

Triton Cheatsheet_{v1.4}

Full info: <http://scicomp.aalto.fi/triton/>

About Triton

- Over 7000 CPUs, 80 GPUs available, up to 256GB(1TB) memory.
- Available for all Aalto staff for any research.
- Good integration with CS/NBE 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 “`srunch -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. Contact us for info.

Getting help *docs: User guide/Getting help*

- All information on our wiki. See URL at top. Includes quickstart tutorial.
- Issue tracker (all issues): <https://version.aalto.fi/gitlab/AaltoScienceIT/triton/issues>
- triton-users mailing list: for announcements, you are automatically added.
- CS, NBE, and PHYS IT include Triton administrators and provide support in person.

Accounts *docs: Account on Triton*

- Accounts are the same as Aalto accounts, but needs activation. Contact `esupport-triton@aalto.fi` or CS/NBE/PHYS IT.
- 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 cross mounted to (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.
- Remote access: department workstations, `talta.aalto.fi`.

B=backded up, S=shared

	B	S	
\$HOME	O	O	Home dir, 1GB. Codes and configuration, not calculation files.
<code>/m/\$dept/scratch/\$project/</code>		O	Shared Lustre FS. Large and fast. Per-project. NO BACKUPS.
<code>/m/\$dept/work/\$username/</code>		O	Same as above, per-user.. NO BACKUPS
<code>/tmp/</code>			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. “`env`” prints these.
- Admins can install common software for you: just ask.
- The “module” function makes software available. Example: `module load FFTW` or `module load FFTW/3.3.4/gompi-2016b`.
- 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.

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

Software development

- Modules contain a variety of compilers and other tools.

Common software *docs: User guide/Applications*

- Multiple versions are available for all of these. By default you load the latest, otherwise `module load $package/$version`
- Python: we recommend the Anaconda modules for general-purpose Python. “`module load anaconda2/latest`” and “`module load anaconda3/latest`” for Python 2 and 3 respectively.
- R: `module load R`
- Matlab: `module load matlab`
- Mathematica: `module load mathematica`
- And so on... see wiki 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*

- Once you run interactively, you can make batch jobs which you do not have to wait for.
- Example script at left. Options can be inside the script. Output goes to files in 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. It will be allocated as many CPUs as you request (`-c $n`). If you request multiple tasks or multiple nodes, it run the process once per task and it is up to you to make them communicate.

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

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

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

- 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 `$_SLURM_ARRAY_TASK_ID`. The command is run once with each parameter. Good for parameter sweeps.
 - Example at left: Run with `sbatch --array 0-3 script.sh`
- MPI, OpenMP, etc instructions in wiki.
- OpenMP: `export OMP_NUM_THREADS=$_SLURM_CPUS_PER_TASK`
- MPI: see wiki

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

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

Slurm *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

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

Slurm commands

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

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

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

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