

Save Your Progress: Checkpointing on FASRC Clusters

Harvard - FAS Research Computing

Who We Are?

- Team of network, software, hardware engineers & scientists to
  - Support data-intensive & computational research across disciplines
  - Enable & facilitate collaboration, innovation, and reproducibility
- Provide research support to faculty, students, and research staff
  - Collaborate with departments, labs, and libraries to support full research lifecycles requirements
- Expertise across high performance computing (HPC), data management, and scientific software management
  - Focus on performance, reliability, and secure research infrastructure

What we offer?



Objectives

- Understand why jobs get interrupted and what that costs
- Learn the checkpoint/restart pattern and what state to save
- Use SLURM signals and requeue to checkpoint and restart automatically
- Add checkpointing to your own code (Python, C, PyTorch)
- Checkpoint unmodified programs and MPI codes with DMTCP* and MANA*
- Use built-in checkpointing of applications (GROMACS)
- Choose the right approach for your workload

*DMTCP = Distributed MultiThreaded CheckPointing; MANA = MPI-Agnostic Network-Agnostic transparent checkpointing

Outline

- Introduction & Background
- Application-Level Checkpointing: Python
- Transparent Checkpointing: DMTCP and MANA
- Built-in Checkpointing: GROMACS
- Checkpointing Deep Learning: PyTorch
- Best Practices, Wrap-up & Q&A

Training Materials

➔ Download slides

https://docs.rc.fas.harvard.edu/kb/training-materials/

➔ Login to Cannon

```
ssh <username>@login.rc.fas.harvard.edu
```

➔ Clone FASRC User Codes repository https://github.com/fasrc/User_Codes/tree/master

SSH: `git clone git@github.com:fasrc/User_Codes.git`

HTTPS: `git clone https://github.com/fasrc/User_Codes.git`

➔ Go to the training folder

```
cd User_Codes/Training/Checkpointing/
```

Hands-On Practice: Exercise Setup

Create space for the workshop materials and clone the repository

```
# Create space for the workshop materials, e.g.,
mkdir -p /n/netscratch/<LAB_NAME>/Lab/<USER_NAME>/Workshop
cd /n/netscratch/<LAB_NAME>/Lab/<USER_NAME>/Workshop

# Clone the User_Codes GitHub repository
git clone https://github.com/fasrc/User_Codes.git

# Go to the exercise directory
cd User_Codes/Training/Checkpointing/
```

Every exercise folder has a `README.md` with the same sections:
Introduction, **Content**, **Workflow**, **Cleanup**.

Example outputs (`*_JOBID.out`) are included so you can compare your results.

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing

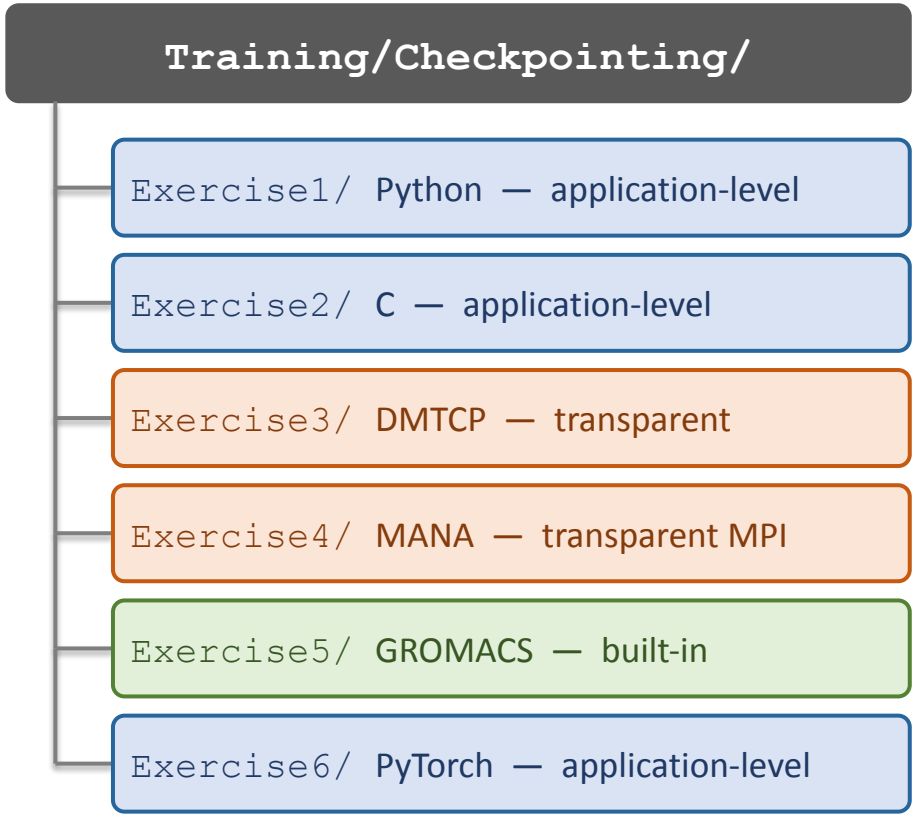
Hands-On Practice: Exercise Setup

Create space for the workshop materials and clone the repository

```
# Create space for the workshop materials, e.g.,
mkdir -p /n/netscratch/<LAB_NAME>/Lab/<USER_NAME>/Workshop
cd /n/netscratch/<LAB_NAME>/Lab/<USER_NAME>/Workshop

# Clone the User_Codes GitHub repository
git clone https://github.com/fasrc/User_Codes.git

# Go to the exercise directory
cd User_Codes/Training/Checkpointing/
```



Every exercise folder has a `README.md` with the same sections: **Introduction**, **Content**, **Workflow**, **Cleanup**.

Example outputs (`*_JOBID.out`) are included so you can compare your results.

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing

The Six Exercises

#	Exercise	Technique	What it covers	Runs on
1	Checkpointing in Python	Application-level	Monte Carlo π: atomic checkpoints, SIGUSR1, automatic requeue	CPU
2	Checkpointing in C	Application-level	Same four steps in C: binary struct, fsync/rename, sigaction	CPU
3	DMTCP	Transparent	Unmodified C program: periodic checkpoints, restart, requeue	CPU
4	MANA	Transparent (MPI)	4-rank MPI program built with mpicc_mana: checkpoint, restart	CPU (MPI)
5	GROMACS	Built-in	Water box: mdrun -cpt, stop with -maxh, restart with -cpi	CPU (MPI)
6	PyTorch	Application-level	MNIST CNN: basic vs complete training checkpoints	GPU

Suggested order: 1 → 2 (same ideas, lower level) → 3 → 4 (transparent alternative) → 5 and 6 in any order.

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing

The Six Exercises

#	Exercise	Technique	What it covers	Runs on
1	**Checkpointing in Python**	Application-level	Monte Carlo π: atomic checkpoints, SIGUSR1, automatic requeue	CPU
2	Checkpointing in C	Application-level	Same four steps in C: binary struct, fsync/rename, sigaction	CPU
3	**DMTCP**	Transparent	Unmodified C program: periodic checkpoints, restart, requeue	CPU
4	MANA	Transparent (MPI)	4-rank MPI program built with mpicc_mana: checkpoint, restart	CPU (MPI)
5	GROMACS	Built-in	Water box: mdrun -cpt, stop with -maxh, restart with -cpi	CPU (MPI)
6	**PyTorch**	Application-level	MNIST CNN: basic vs complete training checkpoints	GPU

Suggested order: 1 → 2 (same ideas, lower level) → 3 → 4 (transparent alternative) → 5 and 6 in any order.

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing

Outline

- Introduction & Background
- Application-Level Checkpointing: Python
- Transparent Checkpointing: DMTCP and MANA
- Built-in Checkpointing: GROMACS
- Checkpointing Deep Learning: PyTorch
- Best Practices, Wrap-up & Q&A

Why Do Jobs Get Interrupted?

1

Wall-time limits

Every partition has a maximum run time (e.g. 3 days on `sapphire`, `shared` and `gpu`). At the limit Slurm ends the job — finished or not.

Why Do Jobs Get Interrupted?

1

Wall-time limits

Every partition has a maximum run time (e.g. 3 days on `sapphire`, `shared` and `gpu`). At the limit Slurm ends the job — finished or not.

2

Preemption

Jobs on `serial_requeue` and `gpu_requeue` run on other groups' idle nodes and are killed and requeued when the owners need them.

Why Do Jobs Get Interrupted?

1

Wall-time limits

Every partition has a maximum run time (e.g. 3 days on `sapphire`, `shared` and `gpu`). At the limit Slurm ends the job — finished or not.

2

Preemption

Jobs on `serial_requeue` and `gpu_requeue` run on other groups' idle nodes and are killed and requeued when the owners need them.

3

Hardware & node failures

Memory errors, disks, network, GPUs falling off the bus. Rare for one node — routine across a large cluster.

Why Do Jobs Get Interrupted?

1

Wall-time limits

Every partition has a maximum run time (e.g. 3 days on `sapphire`, `shared` and `gpu`). At the limit Slurm ends the job — finished or not.

2

Preemption

Jobs on `serial_requeue` and `gpu_requeue` run on other groups' idle nodes and are killed and requeued when the owners need them.

3

Hardware & node failures

Memory errors, disks, network, GPUs falling off the bus. Rare for one node — routine across a large cluster.

4

Maintenance & outages

Scheduled maintenance windows, power or cooling events, filesystem problems.

Why Do Jobs Get Interrupted?

1

Wall-time limits

Every partition has a maximum run time (e.g. 3 days on `sapphire`, `shared` and `gpu`). At the limit Slurm ends the job — finished or not.

2

Preemption

Jobs on `serial_requeue` and `gpu_requeue` run on other groups' idle nodes and are killed and requeued when the owners need them.

3

Hardware & node failures

Memory errors, disks, network, GPUs falling off the bus. Rare for one node — routine across a large cluster.

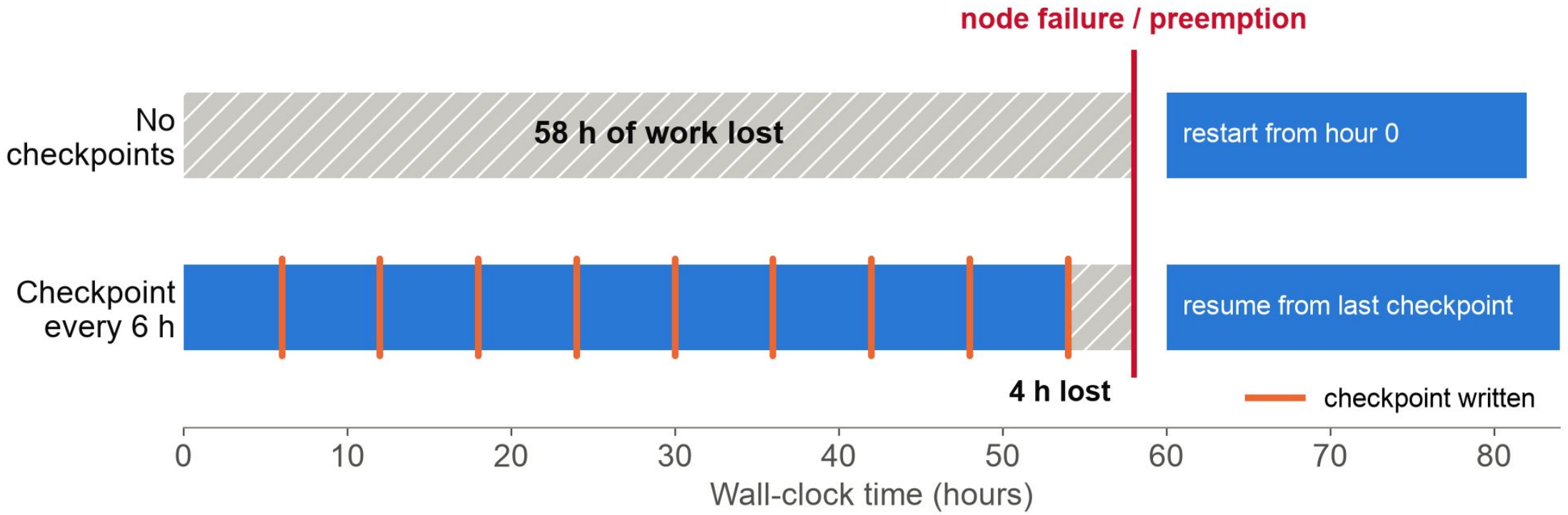
4

Maintenance & outages

Scheduled maintenance windows, power or cooling events, filesystem problems.

The question is not **if** a long job will be interrupted, but **when** — and how much work you lose.

The Cost of Losing Work



Without checkpoints an interruption costs **everything computed so far**. With checkpoints it costs only the work since the **last checkpoint** — plus the time to write them.

What Is Checkpoint/Restart?

Checkpoint: save the state of a computation to disk. **Restart:** a later run loads that state and continues instead of starting over.



Why checkpoint?

- Survive node failures and preemption
- Run longer than the wall-time limit
- Use preemptible (requeue) partitions safely
- Debug or inspect a long run mid-way
- Monitor progress; branch new runs from a saved state
- Reproduce results exactly (with RNG state)

NOTE: This assumes both checkpoint and restart are done by the same version of the application.

https://hpc.nmsu.edu/discovery/slurm/backfill-and-checkpoints/

FASRC Partitions & Time Limits

Partition	Max time	Notes
sapphire, shared	3 days	General-purpose CPU
gpu, gpu_h200	3 days	GPU nodes
bigmem	3 days	Large memory (bigmem_intermediate: 14 days)
intermediate	14 days	Long jobs, fewer nodes
unrestricted	365 days	Few nodes, long waits
test, gpu_test	12 hours	Testing & interactive work; per-user limits
serial_requeue, gpu_requeue	3 days	Preemptible: may be killed and requeued

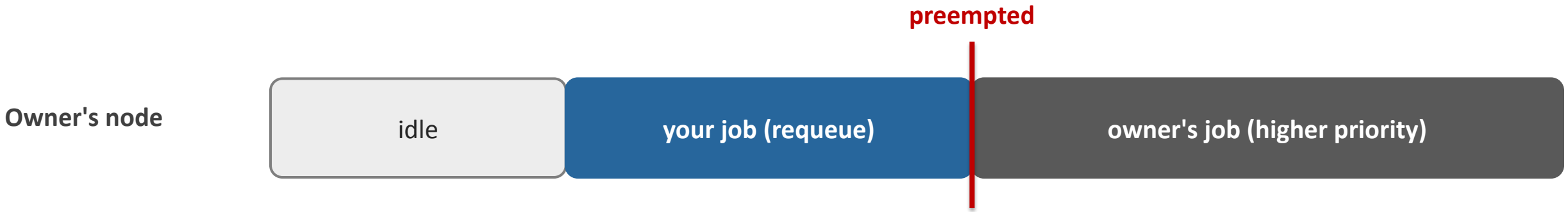
Key points

- A run longer than the limit **must** be split into checkpointed pieces
- Shorter `--time` requests usually start sooner
- `spart` shows the partitions you can use
- Requeue partitions: more resources, but your job can be killed at any time

https://docs.rc.fas.harvard.edu/kb/running-jobs/

Requeue Partitions: Free Cycles, With a Catch

`serial_requeue` and `gpu_requeue` run your job on **idle resources owned by other groups**. When a higher priority job comes in, your job is killed and put back in the queue.

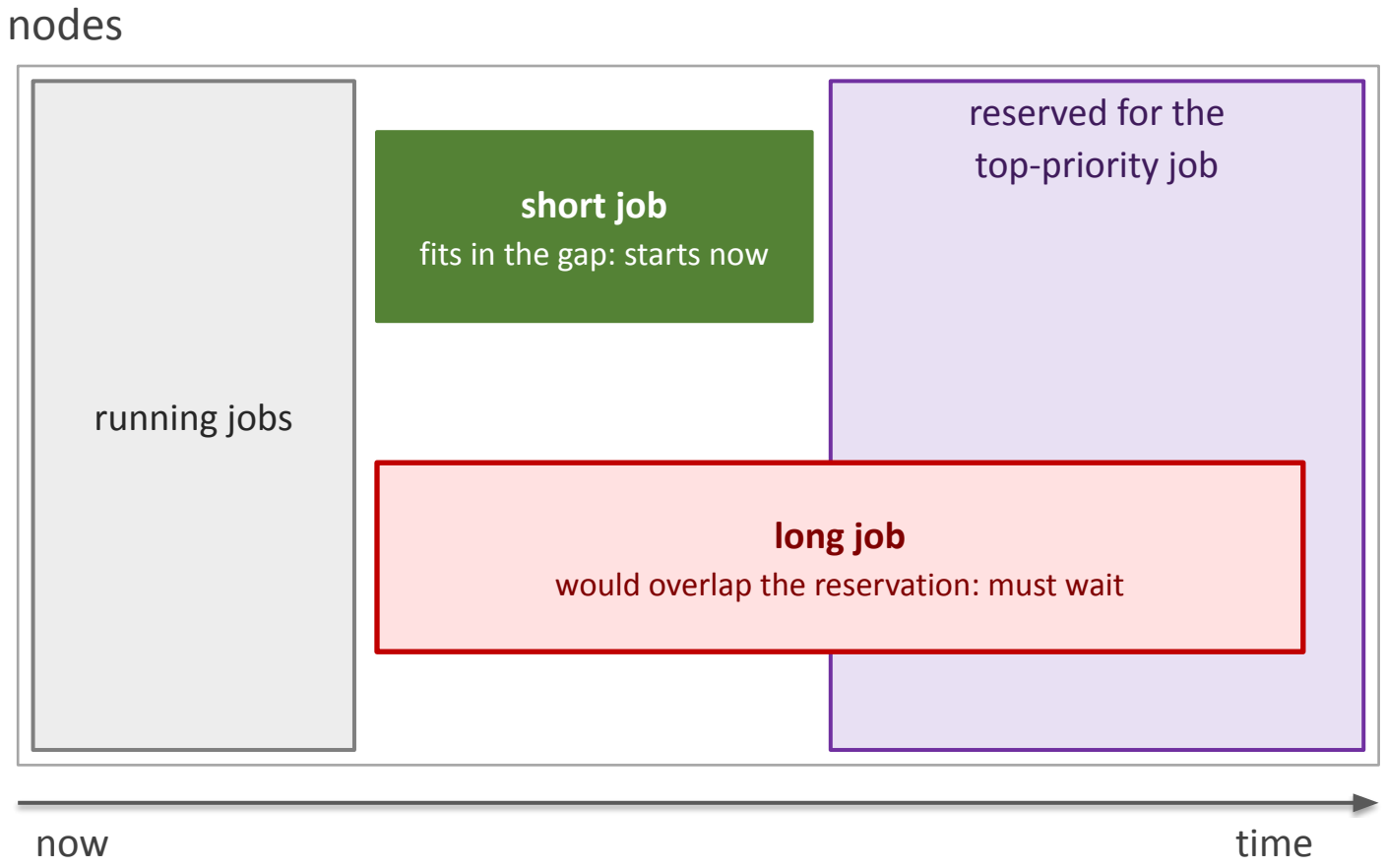


Without a checkpoint
Job restarts from the beginning — possibly again and again

With a checkpoint
Job picks up where it left off on whatever node it lands on next

There may be little or no warning: write checkpoints **periodically**, and make the program **find its checkpoint automatically** when it starts.

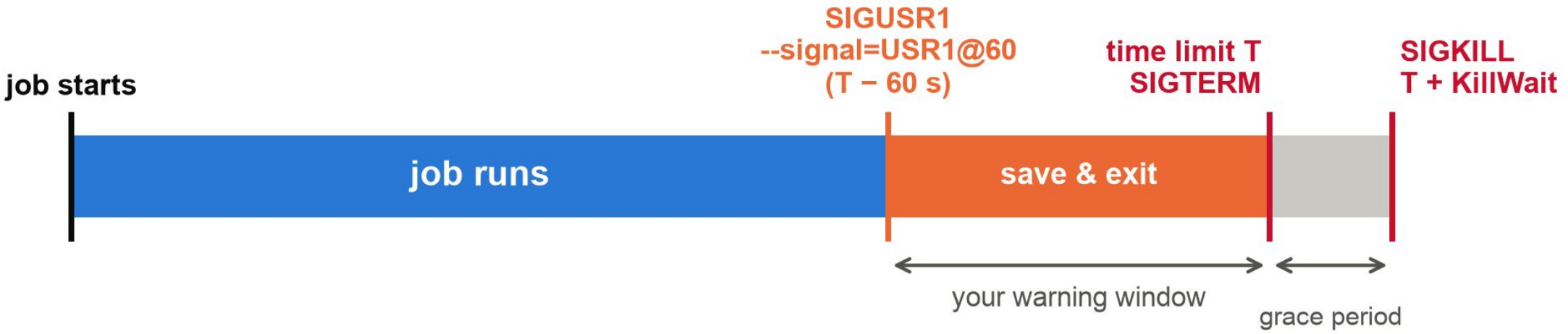
Shorter Jobs Start Sooner: Backfill



- Slurm reserves nodes for the highest-priority pending jobs
- **Backfill** starts lower-priority jobs in the gaps — only if they finish before the reservation
- Accurate, shorter `--time` requests fit more gaps
- Checkpointing lets you split one 7-day run into many shorter, backfill-friendly jobs

https://hpc.nmsu.edu/discovery/slurm/backfill-and-checkpoints/

How Slurm Ends a Job



- At the time limit Slurm sends `SIGTERM`, then `SIGKILL` after a short grace period (`KillWait`) — preemption works the same way
- The default action of `SIGTERM` is to terminate at once: **no checkpoint unless your program handles it**
- `--signal` asks for an earlier warning, so there is time to save and exit cleanly

Asking Slurm for a Warning: --signal

Option	Signal goes to	Use when
`--signal=USR1@60`	every job step (processes started with `srun`)	the application handles the signal itself
`--signal=B:USR1@60`	the batch shell only	the batch script traps the signal (requeue, DMTCP)

```
#SBATCH --signal=B:USR1@60

handler() {
    echo "USR1: time is almost up"
    scontrol requeue "$SLURM_JOB_ID"
}
trap handler USR1

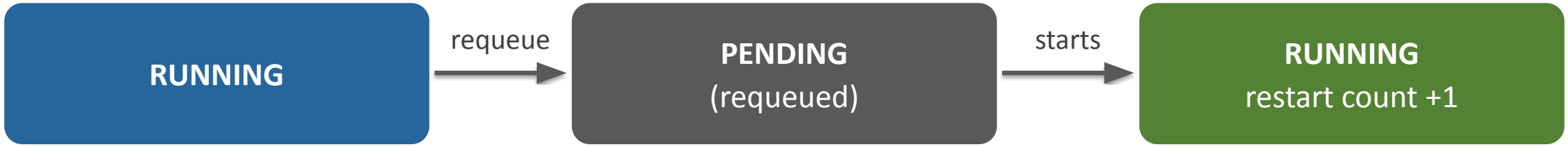
srun ./my_app &    # run in the background ...
wait $!            # ... so the trap can run
```

Key points

- `@60` = seconds before the time limit; Slurm may send it up to ~60 s early
- bash runs a trap only after the foreground command ends → start the program with `&` and `wait`

https://slurm.schedmd.com/sbatch.html#OPT_signal

Re-queueing a Job



Piece	What it does
`#SBATCH --requeue`	allow the job to be requeued
`scontrol requeue $SLURM_JOB_ID`	put the running job back in the queue (same job ID)
`$SLURM_RESTART_COUNT`	how many times this job has been requeued (unset or 0 on the first run)
`#SBATCH --open-mode=append`	append the output of every run to the same `.out` file

Key points

- The requeued job may run on a **different node**
- The program must **find its checkpoint by itself**
- Make sure every run makes progress — avoid endless requeue loops

https://slurm.schedmd.com/scontrol.html#OPT_requeue

Three Approaches to Checkpointing

1

Application-level

Your program saves and reloads exactly the state it needs.

You get: Small, portable checkpoints; full control

You need: Changes to the code

Exercises 1, 2, 6

2

Built into the application

The application already has checkpoint/restart options — use them.

You get: Works and is maintained for you

You need: An application that provides it

Exercise 5

3

Transparent (system-level)

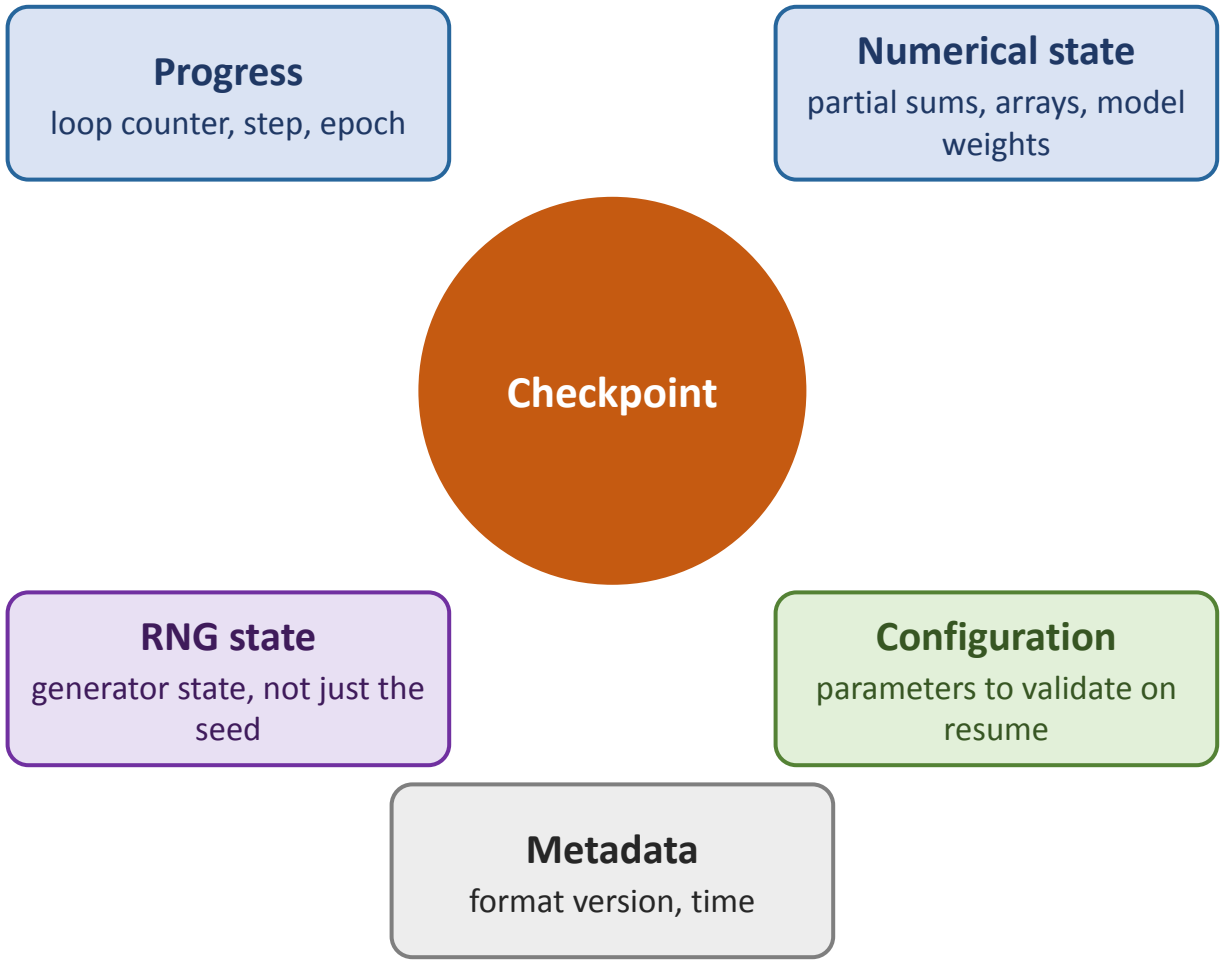
An external tool snapshots the whole process — the program does not know.

You get: No code changes at all

You need: A supported environment (DMTCP, MANA)

Exercises 3, 4

What Goes Into a Checkpoint?



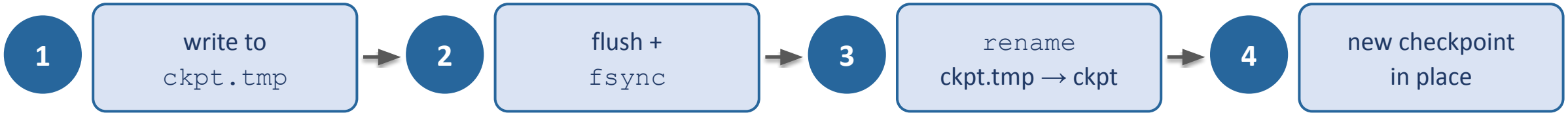
Exercise 1 — pi_checkpoint.py

```
checkpoint = {
    "version":          CHECKPOINT_VERSION,
    "completed_darts":  completed_darts,
    "inside_circle":    inside_circle,
    "rng_state":        rng.getstate(),
    "n_darts":          n_darts,
    "seed":             seed,
    "elapsed":          elapsed,
}
```

Key points

- Save the **minimum** needed to continue exactly
- Anything you can recompute cheaply does not belong in it

Writing Checkpoints Safely: Atomic Writes

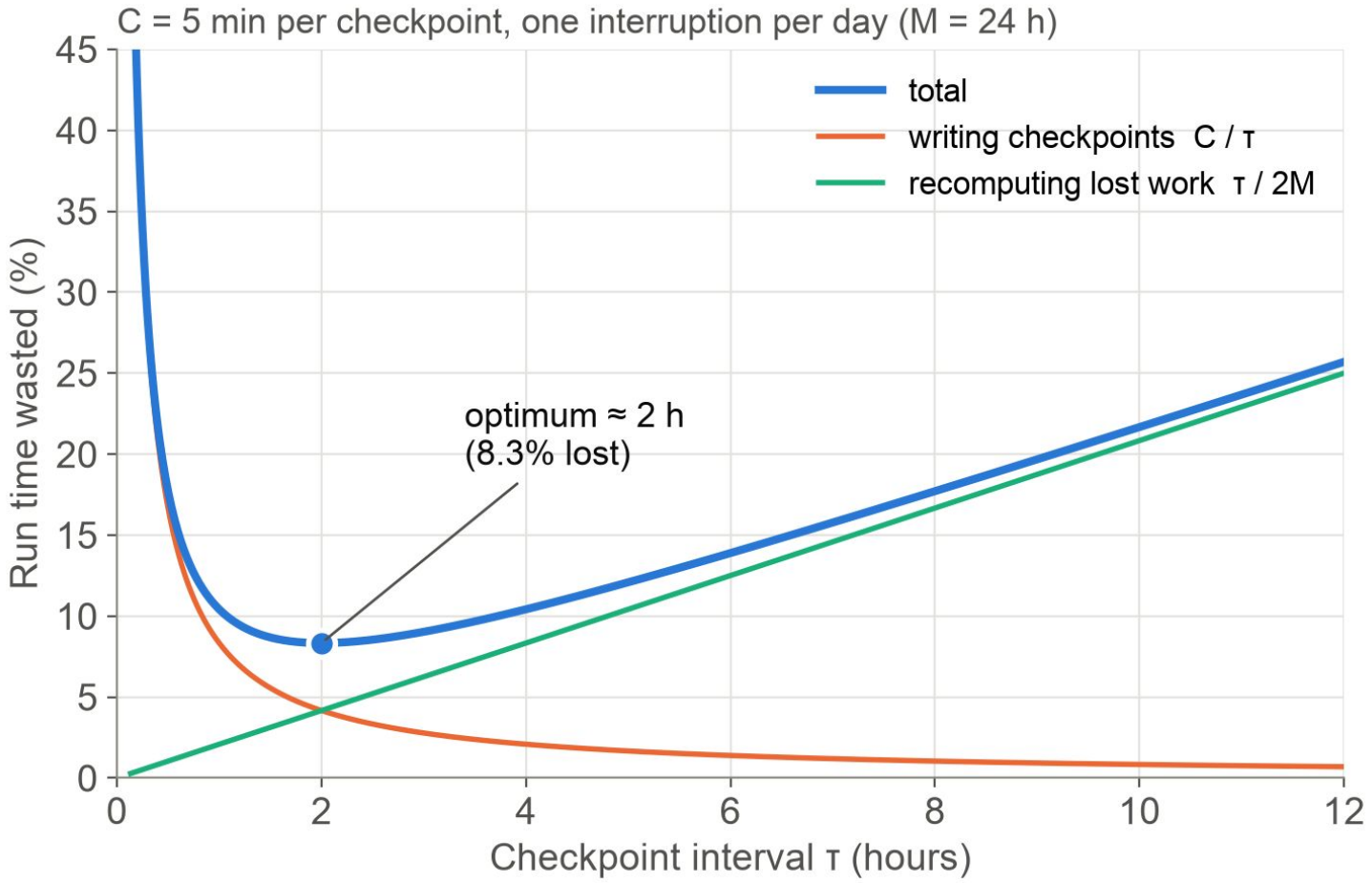


If the job dies …	… you are left with
while writing `ckpt.tmp`	a partial `ckpt.tmp` — the old `ckpt` is still intact
after `fsync`, before `rename`	a complete `ckpt.tmp` — the old `ckpt` is still intact
after `rename`	the new, complete `ckpt`

Key points

- Never overwrite your only good checkpoint in place
- `rename` / `os.replace` is atomic only **on the same filesystem**
- `fsync` makes sure the data is on storage, not just in memory
- Optionally keep the previous checkpoint too (GROMACS keeps `md_prev.cpt`)

How Often Should You Checkpoint?



Key points

- Two costs: **writing** checkpoints (often = expensive) vs **recomputing** lost work (rarely = expensive)
- Young's formula: $T \approx \sqrt{(2 \cdot C \cdot M)}$
  - C = time to write one checkpoint
  - M = mean time between interruptions
- Example: C = 5 min, M = 1 day → τ ≈ 2 h
- In practice: every **30 min – a few hours**, overhead ≲ 5 %
- For time limits, the warning signal gives you the last checkpoint

https://docs.nersc.gov/development/checkpoint-restart/

Where to Store Checkpoints

Location	Size	Lifetime	For checkpoints?
Home ~	100 GB	Permanent, backed up	Small checkpoints only; avoid heavy I/O
Lab storage (`/n/holylabs/…`)	Per lab	Permanent	Checkpoints you must keep
Netscratch (`/n/netscratch/<lab>`)	Large, shared	Files removed after 90 days	Recommended for job checkpoints
Node-local `/scratch`	Per node	Deleted when the job ends	Not for restart files — the next job may run elsewhere

Key points

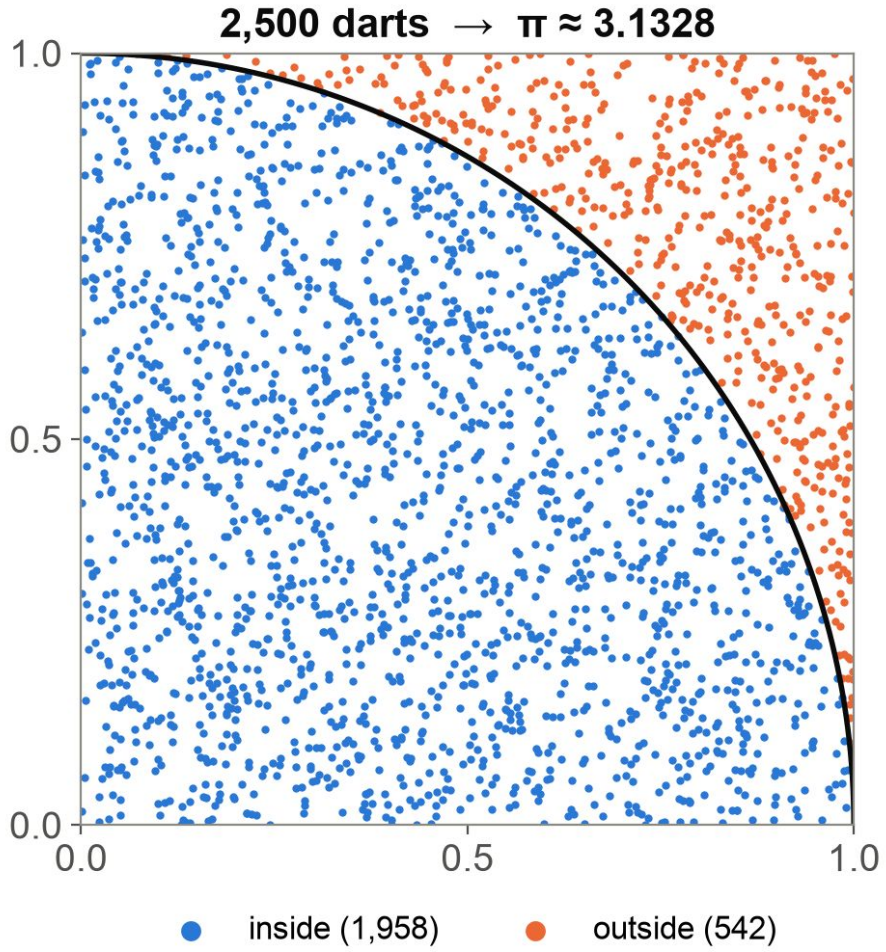
- Estimate **size x number** of checkpoints you keep; delete old ones
- Write to the directory the restart will read from — on shared storage

https://docs.rc.fas.harvard.edu/kb/cluster-storage/

Outline

- Introduction & Background
- Application-Level Checkpointing: Python
- Transparent Checkpointing: DMTCP and MANA
- Built-in Checkpointing: GROMACS
- Checkpointing Deep Learning: PyTorch
- Best Practices, Wrap-up & Q&A

Exercise 1: Monte Carlo π



Throw random darts (x, y) into the unit square; count those inside the circle $x^2 + y^2 \leq 1$:

$\pi \approx 4 \times$ (darts inside) / (total darts)

- A long loop with a little state: two counters + the RNG
- Easy to check: the error shrinks like $1/\sqrt{N}$
- `--sleep` slows the loop so there is time to interrupt it
- Same vehicle in Exercises 1, 2 and 4 (MPI)

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing/Exercise1

Hardening a Program in Four Steps

Each step adds one idea to the same program. Exercise 2 repeats the four steps in C.

1 No checkpointing

`pi_naive.py`

interactive

an interruption loses everything

2 Periodic checkpoints

`pi_checkpoint.py`

interactive

atomic writes; manual --resume

3 Signal-aware

`pi_signal.py`

batch

SIGUSR1 from Slurm → save & exit

4 Automatic requeue

`pi_requeue.py`

batch

finds its checkpoint; job requeues itself

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing/Exercise1

Hands-On Practice: Python Checkpointing

Exercise 1 — Step 1: No checkpointing (pi_naive.py)

```
salloc --partition=test --time=00:20:00 \
       --cpus-per-task=1 --mem=1G
module load python
mkdir -p checkpoints

python pi_naive.py --darts 50000000 --seed 42 \
       --report-every 1000000 --sleep 1
```

- Press **Ctrl-C** after a few reports
- Rerun the same command
- It starts again from dart 1 — **nothing was saved**

```
...
darts =     1,000,000 / 50,000,000   pi = 3.14024400   error = 1.349e-03   elapsed = 0.22 s
darts =     2,000,000 / 50,000,000   pi = 3.14123400   error = 3.587e-04   elapsed = 1.44 s
darts =     3,000,000 / 50,000,000   pi = 3.14173733   error = 1.447e-04   elapsed = 2.67 s
...
```

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing/Exercise1

Saving the State in Python

pi_checkpoint.py

```
def save_checkpoint(checkpoint_file, completed_darts,
                    inside_circle, rng, n_darts, seed, elapsed):
    checkpoint = {
        "version": CHECKPOINT_VERSION,
        "completed_darts": completed_darts,
        "inside_circle": inside_circle,
        "rng_state": rng.getstate(),
        "n_darts": n_darts, "seed": seed, "elapsed": elapsed,
    }
    tmp_file = checkpoint_file.with_name(
        checkpoint_file.name + ".tmp")

    with open(tmp_file, "wb") as f:
        pickle.dump(checkpoint, f,
                    protocol=pickle.HIGHEST_PROTOCOL)
        f.flush()                # Python buffers -> OS
        os.fsync(f.fileno())     # OS -> storage

    # Atomic replacement on the same filesystem
    os.replace(tmp_file, checkpoint_file)
```

Key points

- Everything needed to continue goes into one dictionary
- `rng.getstate()` captures the **full** generator state
- `n_darts` and `seed` are saved so a resume can be validated
- Write `.tmp` → `flush` → `fsync` → `os.replace` = atomic
- Called every `--checkpoint-every` darts

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing/Exercise1

Loading and Validating a Checkpoint

```
def load_checkpoint(checkpoint_file, expected_darts,
                    expected_seed):
    with open(checkpoint_file, "rb") as f:
        checkpoint = pickle.load(f)
    if checkpoint.get("version") != CHECKPOINT_VERSION:
        raise RuntimeError("... version is incompatible")
    if checkpoint["n_darts"] != expected_darts:
        raise RuntimeError("... different number of darts")
    if checkpoint["seed"] != expected_seed:
        raise RuntimeError("... different random seed")
    return checkpoint
```

```
rng = random.Random(args.seed)
rng.setstate(checkpoint["rng_state"])      # on --resume
for i in range(completed_darts + 1, args.darts + 1):
```

Key points

- **Refuse** to resume from a checkpoint that does not match the run
- Restoring the RNG state → the resumed run draws **exactly** the darts an uninterrupted run would
- Restoring only the seed would replay darts from the beginning
- Only unpickle files you trust

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing/Exercise1

Hands-On Practice: Python Checkpointing

Exercise 1 — Step 2: Periodic checkpoints (pi_checkpoint.py)

```
rm -f checkpoints/pi_checkpoint.pkl*

python pi_checkpoint.py \
    --darts 50000000 --seed 42 \
    --report-every 1000000 \
    --checkpoint-every 5000000 \
    --checkpoint-file checkpoints/pi_checkpoint.pkl \
    --sleep 1

# Ctrl-C after 2 checkpoints, then rerun with --resume
```

first run

```
darts =  5,000,000 / 50,000,000   pi = 3.14223440
  --> checkpoint saved at dart 5,000,000
darts =  6,000,000 / 50,000,000   pi = 3.14191867
```

with --resume

```
Checkpoint found.
Resuming from dart : 10,000,000
Inside circle      : 7,855,434
Previous runtime   : 12.81 s
```

- Work after the last checkpoint is lost and redone (e.g., darts 10,000,001 – 13,000,000)
- Try a different `--seed` with `--resume`: the program refuses to continue

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing/Exercise1

Reacting to Slurm's Warning

pi_signal.py

```
stop_requested = False

def signal_handler(signum, frame):
    # Do NOT perform checkpoint I/O here -
    # just record that a signal was received.
    global stop_requested, received_signal
    stop_requested = True
    received_signal = signum

signal.signal(signal.SIGUSR1, signal_handler)
signal.signal(signal.SIGTERM, signal_handler)

for i in range(completed_darts + 1, args.darts + 1):
    ...                          # throw one dart
    if stop_requested:           # safe point
        save_checkpoint(...)
        print("Exiting cleanly.")
        return
```



https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing/Exercise1

Hands-On Practice: Python Checkpointing

Exercise 1 — Step 3: Signal-aware checkpointing (pi_signal.sbatch)

pi_signal.sbatch (excerpt)

```
#SBATCH --time=00:03:00
#SBATCH --signal=USR1@60

srun python pi_signal.py \
    --darts 10000000000 --seed 42 \
    --report-every 1000000 \
    --checkpoint-every 0 \
    --checkpoint-file
checkpoints/pi_signal.pkl
```

```
sbatch pi_signal.sbatch

tail -f pi_signal_JOBID.out

sbatch pi_signal_restart.sbatch     # same +
--resume
```

pi_signal_15362.out

```
darts = 342,000,000 / 10,000,000,000
======================================
Termination signal received
======================================
Signal              : SIGUSR1
Completed darts     : 342,615,676
Saving checkpoint ...
Checkpoint saved    : checkpoints/pi_signal.pkl
Exiting cleanly.
```

pi_signal_restart_15363.out

```
Restarting from checkpoint
Completed darts     : 342,615,676
Previous runtime    : 91.57 s
```

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing/Exercise1

Automatic Requeue: the Batch Script

pi_requeue.sbatch (excerpt)

```
#SBATCH --time=00:03:00
#SBATCH --open-mode=append
#SBATCH --requeue
#SBATCH --signal=B:USR1@60

requeue_job() {
    trap - USR1                  # run only once
    echo "SIGUSR1 received: $(date)"
    ls -lh "${CHECKPOINT_FILE}"
    scontrol requeue "${SLURM_JOB_ID}"
    exit 0
}
trap requeue_job USR1

# no --resume: pi_requeue.py finds the checkpoint itself
srun python pi_requeue.py --darts 1000000000 --seed 42 \
    --checkpoint-every 5000000 \
    --checkpoint-file "${CHECKPOINT_FILE}" &
wait $!
```

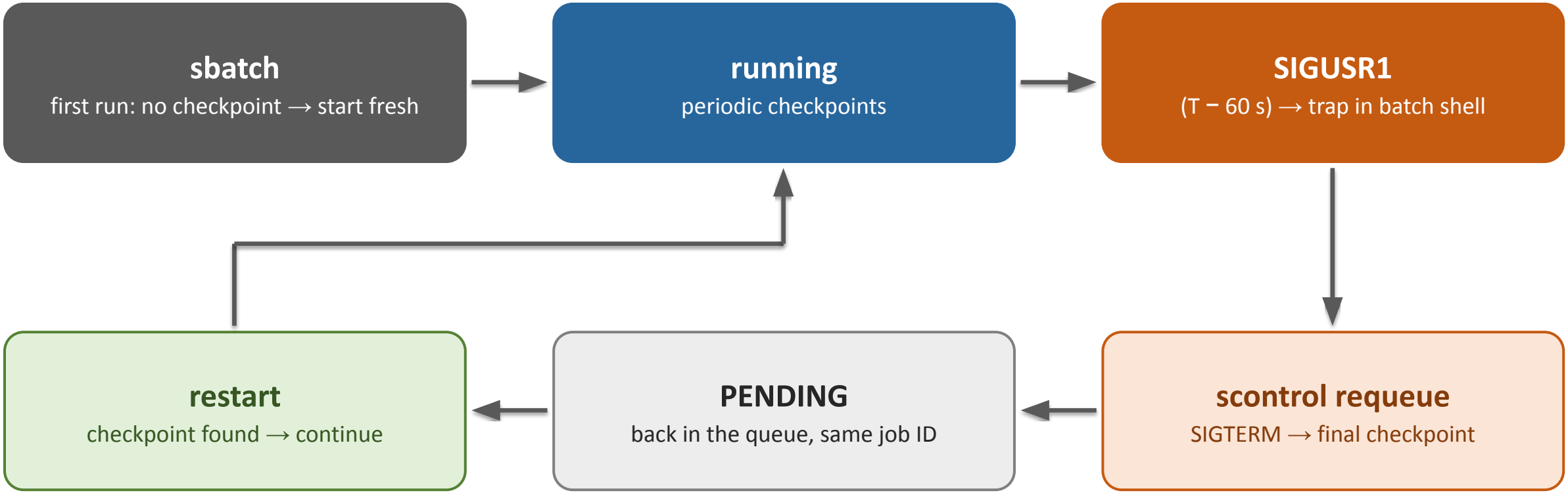
Key points

- `B:` — the **batch shell** gets the warning and runs the trap
- `scontrol requeue` sends `SIGTERM` to the program, which saves a final checkpoint

  NOTE: After SLURM sends SIGTERM, it waits for KillWait seconds (30 sec by default) before sending SIGKILL, so the checkpoint must have completed by then
- App runs in the background (`&` + `wait`) so the trap can fire
- Real runs: `--partition=serial_requeue`
- **Never** hard-code `--fresh` here — it would discard the checkpoint on every restart

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing/Exercise1

The Requeue Loop



`$SLURM_RESTART_COUNT` = 0, 1, 2 …

Hands-On Practice: Python Checkpointing

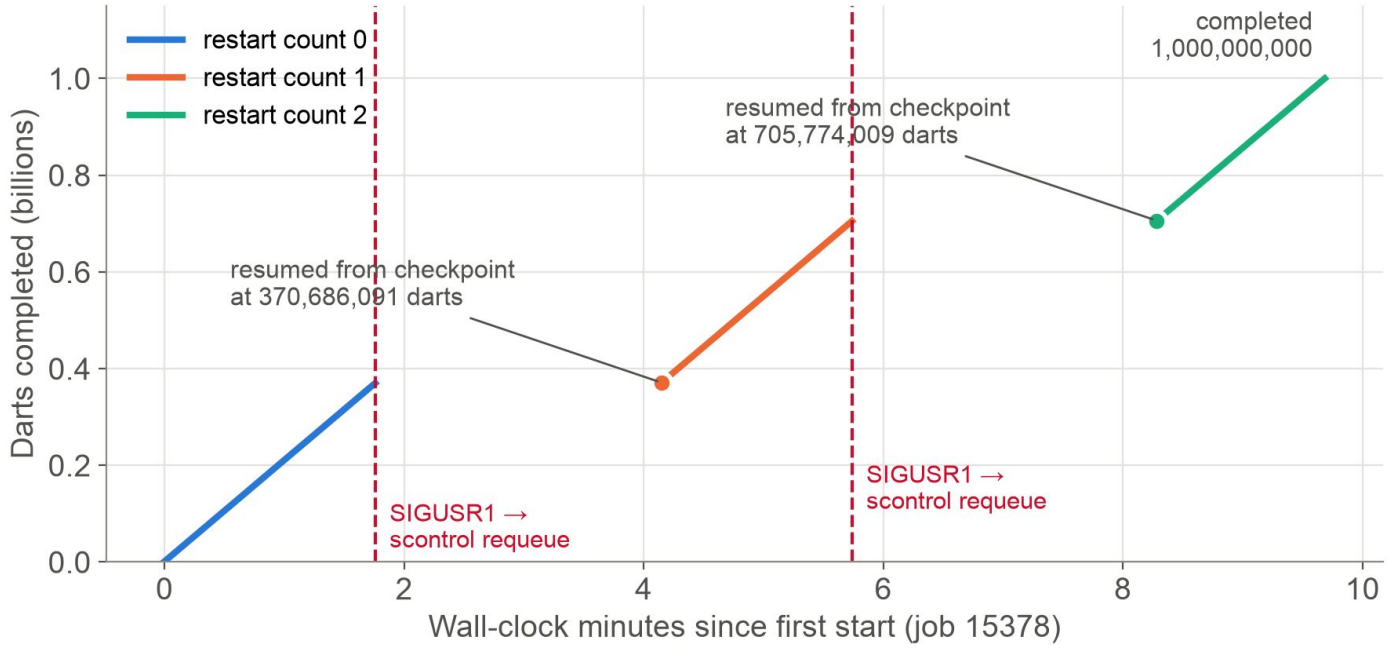
Exercise 1 — Step 4: Automatic requeue (pi_requeue.sbatch)

```
rm -f checkpoints/pi_requeue.pkl*
JOBID=$(sbatch --parsable \
        pi_requeue.sbatch)
tail -f pi_requeue_${JOBID}.out

sacct -j $JOBID -X \
   -o JobID,State,Elapsed,Restarts
```

pi_requeue_15378.out

```
Slurm restart count : 1
============================
CHECKPOINT FOUND -
     AUTOMATIC RESTART
============================
Completed darts   : 370,686,091
Inside circle     : 291,139,215
```



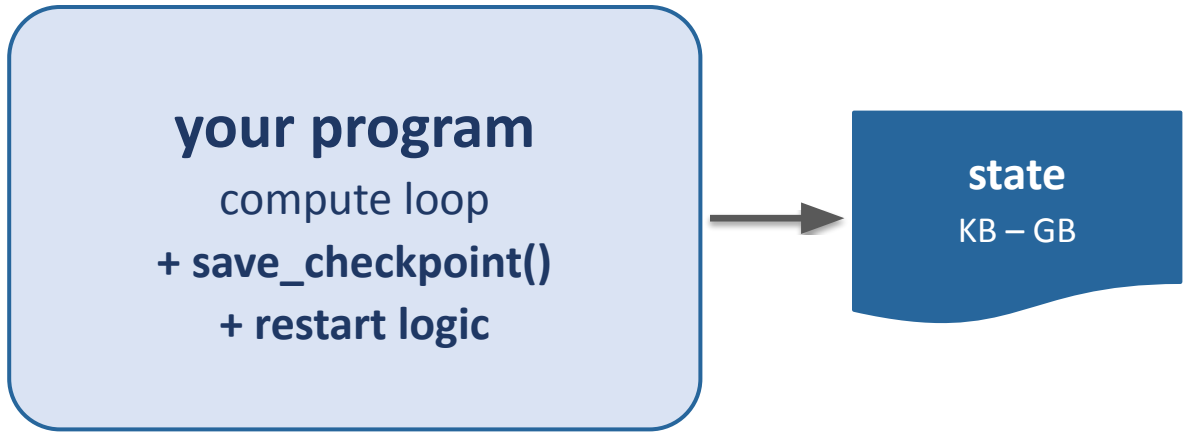
https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing/Exercise1

Outline

- Introduction & Background
- Application-Level Checkpointing: Python
- Transparent Checkpointing: DMTCP and MANA
- Built-in Checkpointing: GROMACS
- Checkpointing Deep Learning: PyTorch
- Best Practices, Wrap-up & Q&A

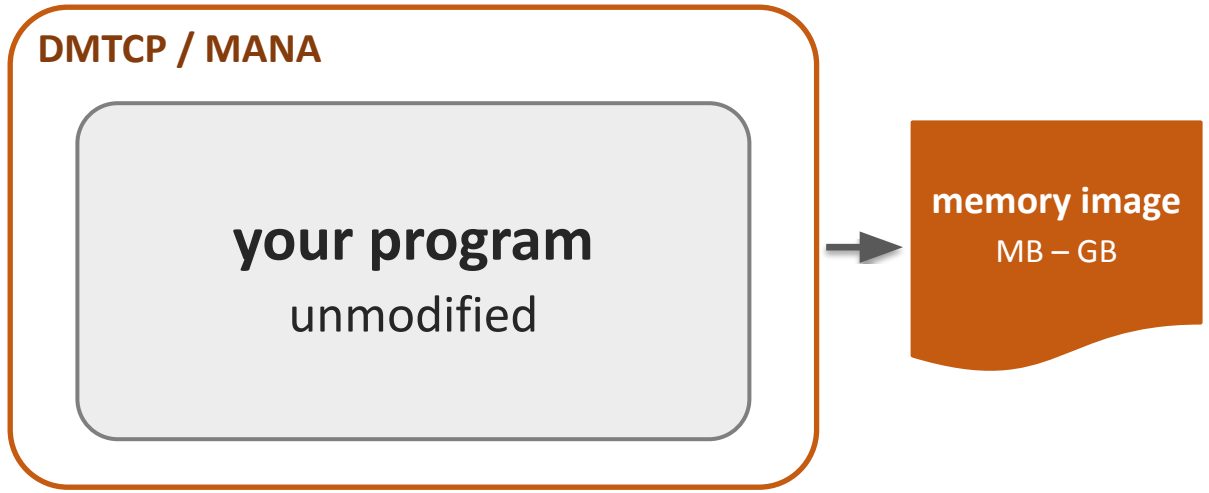
Transparent Checkpointing

Application-level



- Program chooses what to save
- Program knows how to restart

Transparent



- Tool saves the **whole process**: memory, threads, open files
- Restart recreates the process — the program never notices

What Is DMTCP?

Distributed **M**ulti**T**hreaded **C**heck**P**ointing — transparent checkpoint/restart of threaded and multi-process applications.

No code changes
works on the unmodified executable

User space
no kernel modules, no root privileges

Many languages
C/C++, Fortran, Python, R, MATLAB, OpenMP, shell scripts

Processes & threads
multi-threaded and multi-process programs, sockets

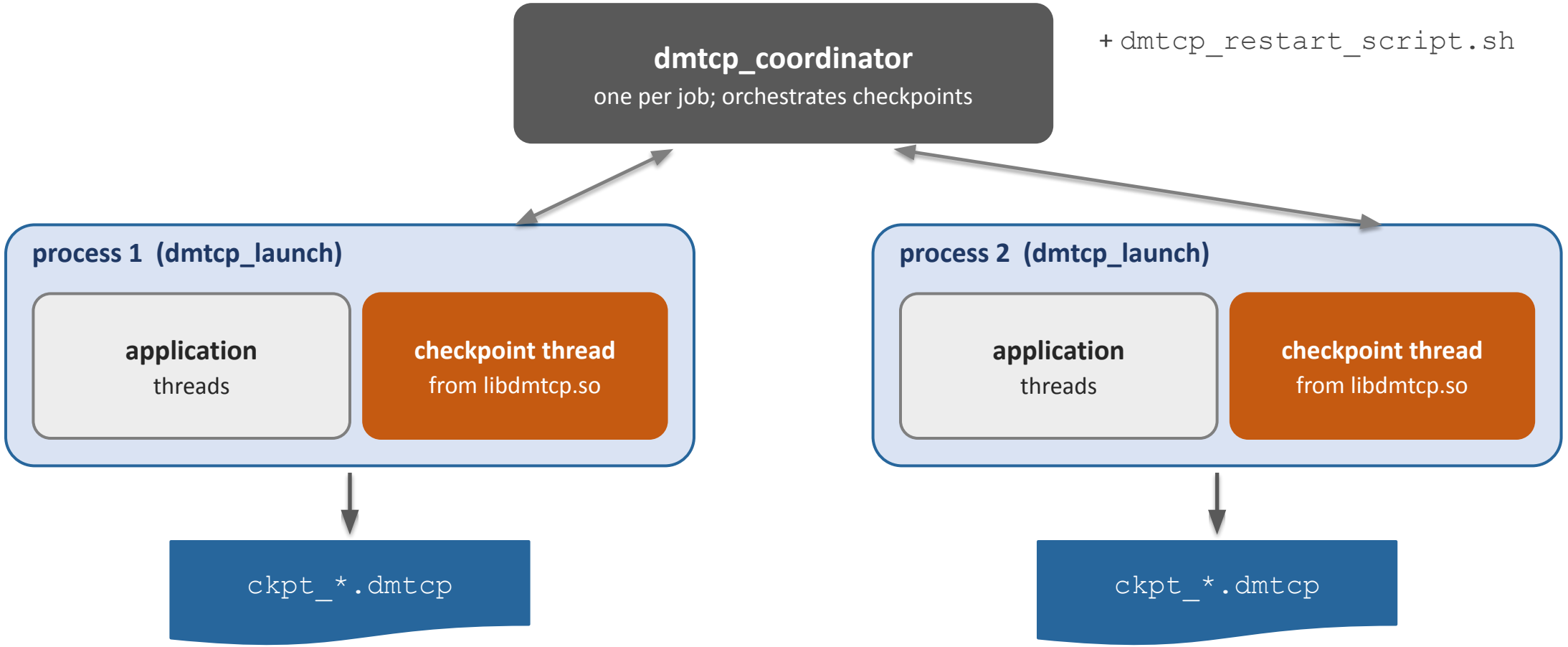
Periodic or on demand
checkpoint every N seconds or with a command

Restart anywhere
on another node of the same cluster

Developed at Northeastern University; on the cluster: `module load dmtcp`

https://dmtcp.sourceforge.io/index.html

How DMTCP Works



`dmtcp_launch` preloads `libdmtcp.so` (LD_PRELOAD) → programs must be **dynamically linked**.

https://dmtcp.sourceforge.io/index.html

DMTCP Commands

Command	What it does
`dmtcp_coordinator --daemon --interval 60`	start a coordinator; checkpoint every 60 s (`--port 0` picks a free port)
`dmtcp_launch ./app args`	run the program under DMTCP (joins the coordinator)
`dmtcp_command --checkpoint`	request a checkpoint now (non-blocking; `--bcheckpoint` is blocking)
`dmtcp_command --status`	show coordinator and processes
`dmtcp_command --kill`	stop the computation
`./dmtcp_restart_script.sh`	restart from the most recent checkpoint
`dmtcp_restart ckpt_*.dmtcp`	restart from specific image files

`DMTCP_COORD_HOST` / `DMTCP_COORD_PORT` tell every command which coordinator to talk to.

https://dmtcp.sourceforge.io/index.html

DMTCP: Requirements & Limitations

Requirements

- Dynamically linked executable (shared `.so` libraries)
- Same modules, libraries and paths at restart
- Files the program had open must still exist
- Enough disk for full memory images
- Pin a module version (e.g., `dmtcp/4.1.0-fasrc01`)

Limitations

- Image size ≈ process memory (compressed by default)
- GPU / CUDA state is **not** captured
- MPI over InfiniBand → use **MANA** (next)
- External connections (databases, licence servers) may not survive
- Restart on the same OS / CPU architecture

NOTE: To reassure you restart on nodes with the same CPU architecture, on the `serial_requeue` and `gpu_requeue` partitions one needs to also to use the `--constraint` option with suitable constraint

https://dmtcp.sourceforge.io/index.html

Hands-On Practice: DMTCP

Exercise 3 — Setup and a run without DMTCP

```
# Request an interactive session
salloc --partition=test --time=00:30:00 \
       --cpus-per-task=1 --mem=1G

# Load required modules
module load gcc/15.2.0-fasrc01
module load dmtcp/4.1.0-fasrc01
make                    # builds long_run.x
mkdir -p checkpoints/

# Run the code
./long_run.x --iterations 100 --sleep-ms 1000
```

```
================================================
DMTCP long-running demo application
================================================
Iterations      : 100
Sleep/iter      : 1000 ms
Checkpoint code: NONE
================================================
iteration =    1   accumulator = 1.00000000
iteration =    2   accumulator = 2.41421356
iteration =    3   accumulator = 4.14626437
...
```

- `long_run.c` contains **no** checkpoint code
- Ctrl-C and rerun → starts again at iteration 1

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing/Exercise3

Hands-On Practice: DMTCP

Exercise 3 — Interactive checkpoint and restart

1. Launch under DMTCP

```
# Shell 1
dmtcp_launch --new-coordinator --coord-port 0 \
             --port-file dmtcp_coord_port   \
             ./long_run.x --iterations 100 \
             --sleep-ms 1000
```

2. Checkpoint from outside

```
# Shell 2 (same node)
module load dmtcp/4.1.0-fasrc01
export DMTCP_COORD_PORT="$(cat dmtcp_coord_port)"
dmtcp_command --bcheckpoint
```

3. Inspect, then stop

```
# Shell 2 (same node)
ls -lh ckpt_long_run_*.dmtcp
ls -l dmtcp_restart_script.sh
dmtcp_command --kill
```

4. Restart

```
# Shell 1
./dmtcp_restart_script.sh
```

Key points

- The program resumes at about the checkpointed iteration — and `long_run.c` was never modified
- Find your node for shell 2: `squeue -u $USER` or `scontrol show job <JOBID>`, then `ssh` to it

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing/Exercise3

Batch: Periodic Checkpoints with DMTCP

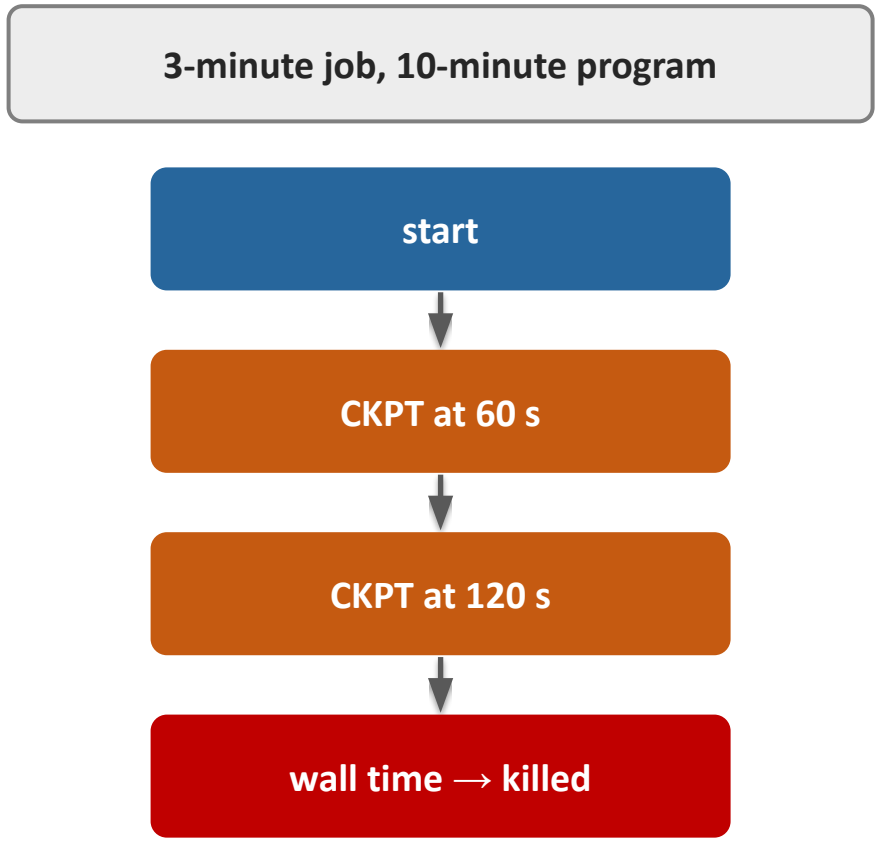
dmtcp_checkpoint.sbatch (excerpt)

```
#SBATCH --time=00:03:00
module load dmtcp/4.1.0-fasrc01
mkdir -p checkpoints && cd checkpoints

# Random coordinator port: no collisions with other users
PORT_FILE="${SLURM_TMPDIR:-/tmp}/dmtcp_port_${SLURM_JOB_ID}"
dmtcp_coordinator --daemon --port 0 \
    --port-file "$PORT_FILE" --interval 60
while [[ ! -s "$PORT_FILE" ]]; do sleep 0.1; done

export DMTCP_COORD_HOST="$(hostname)"
export DMTCP_COORD_PORT="$(cat "$PORT_FILE")"

# The application itself contains NO checkpointing code
dmtcp_launch --join-coordinator \
    ../long_run.x --iterations 600 --sleep-ms 1000
```




```
sbatch dmtcp_checkpoint.sbatch
```

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing/Exercise3

Batch: Restart in a New Job

dmtcp_restart.sbatch (excerpt)

```
cd checkpoints
if [[ ! -f dmtcp_restart_script.sh ]]; then
    echo "ERROR: dmtcp_restart_script.sh not found."
    exit 1
fi

# Fresh coordinator for the new allocation,
# still checkpointing every 60 s
dmtcp_coordinator --daemon --port 0 \
    --port-file "$PORT_FILE" --interval 60
...
./dmtcp_restart_script.sh
```

```
Checkpoint job:
  iteration = 1
  ...
  iteration = 60     <- CKPT
  ...
  iteration = 120    <- CKPT
  ...
  job ends

Restart job:
  iteration = 121
  iteration = 122
  ...
```

- A **new** job, possibly on a **different node** — it resumes from the most recent successful checkpoint

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing/Exercise3

Automatic Checkpoint & Requeue with DMTCP

dmtcp_auto_requeue.sbatch (excerpt)

```
#SBATCH --signal=B:USR1@60
#SBATCH --requeue
#SBATCH --open-mode=append

checkpoint_and_requeue() {
    dmtcp_command --bcheckpoint       # final checkpoint
    scontrol requeue "${SLURM_JOB_ID}"
    exit 0
}
trap checkpoint_and_requeue USR1

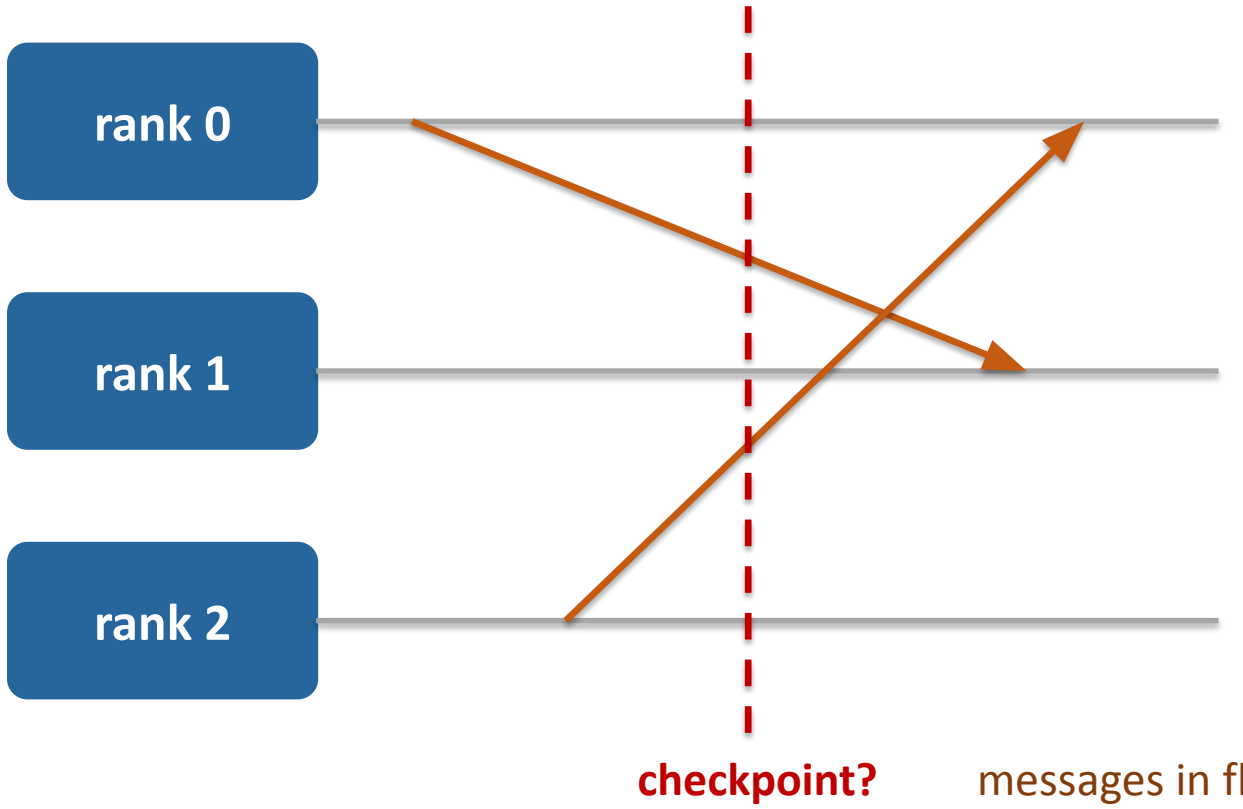
if [[ ${SLURM_RESTART_COUNT:-0} -eq 0 ]]; then
    dmtcp_launch --join-coordinator ../long_run.x \
        --iterations 600 --sleep-ms 1000 &
else
    ./dmtcp_restart_script.sh &
fi
wait $!
```

Key points

- Same pattern as Exercise 1, step 4
- `SLURM_RESTART_COUNT` chooses **launch** or **restart**
- The trap asks DMTCP for a final checkpoint, then requeues
- Periodic checkpoints still protect against unannounced failures

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing/Exercise3

Why Is MPI Hard to Checkpoint?



Key points

- **Messages in flight** belong to no single process
- All ranks must be checkpointed **consistently** — at the same logical moment
- The MPI library and network hardware (InfiniBand, drivers) hold state that cannot simply be copied
- Every MPI implementation and network is different

https://docs.nersc.gov/development/checkpoint-restart/mana/

MANA: MPI-Agnostic, Network-Agnostic

Transparent checkpoint/restart for MPI programs, built on DMTCP — independent of the MPI implementation and the network.

Coordinated
all ranks drain in-flight messages and checkpoint together

MPI-agnostic
works with standard MPI libraries (here: MPICH)

Network-agnostic
no network state in the image

One directory per rank
`ckpt_rank_0/, ckpt_rank_1/, ...`

Developed by Northeastern University (DMTCP team) with NERSC; used in production on NERSC systems.

https://github.com/mpickpt/mana

Outline

- Introduction & Background
- Application-Level Checkpointing: Python
- Transparent Checkpointing: DMTCP and MANA
- Built-in Checkpointing: GROMACS
- Checkpointing Deep Learning: PyTorch
- Best Practices, Wrap-up & Q&A

Many Applications Already Checkpoint

Application	Write checkpoints	Restart
GROMACS	`mdrun -cpt <min>` (default 15 min)	`mdrun -cpi state.cpt`
LAMMPS	`restart N file.*` / `write_restart`	`read_restart file`
NAMD	`restartfreq, restartname`	`bincoordinates, binvelocities, extendedSystem`
OpenFOAM	`writeInterval` in `controlDict`	`startFrom latestTime;`
Quantum ESPRESSO	`max_seconds` (clean stop)	`restart_mode = 'restart'`
CP2K	`*.restart` files	`&EXT_RESTART`
Gaussian	`%Chk=file.chk`	`Opt=Restart, Geom=AllCheck`
PyTorch Lightning / HF	`ModelCheckpoint` / `save_steps`	`ckpt_path=` / `resume_from_checkpoint`

Check your application's manual for “restart” or “checkpoint” before writing your own.

Exercise 5: GROMACS Water Box

Pure water

900 H_2O molecules

2,700 atoms

5,000,000 steps = 5 ns

4 MPI ranks

File	Purpose
`system.gro`, `system.top`, `system.mdp`	structure, topology, run parameters
`md.tpr`	run input (made with `gmx_mpi grompp`)
`gromacs_setup.sh`	GCC + OpenMPI + GROMACS 2026.3
`gromacs_checkpoint.sbatch`	run, stop after 90 s, leave `md.cpt`
`gromacs_restart.sbatch`	continue from `md.cpt`

The full run would need about 200 slices of the length used here — we do one stop and one restart.

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing/Exercise5

Outline

- Introduction & Background
- Application-Level Checkpointing: Python
- Transparent Checkpointing: DMTCP and MANA
- Built-in Checkpointing: GROMACS
- Checkpointing Deep Learning: PyTorch
- Best Practices, Wrap-up & Q&A

Training Is a Long Loop



save a checkpoint after every epoch (or every N steps)

- Training runs for hours to days — often longer than the 3-day `gpu` limit
- GPU time is precious: losing a day of training is expensive
- `gpu_requeue` gives access to more GPUs — if your training can be preempted
- Checkpoints are also how you **evaluate**, **fine-tune** and **share** models

What a Training Checkpoint Must Contain

Basic checkpoint (top row) + **complete** checkpoint (bottom row)

Model weights
`model.state_dict()`

Optimizer state
`optimizer.state_dict()` — e.g. Adam moments

Progress
completed epoch (or step)

LR scheduler
`scheduler.state_dict()`

Training-loop state
best val loss, early-stopping counter

RNG states
CPU, CUDA, DataLoader generator

Without the bottom row a resumed run trains — but not the way the original run would have.

Exercise 6: MNIST CNN on a GPU



Version	Script	Checkpoint contents
No checkpoint	`mnist_naive.py`	—
Basic	`mnist_checkpoint.py`	epoch, model, optimizer
Complete	`mnist_complete_checkpoint.py`	+ scheduler, best val loss, early stopping, RNG states

- Adam optimizer; saved atomically after every epoch with `torch.save` → `.tmp` → `os.replace`
- Runs on `gpu_test` (an A100 MIG slice is plenty)

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing/Exercise6

Hands-On Practice: PyTorch

Exercise 6 — Setup and baseline training

```
salloc --partition=gpu_test --gres=gpu:1 --cpus-per-task=4 --mem=16G --time=01:00:00
module load python
mamba activate /n/holylabs/rc_admin/Everyone/checkpoint-training/pt2.14.0_cuda13.2
python -c "import torch; print(torch.__version__,torch.cuda.is_available())"
```

```
python mnist_naive.py --epochs 10 --sleep 2 --device cuda
```

```
Epoch 1/10 | train loss 0.1709 | train acc 94.70% | val loss 0.0422 | val acc 98.63% | time 5.32 s
Epoch 2/10 | train loss 0.0616 | train acc 98.13% | val loss 0.0348 | val acc 98.92% | time 2.38 s
...
```

- Expected: `2.14.0+cu132 True`
- First run downloads MNIST (~12 MB) into `./data`
- Ctrl-C after a few epochs, rerun → back to epoch 1

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing/Exercise6

Basic Checkpoint: Model + Optimizer

mnist_checkpoint.py

```
def save_checkpoint(checkpoint_path, epoch,
                    model, optimizer):
    checkpoint = {
        "epoch": epoch,
        "model_state_dict": model.state_dict(),
        "optimizer_state_dict": optimizer.state_dict(),
    }
    tmp_path = checkpoint_path + ".tmp"
    with open(tmp_path, "wb") as f:
        torch.save(checkpoint, f)
        f.flush()
        os.fsync(f.fileno())

    os.replace(tmp_path, checkpoint_path)
```

```
Loading checkpoint: mnist_checkpoint.pt
Checkpoint restored from epoch 4.
Training will resume at epoch 5.
```

Key points

- Adam keeps two running averages (**moments**) per parameter
- Weights alone → the optimizer restarts cold: a different trajectory
- Checkpoint ≈ 3 × model size (weights + 2 moments): 38 MB here
- Still missing: RNG, scheduler, early stopping → data order differs after the restart

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing/Exercise6

Complete Checkpoint

mnist_complete_checkpoint.py

```
checkpoint = {
    "version": 1,
    "epoch": epoch,                    # last completed epoch
    "model_state_dict": model.state_dict(),
    "optimizer_state_dict": optimizer.state_dict(),
    "scheduler_state_dict": scheduler.state_dict(),
    "best_val_loss": best_val_loss,
    "early_stopping_counter": early_stopping_counter,
    "torch_rng_state": torch.get_rng_state(),
    "train_generator_state": train_generator.get_state(),
    "saved_device_type": device.type,
}
if device.type == "cuda":
    checkpoint["cuda_rng_state"] = \
        torch.cuda.get_rng_state(device).cpu()

with open(tmp_path, "wb") as f:
    torch.save(checkpoint, f)
    f.flush(); os.fsync(f.fileno())
os.replace(tmp_path, checkpoint_path)
```

Key points

- `ReduceLROnPlateau` halves the LR after 2 epochs without improvement
- Early stopping after 5 epochs without improvement
- Their state decides **when** the LR drops and **when** training stops
- `train_generator` controls the DataLoader shuffling

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing/Exercise6

Restoring Correctly

```
checkpoint = torch.load(checkpoint_path,
                        map_location= "cpu",
                        weights_only= True)

model.load_state_dict(checkpoint[ "model_state_dict" ])
optimizer.load_state_dict(
    checkpoint["optimizer_state_dict" ])
move_optimizer_state_to_device(optimizer, device)
scheduler.load_state_dict(
    checkpoint["scheduler_state_dict" ])

best_val_loss = checkpoint[ "best_val_loss" ]
early_stopping_counter = checkpoint[ "early_stopping_counter" ]

torch.set_rng_state(checkpoint[ "torch_rng_state" ].cpu())
train_generator.set_state(
    checkpoint["train_generator_state" ].cpu())
# + CUDA RNG state if running on a GPU
start_epoch = checkpoint[ "epoch"] + 1
```

Key points

- `map_location="cpu"`: load anywhere, then move to the device — a GPU run can resume on a CPU
- `weights_only=True`: safe loading of tensors and plain types
- Build model, optimizer and scheduler **first**, then load their states
- Move optimizer tensors (Adam moments) to the GPU

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing/Exercise6

Hands-On Practice: PyTorch

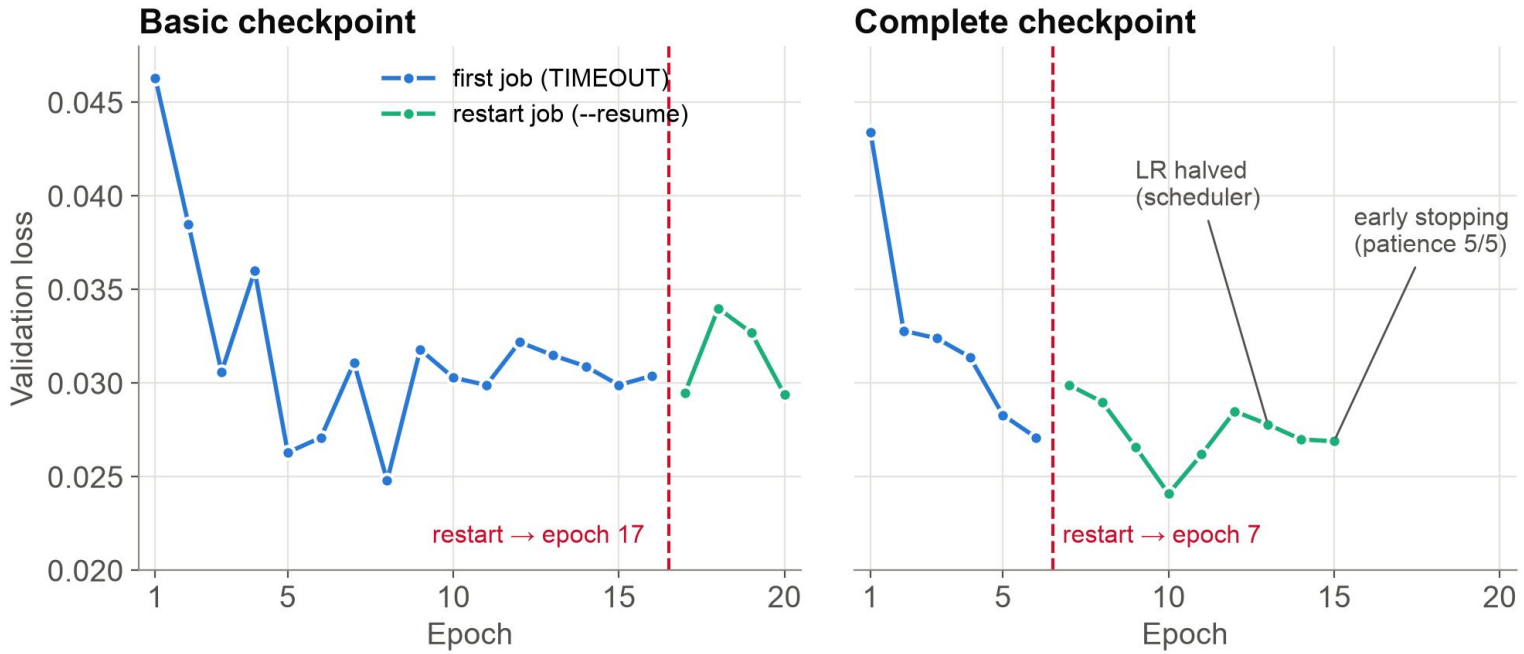
Exercise 6 — Batch checkpoint and restart on gpu_test

```
sbatch mnist_naive.sbatch

JOBID=$(sbatch --parsable \
    mnist_basic_checkpoint.sbatch)
sbatch --dependency=afterany:$JOBID \
    mnist_basic_restart.sbatch

# same for mnist_complete_*.sbatch
```

NOTE: You will need to exit the interactive GPU job before submitting the batch job



`--time=00:01:00` + `--sleep` → the first job ends in `TIMEOUT`; the restart job continues from the last epoch

https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing/Exercise6

Beyond One GPU

```
# DDP: every rank has the same weights
if dist.get_rank() == 0:
    checkpoint = {
        "epoch": epoch,
        "model": model.module.state_dict(),
        "optimizer": optimizer.state_dict()
    }

    with open(tmp_path, "wb") as f:
        torch.save(checkpoint, f)
        f.flush()
        os.fsync(f.fileno())

    os.replace(tmp_path, ckpt_path)

dist.barrier()
```

Tool	Resume with
PyTorch Lightning	`trainer.fit(model, ckpt_path="last")`
Hugging Face Trainer	`trainer.train(resume_from_checkpoint=True)`
FSDP / sharded models	`torch.distributed.checkpoint`

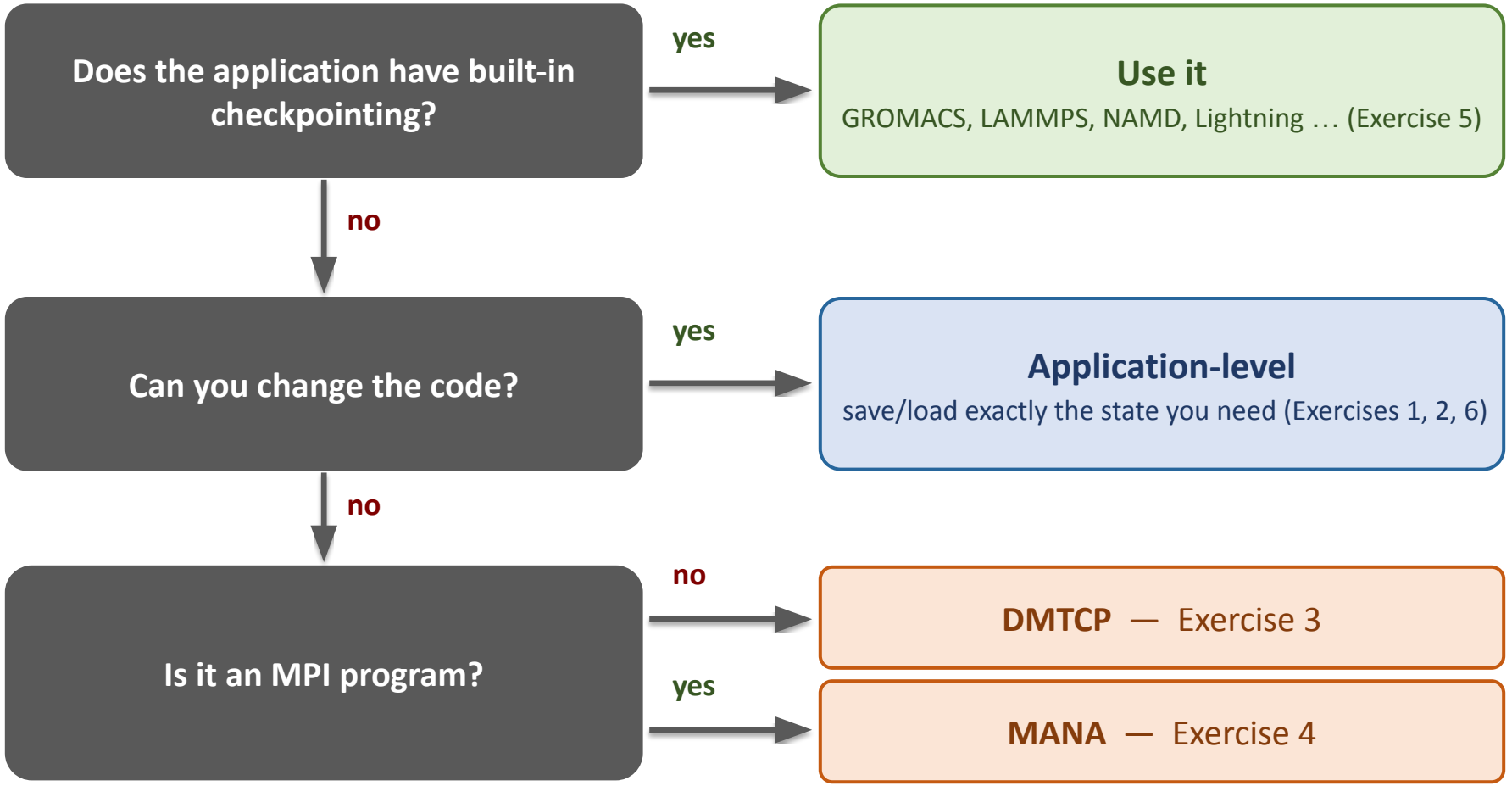
Key points

- Write from **one** rank; load on all (`map_location`)
- Save `model.module.state_dict()` — without the DDP wrapper
- Sharded models: each rank saves its shard in parallel
- Keep **last** (to resume) and **best** (to use)
- Distributed GPU Computing workshop: DDP in depth

Outline

- Introduction & Background
- Application-Level Checkpointing: Python
- Transparent Checkpointing: DMTCP and MANA
- Built-in Checkpointing: GROMACS
- Checkpointing Deep Learning: PyTorch
- Best Practices, Wrap-up & Q&A ←

Choosing an Approach



Then automate

`--signal` for a final checkpoint

`--requeue` or job dependencies

find the checkpoint on startup

test the restart!

Best-Practice Checklist

Design

- Save the **complete** state — including RNG and optimizer
- Put a **version** in the checkpoint; validate parameters on resume
- Write **atomically**: tmp → fsync → rename
- Checkpoint **periodically** and on **SIGTERM / SIGUSR1**
- **Find** the checkpoint automatically on startup

Operations

- **Test the restart** on a short job before the real run
- Store checkpoints on **shared** storage (netscratch)
- Use `--signal`, `--requeue` / `--dependency`, `--open-mode=append`
- Pin **module versions**; restart in the same environment
- Watch disk usage; keep only the last few checkpoints

Common Pitfalls

Pitfall	**Fix**
Overwriting the only checkpoint — the job dies mid-write	Atomic write; keep the previous checkpoint
Checkpoint in node-local `/scratch`	Shared storage — the restart may run on another node
Only weights / only the seed saved	Save optimizer, scheduler and full RNG state
Program in the foreground — the trap never fires	`srun ... &` then `wait`
`--fresh` / `rm checkpoint` inside a requeue script	Clean up only before the **first** submission
Restart with different parameters	Store and validate the configuration
DMTCP on a static binary, MANA with plain `mpicc`	Dynamic linking; build with `mpicc_mana`
The restart was never tested	Kill a short test job on purpose and resume it

Key Takeaways

1. Long jobs **will** be interrupted — plan for it from the start
2. A checkpoint is the **minimum complete state** needed to continue exactly
3. Write it **atomically**, periodically and on Slurm's **warning signal**
4. Let the job **restart itself**: `--requeue`, `SLURM_RESTART_COUNT`, auto-detect
5. Prefer **built-in** → **application-level** → **transparent** (DMTCP, MANA)

References

Workshop exercises: https://github.com/fasrc/User_Codes/tree/master/Training/Checkpointing

NMSU — Backfill and checkpoints: https://hpc.nmsu.edu/discovery/slurm/backfill-and-checkpoints/

Northeastern RC — Checkpointing best practices:
https://rc-docs.northeastern.edu/en/latest/best-practices/checkpointing.html

NERSC — Checkpoint/restart: https://docs.nersc.gov/development/checkpoint-restart/

NERSC — MANA: https://docs.nersc.gov/development/checkpoint-restart/mana/

DMTCP: https://dmtcp.sourceforge.io/index.html

MANA on GitHub: https://github.com/mpickpt/mana

MSU ICER — Checkpointing: https://docs.icer.msu.edu/checkpointing/

Slurm sbatch (--signal, --requeue): https://slurm.schedmd.com/sbatch.html

FASRC documentation

- FASRC docs: FASRC DOCS

- Training
  - Calendar: Training Calendar | FAS Research Computing
  - Material: Training Materials – FASRC DOCS

- Getting help
  - Office hours: Virtual Office Hours | FAS Research Computing
  - Ticket:
    - HUIT Service Portal -> Submit Ticket: Submit Ticket - IT Help
    - Email: rchelp@rc.fas.harvard.edu

Training session evaluation

Please, fill out our training session evaluation. Your feedback is essential for us to improve our trainings!!

https://tinyurl.com/5n777e7f



Questions, Comments, Concerns?