

Introduction to Supercomputing: teacher workbook

Slide notes, demonstrations, worked answers and classroom recovery

Rohit Goswami

2026-10-27

Contents

1	How to teach from this workbook	3
1.1	Four-hour Part I running order	3
1.2	Part I demonstrations and stopping points	4
1.3	Optional survey after the core checkpoints	5
1.4	Workbench exercise checkout	5
1.5	Opening checklist	5
2	Concise instructor lesson: PDF pages 1–22	6
2.1	Pages 1–3: establish the destination	6
2.2	Pages 4–6: shell, modules and Python	6
2.3	Pages 7–9: resources, threads and MPI	6
2.4	Pages 10–13: storage, dependencies and explanation	6
2.5	Pages 14–18: backup divider and runnable listings	6
2.6	Pages 19–22: numerical checks and extended references	7
3	Fundamentals deck: PDF pages 1–51	7
3.1	Pages 1–5: purpose, agenda and HPC	7
3.2	Pages 6–10: a computer, a node and a cluster	7
3.3	Pages 11–17: facilities, architecture and accounting	7
3.4	Pages 18–21: storage by lifetime	8
3.5	Pages 22–26: terminal, browser and classroom access	8
3.6	Pages 27–30: LUMI and portability	8
3.7	Pages 31–33: scheduler and commands	8
3.8	Pages 34–39: job requests and script anatomy	9
3.9	Pages 40–45: software and environment modules	9
3.10	Pages 46–48: first Python and OpenMP jobs	9
3.11	Pages 49–51: consolidation	10
4	Filesystem deck: concepts and hands-on	10
4.1	Overview of applications, communications and I/O	10
4.2	Filesystem overview and storage decisions	10
4.3	Exploring filesystems and file management	10
4.4	Directory management and permissions	11
4.5	Disk usage and bounded write exercise	11
5	Parallel jobs: worked numerical answers	11

5.1	AXPY, serial and MPI	11
5.2	Dot product and reduction	12
5.3	Matrix multiplication	12
6	QCG-PilotJob deck: PDF pages 1–38	12
6.1	Pages 1–6: independent work and the choice of mechanism	12
6.2	Pages 7–13: arrays, limitations and control	12
6.3	Pages 14–17: hands-on examples 1 and 2	13
6.4	Pages 18–23: pilot job and Slurm wrapper	13
6.5	Pages 24–29: Python manager and creating tasks	13
6.6	Pages 30–33: descriptions and execution models	13
6.7	Pages 34–36: dependent means, versions and the critical checkpoint	14
6.8	Pages 37–38: close the workflow argument	14
7	Workflow alternatives deck: PDF pages 1–11	14
7.1	Pages 1–3: one scientific problem, several representations	14
7.2	Page 4: HyperQueue	14
7.3	Page 5: Snakemake	14
7.4	Page 6: FireWorks	15
7.5	Pages 7–9: measurements and idle allocation	15
7.6	Pages 10–11: choosing a tool	15
8	Performance deck: PDF pages 1–41	15
8.1	Pages 1–7: measure the application, not an isolated slogan	15
8.2	Pages 8–15: I/O, communication, compute and memory	15
8.3	Pages 16–18: strong and weak scaling	16
8.4	Pages 19–20: standard optimization techniques	16
8.5	Pages 21–28: roofline, one reveal at a time	16
8.6	Pages 29–34: FLOP counts and worked arithmetic	16
8.7	Pages 35–36: energy footprint	17
8.8	Pages 37–41: Slurm evidence and close	17
9	Data-movement deck: PDF pages 1–18	17
9.1	Pages 1–3: why copying hurts and where data lives	17
9.2	Page 4: pack and verify	17
9.3	Page 5: resumable rsync	18
9.4	Pages 6–8: several streams, mpiFileUtils and a job script	18
9.5	Pages 9–10: from outside the cluster	18
9.6	Pages 11–12: stage in, stage out and the checklist	18
9.7	Pages 13–18: hands-on and debrief	18
10	Questions, answers and teaching recovery	19
10.1	A demonstration fails	19
10.2	Learners finish at different rates	19
10.3	Closing artifacts	19
11	References and provenance	19
12	Source slide atlas	20
12.1	Concise instructor lesson	20
12.2	Fundamentals	20

12.3 Filesystems	21
12.4 QCG-PilotJob	22
12.5 Workflow alternatives	23
12.6 Performance analysis	23
12.7 Data movement	24

1 How to teach from this workbook

Keep the learner slides on the projector and this workbook on a separate screen. The numbered references below use PDF pages, including divider and closing pages. The filesystem deck also contains incremental reveals: finish the current argument before advancing to the next reveal. The printed slide atlas at the end gives every page a stable reference.

The essential outcome is that every learner can explain where a job runs, request a small allocation, inspect the result, and keep durable work separate from disposable data. A faster run is an observation to explain, not the definition of a successful exercise. Establish correctness before comparing timings. Ask for predictions before running a command.

The course is taught on two different kinds of system. Snellius slides describe a national production service. A controller-managed SRC classroom reproduces the workflow with different hardware and resource limits. The private instructor handoff supplies its actual addresses, source revision, partition names and account links. Never read demonstration credentials from a privileged file or put the class enrollment export on the projector.

1.1 Four-hour Part I running order

Use <https://supersurf.rgoswami.me/> beside the course portal. The browser lessons contain commands and fold-out solutions; they do not provide a remote computing session. Keep this workbook on a separate instructor screen. Part I is the four-hour Introduction to HPC class: fundamentals and EESSI, with first jobs, resource requests and a short parallel example. The detailed filesystem, workflow and performance chapters support Parts II and III.

Times are elapsed from the start. Add the room's actual start time. The plan includes a 15-minute break and a ten-minute exit check and recovery margin. Do not add the full browser episode estimates to this selected timetable.

Elapsed	Minutes	Block and learner checkpoint
00:00-00:10	10	Purpose: identify the scientific task and destination
00:10-00:35	25	Access: open a shell and identify the login host
00:35-00:55	20	Architecture: distinguish nodes, processes and cores
00:55-01:25	30	EESSI: load Python and record the environment
01:25-01:55	30	First batch job: save its ID and correct sum
01:55-02:10	15	Break
02:10-02:40	30	Resources: explain requests and calculate 32 SBU
02:40-03:20	40	OpenMP: compare correct one- and two-thread results
03:20-03:40	20	Arrays: map an index and concurrency limit
03:40-03:50	10	Storage: keep the script and result durably
03:50-04:00	10	Exit check, questions and recovery margin

The full instructor route is at <https://supersurf.rgoswami.me/instructors.html>. It names the browser

pages and exact commands for each block. Use selected pages of the extended fundamentals deck. The concise 22-page controller lesson is an alternative continuous explanation, not another required deck.

1.2 Part I demonstrations and stopping points

Opening uses fundamentals pages 1-5. Establish the goal: a correct job, resource request, saved output and an explanation of where the work ran. Use one scientific problem from the room to distinguish memory, arithmetic and many independent inputs. Keep the outcome visible through the class.

Access uses fundamentals pages 22-26: ten minutes to sign in and fifteen for `whoami`, `pwd`, `hostname` and the working directory. The browser page is `fundamentals/11-connecting.html`. Pair a learner with an access problem while the instructor handles account repair privately. Never project the roster or individual setup links.

Architecture uses pages 6-17 for twenty minutes, alongside `fundamentals/12-cluster.html`. Ask pairs to identify login and compute nodes, a process and its threads. Read actual `sinfo` values. A node allocation does not make one serial Python program parallel.

Software uses pages 27-30 and 40-45: fifteen minutes of explanation and fifteen to load Python, record its version and save `module list`. The browser routes are `eessi/shared-stack.html` and `eessi/finding-modules.html`. Select the course-cluster tab. Building software and downloading a client container are reference activities for a group with working cluster logins.

The first job uses pages 31-39 and 46-48: ten minutes to explain the batch request and twenty for the Python submission and output. Use `fundamentals/13-scheduler.html` and `fundamentals/python-and-openmp.html`. The published `python.sh` prints `sum: 499999500000`. Its module version is `Python/3.13.1-GCCcore-14.2.0`; the discovery example uses `Python/3.13.5-GCCcore-14.3.0`. Keep each version with its source script. Save the job ID. Do not submit a second copy because the first is pending.

Leave the 02:10 return time visible during the fifteen-minute break. A pending job can finish while the room is away. Preserve the break if a learner needs more login help.

Resources uses pages 11-21 and 34-39: fifteen minutes of explanation and fifteen for the `fundamentals/surf-accounting.html` exercise. Sixteen cores for two hours on Snellius rome cost 32 SBU. Explain its 16-core minimum, then distinguish that production policy from the smaller classroom request. SRC workspace credits and job SBU are different accounting systems.

OpenMP uses pages 46-48 and `fundamentals/python-and-openmp.html`: ten minutes on processes versus threads, twenty to compile and submit, ten to compare. Compile within a two-CPU allocation. Override the published 32-CPU request with `sbatch --cpus-per-task=2 pi.sh`. Expect two `pi = 3.141592654` lines with thread counts 1 and 2. The elapsed times can differ; the nine decimal digits must agree. Do not promise speedup for a short loop.

Arrays uses QCG pages 1-13 and `fundamentals/21-array.html`: ten minutes of explanation and ten for a four-element classroom example. Indices are 0, 1, 2 and 3; a concurrency limit of two lets at most two elements run at once. This demonstrates indexing, not speedup. The full pilot, MPI, workflow-tool comparisons and 100-input exercises belong in extended classes.

Storage uses fundamentals pages 18-21 and `fundamentals/16-transferring-files.html` for ten minutes. Ask pairs to choose a durable location for the script and result. Distinguish home and project from shared scratch and node-local temporary files. Detailed copy tools and filesystem experiments belong in Part II.

Use the final ten minutes for each pair to explain its job ID, environment, `sum`, two `pi` lines and output location. It also absorbs short queue and access delays. Preserve the first successful job and the break;

shorten the array discussion when a learner needs more help. Printed output supports explanation during an outage, but does not establish a successful live run.

1.3 Optional survey after the core checkpoints

When pairs have the correct Python sum, the two pi lines and a resource explanation by 03:20, replace the 03:20-03:50 array/storage block with this survey. It does not extend the four-hour class. Choose individual previews when the group needs more practice, and preserve the 03:50 exit check.

- Five minutes: filesystem lifetimes and a destination for the result.
- Five minutes: data movement, packing and a checksum.
- Five minutes: an array compared with a pilot allocation.
- Five minutes: MPI ranks and OpenMP threads as a diagram.
- Five minutes: a fair timing comparison. From 8.0 seconds on one core and 1.6 seconds on eight cores, speedup is 5 and efficiency is 0.625.

The previews total 25 minutes, leaving five minutes of transition margin. Use the detailed browser lessons and workbook chapters as references. The example timing table is not a benchmark of the classroom nodes. No preview requires a database, new software installation, profiler or full hybrid run.

1.4 Workbench exercise checkout

The learner website and the course portal are separate browser tabs. Workbench scripts require a separate editable source tree beside `~/introduction-to-supercomputing`. Clone once; inspect an existing checkout rather than cloning over it.

```
git clone https://github.com/HaoZeke/supercomputing_eessi_workbench.git "$HOME/
supercomputing_eessi_workbench"
cd "$HOME/supercomputing_eessi_workbench/lessons/fundamentals/episodes/files/hands-on"
sbatch --partition=rome --cpus-per-task=1 python.sh
queue -u "$USER"
```

Read `slurm-JOBID.out` using the ID returned by `sbatch`. The first line names the allocated host; the next gives `sum: 499999500000`. Compile and submit the parallel example from the same hands-on directory:

```
srunk --partition=rome --ntasks=1 --cpus-per-task=2 --time=00:05:00 bash compilepi
sbatch --partition=rome --cpus-per-task=2 pi.sh
```

Keep the published scripts intact and use command-line resource overrides. The site instructor guide supplies the small array command and the complete pre-class checklist. Run the exact commands before teaching and retain the job IDs and outputs privately. The original course exercises remain in `~/introduction-to-supercomputing`; do not substitute its paths for these.

1.5 Opening checklist

1. Sign in with your own instructor account and required second factor.
2. Open the portal, a learner shell and the roster on separate tabs. Only the portal and learner shell belong on the projected screen.
3. Inspect node state, running jobs, available memory and disk space.
4. Confirm the source directory and PDF revision against the handoff.
5. Run the maintained rehearsal before learners open interactive sessions.
6. Put the portal address and the first exercise directory where everyone can find them. Give individual setup links privately.

The rehearsal command runs inside a two-task CPU allocation with the site modules available. `COURSE` is the exact course directory supplied by the instructor; the command copies exercises into its own rehearsal directory.

```
pixi run rehearse --course "$COURSE" \  
  --module-path /opt/surfcontroller-mc/modules
```

Expected: numerical and workflow checks report PASS, including injected producer failure. A nonzero exit requires investigation. Do not edit an assertion to make a demonstration pass.

2 Concise instructor lesson: PDF pages 1–22

This is the controller-local teaching route. It is separate from the extended course decks and is not an extra prerequisite. Use it when the class needs one continuous path from login to an explained result.

2.1 Pages 1–3: establish the destination

Tell learners that the finished artifact is a correct job with a resource request, a saved output and an explanation of where its data lives. Show the exercise sequence before introducing the architecture vocabulary. Ask everyone to find the same editable course directory. A visible checkpoint is more useful than asking whether everyone is following.

2.2 Pages 4–6: shell, modules and Python

Demonstrate `whoami`, `pwd`, `hostname` and the home directory. Load the software environment and submit the provided Python script. Explain that the script is displayed from the runnable source, not a separate classroom transcription. The expected sum is 499999500000. Keep the job ID and inspect accounting after it leaves the queue.

2.3 Pages 7–9: resources, threads and MPI

Explain one task with two CPUs before comparing it with two tasks. The OpenMP example runs one- and two-thread cases inside the allocation; both must approximate pi correctly. MPI divides work and combines results. Have a learner narrate which operation is local and which is collective. The detailed numerical answer sheet below supplies independent checks.

2.4 Pages 10–13: storage, dependencies and explanation

Choose storage by lifetime, then draw producers and a consumer. The final aggregate must wait for all selected successful producers. Comparing tools requires the same inputs, resources and numerical contract. End the core route by asking pairs to explain their job ID, result, requested CPUs and output location to each other.

2.5 Pages 14–18: backup divider and runnable listings

The divider marks reference material. Use the Python job, compiler script, OpenMP job and pi source only when a learner needs to trace the exact command or implementation. Explain the reduction and thread setting; do not replace a working script with code reconstructed from the screen. These listings are included from runnable course assets by the lesson build.

2.6 Pages 19–22: numerical checks and extended references

Use the answer sheet to distinguish a correct result from an attractive timing. Resource examples must fit the actual classroom nodes. Workflow versions are teaching steps that expose dependencies and failure handling, not five unrelated scientific calculations. The final reference page points to the extended module decks covered in this workbook.

3 Fundamentals deck: PDF pages 1–51

3.1 Pages 1–5: purpose, agenda and HPC

Say: "We use a shared system when the scientific problem exceeds the useful capacity of one workstation, or when many independent calculations need to finish together. The objective is a trustworthy answer within a useful time and resource budget. A large machine does not make a serial program parallel."

Use one simulation, one image-analysis task and one parameter sweep from the room. Ask which needs more memory, more arithmetic, or more independent runs. Accept more than one classification when the learner gives a concrete reason. Avoid promising that every task becomes faster on a GPU.

Ask: "If I run ten independent inputs, have I made one input ten times faster?" Expected answer: no; throughput can improve while each input takes the same time. This distinction prepares both the scheduler and workflow sections. Advance when learners can name their own limiting resource.

3.2 Pages 6–10: a computer, a node and a cluster

Point at processor, memory and storage before naming the interconnect. A core executes instructions; a process has its own address space; threads within a process share memory. A cluster joins nodes through a network. Remote memory is not automatically another part of the local array.

Ask the room to draw two nodes, each with two processes. Then ask where an ordinary Python list lives. Expected: in one process's memory unless a library explicitly distributes or shares it. MPI exchanges data between processes; OpenMP creates cooperating threads within a process.

Misconception: a two-node allocation makes `python script.py` start twice. Correction: allocation and process launch are separate. A batch script starts once; `srun` or another supported launcher starts the requested tasks. Connect this directly to the hostname demonstration below.

3.3 Pages 11–17: facilities, architecture and accounting

Use the location and hardware pictures to connect the abstraction to a physical service. State that the deck's Snellius hardware describes that service and that the class has its own resource inventory. Read actual allocated CPU and memory values rather than treating a diagram as a promise. The classroom inventory can contain emulated devices used only for scheduler instruction; a real numerical GPU demonstration must use the physical type.

Show `sinfo -N -o '%N %c %m %G'` and explain one row. `%c` is the configured CPU count, `%m` is configured node memory in MiB, and `%G` lists generic resources. Do not add repeated rows from overlapping partitions as if they were additional nodes.

Accounting exercise: two allocated cores held for 30 minutes consume one core-hour, even if the program uses only one core. Production billing can apply partition-specific factors and minimum allocations. SRC charges its collaboration wallet for workspaces; `sacct` remains useful for job accounting but is not the collaboration's credit invoice. Distinguish requested wall time, elapsed time and accumulated CPU time.

3.4 Pages 18–21: storage by lifetime

Ask "What would you regret losing?" before discussing throughput. Source, configuration and small results belong in durable storage with a stated backup policy. Recomputable intermediates belong on scratch. Job-local storage belongs to a particular job and is not a shared communication path.

Snellius filesystem names and retention rules belong to Snellius. On the course cluster use the private handoff's mapping: shared home/project, shared scratch and job-local \$TMPDIR. A successful `df` reports capacity, not a personal quota or proof of a backup. A cloud disk can be ephemeral even when its contents are shared over a filesystem.

Ask where to put a 20 GB intermediate file, a job script and a final table. Expected: scratch, durable home/project, durable home/project respectively; move the final table off job-local storage before the allocation ends.

3.5 Pages 22–26: terminal, browser and classroom access

Demonstrate the learner route, not the instructor's privileged shell. An enrollment link sets a credential; the credential must be saved before login works. A learner who attended both courses keeps one identity. Browser access is still a remote session: files displayed there live on the cluster.

Show `whoami`, `hostname`, `pwd` and `printf '%s\n' "$HOME"`. Explain the difference between a command, its output and the prompt. Ask learners not to type the prompt copied from an example. The `hostname` changes when a process runs on a compute node; the shared home directory can remain the same.

If a login fails, check account name, saved credentials and the exact portal URL. Do not repeatedly replace links or reset credentials without checking the existing account. Keep the class moving with a partner's screen while the support instructor handles an individual login privately.

3.6 Pages 27–30: LUMI and portability

Treat these pages as a comparison of architectures and services, not as instructions to run Snellius commands on another machine. Processor type, accelerator vendor, interconnect, storage and software environment all affect how a workload is built and launched. A portable source program can still need different binaries, modules and job requests.

Ask: "Can a binary compiled for one instruction set run everywhere?" Expected: no. Source portability, binary compatibility and numerical reproducibility are different properties. Relate this to selecting the correct software stack and checking a numerical answer after a platform change. The hardware detail is an extension if learners are still working through their first login.

3.7 Pages 31–33: scheduler and commands

Say: "A scheduler is a resource allocator with a queue. It needs a description of the work, a resource request and a time limit. Acceptance into the queue does not mean the program has run successfully."

Use this sequence, explaining each output before typing the next command:

```
sinfo
sbatch python.sh
squeue -u "$USER"
sacct -X --format=JobID,JobName,State,ExitCode,Elapsed
```

Expected: `sbatch` returns a job ID, `squeue` reports pending/running work, and `sacct` retains a terminal state after the job leaves the queue. An empty queue is consistent with either completion or failure. `scancel JOBID` cancels only the job whose ID the learner has checked.

Ask: "Is Submitted batch job 123 proof of a correct answer?" Expected: no; inspect state, exit code, output and the scientific result.

3.8 Pages 34–39: job requests and script anatomy

Walk through the resource directives before the executable. `--nodes` is a node count, `--ntasks` is a process/task count, and `--cpus-per-task` reserves CPUs for each task. A threaded program needs its own thread setting. Asking for eight CPUs does not cause a serial program to create eight threads.

All `#SBATCH` directives must precede executable shell statements. Shell variables in those directives are not expanded by the shell. Supply variable values through command-line options or write a concrete directive. Command line submission options override the corresponding script directives.

Have learners predict the difference between one task with two CPUs and two tasks with one CPU each. Then run `srunch -N2 -n2 hostname` in a suitable small allocation. Expect two task lines; identical versus different node names depend on the explicit node request. This is a launch demonstration, not a network benchmark.

Common failure: a request retains a production node's 32-core default on a two-core classroom node. Inspect the script and submit with a two-core override. Do not change the production resource request in the shared course copy for everyone.

3.9 Pages 40–45: software and environment modules

Say: "A module changes the current shell's environment so that a selected software installation and its libraries can be found. It does not install software into your home, allocate hardware or submit a job."

```
module list
module load 2025
module spider Python
module load Python/3.13.1-GCCcore-14.2.0
command -v python3
python3 --version
```

Expected: the executable path and version identify the selected installation. Load modules inside the job script so another login shell can reproduce the environment. Compiler and MPI libraries must belong to a compatible stack. The course's site aliases can expose supported teaching names without making every package part of the upstream EESSI release.

Ask: "Why did loading a compiler in a different terminal not fix this job?" Expected: environments belong to processes and are inherited when processes are launched; an unrelated terminal is not retroactively changed.

3.10 Pages 46–48: first Python and OpenMP jobs

Start in `~/introduction-to-supercomputing/course-module-fundamentals-of-hpc/hands-on`. The directory is an editable learner copy; `$COURSE` can instead identify a separately prepared copy. The Python script and OpenMP sources are runnable course assets, not code transcribed from a screenshot.

```
sbatch python.sh
bash compilepi
sbatch --cpus-per-task=2 pi.sh
```

The Python answer is `sum: 499999500000`. The integral example approximates pi as 3.141592654 to the printed precision. With a two-CPU override, the script runs its one- and two-thread cases and omits larger cases. Each must give the same numerical result within rounding; elapsed time may vary.

Explain the midpoint rule: divide $[0,1]$ into equal intervals and sum $4/(1+x*x)$ at their midpoints. OpenMP divides independent loop iterations; the reduction combines partial sums without a shared-update race. The floating-point summation order can change the final low-order bits.

Before accepting the exercise, inspect `COMPLETED` and exit `0:0`, find the numerical line in the output, and ask the learner which node ran the job. If compilation fails, inspect `module list` and the compiler path. If the job is pending, inspect its reason instead of submitting many duplicates.

3.11 Pages 49–51: consolidation

Ask each pair to explain one of these without reading a slide: login versus compute node; allocation versus launch; process versus thread; source versus executable; environment versus installation; home versus scratch. Expand an abbreviation only when it helps the explanation.

Exit checkpoint: each learner has an identity file in home, a completed Python job, a correct pi result and a saved job ID. A learner with those four artifacts can continue even if the optional architecture comparisons were shortened. Transition: "The program works; where should its data live?"

4 Filesystem deck: concepts and hands-on

The filesystem PDF has 20 pages because some frames reveal additional content. Follow the named frame and its reveals together. The slide atlas provides the physical page numbers.

4.1 Overview of applications, communications and I/O

Separate three kinds of time: computation, communication and storage access. Increasing the number of processes may reduce local arithmetic while increasing synchronization or filesystem contention. A shared filesystem does not make simultaneous tiny-file access free.

Use a checkpoint as the concrete example. It contains state sufficient to resume; a visualization file may contain only a reduced view. Progress logs serve a different purpose again. Ask which outputs need full numerical precision and which can be sampled. Expected: restart correctness dictates checkpoint precision; a plot can often use fewer fields or frames.

4.2 Filesystem overview and storage decisions

Read the diagram as a lifetime map. Distinguish a project's allocation from the filesystem's total capacity. Explain that a fast cache or scratch tier is not a backup. The private class handoff supplies the actual SRC mapping; the Snellius retention numbers on the slides must not be transferred to it.

Classify source, raw inputs, temporary arrays, checkpoints and final tables with the room. For every answer, ask "What happens when the job ends? What happens when the VM is lost? Where is the independent copy?" RAID, a shared mount and an independent restore are different properties.

4.3 Exploring filesystems and file management

```
pwd
df -h /home /scratch
du -sh "$HOME"
ls -lh
```

Explain that `df` reports a filesystem, `du` walks directory contents and `ls -l` reports individual files. Sparse allocation, permissions and shared targets can make superficially similar totals differ. Do not sum capacity reports for two targets backed by the same physical disk.

Use a newly created exercise directory. Demonstrate `cp` and `mv` on files the learner has just created. Require a prediction of the resulting path. The copy leaves a source; the move changes a name/location. Neither proves that a second independent backup exists.

4.4 Directory management and permissions

A directory's execute permission permits traversal; its write permission permits changing directory entries. File write permission governs the file's contents. A readable file can still be inaccessible if a parent directory cannot be traversed. Use `id`, `ls -ld DIRECTORY` and `namei -l PATH` to diagnose access without granting broad write permissions.

Ask what `rwX` means for a directory rather than a file. Expected: listing, entry changes and traversal have different roles. Do not teach `chmod 777` as a universal repair. Shared project permissions should be administered by the instructor; learners inspect their own directory and test a small file.

4.5 Disk usage and bounded write exercise

The exercise writes a vector in text or binary form. Its default production range is too large to use blindly in a classroom. Keep the writes in a dedicated scratch directory and use a small exponent range. The shell variables below are exported to `sbatch` and then read by the script.

```
cd ~/introduction-to-supercomputing/course-module-snellius-fs/ex1
mkdir -p "$HOME/scratch/fs-exercise"
ibeg=6 iend=7 wdir="$HOME/scratch/fs-exercise" sbatch run.sh
ibeg=6 iend=7 wdir=TMPDIR sbatch run.sh
```

One million binary doubles occupy exactly 8,000,000 payload bytes. ASCII size depends on formatting and values. Compare the same element count and format on each destination. `$TMPDIR` must be inspected inside the job; its path and lifetime are allocation-specific. A shared node may change a timing, so repeat a bounded case and explain variation before ranking storage.

Checkpoint: inspect the generated file, job output and chosen destination. Keep the write bounded; cleanup applies only to verified exercise artifacts under the learner's own directory. A failed write is not a valid benchmark.

5 Parallel jobs: worked numerical answers

These exercises use `course-module-snellius-hybrid-jobs/hands-on/ex1` through `ex5`. Read the actual script names in each directory. Work from the published class copy, load its compatible compiler/MPI modules and use small inputs before requesting more resources.

5.1 AXPY, serial and MPI

AXPY means $y = a \cdot x + y$. With $a=2$, $x[i]=1$, $y[i]=2$ and 100 repetitions, the printed `y[0]` must be 202. This is independent of task count. The two arrays need approximately $16 \cdot N$ bytes, excluding runtime overhead.

For $N=10003$ and two ranks, an equal-only split would lose an element. The partition must cover all global indices exactly once. Ask learners to show the local lengths and offsets, then compare the numerical answer with the serial case. MPI rank count and OpenMP thread count are separate axes. Use small input sizes before production-scale vectors.

```
# Run from the corresponding ex1 or ex2 directory.
sbatch --cpus-per-task=1 submit-axpy.sh 10003
```

```
sbatch --ntasks=2 --cpus-per-task=1 submit-axpy-hybrid.sh 10003
```

Explain why `clock()` and elapsed wall-clock time can tell different stories for parallel code. The learner should report the timer used rather than comparing unlike measurements as a speedup.

5.2 Dot product and reduction

Both audited serial and MPI programs sum 100 repetitions of `sin(i) * 0.5*cos(i)`. For $N=10003$ the expected printed result is about `-3.690724574`. One independent closed form is $25*\sin(N)*\sin(N-1)/\sin(1)$. A constant result copied from another input size is not an oracle. The reduction combines contributions from every rank; OpenMP's local reduction does not replace the global MPI operation.

Ask: "Could rank zero's local sum look plausible while most of the vector is missing?" Expected: yes; compare against an independent global answer and include an uneven input size. Use the maintained rehearsal when the class wants the exact compiler, launch and numerical checks together.

5.3 Matrix multiplication

With all entries of A equal to 1 and B equal to 2, each entry of C is $2*N$. Summing all $N*N$ entries gives checksum $2*N^3$. For $N=40$ the checksum is

1. Repeated multiplication overwrites C ; it does not multiply this

checksum by the repeat count. Memory for three double matrices is roughly $24*N*N$ bytes; arithmetic is roughly $2*N^3$ operations per multiplication.

Show why matrix multiplication has more reuse than a single pass over two vectors. Ask learners to distinguish a mathematical operation count from hardware instruction count. Use the printed checksum, then timings. The production example's larger N is an extension, not a prerequisite.

6 QCG-PilotJob deck: PDF pages 1–38

6.1 Pages 1–6: independent work and the choice of mechanism

Start with 100 independent CSV means, then add one final aggregation. Draw producer boxes and a single consumer. Independent producers can run in any order; the consumer needs every selected successful producer. "Parallel" does not imply "no dependencies anywhere."

Compare shell loops, background processes, job arrays and pilot jobs fairly. A loop is easy to read and can be sufficient. An array is handled directly by the cluster scheduler. A pilot allocates resources once and schedules smaller tasks inside them. A pilot's startup and per-task overhead can dominate extremely short work, so avoid claiming universal superiority.

Ask how many resources the consumer needs while waiting. Expected: none need be actively computing, but a pilot allocation may still be held and billed. This sets up the idle-core measurement.

6.2 Pages 7–13: arrays, limitations and control

Each array index identifies an independent input. Show `sbatch --array=0-9%2 array_job.sh`: ten elements are eligible, at most two run simultaneously. The percent limit controls concurrency, not the total number of elements. Unique output names prevent competing tasks from overwriting one file.

Dependencies need a policy for failure. Distinguish "finished" from "succeeded". A producer that exits nonzero must not silently count as a valid mean. Application-level dependencies and scheduler-level dependencies can solve different layers of the workflow.

Ask learners what happens if the aggregator simply sleeps for 30 seconds. Expected: machine load and input size can exceed the guess; a sleep is not a dependency. Similarly, checking that some outputs exist does not prove that the complete selected set exists.

6.3 Pages 14–17: hands-on examples 1 and 2

Read the examples' README before submission. Establish the input-to-output mapping with one small task, then submit the bounded array. Inspect one output, one error log and the accounting state. Count successful outputs against the selected index set, not against an unqualified directory glob.

Classroom request from the example 2 directory:

```
sbatch --array=0-19%4 array_job.sh
```

Expected: 20 selected tasks eventually finish, with up to four running in that array at a time when resources permit. Use the job's output names from the actual script. A terminal submission success is not the task's exit code.

6.4 Pages 18–23: pilot job and Slurm wrapper

Explain the three files in order: a Slurm script allocates resources and loads software; a Python driver creates the pilot manager and submits work; the executable processes an individual input. A Python API call does not create cloud VMs or enlarge a Slurm allocation.

The wrapper stages data into job scratch and copies selected results back. Relative paths are interpreted from the driver's working directory. Ask a learner to point to the input path before submitting. If input is supplied as a tar archive, extract it in the exercise copy before running the driver. An empty input set is an error to diagnose, not a fast successful workflow.

Example 3 on small classroom nodes can use `sbatch --nodes=3 --ntasks-per-node=2 qcg_job.sh`. Inspect the live topology and use the handoff's resource override if the class differs.

6.5 Pages 24–29: Python manager and creating tasks

Walk through manager construction, task descriptions, submission, waiting and shutdown in the runnable driver. Separate a task identifier from its executable and arguments. Show the selected list before execution so the class knows what completeness means.

Ask whether a Python exception in the driver and a nonzero exit in a task are the same event. Expected: no; the driver must inspect/report task failures and return a failing workflow status when required work fails. Keep the driver's logs as well as task stdout/stderr.

6.6 Pages 30–33: descriptions and execution models

Distinguish task resource requests, working directory, environment, input arguments and output files. Use one concrete task and trace its lifecycle. Requesting more task resources than the pilot owns cannot create capacity. For external programs, the environment and executable path must be valid where the task actually launches.

Avoid mixing shell quoting with a structured argument list. A filename containing spaces is a useful discussion prompt: it remains one argument when the API accepts an argument list. The displayed listing is a guide to the runnable driver, not an instruction to reconstruct it by hand.

6.7 Pages 34–36: dependent means, versions and the critical checkpoint

In example 4 use a small selected set, one version at a time:

```
QCG_MAX_INPUTS=5 sbatch --ntasks=2 qcg_job.sh
```

The source wrapper selects its driver version explicitly. To compare versions, use a separate exercise copy and select exactly one runnable driver in that copy; do not let concurrent runs write the same result file. The result is copied into `results/result.csv` by the wrapper.

For each input, independently compute its arithmetic mean; compare the corresponding aggregate row and the total selected row count. Version 4 uses a manifest and checks every selected producer. Ask why scanning all files in an old output directory can produce a plausible but wrong result. Expected: stale files can be mistaken for this run's outputs.

In example 5 compare `pipeline.sh` and `qcg_job.sh` with the same `TIMES_FILE=times_short.txt` and `--ntasks=2`. Keep worker count, task durations and inputs fixed. The block-based pipeline waits at a group boundary; a dynamic scheduler can start the next ready task when a slot becomes free. Startup cost can reverse the ranking for very short tasks.

6.8 Pages 37–38: close the workflow argument

Ask the class to name the unit of allocation, unit of work and dependency condition. Expected: Slurm allocation, application task, successful completion of the required predecessors. A correct result is not sufficient if failures are hidden; a failed producer must make the required consumer or workflow fail. This is why the rehearsal injects a task failure.

7 Workflow alternatives deck: PDF pages 1–11

7.1 Pages 1–3: one scientific problem, several representations

The comparison holds the input set and mathematical operation fixed. The tools differ in how work is represented, how dependencies are expressed, how state is kept and how workers use an allocation. Compare those properties before comparing a bar chart. The file-backed FireWorks example has one worker; it is not an equal-worker performance comparison with a 16-task run.

Ask learners to redraw the same producer/consumer graph without naming a tool. If they can do that, they can inspect whether any implementation preserves the dependency and the required failure policy.

7.2 Page 4: HyperQueue

The server and worker belong to the allocation. A submission consumes the selected `files.txt` inputs, and `--wait` waits for the submitted work before the next shell command aggregates it. Explain where a command's failing status must stop the shell. The same server directory must not be shared accidentally between unrelated exercise runs.

Ask: "Can a completed wait hide a failed task?" Inspect the tool's reported task status and the wrapper's exit handling rather than assuming that wait means success. Use the existing rehearsal's injected-failure check.

7.3 Page 5: Snakemake

Explain files as the interface between rules: inputs describe prerequisites and outputs describe products. The execution graph comes from requested targets and these relationships. A rule that reports success without writing its declared output is not a valid producer.

Ask why merely changing a filename can change what runs. Expected: filenames are part of the dependency specification. The exercise's manifest bounds the selected inputs; do not compare a five-input run with a thousand-input run because both create a file named `result.csv`.

7.4 Page 6: FireWorks

Explain a workflow as linked task objects with a persistent state store. The classroom file-backed example is intentionally a single-worker route; do not start multiple workers against that store for the sake of matching a bar chart. A production deployment can use a different supported state backend. Persistence, retry policy and worker concurrency are distinct design choices.

Ask what must be retained to inspect a failed workflow. Expected: state, task identity, parameters and relevant logs, not only the final aggregate.

7.5 Pages 7–9: measurements and idle allocation

Read axes, input count, allocation and worker count aloud before discussing the winner. Define idle core-time as allocated core-time minus occupied task-time under the measurement's assumptions. Task duration variation makes fixed-size batches leave slots idle near batch boundaries.

For a two-slot toy example with durations 1, 4, 1 and 4 seconds, a block-based schedule pairs each short task with a long one and takes about 8 seconds ignoring overhead. Dynamic scheduling can use a free slot at 1 second, then start the last task at 2 seconds and finish near 6 seconds. The exact result depends on task order and overhead; draw the timeline.

7.6 Pages 10–11: choosing a tool

Ask for a justified choice, not a universal winner. A small script, familiar file-based rules, explicit task state, retries, deployment constraints and observability can each matter. A learner should state both a required property and the cost of obtaining it. End by reconnecting every choice to the same numerical and failure checks.

8 Performance deck: PDF pages 1–41

8.1 Pages 1–7: measure the application, not an isolated slogan

State the scientific question, input size, correct result and performance quantity before profiling. Wall time answers the user's waiting-time question; CPU time, throughput, memory and energy answer other questions. Report the measurement boundary, repetitions, software and allocated resources. A fast incorrect run is not an optimization.

Use the application lifecycle to distinguish verification, representative measurement, bottleneck diagnosis, a targeted change and a repeated check. Ask what evidence would falsify the proposed bottleneck. A profile is a measurement with overhead and sampling limits, not a complete explanation by itself.

8.2 Pages 8–15: I/O, communication, compute and memory

For I/O, inspect bytes, call count, file count and access pattern. Many tiny files can be metadata-bound even when total bytes are small. Cached reads can hide storage cost. For communication, separate latency, bandwidth, collectives, synchronization and load imbalance. A rank waiting in a collective may be waiting for slower work on another rank.

For compute, examine instruction mix and vectorization rather than reading CPU utilization as useful work. For memory, distinguish capacity from bandwidth and locality. A program can fit in RAM and still spend most time waiting for data. Ask the class to propose a measurement for each of the four categories before naming a tool.

8.3 Pages 16–18: strong and weak scaling

Strong scaling holds the total problem fixed while increasing resources. Speedup is $T(1)/T(p)$ and efficiency is speedup divided by p . With $T(1)=100$ s and $T(4)=40$ s, speedup is 2.5 and efficiency is 62.5%. Weak scaling holds work per worker approximately fixed while total work grows; ideal elapsed time remains roughly constant.

Ask why a strong-scaling efficiency can fall without a new bug. Expected: serial work, communication, synchronization, imbalance and decreasing work per worker can dominate. A classroom mesh and small VMs do not reproduce the production interconnect, so use them to teach the method rather than promise national-system scaling ratios.

8.4 Pages 19–20: standard optimization techniques

Change one identified cause, keep the numerical contract and repeat the measurement. Algorithm changes can dominate compiler-level tuning. Data layout, locality, vectorization and reduced synchronization address different bottlenecks. Document the baseline and the changed configuration.

Ask whether doubling requested CPUs is an optimization when elapsed time is unchanged. Expected: it can increase resource cost without improving the selected outcome. Consider throughput and core-hours as well as time.

8.5 Pages 21–28: roofline, one reveal at a time

The vertical axis is achieved floating-point operations per second; the horizontal axis is arithmetic intensity, operations per byte moved at the specified memory level. The roof is the smaller of peak compute throughput and bandwidth times intensity. Say which memory level the byte count refers to; cache reuse changes the relevant traffic.

Use an invented machine explicitly: bandwidth 100 GB/s and peak compute 1000 GFLOP/s. The ridge point is 10 FLOP/byte. An intensity of 0.25 FLOP/byte gives a bandwidth roof of 25 GFLOP/s; an intensity of 20 is capped by the 1000 GFLOP/s compute roof. These are teaching numbers, not measured class hardware specifications.

For each reveal, ask whether moving right, raising the bandwidth roof or raising peak compute could help the highlighted kernel. A point below both roofs can reflect inefficiency, latency, dependencies or another resource; the chart does not uniquely diagnose the cause.

8.6 Pages 29–34: FLOP counts and worked arithmetic

One multiply and one add conventionally count as two FLOPs, including a fused multiply-add. AXPY does about two FLOPs per element. Reading x and y and writing y gives an idealized 24 bytes per double-precision element, so intensity is about $1/12$ FLOP per byte before additional traffic/reuse.

A dot product also does about two FLOPs per element and reads two double vectors, about $1/8$ FLOP per byte in the idealized streaming case. Dense matrix multiplication does roughly $2N^3$ FLOPs; reuse can make its arithmetic intensity grow with N . State whether repeat counts, initialization, allocation and reduction are inside the timing boundary.

Ask learners to count one loop iteration themselves. Accept an alternative byte model when its cache and write assumptions are explicit. Do not equate an algorithmic FLOP count with the number of machine instructions.

8.7 Pages 35–36: energy footprint

Energy is the integral of power over time. A constant average power of 200 W for 30 seconds corresponds to 6000 J, about 1.67 Wh. Faster elapsed time need not minimize energy, and device power is not the whole-system power. State the measurement boundary and sensor availability.

The SRC classroom may not expose production hardware counters or energy telemetry. An unavailable sensor is not zero energy. Use the dimensional calculation and explain what extra measurement would be needed for a defensible real-system result.

8.8 Pages 37–41: Slurm evidence and close

Use `sacct` for terminal state and elapsed resource accounting, `seff` after completion, and a suitable profiler for code behavior. A sampled maximum RSS can miss a brief peak; the configured collection interval matters. Requesting memory and measuring memory answer different questions.

Exit question: "Which observation supports your bottleneck claim, and what would you measure after changing the code?" Require a resource, a measurement boundary and a correctness check. End with a learner explanation rather than a collection of profiler screenshots.

9 Data-movement deck: PDF pages 1–18

9.1 Pages 1–3: why copying hurts and where data lives

Copy time includes discovery, metadata, bytes, network and destination work. A million tiny files and one large file with the same payload need not take the same time. Locate both endpoints and identify which machine interprets each path. A command started on a laptop sees local paths there; a command started in a job sees the compute node's paths.

Ask which of network, metadata or bulk throughput limits an example before introducing parallel streams. An extra process is useful only if it can use capacity not already saturated elsewhere.

9.2 Page 4: pack and verify

Packing reduces per-file transfer overhead and can make a directory easier to verify as one object. Sorting archive entries does not by itself make archive bytes reproducible: timestamps, ownership and format metadata also matter. A checksum verifies the bytes actually read, not that the dataset contains the scientifically intended records.

```
tar -cf run.tar run/
sha256sum run.tar > run.tar.sha256
# Copy both files, then run at the destination:
sha256sum -c run.tar.sha256
```

Expected: `run.tar: OK`. Keep verification inside the measurement boundary when comparing complete transfer workflows. A chart measuring copy of a prepared archive excludes the cost of creating that archive.

9.3 Page 5: resumable rsync

Explain source trailing slash with a tiny directory before copying a large tree. `-a` requests archive behavior; `--partial` retains incomplete data; `--info=progress2` displays aggregate progress. `--checksum` reads content when size and timestamp are insufficient, at the cost of full reads.

```
rsync -a --partial --info=progress2 source/ destination/  
rsync -ain --checksum source/ destination/
```

The second command is an itemized dry run. Inspect all reported differences. No deletion option is needed for this exercise. A resumed transfer should be checked at the destination before treating a source as disposable.

9.4 Pages 6–8: several streams, mpiFileUtils and a job script

Parallel streams can help a set of large independent shards, but also cause metadata contention or network saturation. Begin with one stream and measure. The destination tree and output ownership must be correct before parallel copies start. A distributed copy tool needs a supported allocation and filesystem visibility on every participating node.

Read a real job script top to bottom: resource request, modules, paths, copy, verification, exit handling and result retention. Directory and variable quoting must survive spaces. The fastest command that silently misses a shard is not a successful copy.

9.5 Pages 9–10: from outside the cluster

Draw laptop, login host, shared filesystem and compute job. Explain which endpoint needs authentication and where a transfer is permitted by service policy. A university VPN, two-factor login and a cloud workspace's SSH key are different mechanisms. Use the actual service instructions; do not teach bypasses for an authentication problem.

Ask a learner to narrate a source and destination path before running a command. The colon in `host:path` changes which machine resolves the path. Copy only the exercise data and do not include credential directories.

9.6 Pages 11–12: stage in, stage out and the checklist

Job-local staging trades a bounded copy for repeated fast local reads. The source must remain authoritative; results must be copied back before the allocation ends. A failed application should retain useful diagnostic information, and a failed copy must not be hidden by a later successful cleanup command.

The checklist is: intended source; intended destination; sufficient space; bounded concurrency; correct permissions; successful process status; verified bytes; durable result location. Ask which item a progress bar does not establish. Expected: most of them.

9.7 Pages 13–18: hands-on and debrief

Have pairs predict whether packing helps before timing both paths. Record file count, bytes, cache conditions, stream count and whether packing and verification were timed. Repeat the same bounded case rather than inflating the data until a shared disk fills.

Interrupt only a learner-owned copy in the dedicated exercise directory, rerun it with retained partial data and verify the result. Stage a small dataset in `$TMPDIR` inside a job and show that durable results are copied to a location outside that directory. Do not delete a source as part of proving the copy worked.

End with two different explanations: performance describes why the transfer took that time; correctness describes why the destination can be trusted.

10 Questions, answers and teaching recovery

Question	Answer to establish
My queue is empty. Did my job succeed?	Check accounting, exit status and output.
Can I use all free cores shown on a node?	Only the allocation's resources belong to the job.
Does a module allocate a GPU?	No; software selection and resource allocation are separate.
Is shared scratch a backup?	No; consult its lifetime and independent recovery policy.
Is the fastest workflow always best?	Required correctness, state, retries and operating constraints matter.
Can every learner reserve a whole card?	Whole-card concurrency is limited by physical cards; use the shared option for suitable exercises.
Why is my result different at low precision?	Check input, algorithm and tolerance before attributing it to floating-point order.

10.1 A demonstration fails

Keep the exact error and the job ID. First classify the failure: access, environment, pending allocation, application error, missing input or missing result. A pending allocation is not an application crash. A drained node should be diagnosed by the operator; blindly resuming it can conceal a real configuration problem.

For a numerical mismatch, return to the smallest verified input and compare the independent answer. For missing inputs, inspect the manifest and working directory. For an unavailable profiler, teach the measurement question and read the saved example rather than inventing a live result. Label saved output explicitly and do not present it as execution from the current job.

10.2 Learners finish at different rates

The core checkpoint is a correct finished job and an explanation. Faster learners vary one resource or input parameter and predict the consequence. Pair a learner who can explain an error with a learner who can operate the commands. Do not give everyone a longer queue merely because a few learners have finished. Keep extensions in separate directories so they cannot overwrite the core exercise's evidence.

10.3 Closing artifacts

Each learner keeps the job script, module versions, job IDs, selected input manifest, numerical result and a short interpretation of one measurement. Discuss how those artifacts support a repeatable scientific result. Remind learners of the class's retention/expiry notice and the destination for work they want to keep. Only the operator handles cluster lifecycle actions.

11 References and provenance

- Source course: <https://github.com/sara-nl/introduction-to-supercomputing>
- Controller lesson and rehearsal: `teaching/lesson.tex` and `e2e/rehearse.py`.

- Slurm batch semantics: <https://slurm.schedmd.com/sbatch.html>
- EESSI documentation: <https://www.eessi.io/docs/>
- Transfer and checksum commands should be checked against their installed manuals when taught on a different service.

The generated slide atlas records PDF hashes and every PDF page used by the workbook. Branch-specific decks remain separate; building this workbook does not merge the upstream slide branches or change the deployed course.

12 Source slide atlas

References are PDF page numbers, not animation clicks. Full hashes are in `workbook-provenance.json`.

12.1 Concise instructor lesson

Source: `lesson.pdf`. SHA-256 prefix: `8c5c61be7da775f6`.

Every PDF page below points to the named teaching discussion; repeated reveals share a discussion.

- PDF page 1: Destination and route.
- PDF page 2: Destination and route.
- PDF page 3: Destination and route.
- PDF page 4: Shell, modules and Python.
- PDF page 5: Shell, modules and Python.
- PDF page 6: Shell, modules and Python.
- PDF page 7: Resources, OpenMP and MPI.
- PDF page 8: Resources, OpenMP and MPI.
- PDF page 9: Resources, OpenMP and MPI.
- PDF page 10: Storage, dependencies and explanation.
- PDF page 11: Storage, dependencies and explanation.
- PDF page 12: Storage, dependencies and explanation.
- PDF page 13: Storage, dependencies and explanation.
- PDF page 14: Backup listings.
- PDF page 15: Backup listings.
- PDF page 16: Backup listings.
- PDF page 17: Backup listings.
- PDF page 18: Backup listings.
- PDF page 19: Checks and references.
- PDF page 20: Checks and references.
- PDF page 21: Checks and references.
- PDF page 22: Checks and references.

12.2 Fundamentals

Source: `fundamentals.pdf`. SHA-256 prefix: `16220ea502311efe`.

Every PDF page below points to the named teaching discussion; repeated reveals share a discussion.

- PDF page 1: Purpose and HPC.
- PDF page 2: Purpose and HPC.
- PDF page 3: Purpose and HPC.
- PDF page 4: Purpose and HPC.
- PDF page 5: Purpose and HPC.

- PDF page 6: Computer, node and cluster.
- PDF page 7: Computer, node and cluster.
- PDF page 8: Computer, node and cluster.
- PDF page 9: Computer, node and cluster.
- PDF page 10: Computer, node and cluster.
- PDF page 11: Facilities and accounting.
- PDF page 12: Facilities and accounting.
- PDF page 13: Facilities and accounting.
- PDF page 14: Facilities and accounting.
- PDF page 15: Facilities and accounting.
- PDF page 16: Facilities and accounting.
- PDF page 17: Facilities and accounting.
- PDF page 18: Storage lifetimes.
- PDF page 19: Storage lifetimes.
- PDF page 20: Storage lifetimes.
- PDF page 21: Storage lifetimes.
- PDF page 22: Access.
- PDF page 23: Access.
- PDF page 24: Access.
- PDF page 25: Access.
- PDF page 26: Access.
- PDF page 27: Portability.
- PDF page 28: Portability.
- PDF page 29: Portability.
- PDF page 30: Portability.
- PDF page 31: Scheduler.
- PDF page 32: Scheduler.
- PDF page 33: Scheduler.
- PDF page 34: Job requests.
- PDF page 35: Job requests.
- PDF page 36: Job requests.
- PDF page 37: Job requests.
- PDF page 38: Job requests.
- PDF page 39: Job requests.
- PDF page 40: Modules.
- PDF page 41: Modules.
- PDF page 42: Modules.
- PDF page 43: Modules.
- PDF page 44: Modules.
- PDF page 45: Modules.
- PDF page 46: First jobs.
- PDF page 47: First jobs.
- PDF page 48: First jobs.
- PDF page 49: Consolidation.
- PDF page 50: Consolidation.
- PDF page 51: Consolidation.

12.3 Filesystems

Source: `filesystems.pdf`. SHA-256 prefix: `15d1f6b6ac8b0f9a`.

Every PDF page below points to the named teaching discussion; repeated reveals share a discussion.

- PDF page 1: Follow the named frame and its incremental reveals in the filesystem chapter.
- PDF page 2: Follow the named frame and its incremental reveals in the filesystem chapter.
- PDF page 3: Follow the named frame and its incremental reveals in the filesystem chapter.
- PDF page 4: Follow the named frame and its incremental reveals in the filesystem chapter.
- PDF page 5: Follow the named frame and its incremental reveals in the filesystem chapter.
- PDF page 6: Follow the named frame and its incremental reveals in the filesystem chapter.
- PDF page 7: Follow the named frame and its incremental reveals in the filesystem chapter.
- PDF page 8: Follow the named frame and its incremental reveals in the filesystem chapter.
- PDF page 9: Follow the named frame and its incremental reveals in the filesystem chapter.
- PDF page 10: Follow the named frame and its incremental reveals in the filesystem chapter.
- PDF page 11: Follow the named frame and its incremental reveals in the filesystem chapter.
- PDF page 12: Follow the named frame and its incremental reveals in the filesystem chapter.
- PDF page 13: Follow the named frame and its incremental reveals in the filesystem chapter.
- PDF page 14: Follow the named frame and its incremental reveals in the filesystem chapter.
- PDF page 15: Follow the named frame and its incremental reveals in the filesystem chapter.
- PDF page 16: Follow the named frame and its incremental reveals in the filesystem chapter.
- PDF page 17: Follow the named frame and its incremental reveals in the filesystem chapter.
- PDF page 18: Follow the named frame and its incremental reveals in the filesystem chapter.
- PDF page 19: Follow the named frame and its incremental reveals in the filesystem chapter.
- PDF page 20: Follow the named frame and its incremental reveals in the filesystem chapter.

12.4 QCG-PilotJob

Source: `qcg.pdf`. SHA-256 prefix: `9beeea66cd53f186`.

Every PDF page below points to the named teaching discussion; repeated reveals share a discussion.

- PDF page 1: Independent work.
- PDF page 2: Independent work.
- PDF page 3: Independent work.
- PDF page 4: Independent work.
- PDF page 5: Independent work.
- PDF page 6: Independent work.
- PDF page 7: Arrays and control.
- PDF page 8: Arrays and control.
- PDF page 9: Arrays and control.
- PDF page 10: Arrays and control.
- PDF page 11: Arrays and control.
- PDF page 12: Arrays and control.
- PDF page 13: Arrays and control.
- PDF page 14: Examples 1 and 2.
- PDF page 15: Examples 1 and 2.
- PDF page 16: Examples 1 and 2.
- PDF page 17: Examples 1 and 2.
- PDF page 18: Pilot and wrapper.
- PDF page 19: Pilot and wrapper.
- PDF page 20: Pilot and wrapper.
- PDF page 21: Pilot and wrapper.
- PDF page 22: Pilot and wrapper.
- PDF page 23: Pilot and wrapper.

- PDF page 24: Python manager.
- PDF page 25: Python manager.
- PDF page 26: Python manager.
- PDF page 27: Python manager.
- PDF page 28: Python manager.
- PDF page 29: Python manager.
- PDF page 30: Task descriptions.
- PDF page 31: Task descriptions.
- PDF page 32: Task descriptions.
- PDF page 33: Task descriptions.
- PDF page 34: Dependent means.
- PDF page 35: Dependent means.
- PDF page 36: Dependent means.
- PDF page 37: Close.
- PDF page 38: Close.

12.5 Workflow alternatives

Source: `workflows.pdf`. SHA-256 prefix: 80b9b5440e1085c6.

Every PDF page below points to the named teaching discussion; repeated reveals share a discussion.

- PDF page 1: Equivalent problems.
- PDF page 2: Equivalent problems.
- PDF page 3: Equivalent problems.
- PDF page 4: HyperQueue.
- PDF page 5: Snakemake.
- PDF page 6: FireWorks.
- PDF page 7: Measurements.
- PDF page 8: Measurements.
- PDF page 9: Measurements.
- PDF page 10: Choice.
- PDF page 11: Choice.

12.6 Performance analysis

Source: `performance.pdf`. SHA-256 prefix: c38235ca140eb94e.

Every PDF page below points to the named teaching discussion; repeated reveals share a discussion.

- PDF page 1: Measurement method.
- PDF page 2: Measurement method.
- PDF page 3: Measurement method.
- PDF page 4: Measurement method.
- PDF page 5: Measurement method.
- PDF page 6: Measurement method.
- PDF page 7: Measurement method.
- PDF page 8: Bottleneck classes.
- PDF page 9: Bottleneck classes.
- PDF page 10: Bottleneck classes.
- PDF page 11: Bottleneck classes.
- PDF page 12: Bottleneck classes.
- PDF page 13: Bottleneck classes.

- PDF page 14: Bottleneck classes.
- PDF page 15: Bottleneck classes.
- PDF page 16: Scaling.
- PDF page 17: Scaling.
- PDF page 18: Scaling.
- PDF page 19: Optimization.
- PDF page 20: Optimization.
- PDF page 21: Roofline reveals.
- PDF page 22: Roofline reveals.
- PDF page 23: Roofline reveals.
- PDF page 24: Roofline reveals.
- PDF page 25: Roofline reveals.
- PDF page 26: Roofline reveals.
- PDF page 27: Roofline reveals.
- PDF page 28: Roofline reveals.
- PDF page 29: FLOP counts.
- PDF page 30: FLOP counts.
- PDF page 31: FLOP counts.
- PDF page 32: FLOP counts.
- PDF page 33: FLOP counts.
- PDF page 34: FLOP counts.
- PDF page 35: Energy.
- PDF page 36: Energy.
- PDF page 37: Slurm evidence.
- PDF page 38: Slurm evidence.
- PDF page 39: Slurm evidence.
- PDF page 40: Slurm evidence.
- PDF page 41: Slurm evidence.

12.7 Data movement

Source: `data-movement.pdf`. SHA-256 prefix: `b7e2b518a7f0894d`.

Every PDF page below points to the named teaching discussion; repeated reveals share a discussion.

- PDF page 1: Copy costs and paths.
- PDF page 2: Copy costs and paths.
- PDF page 3: Copy costs and paths.
- PDF page 4: Pack and verify.
- PDF page 5: rsync.
- PDF page 6: Parallel copies.
- PDF page 7: Parallel copies.
- PDF page 8: Parallel copies.
- PDF page 9: Outside endpoints.
- PDF page 10: Outside endpoints.
- PDF page 11: Staging.
- PDF page 12: Staging.
- PDF page 13: Exercises.
- PDF page 14: Exercises.
- PDF page 15: Exercises.
- PDF page 16: Exercises.

- PDF page 17: Exercises.
- PDF page 18: Exercises.