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

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>$HOME</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>/m/$dept/scratch/$project/</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>/m/$dept/work/$username/</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td>/tmp/</td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="https://version.aalto.fi">https://version.aalto.fi</a></td>
<td>Aalto git repository.</td>
<td></td>
</tr>
<tr>
<td>SXDG_RUNTIME_DIR</td>
<td>Ramfs (in-memory filesystem): very temporary but fast space</td>
<td></td>
</tr>
</tbody>
</table>

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.

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

Software development docs: User guide/Applications

- 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

- 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 (-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

- 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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>slurm queue</code> (slurm <code>qq</code>)</td>
<td>Your currently queued jobs, or slurm <code>watch</code> queue for updating view</td>
</tr>
<tr>
<td>`slurm history 1day</td>
<td>2hour</td>
</tr>
<tr>
<td><code>slurm job $jobid</code></td>
<td>Info on a certain job.</td>
</tr>
<tr>
<td><code>seff $jobid</code></td>
<td>Check effectiveness of requested resources.</td>
</tr>
<tr>
<td><code>squeue / sacct / scontrol</code></td>
<td>Advanced info on waiting jobs / finished jobs / running jobs.</td>
</tr>
</tbody>
</table>

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

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

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

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

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

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

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

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

- `-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 $job-name` Specify memorable job name
- `-o $file, -e $file` Job stdout/stderr is saved to this file name. Default to same dir+$jobid.
- `-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, wsp, 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),