

Introduction to Wulver: Job Efficiency and Resource Utilization

Automate many jobs, chain workflows, and keep long runs going on Wulver

10/21/2026

Abhishek Mukherjee

Research Computing Facilitator, High Performance Computing, NJIT



Today's Agenda

From one job at a time to automated, restartable workflows



01

Interactive Jobs

Get nodes with `salloc` and run job steps with `srun`



02

Job Dependencies

Chain pre-processing, simulation and analysis into a pipeline



03

Job Arrays

Run many similar tasks from one script and a single `sbatch`



04

Checkpoint & Requeue

Save progress, recover from preemption and run past 72 hours



05

Common Problems

Why jobs run slow and how to fix the script

Quick Recap: Wulver Rules That Shape Your Jobs



From the last session, the limits these techniques are built around

72 h

Maximum walltime

For standard, low and high QoS. Debug partitions stop at 8 h

Low

Free, but preemptible

`--qos=low` costs no SUs, but standard and high jobs can stop it

2nd Tue

Monthly maintenance

Jobs whose walltime overlaps the window are held until it ends

1 SU

= 1 CPU-hour

Or 4 GB of RAM for one hour, charged to your PI's allocation. SU charge varies for different partition.

https://hpc.njit.edu/Running_jobs/service-units/



Keep these four rules in mind for the rest of the session

Every technique today builds on the basic `sbatch` script from the last session and has to work within the walltime limit, and preemption on low QoS.

PART 1

Interactive Jobs

Get resources, then run job steps inside them



Interactive Jobs with salloc

Get an allocation first, then launch job steps inside it

```
$ salloc --ntasks=128 \
  --account=<pi_ucid> \
  --time=01:00:00 \
  --qos=standard \
  --partition=general
salloc: Granted job allocation 17
salloc: Nodes node[00-29] are ready

# job steps run side by side
$ srun -n 100 app1 &
$ srun -n 20 app2 &
$ srun -n 8 app3 &
$ wait          # until all 3 finish

$ sacct -j 17    # one line per step
$ exit          # release the nodes
```

1

salloc reserves resources

Opens a shell with SLURM variables set, for testing and debugging

2

Where the shell runs

Login node or first compute node, set by `use_interactive_step`

3

srun launches job steps

Each `srun` is a step; ending a line with `&` runs the steps together

4

Check steps, then exit

`sacct -j <job_id>` lists each step; `exit` frees the nodes

Exclusive Nodes with `--exclusive`

Ask for 4 cores, get the whole node, and pay SUs for the whole node

```
$ salloc --ntasks=4 \
  --exclusive \
  --account=<pi_ucid> \
  --time=01:00:00 \
  --qos=standard \
  --partition=general
salloc: Granted job allocation 18
salloc: Nodes node05 are ready

# 4 cores requested, full node held
$ srun -n 4 appl          # uses 4 cores
$ exit                   # release the node
```

1

`--exclusive` locks the node

No other job can share the node, even when you request only 4 cores

2

SUs cover the full node

Every core on the node is billed for the whole run, not just the 4 you asked for

3

When it is worth it

Benchmarking, or jobs that need all of a node's memory or bandwidth

4

Otherwise, leave it off

On a shared node, SUs track only the cores you request

PART 2

Job Dependencies

Build pipelines that run themselves



What Are Job Dependencies?

Start a job only when a condition on another job is met

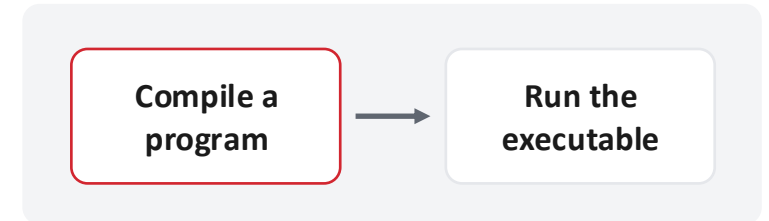
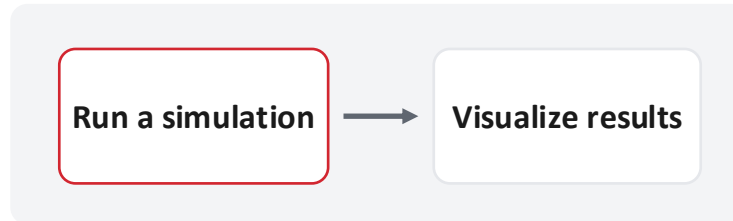


SLURM keeps the job in the queue until the condition is met, then starts it for you

You submit every step at once instead of watching the queue and submitting the next step by hand.

```
sbatch --dependency=<type>:<job_id>[:<job_id>...] next_step.sh
```

COMMON WORKFLOWS



No babysitting

SLURM starts each step when its parent meets the condition



Visible in the queue

Waiting steps show ST=PD with reason (Dependency)

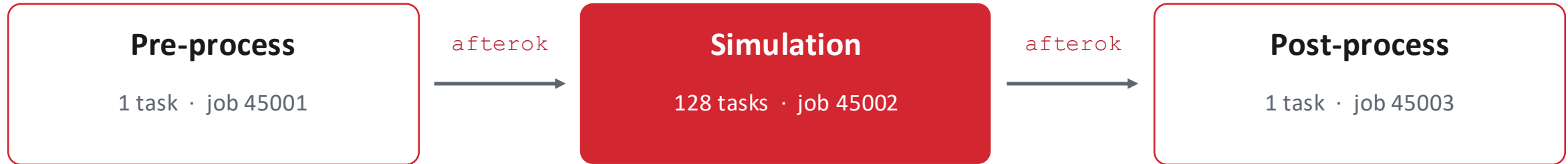


Failures stop the chain

A step can wait for its parent to succeed, so bad output never moves on

Submitting a Sequence of Jobs

Three batch jobs submitted at once; each waits for the one before it



```

$ sbatch --ntasks=1 --parsable pre_process.sh
45001
$ sbatch --ntasks=128 --parsable \
  --dependency=afterok:45001 do_work.sh
45002
$ sbatch --ntasks=1 --parsable \
  --dependency=afterok:45002 post_process.sh
45003
  
```

OPTIONS USED

--ntasks

Number of tasks; by default, the number of cores

--parsable

Print only the job ID, so it is easy to reuse

--dependency

The condition on another job, here `afterok`

slurm.schedmd.com/sbatch.html#OPT_dependency

Dependency Types

Pick the condition that matches what the next step needs

Condition	The job can start when...	Typical use
<code>after:<id></code>	the listed jobs have started running	A monitor or helper that runs alongside
<code>afterok:<id></code>	the listed jobs finished with exit code 0	The next step of a pipeline
<code>afternotok:<id></code>	the listed jobs failed: non-zero exit, timeout, node failure	Clean-up, alerts or a retry job
<code>afterany:<id></code>	the listed jobs ended, in any state	Collecting logs; chaining checkpointed runs
<code>aftercorr:<id></code>	the matching task of another array succeeded	Array-to-array pipelines
<code>singleton</code>	earlier jobs with the same name and user have ended	One-at-a-time runs of a long job



Each type takes one or more IDs, `job_id[:job_id...]`. Combine conditions with `,` (all must be met) or `?` (any one is enough).

When Dependencies Go Wrong

Read the reason column in `squeue` before you resubmit

WHAT YOU SEE



PD (Dependency)

The parent job has not finished yet



PD (DependencyNeverSatisfied)

The parent failed, so `afterok` can never be met



Wrong parent job ID

The job waits on the wrong step

WHAT TO DO



Nothing: this is normal, the job starts on its own



scancel it, or
`scontrol update JobId=<id>`
`Dependency=afternotok:<first_job_id>`



`scontrol update JobId=<id>`
`Dependency=afterok:<new_id>`

The singleton Dependency

Only one job with a given name runs at a time, per user

```
# run.sh
#!/bin/bash -l
#SBATCH --job-name=longsim
#SBATCH --partition=general
#SBATCH --qos=standard
#SBATCH --account=<PI_UCID>
#SBATCH --ntasks=8
#SBATCH --time=72:00:00
#SBATCH --dependency=singleton
#SBATCH --output=%x.%j.out
# continue from the last checkpoint
srun ./my_sim --resume latest.chk
```

```
# submit it three times, all at once
$ sbatch run.sh
$ sbatch run.sh
$ sbatch run.sh
```

```
$ squeue -u $LOGNAME
JOBID  NAME      ST  NODELIST(REASON)
45001  longsim   R   n0042
45002  longsim   PD   (Dependency)
45003  longsim   PD   (Dependency)
```

WHAT HAPPENS

45001 runs first

45002 starts when 45001 ends, whether it completed, hit the 72 h limit, or failed

45003 starts when 45002 ends

Each run resumes from the checkpoint the previous one saved

OR SET IT ON THE COMMAND LINE

Leave the `--dependency` line out of `run.sh` and use:

```
sbatch --job-name=longsim \
    --dependency=singleton run.sh
```

PART 3

Job Arrays

One script, one sbatch, many tasks



What Are Job Arrays?

Submit and manage a collection of similar jobs easily



One `sbatch` creates many tasks that run the same script

Each task gets its own task ID, so each one can work on a different input file or parameter.

```
$ sbatch --array=1-50 -N1 -i my_in_%a -o my_out_%a my.bash
```

50-element job array

```
Submitted batch job 123
```

```
# 50 tasks: 123_1 ... 123_50      -N1: 1 node per task    -i: input file    -o: output file
```

```
# default logs: slurm-<job_id>_<task_id>.out, e.g. slurm-123_1.out, slurm-123_2.out
```



Batch jobs only

Arrays are supported for `sbatch` jobs, not interactive sessions



Fast to submit

The big benefit: under 1 second for all 50 tasks



%a in file names

Mapped to the task ID: task 7 reads `my_in_7`



One log per task

By default `slurm-123_1.out`, `slurm-123_2.out`, ...

Job Array Example

array.sh: 30 tasks, one input file each

```
#!/bin/bash -l
#SBATCH -J myprogram
#SBATCH --partition=general
#SBATCH --qos=standard
#SBATCH --account=<PI_UCID>
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --array=1-30
#SBATCH --output=myprogram%A_%a.out
#SBATCH --error=myprogram%A_%a.err
# %A = array job ID, %a = array index
#SBATCH --time=71:59:59

./myprogram input$SLURM_ARRAY_TASK_ID.dat
sleep 10
```

1

--array=1-30

Creates 30 tasks with IDs 1 to 30

2

%A_%a in the log names

One .out and .err per task, nothing overwritten

3

\$SLURM_ARRAY_TASK_ID

Task 7 runs ./myprogram input7.dat

4

Resources are per task

Each task gets 1 node, 1 core and up to 72 h

5

Try it yourself

Copy the example from [/apps/testjobs/job_array](/apps/testjobs/job_array)

Why Use Job Arrays?

Use case: run app on every line of a dataset, then compare the runs against each other

BEFORE: A LOOP OF SBATCH CALLS

```
submit_all.sh
#!/bin/bash
DATASET=dataset.txt
scriptnum=0
while read LINE; do
    echo "app $LINE" > ${scriptnum}.sh
    sbatch ${scriptnum}.sh
    scriptnum=$(( scriptnum + 1 ))
done < $DATASET
```



Needlessly complex and slow: 100 scripts, and `sbatch` has to verify each job as it is submitted

AFTER: ONE ARRAY JOB

```
array.sh
#!/bin/bash -l
#SBATCH --job-name=app
#SBATCH --array=1-100
#SBATCH --output=app_%A_%a.out
#SBATCH --time=01:00:00

# task N reads line N of the dataset
LINE=$(sed -n "${SLURM_ARRAY_TASK_ID}p" dataset.txt)
srun app $LINE
```



One script, one job ID, submitted in under a second. Use `wc -l dataset.txt` to size the array

Array Syntax: --array

Choose exactly which task IDs to create

Specification	Task IDs created	Use it for
<code>--array=1-30</code>	1, 2, 3, ... 30	One task per numbered input file
<code>--array=0-20:5</code>	0, 5, 10, 15, 20	Stepping through a parameter range
<code>--array=1, 4, 9</code>	1, 4, 9	Re-running only the tasks that failed
<code>--array=1-500%20</code>	1 ... 500, at most 20 running at once	Throttling to protect SUs, storage and the queue



Each array task is its own job

The `#SBATCH` resources apply to every task, each task waits in the queue on its own, and each is charged SUs separately. A 100-task array of 4-core jobs can use up to 400 cores.

Array Variables and File Names

How each task knows which piece of work is its own

Variable	Value
SLURM_ARRAY_TASK_ID	This task's index, e.g. 7
SLURM_ARRAY_JOB_ID	Job ID shared by the whole array
SLURM_JOB_ID	Unique job ID of this task
SLURM_ARRAY_TASK_MIN	Lowest task ID in the array
SLURM_ARRAY_TASK_MAX	Highest task ID in the array
SLURM_ARRAY_TASK_COUNT	Number of tasks in the array

FILE NAME PATTERNS

- %A** Array job ID
- %a** Array task ID
- %j** Job ID of this task
- %x** Job name

```
--output=run_%A_%a.out
→ run_45001_7.out
```

Leave out %a and every task writes to one file

Mapping Task IDs to Inputs

Four common patterns; pick the one that matches your data



Numbered input files

input1.dat, input2.dat, ... input30.dat

```
#SBATCH --array=1-30
./myprogram input${SLURM_ARRAY_TASK_ID}.dat
```



One line per task

Each line of a text file is one case

```
#SBATCH --array=1-100
LINE=$(sed -n "${SLURM_ARRAY_TASK_ID}p" cases.txt)
./app $LINE
```



Parameter sweep

Values listed in a bash array, indexed from 0

```
#SBATCH --array=0-3
LR=(0.1 0.01 0.001 0.0001)
python train.py --lr ${LR[$SLURM_ARRAY_TASK_ID]}
```



Every file in a folder

Size the array with `ls data/*.csv | wc -l`

```
#SBATCH --array=0-49
FILES=(data/*.csv)
./app ${FILES[$SLURM_ARRAY_TASK_ID]}
```

Monitoring and Controlling Arrays

Address the whole array, one task, or a range of tasks

List every task

Pending tasks are shown collapsed unless you add -r

```
squeue -u $LOGNAME -r
```

Cancel the whole array

Use the array job ID alone

```
scancel 45001
```

Cancel one task

Array job ID, underscore, task ID

```
scancel 45001_7
```

Cancel a range

Brackets select a range of task IDs

```
scancel 45001_[10-20]
```

Change the throttle

Raise or lower how many tasks run at once

```
scontrol update JobId=45001  
ArrayTaskThrottle=5
```

Find failed tasks

Resubmit just the failed IDs, e.g. 4,9

```
sacct -j 45001  
-o JobID,State,ExitCode
```

Dependencies with Job Arrays

Dependencies work by individual task IDs or by an entire job array

Start once a specific task begins

```
--dependency=after:45001_4
```

Wait for specific tasks to succeed

```
--dependency=afterok:45001_4:45001_8
```

Task N waits for task N

```
--dependency=aftercorr:45001
```

Wait for the whole array

```
--dependency=afterok:45001
```

45001

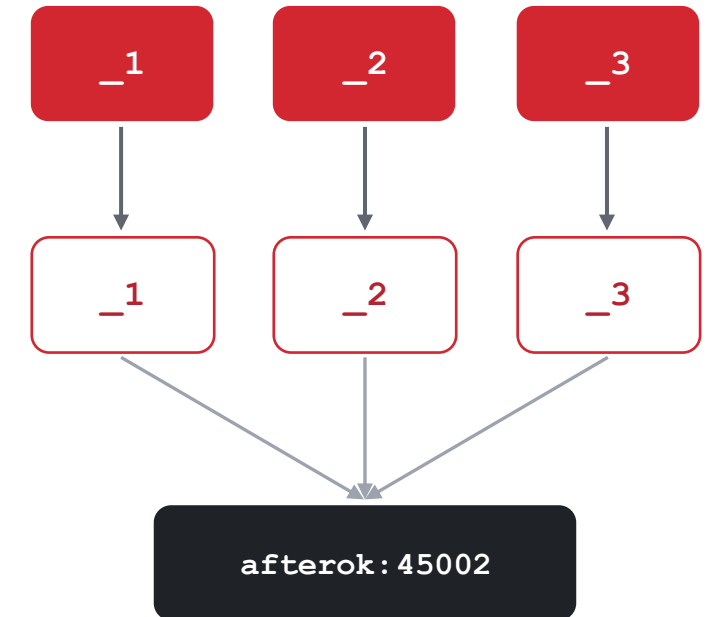
simulate

45002

post-process
aftercorr:45001

45003

summarize



Task `_2` of 45002 starts as soon as task `_2` of 45001 succeeds. 45003 waits for every task of 45002.

PART 4

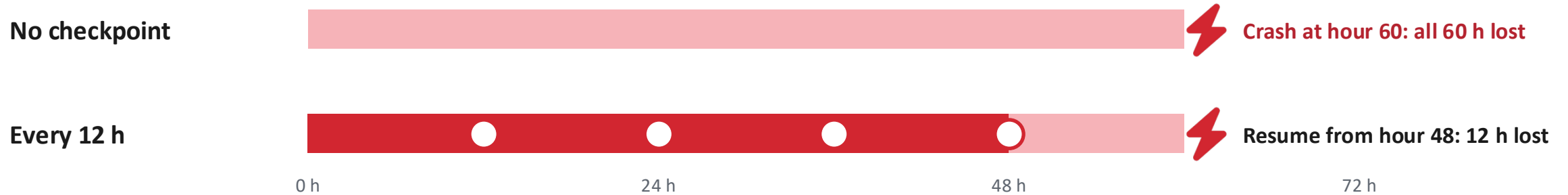
Checkpointing and Requeue

Save progress, survive interruptions, run past 72 hours



What Is Checkpointing?

Save the state of a running job so it can resume instead of starting over



Failure recovery

Resume after a node crash
or a job that hits its time
limit



No wasted SUs

Interrupted work is not
recomputed from the start



Preemption tolerance

Makes free low-QoS jobs
safe to stop and restart



Past the 72 h limit

Split one long run into a
chain of shorter jobs

Most Software Already Checkpoints

Look for "restart" or "checkpoint" in the manual before writing your own

Software	Save a checkpoint	Restart from it
GROMACS	<code>gmx mdrun -deffnm sim -cpt 15</code>	<code>gmx mdrun -deffnm sim -cpi sim.cpt</code>
LAMMPS	<code>write_restart restart.equil</code>	<code>read_restart restart.equil</code>
OpenFOAM	<code>writeInterval 100; (controlDict)</code>	<code>startFrom latestTime;</code>
PyTorch	<code>torch.save(state, 'ckpt.pth')</code>	<code>model.load_state_dict(...)</code>
TensorFlow / Keras	<code>callbacks.ModelCheckpoint(...)</code>	<code>model.load_weights(path)</code>
Plain Python	<code>pickle.dump(state, f)</code>	<code>state = pickle.load(f)</code>



GROMACS `-cpt 15` writes a checkpoint every 15 minutes. Examples for each tool: hpc.njit.edu/Running_jobs/checkpointing

Writing Your Own Checkpoint

The resume pattern, shown in PyTorch

train.py

```
ckpt = "checkpoint.pth"
start_epoch = 0
if os.path.exists(ckpt):          # resume?
    state = torch.load(ckpt)
    model.load_state_dict(state["model"])
    optimizer.load_state_dict(state["optim"])
    start_epoch = state["epoch"] + 1

for epoch in range(start_epoch, num_epochs):
    train_one_epoch(model, optimizer)
    torch.save({"epoch": epoch,
               "model": model.state_dict(),
               "optim": optimizer.state_dict()},
              ckpt + ".tmp")
    os.replace(ckpt + ".tmp", ckpt)  # atomic
```

1

Load it if it exists

The same script works for a fresh start and a restart

2

Save everything you need

Model, optimizer and the step or epoch counter

3

Save on a steady interval

Every epoch or every N minutes, not every step

4

Write, then rename

A job killed mid-save never leaves a broken file

5

Use shared storage

PROJECT or SCRATCH, never node-local / tmp

DMTCP: Checkpoint Any Program

Transparent, system-level checkpointing for code with no restart option

FIRST RUN · `sbatch dmtcp.submit.sh` (examples:
`/apps/testjobs/DMTCP`)

```
#!/bin/bash -l
#SBATCH --job-name=dmtcp-test
#SBATCH --output=%x.%j.out
#SBATCH --partition=general
#SBATCH --qos=standard
#SBATCH --ntasks-per-node=4
#SBATCH --time=30:00

module purge
module load foss/2025a DMTCP/4.0.0
source start_coordinator.sh
start_coordinator -i 60 # every 60 s
dmtcp_launch -j ./test.py
```

RESTART · `sbatch dmtcp.restart.sh`

```
#!/bin/bash -l
#SBATCH --job-name=dmtcp-restart
#SBATCH --output=%x.%j.out
#SBATCH --partition=general
#SBATCH --qos=standard
#SBATCH --ntasks-per-node=4
#SBATCH --time=30:00

module purge
module load foss/2025a DMTCP/4.0.0
source start_coordinator.sh
start_coordinator -i 60 # every 60 s
./dmtcp_restart_script.sh
```



MPI codes: `dmtcp_launch --rm mpiexec ./app` or `srun dmtcp_launch --rm ./app` (PMI1; PMI2 is not supported)

Running Past 72 Hours: Chain Checkpointed Jobs



One 250-hour run as four jobs of up to 72 hours, each resuming from the last checkpoint

Job 1 · hours 0–72

Job 2 · hours 72–144

Job 3 · hours 144–216

Job 4 · hours 216–250

Option A: singleton

Same job name; only one runs at a time

```
for i in 1 2 3 4; do
  sbatch --job-name=longrun \
        --dependency=singleton run.sh
done
```

Option B: afterany chain

Each job waits for the one before it to end

```
jid=$(sbatch --parsable run.sh)
for i in 2 3 4; do
  jid=$(sbatch --parsable \
        --dependency=afterany:$jid run.sh)
done
```



Use `afterany`, not `afterok`: a job stopped at its time limit ends as `TIMEOUT`, which counts as a failure. `run.sh` must resume from the latest checkpoint on its own.

Requeue: Let SLURM Restart Your Job

The job returns to the queue with the same job ID and runs the script again

```
#!/bin/bash -l
#SBATCH --job-name=sweep
#SBATCH --partition=general
#SBATCH --qos=low           # free, preemptible
#SBATCH --account=<PI_UCID>
#SBATCH --ntasks=8
#SBATCH --time=72:00:00
#SUBATCH --requeue           # come back if stopped
#SUBATCH --open-mode=append # keep earlier logs
#SBATCH --output=%x.%j.out

module load foss/2025a OpenFOAM
source $FOAM_BASH

# Run the job using
requeue_job mpirun interFoam -parallel
```

1

When it happens

Preemption or node failure requeues it automatically;
`requeue_job` handles the 72 h limit

2

--requeue

Lets SLURM put the job back in the queue; `--no-requeue` forbids it

3

--open-mode=append

Each run adds to the log instead of replacing it

4

requeue_job

Runs your command, then requeues the job at 71.5 hours, before the 72-hour limit

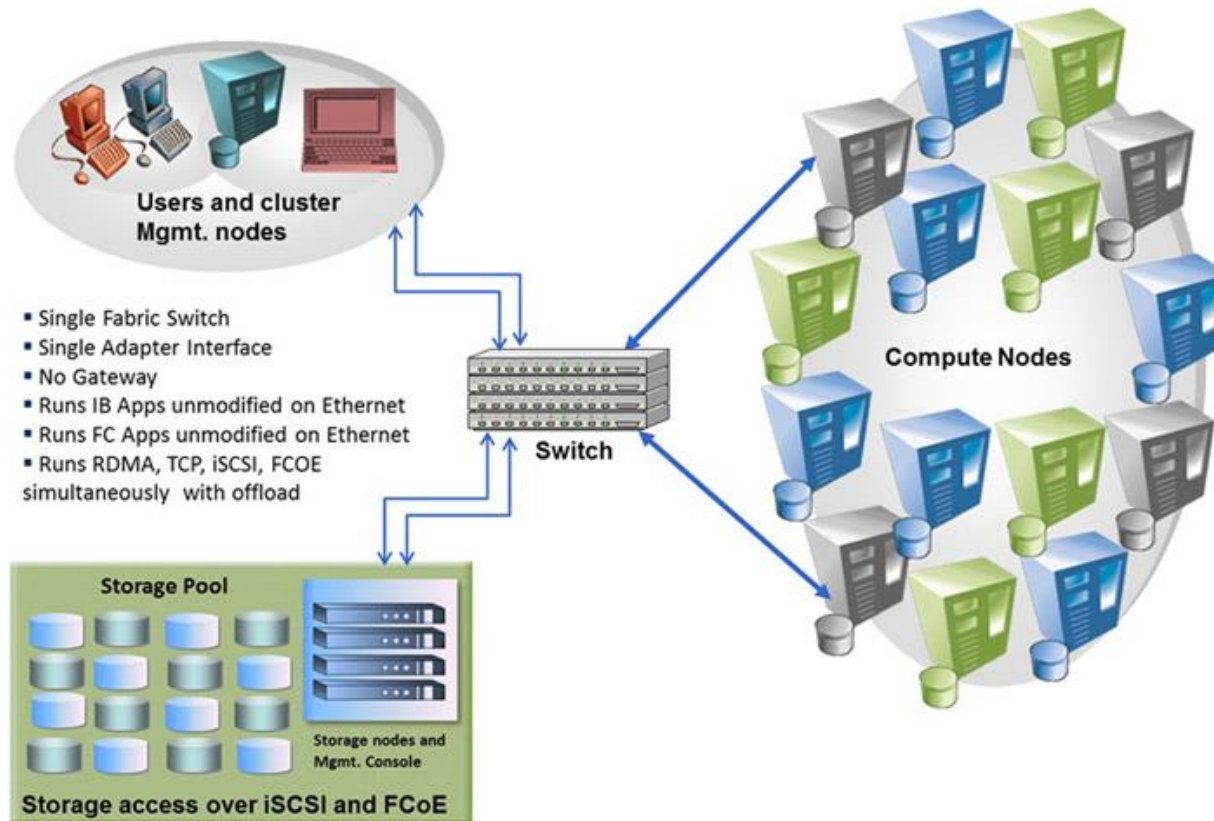
5

Make it restart-safe

Resume from a checkpoint; never delete earlier results

A Network Switch Links a Group of Nodes

Traffic between nodes on different switches takes extra hops



1

What a switch is

Network hardware that connects a group of compute nodes over InfiniBand

2

Same switch: one hop

Nodes under one switch exchange MPI messages directly, with the lowest latency

3

Different switches: more hops

Messages travel up to a core switch and back down, adding delay

4

Why it matters

--ntasks=128 may land on 16 nodes × 8 cores, spread across several switches

Diagram source: Chelsio, High Performance Cluster Computing

Keep Multi-Node Jobs on One Network Switch

--switches trades extra queue wait for lower inter-node latency

```
#!/bin/bash -l
#SBATCH --job-name=switch-test
#SBATCH --output=%x.%j.out # JobName.JobID
#SBATCH --error=%x.%j.err
#SBATCH --partition=general
#SBATCH --qos=standard
#SBATCH --ntasks=64
#SBATCH --nodes=4
#SBATCH --time=30:00
#SBATCH --switches=1@8:00:00 # 1 switch, wait ≤8 h

srun ./my_mpi_app
```

1

Same switch, fewer hops

Nodes on different switches add latency to every MPI message between them

2

--switches=1@8:00:00

Use nodes under 1 switch; after 8 h of waiting, SLURM starts the job anyway

3

--nodes=1-4

Caps the 64 tasks at 4 nodes instead of scattering them across the cluster

4

Test before adopting

Time the job with and without it; loosely coupled codes may see no gain

Recap: Which Tool for Which Problem?

Four everyday problems, four SLURM features

THE PROBLEM



I need to process 500 input files

Writing 500 job scripts by hand is slow and error-prone



Step 2 must wait for step 1

Checking the queue and submitting by hand wastes hours



My run needs more than 72 hours

Every job stops at the walltime limit



My multi-node MPI job runs slow

Its nodes are spread across switches, adding latency

THE SLURM FEATURE



Job arrays

`--array=1-500`



Job dependencies

`--dependency=afterok:<job_id>`



Checkpoint + chained jobs

`--dependency=singleton` or `requeue_job`



Switch-aware placement

`--switches=1@8:00:00`

Recap: sbatch Options Used Today

The flags from this session in one place

Option	What it does	Example
<code>--exclusive</code>	Reserve whole nodes; no other jobs share them	<code>sbatch --exclusive job.sh</code>
<code>--array</code>	Submit a job array of similar tasks	<code>--array=1-100%10</code>
<code>--dependency</code>	Start only when other jobs meet a condition	<code>--dependency=afterok:12345</code>
<code>--parsable</code>	Print only the job ID, ready for scripts	<code>jid=\$(sbatch --parsable a.sh)</code>
<code>--requeue</code> / <code>--no-requeue</code>	Allow or forbid requeue after preemption or node failure	<code>--requeue</code>
<code>--open-mode</code>	Append to log files instead of overwriting on restart	<code>--open-mode=append</code>
<code>--switches</code>	Keep all nodes under a set number of network switches	<code>--switches=1@8:00:00</code>

PART 5

Common Problems

Misconceptions that make jobs run slower, and how to fix the script



More CPUs Help Only If the Code Runs in Parallel



`srun` starts one copy of a serial script for every task

```
#!/bin/bash -l
#SBATCH --job-name=python
#SBATCH --output=%x.%j.out # JobName.JobID
#SBATCH --error=%x.%j.err
#SBATCH --partition=general
#SBATCH --qos=standard
#SBATCH --ntasks=4 # 4 tasks
#SBATCH --time=30:00 # D-HH:MM:SS

module load foss/2025a Python
srun python test.py # runs test.py 4 times
```

1

--ntasks=4 means 4 copies

`srun` launches the command once per task, so a serial script runs 4 times

2

Serial script? Run one copy

Use `python test.py` or `srun -n1 python test.py`

3

Want a speedup? Parallelize

Add `mpi4py` (MPI) or `Parsl` (parallel tasks) to the script itself

4

Request what the code uses

Idle CPUs still count against your allocation and lengthen the queue wait

Oversubscribed Node: 256 Threads on 128 Cores



Why this GROMACS job runs slower on HPC than expected

```
#!/bin/bash -l
#SBATCH -J gmx-test
#SBATCH -o %x.%j.out
#SBATCH -e %x.%j.err
#SBATCH --partition=gpu
#SBATCH --qos=standard
#SBATCH --time=72:00:00 # max 3 days
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=128 # 128 tasks
#SBATCH --gres=gpu:4

module purge
module load wulver
module load foss/2021b GROMACS/2021.5-CUDA-11.4.1
gmx grompp -f run.mdp -c npt2.gro \
  -r npt2.gro -p topol.top -o run.tpr
srun gmx_mpi mdrun -deffnm run -cpi run.cpt \
  -v -ntomp 2 -pin on -tunepme \ # *2 threads
  -dlb yes -noappend
```

1

128 MPI tasks per node

--ntasks-per-node=128 already fills every core on the node

2

-ntomp 2 doubles the threads

Each task starts 2 OpenMP threads: $128 \times 2 = 256$ threads

3

Threads compete for cores

With 2 threads per core, each one waits for its turn and the run slows down

4

Spot it with checkload

A load near 256 on a 128-core node means the node is oversubscribed

Fix 1: Fit Tasks × Threads on One Node

64 tasks × 2 threads = 128 threads on 128 cores

```
#!/bin/bash -l
#SBATCH -J gmx-test
#SBATCH -o %x.%j.out
#SBATCH -e %x.%j.err
#SBATCH --partition=gpu
#SBATCH --qos=standard
#SBATCH --time=72:00:00 # max 3 days
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=64 # 64 tasks
#SBATCH --cpus-per-task=2 # 2 cores each
#SBATCH --gres=gpu:4
module purge
module load wulver
module load foss/2021b GROMACS/2021.5-CUDA-11.4.1
gmx grompp -f run.mdp -c npt2.gro \
  -r npt2.gro -p topol.top -o run.tpr
srun gmx_mpi mdrun -deffnm run -cpi run.cpt \
  -v -ntomp 2 -pin on -tunepme \
  -dlb yes -noappend
```

1

Check, then cancel

Run checkload; if overloaded, scancel <job_id> and fix the script

2

--ntasks-per-node=64

Halve the MPI tasks so each one has room for its threads

3

--cpus-per-task=2

Reserve 2 cores per task to match -ntomp 2

4

One thread per core

64 × 2 = 128 threads on a 128-core node, no oversubscription

Fix 2: Spread Tasks Across Two Nodes

128 tasks × 2 threads = 256 cores across 2 nodes

```
#!/bin/bash -l
#SBATCH -J gmx-test
#SBATCH -o %x.%j.out
#SBATCH -e %x.%j.err
#SBATCH --partition=gpu
#SBATCH --qos=standard
#SBATCH --time=72:00:00 # max 3 days
#SBATCH --nodes=2          # 2 nodes
#SBATCH --ntasks-per-node=64 # 128 tasks
#SBATCH --cpus-per-task=2   # 256 cores
#SBATCH --gres=gpu:4        # per node
module purge
module load wulver
module load foss/2021b GROMACS/2021.5-CUDA-11.4.1
gmx grompp -f run.mdp -c npt2.gro \
  -r npt2.gro -p topol.top -o run.tpr
srun gmx_mpi mdrun -deffnm run -cpi run.cpt \
  -v -ntomp 2 -pin on -tunepme \
  -dlb yes -noappend
```

1

--nodes=2

Two 128-core nodes give 256 cores for 128 tasks × 2 threads

2

--ntasks-per-node=64

64 tasks on each node, 128 MPI tasks in total

3

--gres counts per node

gpu:4 now means 4 GPUs on each node, 8 in total

4

Add --switches

Keep both nodes under one switch to limit MPI latency between them

Resources to Get Your Answers

Docs, example scripts and support



LEARN

Running Jobs Docs

[Array jobs](#)

[Dependency jobs](#)

[Checkpointing](#)

[Managing jobs](#)



TRY IT

Example Scripts

`/apps/testjobs/job_array`

`/apps/testjobs/DMTCP`

[DMTCP module](#)

[sbatch manual](#)



SUPPORT

Get Help

[HPC Contact](#)

[Email a ticket](#)

[HPC Workshops](#)



UPDATES

System Updates

Message of the Day at login

[NJIT HPC News](#)

`hpc.njit.edu`

`man sbatch`

`hpc@njit.edu`

`hpc.njit.edu/news`

Best Practices and Next Steps

Automate the repetition, plan for interruption



Arrays, not loops

One sbatch, %A_%a logs, throttle with %N



Chain, don't babysit

--parsable plus --dependency builds the pipeline



Checkpoint long runs

Resume automatically; save to PROJECT or SCRATCH



Plan for interruption

--requeue, --open-mode=append, after any chains



TRY IT ON WULVER

Copy the examples in `/apps/testjobs/job_array` and `/apps/testjobs/DMTCP`

Docs: hpc.njit.edu/Running_jobs · Questions: submit a support ticket · Prefer a GUI? Try Open OnDemand