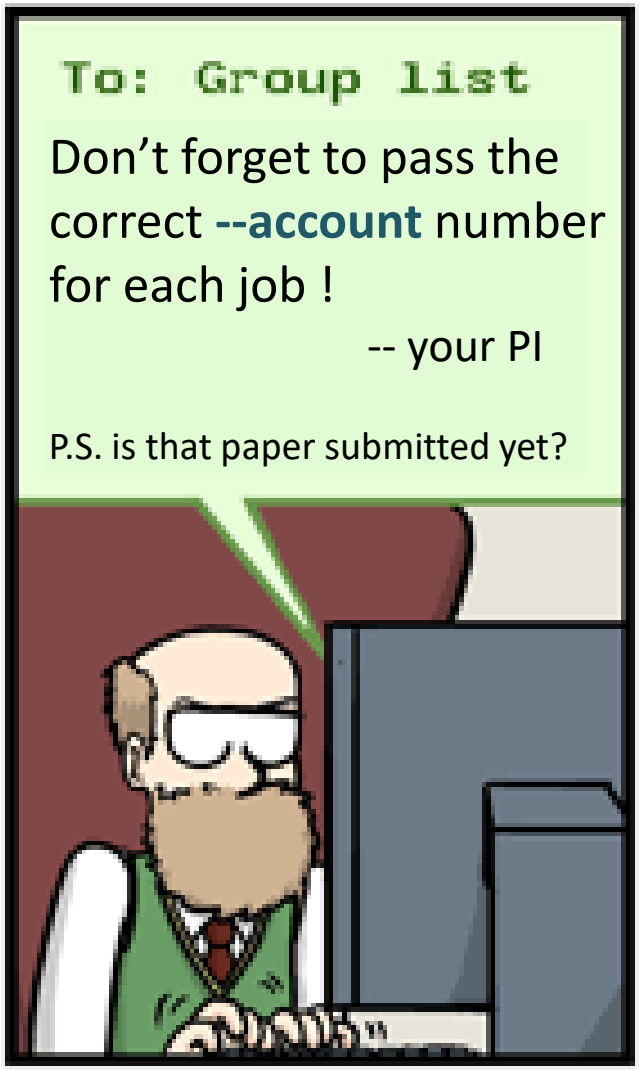
This was not needed previously, so make sure to add it now.

- The **--account** option is needed so that your resource usage can be billed correctly to your PI.
- It has no effect on your actual computing.

long option	short	description
--account	-A	Charge resources used by this job to the specified account.

- If **--account** is omitted, your job will be charged to your default account.

A message from your PI →



adapted from WWW.PHDCOMICS.COM

Which accounts are associated with me ?

Use the following command to list the **--account** values associated your user name:

```
[bob@login1~]$ sacctmgr show user $USER WithAssoc format=User,Account%30,DefaultAccount%30

      User                        Account                        Def Acct
---------- ------------------------------ ------------------------------
       bob                cours_intro_hpc                        core_it
       bob                       core_it                        core_it
```

User name

List of all accounts the user is associated with.

Default account (this will be used if you do not specify **--account** when submitting a job.

Examples:

```
[bob@login1 ~]$ srun --account=cours_intro_hpc --time=2:00:00 --pty bash

[bob@login1 ~]$ sbatch --account=core_it jobScript.sh

[bob@login1 ~]$ srun --pty bash
```

If no account is specified, the job's resource usage are charged to the default account.

Select a partition, name your job

LSF equivalent:

```
#BSUB -q <queue name>
#BSUB -J  <job name>
```

long option	short	description
--partition	-p	Specify partition (i.e. queue). Currently 2 partitions are available: “normal” and “long”.
--job-name	-J	Give a specific name to your job.

- Select the correct **--partition** depending on your job’s execution time:
  - **normal** for jobs < 24h.
  - **long** for jobs up to 10 days.
  - The default partition is **normal**.

- **--job-name** is not essential but can be convenient.

Examples:

“**normal**” is the default, but for reproducibility it’s always good to specify it explicitly.

Long options:

```
#!/bin/bash

#SBATCH --account=PIname_project
#SBATCH --partition=normal
#SBATCH --job-name=testRun

# Commands start here...
```

Short options:

```
#!/bin/bash

#SBATCH -A PIname_project
#SBATCH -p long
#SBATCH -J testRun

# Commands start here...
```

Reminder:
With slurm short options, a space must be used instead of “=“.

Save standard output/error to file

LSF equivalent:

#BSUB -o <stdout file>
#BSUB -e <stderr file>

- The **--output** and **--error** options are used to specify file names where to save stdout and stderr streams.
  If **--output** is specified but **--error** is not, both stdout/stderr are written to the same file.

long option	short	description
--output	-o	Write the job's standard output directly to the specified file.
--error	-e	Write the job's standard error directly to the specified file.

- Slurm **always saves the stdout/stderr of a job to a file, even if the --output or --error options are not passed**. By default both are saved to the same file, named **slurm-<jobID>.out**, in the current working directory.

- Slurm writes to stdout/stderr files in real time (i.e. as the job progresses): **this replaces LSF's "bpeek" function**. To watch these files live, "tail -f <filename>" can be used.

Warning: if an output file already exists, **slurm will overwrite it**!

Slurm vs. LSF

- The options names for -o/-e are the same between slurm and LSF.
- slurm always saves stdout/stderr to file – LSF does only if instructed to do so.
- slurm writes stdout/stderr to files in real time – LSF only when the job has completed.
- slurm overwrites exiting files – LSF appends to existing files.

Filename patterns

To make the naming of stdout/stderr output files easier, slurm provides a number of variables that expand at runtime. The most useful are listed here:

variable	expands to
%j	job ID. Very useful to ensure the stdout/stderr files have unique names.
%x	job name which is passed using the --job-name option.
%u	user name.
%N	name of host running the job.
%J	jobID.stepID of the current task (only applies if tasks submitted with srun).
%A	job array's master job ID number. **Must be used instead of %j in array jobs.**
%a	array index number. Only useful for array jobs (see later slides).

LSF equiv. %J

LSF equiv. %I

Only array jobs !

Filename patterns only work in filenames, and only in the slurm options:

- Things like **--job-name=testJob_%j** does not work (%j will not expand).
- They will not work in your bash commands after the option section.

Examples:

```
#!/bin/bash
#SBATCH --job-name=testRun
#SBATCH --output=%x_%j.out
#SBATCH --error=%x_%j.err
```

Assuming this job gets ID 1859, the stdout/stderr output files will be:

- testRun_1859.out
- testRun_1859.err

```
#!/bin/bash
#SBATCH -o logDir/random_%j.out
#SBATCH -e logDir/random_%j.err
```

Output files can also be saved to a different location than the current working directory.

--time option: set a job's run time limit

LSF equivalent:
#BSUB -W

- By default, a job's maximum runtime is set to the partition's default value: 12h for "normal", 5 days for "long".
- This can be changed with the **--time** option:

long option	short	description
--time	-t	Set limit to the run time of a job (wall time). If the runtime limit is reached, the job gets killed. Accepted time formats: • hours:minutes:seconds • days-hours:minutes:seconds • days-hours:minutes • minutes:seconds • minutes

Warning: if the requested time limit exceeds the partition's time limit, the job will be left in a PENDING state… **indefinitely**!

--time option: examples

Acceptable time format examples for the --time option.

```
#!/bin/bash
#SBATCH --partition=normal
#SBATCH --job-name=testRun

#SBATCH --time 2:30:00        # 2 h 30 min.
#SBATCH --time 0-2:30         # 2 h 30 min.
#SBATCH --time 2:30           # 2 min. 30 sec.
#SBATCH --time 3-00:00:00     # 3 days.
```

Note the difference between 2h30 and 2 min 30 sec. !

Problem when time limit > max runtime of partition.

Note: The --time parameter is passed outside of the script. This is not recommended and is only done here so you can see the passed value.

```
[user@login1 ~]$ sbatch --time 3-1:00:00 basic_cpuStress.sh
Submitted batch job 1433
[user@login1 ~]$
[user@login1 ~]$ squeue
JOBID   PARTITION    NAME      USER   ST     TIME   NODES     NODELIST(REASON)
 1433      normal    test      user   PD     0:00       1    (PartitionTimeLimit)
```

Normal partition has a max runtime of 24h

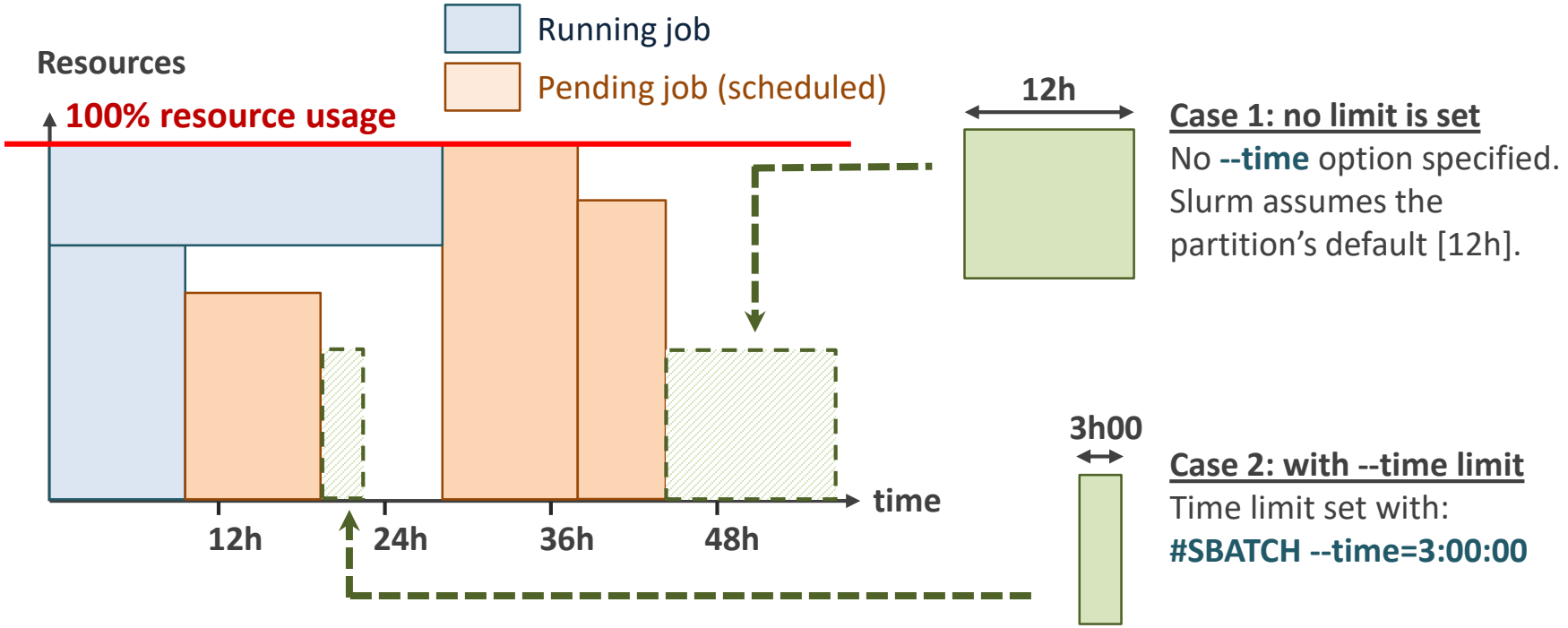
PD = PENDING

Hint that something isn't quite right !

--time option: examples

Why use **--time** to set a restrictive time limit to my job when queues are time limited anyway ?

- Let's assume we want to run a job that will last about 2 hours.
  Since it's less than 24h, we will chose to submit it to the "normal" partition.



Altruism meets personal fitness:

- Specifying **--time** allows slurm to better optimize the usage of the cluster.
- Benefits your jobs directly as slurm can "squeeze-in" jobs in free slots smaller than the queue's default time limit.
- Particularly interesting if the job is much shorter than the default limit of the queue.

Stay tuned: slurm email notifications

LSF equivalent:

```
#BSUB -u <email>
#BSUB -N
```

Slurm can send notification emails to help monitoring your jobs.

long option (no short option available)	description
--mail-user <email address>	Email address to send notifications to.
--mail-type <type>	Type of events for which SLURM should send an email. Default value is ALL. Events types (non-exhaustive list) • BEGIN, END, FAIL. • TIME_LIMIT_90 (reached 90 percent of runtime limit). • TIME_LIMIT_80 (reached 80 percent of runtime limit). • TIME_LIMIT_50 (reached 50 percent of runtime limit). • ALL (equivalent to: BEGIN,END,FAIL). • NONE (no notification).

Example:

```
#!/bin/bash

#SBATCH --mail-user alice@sib.swiss
#SBATCH --mail-type BEGIN,END,FAIL,TIME_LIMIT_80
```

The emails only contain a subject line – the body itself is empty.

	Subject		Correspondents
success →	Slurm Job_id=1362 Name=test Ended, Run time 00:00:30, COMPLETED, ExitCode 0		Slurm batch scheduler user
start →	Slurm Job_id=1362 Name=test Began, Queued time 00:00:01		Slurm batch scheduler user
failed job →	Slurm Job_id=28924 Name=sleepJob Failed, Run time 00:00:20, FAILED, ExitCode 1		slurm scheduler user
memory exceeded →	Slurm Job_id=28920 Name=memBurn Failed, Run time 00:00:59, OUT_OF_MEMORY		slurm scheduler user

Passing options to *sbatch* outside of script

- Job options can be passed directly to ***sbatch***, outside of a script.
- Options passed directly to ***sbatch*** take precedence over options passed inside a script.

```
[user@login1 ~]$
[user@login1 ~]$ sbatch --partition=long --mem=48G exploreJob.sh
Submitted batch job 1492
[user@login1 ~]$
```

Slurm vs. LSF, some things never change…

Just as with LSF, passing options directly to slurm is OK for a quick test, but don't make a habit of it.

not recommended because **not reproducible** !

What could go wrong ?

- yourself running the script after 6 months.
- a colleague to whom you passed your script.

slurm environment variables

LSF equivalent:

```
LSB_JOBID      = SLURM_JOB_ID
LSB_JOBINDEX = SLURM_ARRAY_TASK_ID
LSB_JOBNAME = SLURM_JOB_NAME
```

- Slurm environment variables provide access to useful values that can be used in bash scripts submitted to ***sbatch***.
- Here is a (non-exhaustive) list of the most useful of them:

variable name	description
$SLURM_JOB_ID	Job ID value.
$SLURM_JOB_NAME	Job name – i.e. value passed to --job-name.
$SLURM_CPUS_ON_NODE	Number of CPUs allocated on node running the job.
$SLURM_CPUS_PER_TASK	Number of CPUs allocated to current task.
$SLURM_SUBMIT_DIR	The directory from which sbatch was invoked or, if applicable, the directory specified by the --chdir option.
$SLURM_JOB_NODELIST	List of nodes allocated to the job.

Note: in array jobs, some environmental variables change:

variable name	description
$SLURM_ARRAY_JOB_ID	Job array master ID value. **In array jobs, this must be used instead of $SLURM_JOB_ID.**
$SLURM_ARRAY_TASK_ID	Job array index ID number of the current array replicate.

Why and how to use environment variables in your scripts ?

```
#!/bin/bash

#SBATCH --account=324517
#SBATCH --partition=normal
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=8
#SBATCH --job-name=testJob
#SBATCH --export=NONE

# Commands start here:
echo "Starting job $SLURM_JOB_NAME with ID $SLURM_JOB_ID".


outputDir=run_$SLURM_JOB_ID
mkdir $outputDir


samtools sort --threads $SLURM_CPUS_PER_TASK input.sam > $outputDir/sorted.bam
```

Print info to stdout to keep a record of your job name or ID number.

Create unique output directories that are unique to each job (avoid overwriting outputs)

Match the number of threads in your software to the number of CPUs in your allocation (--cpus-per-task option).

Warning: *sbatch* exports your local environment !

LSF comparison warning !

Unlike LSF, **slurm exports** (by default) **the local shell environment** (from the front end machine) **to the compute nodes that runs the job**.

To avoid this, it is best to always add the following option to your script:

```
#SBATCH --export=NONE
```

This will make sure your environment is clean at the start of your job.

Even with **--export=NONE**, your bash profile ("~/.bashrc" file) will still be loaded when your job starts. So please keep it clean :-)

Warning: *sbatch* exports your local environment !

Example without --export=NONE.

```
[user@login1 ~]$ export THIS_IS_SECRET="Don't tell anyone"
[user@login1 ~]$ module add /software/module/UHTS/Analysis/samtools/1.8
[user@login1 ~]$ sbatch testScript.sh
...
[user@cpt05 ~]$ module list
Currently Loaded Modulefiles:
  1) UHTS/Analysis/samtools/1.8
[user@cpt05 ~]$ echo $THIS_IS_SECRET
Don't tell anyone
[user@cpt05 ~]$
[user@cpt05 ~]$ module purge
[user@cpt05 ~]$ module list
No Modulefiles Currently Loaded.
[user@cpt05 ~]$
```

This happens inside "testScript.sh"…

Example with #SBATCH --export=NONE.

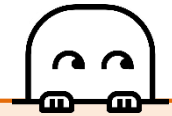
```
[user@login1 ~]$ export THIS_IS_SECRET="Don't tell anyone"
[user@login1 ~]$ module add /software/module/UHTS/Analysis/samtools/1.8
[user@login1 ~]$ sbatch testScript.sh
...
[user@cpt05 ~]$ module list
No Modulefiles Currently Loaded.
[user@cpt05 ~]$ echo $THIS_IS_SECRET

[user@cpt05 ~]$
```

This happens inside "testScript.sh"…

Slurm: requesting resources

Requesting resources – a reminder

Slurm vs. LSF, some things never change…



The higher the resource requirements, the more difficult to find an available compute node.

Do not request (much) more than you need

- more time for your job to get started.
- Unused resources are unavailable to others, as well as for your own other jobs.

Requesting multiple CPUs

LSF equivalent:

```
#BSUB -n <proc. nb>
#BSUB -R "span[hosts=1]"
```

- By default slurm allocates **1 CPU per task** (and 1 task per job).
- Additional CPUs must be explicitly requested with the **--cpus-per-task** option:

long option (no short option available)	**description**
--cpus-per-task	Number of processors to be allocated to each task. • All processors for a task are allocated on a same node. • Without this option slurm will allocate only one processor per task.

Slurm vs. LSF

Unlike LSF, **slurm strictly enforces the CPU limit allocated to each job**. If you ask for 1 processor – or don't specifically ask for more – your job will have only 1 processor available, and if you then run something multi-threaded all threads will share this one processor. So if more than 1 processor is needed, it is essential to explicitly request them.

Requesting multiple CPUs: examples

General case example:

Job is run as a single task on a single node.
For instance:

- shared memory (e.g multi-threaded).
- jobs arrays.

```
#!/bin/bash

#SBATCH --export=NONE
#SBATCH --job-name=testJob
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=24

# Commands start here ...
cd /path/to/workdir
module add UHTS/Analysis/...
```

Total allocation: 24 CPUs on a single node.

Advanced case example:

Job runs multiple tasks, e.g. “MPI” jobs.
The total number of CPUs for the job allocation is:

<number of tasks> * cpus-per-task

```
#!/bin/bash

#SBATCH --export=NONE
#SBATCH --job-name=mpiJob
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=3
#SBATCH --cpus-per-task=8

# Commands start here ...
srun ...
srun ...
srun ...
```

Total allocation: 24 CPUs on a single node.

Each task (started with **srun**) can access only 8 CPUs.

Requesting memory

LSF equivalence

#BSUB -R “rusage[mem=<memory in MB>]”
#BSUB -M <memory in KB>

- By default, jobs get allocated 2 GB of memory (on all partitions).
- Jobs that exceed their memory limit get killed by slurm.
- To request more memory, two **mutually exclusive** options that can be used:

long option (no short option available)	description
--mem=<memory>[M|G|T]**	memory limit **per node**.
--mem-per-cpu=<memory>[M|G|T]**	memory limit per cpu (logical processor).

** 'M','G' or 'T' after the memory value specifies the unit: megabyte [M], gigabyte [G] or terabyte [T].
If no unit is given, slurm defaults to megabytes [M].

For jobs running on a single node, both **--mem** and **--mem-per-cpu** can achieve the same result.
For MPI jobs distributed over several nodes, only --mem-per-cpu should be used.

Example:

2 equivalent ways of requesting 3 CPUs and 24 GB of memory on a single compute node.

```
#!/bin/bash

#SBATCH --ntasks=1
#SBATCH --cpus-per-task=3
#SBATCH --mem=24000
#SBATCH --mem=24000M
#SBATCH --mem=24G
```

These are equivalent. Select one.

```
#!/bin/bash

#SBATCH --ntasks=1
#SBATCH --cpus-per-task=3
#SBATCH --mem-per-cpu=8G
```

How do I know how much resources I need ?

CPU:

- Read the software's documentation: hopefully it will indicate whether it is multithreaded or not.
- (good) multithreaded software will have an option allowing you to select how many CPUs the software should use (e.g. samtools has a --threads option).
- Look at the output of "***sacct***" and compare values in columns "CPUTime" and "TotalCPU".

Memory:

- Make a first run where you set --mem to your best "guesstimate"…
- If job is terminated with "OUT_OF_MEMORY" error, increase memory allocation and try again…
- If job completes, use "***sacct***" and compare the "MaxRSS" column to your --mem request. If you requested much more than the actual usage, decrease your memory request.

Too much memory and CPU requested.

```
[user@login1 ~]$ sacct --format=jobid,jobname,partition,account%30,alloctres%40,cputime,
                 totalcpu,maxrss,submit,state%15,exitcode,elapsed
       JobID    JobName  Partition                          AllocTRES    CPUTime   TotalCPU     MaxRSS
------------ ---------- ----------  ---------------------------------- ---------- ---------- ----------
28917           memBurn     normal   billing=8,cpu=8,mem=30G,node=1   00:08:16  00:57.790
28917.batch       batch                        cpu=8,mem=30G,node=1   00:08:16  00:57.788  15667884K

28919           memBurn     normal   billing=8,cpu=1,mem=20G,node=1   00:01:01  00:57.631
28919.batch       batch                        cpu=1,mem=20G,node=1   00:01:01  00:57.629  15667884K
```

Fixed it ! (only 20G of memory and 1 CPU)

Requesting / excluding specific nodes.

- slurm allows requesting/excluding specific nodes.

long option	short	description
--nodelist	-w	Request specific nodes to run your job.
--exclude	-x	Exclude specific nodes from the resource allocation pool.

- Multiple nodes can be requested/excluded with:
  - a comma separated list “cpt01,cpt02,cpt03”.
  - a dash separated range “cpt[01-03]”.
  - or a combination of both “cpt[01,02-03]”, “cpt01,cpt[02-03]”.

Example:

```
#!/bin/bash

#SBATCH --nodelist=cpt03                    } Job can only run on cpt03.
#SBATCH --nodelist=cpt01,cpt02,cpt03        }
#SBATCH --nodelist=cpt[01,02,03]            }
#SBATCH --nodelist=cpt[01-03]               } Equivalent ways to request cpt01,
#SBATCH --nodelist=cpt01,cpt[02-03]         } cpt02 and cpt03.
#SBATCH --nodelist=cpt[01,02-03]            }

#SBATCH --exclude=cpt04,cpt01               } Job cannot run on cpt01 and cpt04.
```

Validate your slurm script before running it

sbatch --test-only jobScript.sh

- Validate your script's allocation before running it.
- Gives estimate of when job will run.
- **Warning**: does not detect bad **--time** allocations.

sbatchTest_withErrors.sh

```
#!/bin/bash
#SBATCH --account=
#SBATCH --partition=normal
#SBATCH --job-name=testJob
#SBATCH --nodes=1
#SBATCH --ntasts=1
#SBATCH --cpus-per-task=2
#SBATCH --mem=20T
#SBATCH --time=2-00:30:00
#SBATCH --export=NONE
```

```
[user@login1 ~]$ sbatch --test-only sbatchTest_withErrors.sh
sbatch: unrecognized option '--ntasts=1'
allocation failure: Invalid account or account/partition combination specified
allocation failure: Requested node configuration is not available.

... fix errors in script ...
[user@login1 ~]$ sbatch --test-only sbatchTest_withErrors.sh
sbatch: Job 28858 to start at 2019-06-07T17:21:55 using 2 processors on nodes
cpt002 in partition normal

[user@login1 ~]$ sbatch sbatchTest_withErrors.sh
Submitted batch job 28873
[user@login1 ~]$ squeue
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
28873    normal  testJob  rengler PD       0:00      1 (PartitionTimeLimit)
```

Slurm: other commands

	Slurm command	LSF command
Inquire about job status	**squeue / sacct**	**bjobs**
Force termination of running job	**scancel**	**bkill**
Display information about queues/partitions	**sinfo**	**bqueues**
Display information about cluster nodes	**sinfo --Node**	**bhosts**
Move job to a different queue/partition	**scontrol**	**bswitch**

squeue – monitor running jobs

LSF equivalent:

```
bjobs
bjobs -a      # list all jobs.
```

- lists submitted and running jobs or job.steps, with their current status.

General syntax: **squeue <options>**

```
[user@login1 ~]$ squeue
JOBID   PARTITION     NAME     USER    ST   TIME      NODES   NODELIST(REASON)
 1322      normal   testJob   rengler   R   1:20:03       1   cpt03
```

- Job name
- Abbreviated job status. R=RUNNING
- wall time since the jobs started (H:M:S)
- Node(s) running the job.

- Adding the **--long / -l** option expands the job's status and displays its runtime limit.

```
[user@login1 ~]$ squeue --long
JOBID   PARTITION     NAME     USER    STATE   TIME TIME_LIMIT   NODES   NODELIST(REASON)
 1322      normal   testJob   rengler  RUNNING  0:03    2:30:00       1   cpt03
```

- Expanded job status.
- Time limit of job
  Note: a custom time limit was here set with --time.

By default ***squeue*** will only show jobs of the current user that are **PENDING [PD]**, **RUNNING [R]** or **COMPLETING[CG]**.

squeue – monitor running jobs

- To show jobs in other states, the **--states / -t** option can be used. State values can be passed in full or abbreviated form, e.g. "RUNNING" and "R", "COMPLETED" and "CD".

```
[user@login1 ~]$ squeue -l -t RUNNING,COMPLETED
[user@login1 ~]$ squeue -l -t R,CD
JOBID PARTITION     NAME     USER      STATE  TIME  TIME_LIMIT  NODES  NODELIST(REASON)
1338     normal    test1   rengler   COMPLETE  2:01     2:00:00      1  cpt01
1339     normal    test2   rengler    RUNNING  0:04     3:30:00      3  cpt[01-03]
```

- Use **-t all t**o display jobs in any state.

```
[user@login1 ~]$ squeue -t all
JOBID PARTITION     NAME     USER      STATE  TIME  TIME_LIMIT  NODES  NODELIST(REASON)
1338     normal    test1   rengler   COMPLETE  2:01     2:00:00      1  cpt01
1339     normal    test2   rengler    RUNNING  0:04     3:30:00      3  cpt[01-03]
```

Reminder: a “job step” corresponds to a task (started with srun) within a job.

Option summary: (non-exhaustive list)

long option	short	description
--states	-t	Restrict reporting to the jobs with the specified states. Comma separator can be used to specify multiple states (e.g. PD,R,CD). "all" will report jobs in any state.
--long	-l	Display job status values in full, and adds job time limit.
--user	-u	Display jobs of specific user. If more than one, use comma separated list.
--name	-n	Only display info for specified jobs. If more than one, use comma separated.
--step	-s	display job steps instead of jobs. For jobs without steps, nothing is displayed.

sacct – show completed jobs

General syntax: **sacct <options>**

- By default ***sacct*** will only show jobs have finished (or are still running/pending) within the current day.
- To display older jobs, the **--starttime=MM.DD[.YY]** option can be used, specifying the date after which jobs should be listed. Different formats are accepted for entering the date:

```
sacct --starttime=06.14      # June 6th of the current year.
sacct --starttime=0614       # same as above.
sacct --starttime=06.14.19   # June 6th 2019.
sacct --starttime=061419     # same as above.
```

Options: (non-exhaustive list)

long option	short	description
--format	-o	Customize the output's columns. E.g. this will add memory/CPU/time consumption: **--format=jobid,jobname,partition,account%30,alloctres%40,cputime,totalcpu, maxrss,submit,state%15,exitcode,elapsed**
--starttime	-S	Display jobs that finished after the selected date. Several date formats are recognized: MMDDYY, MMDD, MM.DD.YY, MM.DD, MM/DD/YY, MM/DD, YYYY-MM-DD.
--state	-s	Report only jobs matching the selected state(s). Important: when using this option, the **--starttime** option must also be specified (otherwise only running/pending jobs are shown)
--long	-l	Displays more info (warning: really a lot of info!).

sacct – examples

```
[user@login1 ~]$ sacct
JobID           JobName  Partition    Account  AllocCPUS      State ExitCode
------------ ---------- ---------- ---------- ---------- ---------- --------
1062                echo     normal   vital-it          1  COMPLETED  0:0

1340                test2       long   vital-it          3  COMPLETED  0:0
1340.batch          batch              vital-it          3  COMPLETED  0:0

1339                test1     normal   vital-it          3  COMPLETED  0:0
1339.batch          batch              vital-it          1  COMPLETED  0:0
1339.0          printHost+             vital-it          1  COMPLETED  0:0
1339.1          printHost+             vital-it          1  COMPLETED  0:0
1339.2          printHost+             vital-it          1  COMPLETED  0:0

1360                test4     normal   vital-it          2  CANCELLED  0:0
1360.batch          batch              vital-it          1  CANCELLED  0:15
1360.0           samtools+             vital-it          1  CANCELLED  0:15
1360.1           samtools+             vital-it          1  CANCELLED  0:15

1361                test5     normal   vital-it          2     FAILED  1:0
1361.batch          batch              vital-it          1     FAILED  1:0

1363             memBurn     normal   vital-it          4 OUT_OF_MEMORY 0:125
1356.Batch       memBurn              vital-it          4 OUT_OF_MEMORY 0:125
```

"srun" job.

"sbatch" job.

"sbatch + srun" with 3 tasks (3 job steps).

Job cancelled with "scancel".

Job failed.

Job terminated by slurm because it exceeded memory allocation.

Slurm exit codes explained:

The "ExitCode" column has 2 numbers separated by a column, e.g. "0:0", "1:0" or "0:125".

- first value is your script's exit code. E,g, 0 = no error, 1 = error.
- second value is slurm's internal exit code. E.g. 0 = no error, 15 = cancelled by user, 125 = exceeded memory.

To add useful information to the output of ***sacct***, such as job wall time, CPU and memory usage or submit time, the following command can be used:

sacct --format=jobid,jobname,partition,account%30,alloctres%40,cputime,totalcpu,maxrss,submit,state%15,exitcode,elapsed

```
[user@login1 ~]$ sacct --format=jobid,jobname,partition,account%30,alloctres%40,cputime,
                 totalcpu,maxrss,submit,state%15,exitcode,elapsed
       JobID    JobName  Partition                Account                                AllocTRES    CPUTime   TotalCPU
------------ ---------- ---------- ------------------------------ ---------------------------------------- ---------- ----------
28903           cpuBurn     normal    cours_intro_hpc   billing=8,cpu=8,mem=4G,node=1   00:04:08  03:59.013
28903.batch       batch              cours_intro_hpc             cpu=8,mem=4G,node=1   00:04:08  03:59.011

28917           cpuBurn     normal    cours_intro_hpc   billing=8,cpu=8,mem=30G,node=1  00:08:16  00:57.790
28917.batch       batch              cours_intro_hpc             cpu=8,mem=30G,node=1  00:08:16  00:57.788

28919           memBurn     normal    cours_intro_hpc   billing=8,cpu=8,mem=20G,node=1  00:16:08  00:57.631
28919.batch       batch              cours_intro_hpc             cpu=8,mem=20G,node=1  00:16:08  00:57.629

28920           memBurn     normal    cours_intro_hpc   billing=8,cpu=8,mem=10G,node=1  00:07:52  00:56.106
28920.batch       batch              cours_intro_hpc             cpu=8,mem=10G,node=1  00:07:52  00:56.104
```

```
   MaxRSS              Submit             State ExitCode    Elapsed
--------- ------------------- --------------- -------- ----------
          2019-06-13T09:33:37       COMPLETED      0:0   00:00:31
  11596K  2019-06-13T09:33:37       COMPLETED      0:0   00:00:31

          2019-06-13T11:28:04       COMPLETED      0:0   00:01:02
4269804K  2019-06-13T11:28:04       COMPLETED      0:0   00:01:02

          2019-06-13T11:35:00       COMPLETED      0:0   00:02:01
15667884K 2019-06-13T11:35:00       COMPLETED      0:0   00:02:01

          2019-06-13T11:38:07   OUT_OF_MEMORY    0:125   00:00:59
4271244K  2019-06-13T11:38:08   OUT_OF_MEMORY    0:125   00:00:59
```

Column legend:

AllocTRES = requested resources.
Elapsed = wall time.
CPUTime = wall time * cpu requested.
TotalCPU = actual CPU usage time.
MaxRSS = maximum memory usage.

List of the most common slurm job states *(non-exhaustive list)*

Status	Short	Description
PENDING	PD	Job is awaiting resource allocation.
RUNNING	R	Job currently has an allocation.
COMPLETED	CD	Job has terminated all processes on all nodes with an exit code of zero.
FAILED	F	Job terminated with non-zero exit code or other failure condition.
TIMEOUT	TO	Job terminated upon reaching its time limit.
CANCELLED	CA	Job was explicitly canceled by the user or system administrator. The job may or may not have been initiated.
SUSPENDED	S	Job has an allocation, but execution has been suspended and CPUs have been released for other jobs.
COMPLETING	CG	Job is in the process of completing. This state should generally last only a few seconds, and if longer might be indicative of a problem.
OUT_OF_MEMORY	OOM	Job was terminated because it exceeded its memory allocation, i.e. it used more memory (RAM) than what was requested with the **--mem** option.

LSF equivalent: bswitch

scontrol – switch job to a different partition

- ***scontrol*** can be used to modify the partition of a pending job – **but not of a running job !**
- Most ***scontrol*** commands are meant for admins rather than regular users.

General syntax:

scontrol update jobid=<jobID> **partition=**<partition name>

Examples:

Switching a PENDING job to the "long" partition.

```
[user@login1 ~]$ sbatch --partition=normal --time 3-00:00:00 jobScript.sh
Submitted batch job 28878

[user@login1 ~]$ squeue -l
JOBID PARTITION     NAME     USER    STATE  TIME  TIME_LIMIT NODES  NODELIST
28878    normal  testJob  rengler  PENDING  0:00  3-00:00:00 1 (PartitionTimeLimit)

[user@login1 ~]$ scontrol update jobid=28878 partition=long
[user@login1 ~]$ squeue -l
JOBID PARTITION     NAME     USER    STATE  TIME  TIME_LIMIT NODES  NODELIST
28878      long  testJob  rengler  RUNNING  0:23  3-00:00:00     1    cpt002
```

scontrol – doesn't work with running jobs...

Warning: Running jobs cannot be switched anymore !

```
[user@login1 ~]$ sbatch --partition=normal --time 24:00:00 jobScript.sh
Submitted batch job 28878

[user@login1 ~]$ squeue -l
JOBID  PARTITION     NAME     USER    STATE  TIME  TIME_LIMIT NODES   NODELIST
28878     normal  testJob  rengler  RUNNING  23:10   24:00:00     1      cpt005

[user@login1 ~]$ scontrol update jobid=28878 partition=long
Job is no longer pending execution for job 28877
```

partition switch failed!

Warning: Extending time limit requires admin privileges.

```
[user@login1 ~]$ sbatch --partition=normal --time 3:00:00 jobScript.sh
Submitted batch job 28878

[user@login1 ~]$ squeue -l
JOBID  PARTITION     NAME     USER    STATE   TIME  TIME_LIMIT NODES  NODELIST
28878     normal  testJob  rengler  RUNNING  02:30   03:00:00       1    cpt005

[user@login1 ~]$ scontrol update jobid=28878 TimeLimit=24:00:00
Access/permission denied for job 28878
```

runtime limit extension failed!

scancel – terminate jobs and job steps

LSF equivalent:

bkill

- ***scancel*** allows you to terminate jobs or job steps.
- Jobs and job steps can be terminated individually, or by groups (e.g. all jobs of a given user in a given state).

General syntax:

```
scancel <jobID>
scancel <jobID.step>
```

Options: (non-exhaustive list)

long option	short	description
--user	-u	Kill all jobs for a given user.
--name	-n	Kill job(s) that match a given job name.
--state	-t	Kill all jobs in a given state for the current user. States can be given either by their full name, e.g. PENDING, RUNNING, SUSPENDED, or their short form, e.g. PD, R, S.

Examples:

```
[user@login1 ~]$ scancel 45002
[user@login1 ~]$ scancel 45001.1
[user@login1 ~]$ scancel -u bob --state PENDING
```

Terminate job 45002

Terminate step 1 of job 45005

Terminate all jobs from user "bob" that are in state "pending"

sinfo – display info about nodes and partitions

LSF equivalent:

bqueues

- ***sinfo*** displays the list of all partitions and nodes available on the cluster.
- By default, ***sinfo*** shows displays info in "partition oriented" format, where nodes are grouped by state and partitions.

General syntax:

sinfo
sinfo --Node

```
[user@login1 ~]$ $ sinfo
PARTITION AVAIL    TIMELIMIT  NODES   STATE  NODELIST
normal*      up   1-00:00:00      2    fail  cpt[04,07]
normal*      up   1-00:00:00      5    idle  cpt[01-03,05-06]
long*        up  10-00:00:00      7    aloc  cpt[07-10,15,21-22]
```

- Partition name
- Availability up/down
- Max runtime of partition
- Node count
- Node state:
  idle = available to start job.
  aloc = in use.
  fail = node failed.
  maint = under maintenance
- Node list

- Add the **--long / -l** option to display additional info:

```
[user@login1 ~]$ $ sinfo --long
PARTITION AVAIL  TIMELIMIT   JOB_SIZE ROOT OVERSUBS GROUPS NODES  STATE NODELIST
normal*      up 1-00:00:00 1-infinite   no       NO    all     1   fail cpt04
normal*      up 1-00:00:00 1-infinite   no       NO    all     4   idle cpt[01-03,05]
```

- min-max number of nodes that can be requested per job
- User groups that can submit jobs to partition. "all" = everyone.

sinfo – display info about nodes and partitions

LSF equivalent: bhosts

- Use **sinfo --Node / sinfo -N** to display info on individual nodes (i.e., "Node oriented" format).

```
[user@login1 ~]$ $ sinfo -N
[user@login1 ~]$ $ sinfo --Node
NODELIST NODES PARTITION STATE  CPUS  S:C:T  MEMORY  TMP_DISK  WEIGHT  AVAIL_FE REASON
cpt01        1   normal*  idle    32  4:8:1   31950     48000       1   (null)   none
cpt02        1   normal*  idle    32  4:8:1   31950     48000       1   (null)   none
cpt05        1   normal*  idle     8  2:4:1    7800    115000       1   (null)   none
```

- CPU count on node
- Socket : Core : Thread — Number of CPUs = S*C*T
- RAM on node [MB]
- size of /tmp [MB]
- reason a node is unavailable

- The **--nodes / -n** option can additionally be used to restrict display to certain nodes.
  Warning: do not confuse the **--Node** and **--nodes** options.

```
[user@login1 ~]$ $ sinfo -N -n cpt[01-02]
[user@login1 ~]$ $ sinfo --Node --nodes cpt01,cpt02
NODELIST NODES PARTITION STATE  CPUS  S:C:T  MEMORY  TMP_DISK  WEIGHT  AVAIL_FE REASON
cpt01        1   normal*  idle    32  4:8:1   31950     48000       1   (null)   none
cpt02        1   normal*  idle    32  4:8:1   31950     48000       1   (null)   none
```

LSF to slurm:

script conversion example

Script conversion example: single node R script

Run an R script on the cluster with:

- 2 CPU, 64 GB of memory.
- ~ 2 days runtime.
- Job runs on a single node.

LSF script

```
#BSUB -L /bin/bash
#BSUB -J rScript
#BSUB -q long
#BSUB -o rScript_%J.out
#BSUB -e rScript_%J.txt
#BSUB -n 2
#BSUB -R "span[hosts=1]"
#BSUB -R "rusage[mem=64000]"
#BSUB -M 64000000
#BSUB -u user@unil.ch
#BSUB -N

# Commands on next slide...
```

Slurm script

Interpreter line: compulsory with slurm.

Make sure you give the correct account.

```
#!/bin/bash

#SBATCH --account=PIname_project
#SBATCH --job-name=rScript
#SBATCH --partition=long
#SBATCH --output=%x_%j.out
#SBATCH --error=%x_%j.err
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=2
#SBATCH --mem=64G
#SBATCH --mail-user=user@unil.ch
#SBATCH --mail-type=ALL
#SBATCH --time=2-12:00:00
#SBATCH --export=NONE

# Commands on next slide...
```

← Max runtime

← Stop environment export to compute node.

Script conversion example: single core R script

Run an R script on the cluster with:

- 2 CPU, 64 GB of memory.
- ~ 2 days runtime.
- Job runs on a single node.

```
# Load module for R 3.5.1.
module load Bioinformatics/Software/vital-it
module add R/3.5.1

# Change to correct working directory.
cd /scratch/cluster/monthly/$USER
cd /scratch/wally/<Institution>/<faculty>/<department>/<PI>/<project_short_name>

# Run R script
Rscript ~/scripts/dataAnalysis.R

exit 0
```

Wally / Axiom cluster: need to specifically load the Vital-IT software stack.

UNIL cluster: new scratch location.

Script in your home directory. This doesn't change.

Slurm: job arrays

Job arrays – run the same job multiple times

What job arrays are…

- Job arrays are a way to **replicate n times the exact same job**.
- So essentially they replace a bash for loop like:

```
for x in for sample in $(seq 1 5); do
    sbatch jobScript.sh inputSample_${x}.fastq
done
```

with:

```
sbatch jobArrayScript.sh
```

- why bother ?
  - Better reproducibility (all the info is in your script).
  - Easier job management for you (e.g. you get single notification when the job is done).
  - Better job scheduling by slurm = your jobs might be dispatched faster.
  - Compute in style – arrays are just a more elegant solution !

and what they are not…

- Array job != MPI (message passing interface) jobs.
- Individual jobs of an array are likely to run on different compute nodes, but they are fully independent (each with its own global allocation) and do not communicate.
- Resources requested for an array job (memory, CPU, …) are the resources needed for one replicate of the array, not the entire array. E.g. is you run an array with 15 replicates, and each needs 20 GB of memory, you request is “--mem=20G” (and not --mem=300G).

Job arrays – how to submit an array

long option	short	description
--array=<index list>	-a	Submit a job array with indexes "index list". The index list can be passed as ranges (a-b = from a to b), comma separated values (a,b,c = a, b and c) or a combination of both (a-b,c = from a to b, and c). Here are some examples: --array=1-25 : 25 replicates, with indexes from 1 to 25. --array=0-24 : 25 replicates, with indexes from 0 to 24. --array=9,12,23,45 : 4 replicates, with indexes 9, 12, 23 and 45. --array=1-5,7,9-10 : 8 replicates, with indexes 1,2,3,4,5,7,9,10. **Note**: index values must be positive integers.

Custom step value:

Use the **:x** modifier to set the increase step within a range to "x". For instance:

--array=1-9:2 ==> run an array of 5 replicates with indexes 1, 3, 5, 7 and 9.

--array=0-23:5 ==> run an array of 5 replicates with indexes 0, 5, 10, 15 and 20.

Limit number of running jobs:

Use the **%x** modifier to limit the number of simultaneously running jobs to "x" .

--array=1-250%20 ==> This array has 250 jobs, but at most 20 will be running at anytime.

Job arrays – environment variables and filename patterns

Shell environment variables

- With job arrays, it is up to you to use the variable **$SLURM_ARRAY_TASK_ID** at some point in your bash script to modify the inputs/outputs of your job in each replicate of the array.
- **$SLURM_ARRAY_TASK_ID** takes the values of the indexes passed to the --array option. E.g. in a job with "--array=1-5,9", the values of **$SLURM_ARRAY_TASK_ID** will be 1,2,3,4,5 and 9.
- The value of the array index (**$SLURM_ARRAY_TASK_ID**) is the only thing that varies between replicates within an array.

	variable	expands to
most useful	$SLURM_ARRAY_TASK_ID	job array index ID number for the current array replicate.
most useful	$SLURM_ARRAY_JOB_ID	job array master ID value. **In array jobs, this must be used instead of $SLURM_JOB_ID.**
	$SLURM_ARRAY_TASK_COUNT	total number of tasks in a job array.
	$SLURM_ARRAY_TASK_MAX $SLURM_ARRAY_TASK_MIN	job array's maximum/minimum index number.
	$SLURM_ARRAY_TASK_STEP	job array's index increase step size.

Filename patterns variables (to use in file names with the --output/--error options).

variable	expands to
%A	job array's master job ID number (unique job ID for all jobs in the array). **%A must be used instead of %j in array jobs.**
%a	job array index ID number for the current array replicate.

Job arrays – example

Replicate a same R script for 11 samples, named “sample_1.txt”, “sample_2.txt”, ..., “sample_12.txt”. Note that “sample_11.txt” is missing.

```
#!/bin/bash
#SBATCH --account=PIname_project
#SBATCH --array=1-10,12
#SBATCH --job-name=testArray
#SBATCH --partition=normal
#SBATCH --output=%x_%A-%a.out
#SBATCH --error=%x_%A-%a.err
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=8
#SBATCH --mem=16G
#SBATCH --mail-user=user@unil.ch
#SBATCH --mail-type=ALL
#SBATCH --time=3:00:00
#SBATCH --export=NONE

# Commands...
cd scratch/wally/<Institution>/<faculty>/<department>/<PI>/<project_short_name>
module load Bioinformatics/Software/vital-it
module add R/3.5.1

mkdir arrayRun_$SLURM_ARRAY_TASK_ID
Rscript myScript.R sample_$SLURM_ARRAY_TASK_ID.txt
```

List of indexes over which to iterate.

Note the use of **%A** for jobID (instead of **%j** used in regular jobs). This value stays the same for all jobs within the array.

%a expands to the index value. This value changes with every job within the array.

Example output files:

```
testArray_28877_1.out
testArray_28877_1.err

testArray_28877_12.out
testArray_28877_12.err
```

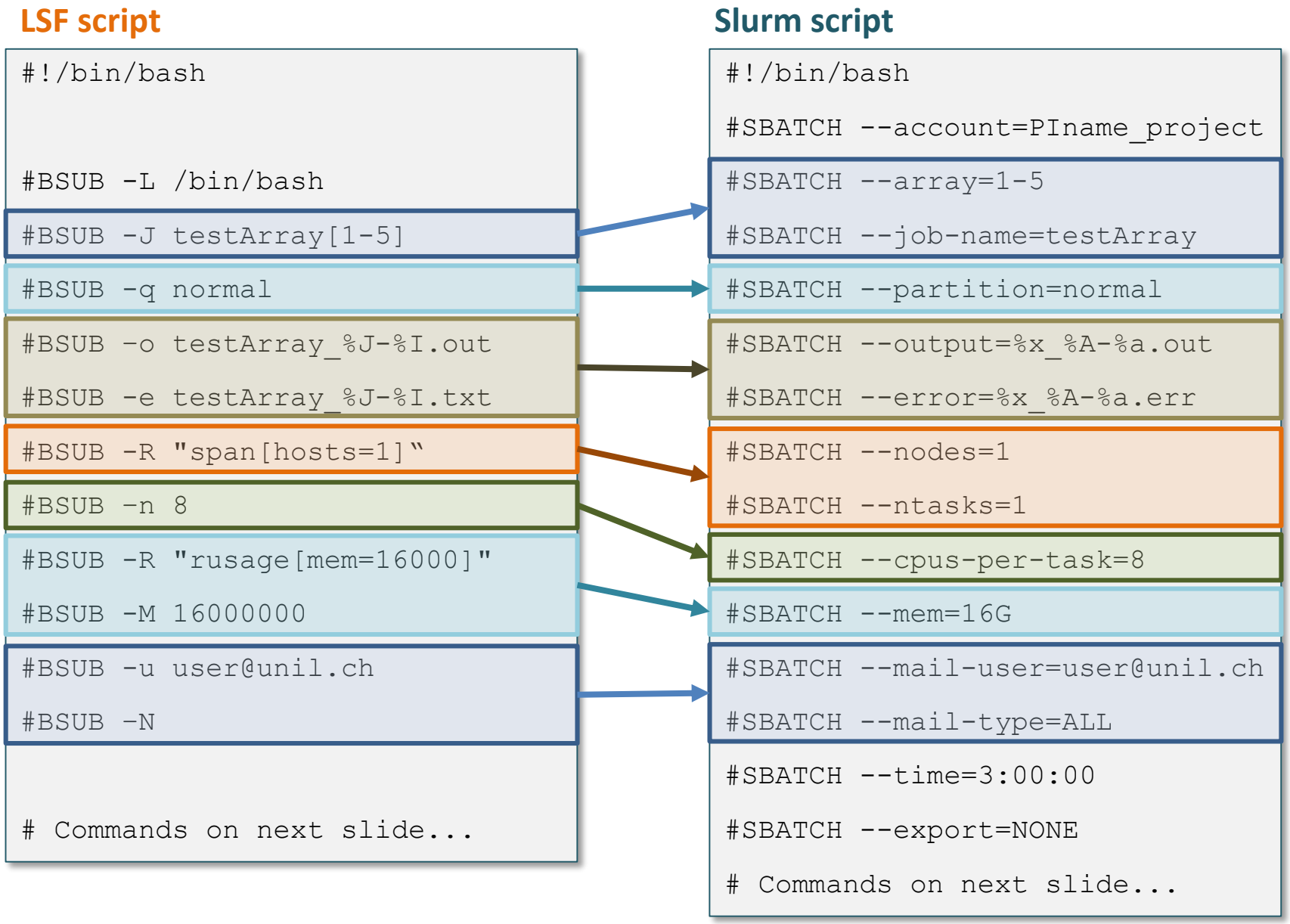
$SLURM_ARRAY_TASK_ID expands to a different value for each job of the array.

LSF to slurm:

script conversion example

Script conversion example: array job script

For each of 5 samples, align reads in the sample against a reference using the “bowtie” aligner.



Script conversion example: array job script

```
# Set input and output directories, as well as sample list to process.
rootDir=/scratch/cluster/monthly/$USER
rootDir=/scratch/wally/<Institution>/<faculty>/<department>/<PI>/<project_name>
inputDir=$rootDir/fastQ
sampleList=( ERR315856 ERR315857 ERR315858 ERR594295 ERR594296 )

# Load modules for bowtie and samtools.
module load Bioinformatics/Software/vital-it
module add UHTS/Aligner/bowtie2/2.3.1
module add UHTS/Analysis/samtools/1.3

# Print progress to stdout.
echo "### Starting log for replicate
[$LSB_JOBINDEX] of array job [$LSB_JOBNAME]."
[$SLURM_ARRAY_TASK_ID/$SLURM_ARRAY_TASK_COUNT] of array job [$SLURM_JOB_NAME]."


# Get name of sample to process in the current job index.
# Reminder: bash arrays are 0-based so we need to subtract 1 to the array ID.
sampleName=${sampleList[(($LSB_JOBINDEX - 1))]}
sampleName=${sampleList[(($SLURM_ARRAY_TASK_ID - 1))]}
echo "### Sample for the current run is: $sampleName"
```

Script conversion example: array job script

```
# Create unique output directory based on job ID name.
outputDir=$rootDir/run_$LSB_JOBID
outputDir=$rootDir/run_$SLURM_ARRAY_JOB_ID
mkdir -p $outputDir
cd $outputDir

# Align sample reads against contig file.
# Run bowtie and pipe the output through samtools.
readFile1="$inputDir/${sampleName}_1_val_1.fq.gz"
readFile2="$inputDir/${sampleName}_2_val_2.fq.gz"
indexFile="$inputDir/$contigIndexFile_$sampleName"
echo "### Running bowtie + samtools..."

cpuCount=8
bowtie2 -q --fast --phred33 --time --threads $cpuCount -x $indexFile \
   -1 $readFile1 -2 $readFile2 2> $outputDir/run_$LSB_JOBINDEX.log | \
   samtools view -h -F 12 -b --threads $cpuCount | \
   samtools sort --threads $cpuCount \
   -o ${outputDir}/${sampleName}_alignedAndSorted.bam

bowtie2 -q --fast --phred33 --time --threads $SLURM_CPUS_PER_TASK -x $indexFile \
    -1 $readFile1 -2 $readFile2 2> $outputDir/run_$SLURM_ARRAY_JOB_ID.log | \
    samtools view -h -F 12 -b --threads $SLURM_CPUS_PER_TASK | \
    samtools sort --threads $SLURM_CPUS_PER_TASK \
    o ${outputDir}/${sampleName}_alignedAndSorted.bam"
```

Additional resources

Official slurm doc: https://slurm.schedmd.com/

helpdesk@unil.ch