Triton Cheatsheet v1.6

Full info: https://scicomp.aalto.fi/triton/

About Triton

- Over 7000 CPUs, 80 GPUs available, up to 256GB or 1TB memory/node.
- Available for all Aalto staff for any research.
- Good integration with department workstations: most filesystems are cross-mounted and you can easily open and process files as if they were local.
- Rather than expect your workstation to do everything, develop to Triton and you can scale up to whatever resources you need.
- Example Triton workflows: Test code on frontend node. Submit interactive test jobs with "`srun -p debug ./your-command`" for *fast* testing. For production runs, do the same but to bigger partitions using more CPUs, or use batch submissions. Examine output on your own workstation via `/m/$dept/scratch/`.
- Own group's server: your own dedicated node for interactive work. Ask for info.

Getting help *docs: User guide/Getting help*

- All information on https://scicomp.aalto.fi/triton/. Includes quickstart tutorials.
- Issue tracker: https://scicomp.aalto.fi/triton/issues (not by email)
- triton-users mailing list: for announcements, you are automatically added.
- CS, NBE, and PHYS IT include Triton administrators and provide support in person.
- SciComp garage: once per week, in-person help and brainstorming session.

Accounts *docs: Account on Triton*

- Accounts are the same as Aalto accounts, but needs activation. Contact `esupport-triton@aalto.fi`.
- Login: ssh to `triton.aalto.fi` with Aalto username/password.

Data Storage *docs: User guide/Data storage*

- /scratch is a Lustre filesystem: 2PB, networked and highly parallel. Also available on (CS,NBE) workstations. All calculation data goes here.
- Using local disks can be more efficient for high I/O processes.
- Other department filesystems (CS,NBE) are on login node and group servers.

B=backed up, S=shared

	B	S	
$HOME	O	O	Home dir, 1GB. Codes and configuration, not calculation files.
/m/$dept/scratch/$project/		**O**	Shared Lustre FS. Large and fast. Per-project. NO BACKUPS.
/m/$dept/work/$username/		**O**	Same as above, per-user. NO BACKUPS
/tmp/			Local disk storage. Not backed up.
https://version.aalto.fi			Aalto git repository.
$XDG_RUNTIME_DIR			Ramfs (in-memory filesystem): very temporary but fast space

Software availability *docs: User guide/Application modules tutorial*

- Most software and libraries are in the "module" system. This allows you to select what you need, including exact versions. It just changes environment variables like `$PATH`, `$LD_LIBRARY_PATH`, etc. Use "`env`" prints these.
- Admins can install common software for you: just ask.
- The "`module`" function makes software available. Example: `module load matlab` or `module load matlab/r2019a`.
- Modules also contain dependencies: if you load E, it will automatically load A, B, C, D if needed. So just request what you need.
- The suffixes are *toolchains*: standard compilers and support libraries. Don't mix and match.
- "`which`" shows exactly what a command name will run.

`module avail $pattern`	Search for modules matching pattern.
`module spider $pattern`	Search (full) for modules matching pattern.
`Module show $name`	Show module details, exactly what it does.
`module load $name`	Load a module. Specify version with `$name/$version`.
`module unload $name`	Unload a module.
`module list`	List currently loaded modules.
`module purge`	Remove all loaded modules from the current session.
`module save $alias`	Save/restore currently loaded modules to a collection. Loading a collection is much faster.
`module restore $alias`	
`module savelist`	List saved collections.

Software development

- Modules contain a variety of compilers and other build tools.

Common software *docs: User guide/Applications*

- Multiple versions are available for all of these. By default you load the latest, otherwise give version: `module load $package/$version`
- Python: we recommend the Anaconda modules for general-purpose Python. "`module load anaconda`" for Python 3 (`anaconda2` for Python 2).
- R: `module load r`
- Matlab: `module load matlab`
- Mathematica: `module load mathematica`
- And so on… see user guide and/or discuss your needs with us.

Interactive jobs *docs: User guide/Interactive jobs tutorial*

- Easiest way to use triton: "**Just add srun!**" to your working command, and specify how much power you need. (details described on next page)
- Example: `srun --mem=50G --time=5:00 -c 6 ./your_command`
- `sinteractive` gets you a shell which is also usable for graphical applications.
- `slurm history` shows detailed CPU/memory usage of the process.

Batch jobs *docs: User guide/Serial jobs tutorial*

```
#SBATCH --mem=50G
#SBATCH --cpus-per-node 20
#SBATCH --nodes=5

srun ./step_1 1 5
srun ./step_2 1 5
```

- Once you run interactively, you can make batch jobs which run in the background.
- Example script at left. Options can be inside the script. Output goes to files in the same directory.
- Submit job with `sbatch script-name.sh`
- Monitor with `slurm queue`.
- `slurm history` shows resource usage, including details on CPU/time/memory for *each srun step*.
- Slurm will run the batch script only once. Within it, each srun command will start as many processes as you request tasks (`-n $n`). It is up to you to get the tasks to communicate (but there may be slurm integration).
- Slurm will start as many processes as you specify tasks with `-n $n`. For basic usage, you will want one. That process will be allocated as many CPUs as you request (`-c $n`). If you request multiple tasks or multiple nodes, it runs the process once per task and it is up to you to make them communicate.

Parallel jobs *docs: User guide/Array jobs tutorial*

```
#!/bin/sh
#SBATCH --time=5:00
#SBATCH -n 4
#SBATCH --array=1-10

srun ./my-command  \
    input_$SLURM_ARRAY_TASK_ID  \
    -o OUTPUT_$SLURM_ARRAY_TASK_ID
```

- Once you run batch jobs, you can easily parallelize to access more resources. (same options work for interactive/batch jobs)
- Easy: Array jobs. Use `--array=M-N` with sbatch and you can easily scan parameters using `$SLURM_ARRAY_TASK_ID`. The command is run once with each parameter. Good for parameter sweeps.
    - Example at left: Run with `sbatch script.sh`
- MPI, OpenMP, etc instructions in wiki.
- OpenMP: Usually with `-c`. `export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK`
- MPI: See docs. Usually with `-n`.
- Python/R/other languages: Usually with `-c`, but depends on the code. Must be checked individually.
- Use `seff $job_id` to verify efficiency.

Slurm details *docs: User guide/Reference, Running programs on Triton*

- *Slurm* is the system which allocates CPU, GPUs, etc. to people doing computation.
- The core is a queuing system which fairly prioritizes users. The less you run, the higher your priority.
- Work is submitted as jobs. CPUs, memory, and time must be declared for jobs. Jobs killed if these limits are exceeded too much.
- In general, just declare what you need and slurm will do the right thing.

The following commands give history about jobs:

`slurm queue (slurm qq)`	Your currently queued jobs, or `slurm watch queue` for updating view
`slurm history 1day|2hour|...`	Your recently completed jobs, including detailed time/memory info.
`slurm job $jobid`	Info on a certain job.
`seff $jobid`	Check effectiveness of requested resources.
`squeue / sacct / scontrol`	Advanced info on waiting jobs / finished jobs / running jobs.

Slurm commands **Complete reference:** https://scicomp.aalto.fi/triton/ref/

The following commands submit jobs. All require some of the slurm options.

`srun`	Run a single command on nodes, I/O connected to terminal.
`srun (in batch script)`	Run a job step so that time/memory can be separately tracked
`srun --pty [bash]`	Run a command (or shell) with full terminal support.
`sinteractive`	Start a shell on a node, usable for graphical applications.
`sbatch`	Run a batch script. Submits and returns immediately.
`scancel $job_id`	Cancel a running job

Slurm options for `srun`, `sbatch`, or `#SBATCH` in batch scripts:

`--time XXX`	Total job run time (`HH:MM[:SS]` or `DD-HH`)
`-c N`	Number of cores (per task)
`--mem nnG`	Total memory per node, only for single node jobs.
`--mem-per-cpu nnG`	Total memory per CPU
`-p $partition`	Partition to use (usually leave off) `slurm p`
`-N $N`	Number of nodes
`-n $n`	Number of tasks to start (number of individual srun processes to start)
`-J $job-name`	Specify memorable job name
`-o $file, -e $file`	Job stdout/stderr is saved to this file name. Default to same dir+jobid.
`--array=N-M`	Array job, easy parallelization (*only* with `sbatch`). `$SLURM_ARRAY_TASK_ID`
`--constraint XXX`	Request hardware type (hsw,ivb,wsm,opt,)
`--gres=gpu:n` (request n GPUs, for gpu partition), `--exclusive` (whole-node), `--constraint=` (limit hardware, e.g. `avx`, `hsw`, …, or GPU generations: `kepler`, `pascal`, `volta`),	