UNIX IV: Running analyses on the cluster

So far you have run programs on the login node or in a short interactive `srun` session. Real analyses (BLAST of whole proteomes, aligning reads for 20 samples, assembling a genome) need more CPUs, more memory and more time than that, and you don't want to sit and watch them. On the HPCC you hand this work to the **SLURM** scheduler as **batch jobs**: a script that says what to run and what resources it needs. SLURM finds a compute node, runs it, and saves the output in a log file.

This lecture covers:

- how the cluster is organized: login and compute nodes, partitions, CPUs, memory, time limits and fair share
- writing an `sbatch` job script, submitting it, watching it, cancelling it, reading its log file
- asking for the right number of CPUs (`-c` and `$SLURM_CPUS_PER_TASK`), memory and time, and checking what a finished job really used (`sacct`, `seff`)
- the common reasons a job fails or never starts
- **job arrays**: one script that runs the same command on every sample, with a complete BLAST example, and job dependencies
- where to put files: home, bigdata and scratch
- keeping a session alive with `tmux`
- installing software yourself with conda, and containers
- finding and processing many files with `find`, `xargs` and `parallel`
- where to go next: workflow managers (Snakemake, Nextflow)

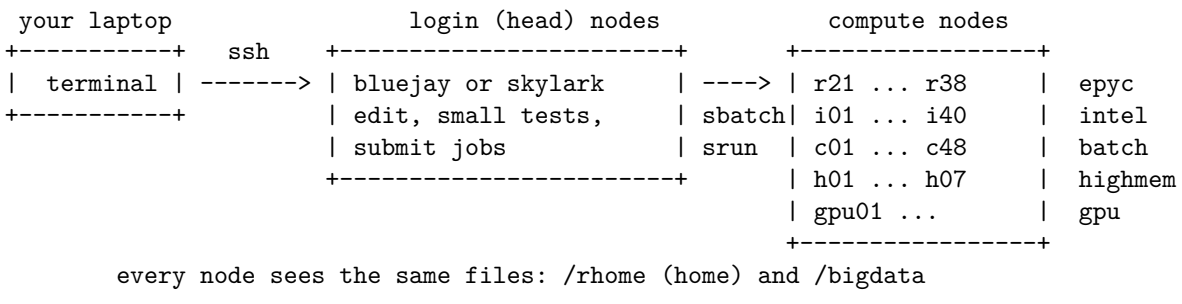
Before you start you should be comfortable with:

- logging in, paths and the login node vs compute nodes (UNIX I)
- modules, `srun`, redirection, wildcards, `grep` and your disk space (UNIX II)
- variables, quoting, `if`, `for` loops, scripts with arguments, `set -euo pipefail`, `cut`, `sed` and `awk` (UNIX III lab)
- running BLAST on the command line (Basics and BLAST)

The HPCC's own documentation is the final word on partition names, limits and commands, and it changes from time to time. The facts here were checked against it in September 2026; see the Sources at the end.

How the cluster is organized

Login nodes and compute nodes



- `ssh cluster.hpcc.ucr.edu` puts you on one of the **login nodes** (the HPCC calls them head nodes: currently `bluejay` and `skylark`). They are shared by everyone who is logged in. They are for editing files, moving data, small tests (the HPCC's rule is under 100% of one CPU and under 1 GB of memory) and **submitting jobs**. Heavy programs on a login node are slowed down or killed automatically.
- The **compute nodes** do the work. Each one is a separate computer with many CPU cores (32 to 128) and a lot of memory (128 GB to 3 TB). You only get onto one by asking SLURM, with `srun` (interactive) or `sbatch` (batch).
- A **core** (SLURM calls it a CPU) runs one thread of a program at a time. A program that can use 8 threads (`blastp -num_threads 8`, `bwa mem -t 8`) needs 8 cores to go 8 times faster. A program that

isn't multithreaded uses one core no matter how many you ask for.
- **Memory** (RAM) is what a program holds in its head while it runs: a genome index, a BLAST database, a table loaded into pandas. If a job uses more memory than it asked for, SLURM kills it.
- All nodes mount the same storage, so a script you write in `~/bigdata` on the login node is right there when your job runs on a compute node.

Partitions

Compute nodes are grouped into **partitions** (queues). You choose one with `-p NAME`. The public partitions on the HPCC are:

Partition	Use it for	Max time	Notes
`short`	tests and anything under 2 hours	2 hours	mix of nodes; often starts fastest
`epyc`	general CPU work, multithreaded programs	30 days	AMD nodes, 64 cores, 1 TB each
`intel`	general CPU work	30 days	older Intel nodes, 32 cores, 512 GB
`batch`	general CPU work	30 days	oldest AMD nodes, 64 cores, 512 GB
`highmem`	jobs that need a lot of memory	30 days	you **must** ask for at least 100 GB
`highclock`	programs that can't use many cores	30 days	fast single cores
`gpu`, `short_gpu`	programs that use GPUs	7 days / 2 hours	request GPUs with `--gres=gpu:1`

If you don't give `-p`, your job goes to a default partition (the HPCC documentation names `epyc` in one place and `intel` in another), so **always give `-p`**. Labs that bought their own nodes also have a private partition named after the lab.

Limits and fair share

The cluster is shared by hundreds of users, so there are limits on what you can use **at the same time**. The main ones (from the HPCC queue policies page):

- per user: 384 cores and 1 TB of memory running at once on the CPU partitions (`epyc`, `intel` and `batch` together; `short` has the same limit); 32 cores on `highmem`
- per lab (group): 768 cores across all partitions - and during this course the whole class shares the `gen220` group
- at most 64 GB of memory per core (ask for more and SLURM raises your core count to match), 5000 jobs queued or running per user, and up to 2500 tasks in one job array

Jobs over a limit don't fail; they wait in the queue until your earlier jobs finish. See your current limits and usage with the HPCC commands:

```
slurm_limits      # your limits on each partition
group_cpus        # cores your group is using right now
sinfo -s          # partitions and how many nodes are busy/idle
```

When the cluster is busy, the order in which waiting jobs start is set by a **priority**. It is mostly your **fair share** score: users and groups that have used a lot recently get lower priority, those who haven't get higher, and a job's priority also grows the longer it waits. Small jobs may be slipped in ahead of you (backfill) if they will finish before your job could start anyway - which is one reason to ask for a realistic time limit rather than the maximum.

```
sshare -u $USER      # your fair share score (0 = used a lot, 1 = used little)
sprio -u $USER       # priority of your pending jobs
squeue -u $USER --start   # SLURM's rough guess of when they will start
```

Batch jobs with sbatch

Your first job script

A job script is an ordinary bash script (see UNIX III) with some special comment lines at the top. Save this as `hello_job.sh`:

```
#!/bin/bash -l
#SBATCH -p short
#SBATCH -c 1
#SBATCH --mem 1G
#SBATCH --time 0:10:00
#SBATCH -J hello

echo "Hello from $(hostname)"
echo "Job $SLURM_JOB_ID started in $(pwd) at $(date)"
sleep 30
echo "Done at $(date)"
```

- `#!/bin/bash -l` runs the script with bash as a **login** shell (`-l`), so the same setup you get when you log in - including the `module` command - is there in the job.
- Lines starting with `#SBATCH` are read by `sbatch` (to bash they are just comments). They must come **before the first command** in the script; `#SBATCH` lines after that are ignored.
- Everything after them runs on the compute node.

Submit it, and SLURM answers with the **job ID**:

```
sbatch hello_job.sh
```

```
Submitted batch job 4321987
```

(All cluster output in this lecture is example output: your job IDs, node names and times will be different.) Watch it with `squeue`:

```
squeue -u $USER
```

```
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           4321987     short    hello  jsmith0 PD       0:00      1 (Priority)
```

`ST` is the state: `PD` pending (waiting; the reason is in the last column), `R` running, `CG` completing. When the job is done it disappears from `squeue`. Its output (everything printed to the screen) is in `slurm-JOBID.out` in the directory you ran `sbatch` from:

```
cat slurm-4321987.out
```

```
Hello from r22
Job 4321987 started in /rhome/jsmith0/bigdata/unix4 at Thu Oct 29 10:02:13 PDT 2026
Done at Thu Oct 29 10:02:43 PDT 2026
```

Notice the job **starts in the directory you submitted from**, so relative paths in the script are relative to that folder. `cd` there before you run `sbatch`.

To stop a job (pending or running):

```
scancel 4321987          # one job
scancel -u $USER         # ALL of your jobs - careful
```

The #SBATCH options you will use

Option	Long form	Meaning
`-p short`	`--partition=short`	which partition
`-c 8`	`--cpus-per-task=8`	CPU cores for your program's threads
`--mem 16G`		memory for the whole job (units `K`, `M`, `G`, `T`; no unit = MB)
`-t 2:00:00`	`--time=2:00:00`	time limit: `MM`, `HH:MM:SS` or `D-HH:MM:SS`
`-J blast`	`--job-name=blast`	a name to recognize it in `squeue`
`-o logs/blast.%j.log`	`--output=...`	where screen output (STDOUT) goes
`-e logs/blast.%j.err`	`--error=...`	STDERR separately (default: same file as `-o`)
`--mail-type=END,FAIL`		email when it ends or fails (`BEGIN`, `ALL`, `NONE` also work)
`--mail-user=NETID@ucr.edu`		where to send it
`-N 1`	`--nodes=1`	number of computers (always 1 in this course)
`-n 1`	`--ntasks=1`	number of separate tasks (processes); 1 unless you run MPI
`-a 1-10`	`--array=1-10`	a job array (below)

In `-o` and `-e` file names, `%j` becomes the job ID, `%x` the job name, and for arrays `%A` the array's job ID and `%a` the task number. **The folder must already exist**: if `logs/` isn't there, SLURM can't write the log and the job fails without any output, so `mkdir -p logs` first.

If you don't ask, the HPCC gives you 1 core, 1 GB of memory and (outside `short`) a 7-day time limit.

Options can also be given on the `sbatch` command line, where they **override** the `#SBATCH` lines. This is handy for a quick change without editing the script:

```
sbatch -p epyc --time 12:00:00 --mem 32G run_blast.sh
```

CPUs: -c, not -n, for multithreaded programs

Almost every bioinformatics program you will run (BLAST, bwa, samtools, HISAT2, kallisto, SPAdes, IQ-TREE) is **one program that runs several threads**. For those, ask for one task with several CPUs, and tell the program to use exactly that many:

```
#SBATCH -N 1 -n 1 -c 8        # 1 node, 1 task, 8 CPUs for that task

CPU=${SLURM_CPUS_PER_TASK:-1}
blastp -query my.pep -db swissprot -num_threads "$CPU" -out my.blastp
```

- `-c` / `--cpus-per-task` gives your one program N cores **on the same node**, which is what threads need. (`-N 1 -n 1` are the defaults, written out here for clarity.)
- `-n` / `--ntasks` asks for N separate *tasks*, meant for MPI programs that run many cooperating copies of themselves, possibly spread across several nodes. A multithreaded program started once uses only the CPUs of one task. You will see older scripts (including some in this course) that use `-n 4` or `-N 1 -n 16` for BLAST or bwa; with `-N 1` this usually works, but `-c` is the correct request and it's what the HPCC documentation recommends for threaded programs.
- SLURM sets `$SLURM_CPUS_PER_TASK` to the `-c` value inside the job. Use it instead of typing the number twice: if you later change `-c 8` to `-c 16` (or override it with `sbatch -c 16`), the program follows automatically. `${SLURM_CPUS_PER_TASK:-1}` means "use 1 if the variable isn't set", so the same script still

works when you test it with plain `bash` outside of SLURM. (Some older scripts use `$SLURM_CPUS_ON_NODE`, which is the number of CPUs allocated on the node; for a `-c` job it is normally the same number, but `$SLURM_CPUS_PER_TASK` says exactly what you asked for.)
- Asking for more cores than the program uses doesn't make it faster; it just makes you wait longer in the queue and takes cores away from your classmates.

A template job script

Copy this and change the parts in CAPITALS. It prints some information at the start and end of the log, which makes it much easier to work out what happened later.

```
#!/bin/bash -l
#SBATCH -p short                # partition: short (<2 h), epyc, intel, batch, highmem
#SBATCH -N 1 -n 1 -c 4          # 1 node, 1 task, 4 CPUs
#SBATCH --mem 8G                # memory for the job
#SBATCH --time 2:00:00          # time limit (HH:MM:SS or D-HH:MM:SS)
#SBATCH -J JOBNAME              # short name shown in squeue
#SBATCH -o logs/%x.%j.log       # log file: logs/JOBNAME.JOBID.log (make logs/ first!)
#SBATCH --mail-type=END,FAIL    # optional: email when it finishes or fails
#SBATCH --mail-user=NETID@ucr.edu

# load software first (module/conda setup scripts don't like 'set -u')
module load MODULE_NAME

set -euo pipefail               # stop at the first error (see UNIX III)

CPU=${SLURM_CPUS_PER_TASK:-1}
echo "Job ${SLURM_JOB_ID:-none} on $(hostname) with $CPU CPUs, started $(date)"

# ---- your commands here, using $CPU for the number of threads ----
PROGRAM --threads "$CPU" INPUT > OUTPUT

echo "Finished $(date)"
```

The `set -euo pipefail` line (from UNIX III) is especially important in job scripts: without it, if step 1 fails the script carries on, step 2 runs on a missing or half-written file, and you find out hours later. It comes *after* the `module load` lines because the scripts behind `module` and `conda activate` can trip over `set -u`.

Test before you submit. Run the commands on a tiny input in an interactive session (or with the job on `-p short`) first. A typo found in 30 seconds is much better than one found after 6 hours in the queue. You can also check a script for common bash mistakes with `shellcheck myjob.sh` if it is installed.

Reading the log files

The log is the first place to look when something goes wrong. It contains everything the script and its programs printed, including error messages. Useful commands:

```
ls -lt logs | head              # the newest logs first
tail logs/blast.4321990.log     # the end of a log
tail -f logs/blast.4321990.log # follow a log while the job runs (Ctrl-C to stop watching)
grep -il error logs/*.log       # logs that mention an error (-i any case, -l names only)
```

How much did my job use? sacct and seff

`squeue` only shows jobs that are waiting or running. For finished jobs use `sacct` (SLURM's accounting records):

```
sacct -u $USER -S 2026-10-29        # all your jobs since a date
sacct -j 4321990 \
```

```
  --format=JobID,JobName%15,State,Elapsed,AllocCPUS,TotalCPU,ReqMem,MaxRSS,ExitCode
```

JobID	JobName	State	Elapsed	AllocCPUS	TotalCPU	ReqMem	MaxRSS	ExitCode
4321990	blast	COMPLETED	00:41:07	8	05:02:11	16G		0:0
4321990.bat+	batch	COMPLETED	00:41:07	8	05:02:11		3.10G	0:0
4321990.ext+	extern	COMPLETED	00:41:07	8	00:00:00		0	0:0

`Elapsed` is the wall-clock time, `TotalCPU` the CPU time summed over all cores, and `MaxRSS` (on the `.batch` line) the most memory the script used. The `State` column is `COMPLETED`, `FAILED` (a command exited with an error), `OUT_OF_MEMORY`, `TIMEOUT` or `CANCELLED`.

`seff` summarizes the same information as percentages of what you asked for (use it once the job has finished):

```
seff 4321990
```

```
Job ID: 4321990
Cluster: hpcc
User/Group: jsmith0/gen220
State: COMPLETED (exit code 0)
Cores: 8
CPU Utilized: 05:02:11
CPU Efficiency: 91.89% of 05:28:56 core-walltime
Job Wall-clock time: 00:41:07
Memory Utilized: 3.10 GB
Memory Efficiency: 19.38% of 16.00 GB
```

This job used its 8 cores well (92%) but only 3.1 GB of the 16 GB it asked for, so next time `--mem 4G` or `--mem 6G` would be plenty. If CPU efficiency is low (say 15% of 8 cores), the program probably used only 1 thread: check that you passed `$CPU` to it.

Choosing resources

Start from a guess, run **one** sample, look at `seff`, then set the resources for the rest with some headroom (about 20-50% more memory and time than the test used, because other samples will be a bit bigger).

Kind of job	CPUs (`-c`)	Memory	Time	Partition
a script with `grep`/`awk`/Python on text files	1	1-2 G	minutes	`short`
installing a conda environment	4	10 G	< 1 h	`short`
BLAST one bacterial proteome against another	4	2-4 G	minutes	`short`
BLAST thousands of proteins against SwissProt or nr	8-16	16-32 G	hours to days	`epyc`
`bwa mem` + `samtools sort`, one bacterial or fungal sample	8	8-16 G	< 2 h	`short` or `epyc`
read counting with kallisto or featureCounts	4-8	4-16 G	< 1 h	`short`
assembling a fungal genome	16-32	64-250 G	1-2 days	`epyc`, `highmem` if > 1 TB

These are only starting points. The program's manual often says how memory grows with input size (e.g. `samtools sort` uses about `-m` (default 768 MB) per thread).

When things go wrong

What you see	What it means	What to do
no log file at all, job gone	the `-o` folder doesn't exist	`mkdir -p logs`, resubmit
`sbatch: error: Batch script contains DOS line breaks`	the script was written on Windows	`dos2unix script.sh`, or write it on the cluster
`module: command not found`	first line isn't `#!/bin/bash -l`	fix the first line
`blastp: command not found`	the module isn't loaded *in the script*	add `module load ...` to the script
`No such file or directory`	a relative path from the wrong folder	submit from the right folder or use full paths
`slurmstepd: error: Detected 1 oom_kill event...`, state `OUT_OF_MEMORY`	used more memory than `--mem`	ask for more (check `seff`), or a smaller input
`*** JOB 4321990 ON r22 CANCELLED AT ... DUE TO TIME LIMIT ***`, state `TIMEOUT`	ran past `--time`	ask for more time or more CPUs; make the script skip finished work
`Invalid partition name specified`	typo in `-p`	check `sinfo -s`
pending with `(Resources)` or `(Priority)`	normal: waiting for free nodes or for higher-priority jobs	wait; smaller/shorter requests start sooner
pending with `(AssocGrpCpuLimit)` or `(AssocGrpMemLimit)`	you or your group are at a core/memory limit	wait for your other jobs to finish
pending with `(QOSMaxWallDurationPerJobLimit)`	time asked for is more than the partition allows (e.g. 3 h on `short`)	`scancel` and resubmit with a shorter time or another partition
pending with `(QOSMinGRES)` on `highmem`	`highmem` needs `--mem` of at least 100 G	use `epyc`, or ask for >= 100 G

`scontrol show job JOBID` shows everything SLURM knows about a pending or running job, including the reason it is waiting.

Interactive jobs

For testing commands, looking at data, or anything where you type as you go, ask for an interactive shell on a compute node (introduced in UNIX II). It takes the same resource options as `sbatch`:

```
srun -p short -c 4 --mem 8G --time 2:00:00 --pty bash -l
```

When the prompt changes from the login node's name to a compute node's name (e.g. `r22`), you are on the compute node. `$SLURM_CPUS_PER_TASK` is set here too. Type `exit` when you're done so the cores go back to the pool - the session also ends if you close your laptop or lose your connection, unless you start it inside `tmux` (see below). For anything that takes more than an hour or two, write a batch script instead.

Job arrays: the same analysis on many samples

The idea

Very often you want to run the same commands on many inputs: align 24 FASTQ files, BLAST 10 proteomes, call variants on 50 strains. You could write a `for` loop in one job, but then the samples run one after another. You could write 24 job scripts, but that is tedious and error-prone. A **job array** submits one script many times;

each copy (a **task**) gets a different number in `$SLURM_ARRAY_TASK_ID`, and uses it to pick its sample. The tasks run at the same time on as many nodes as are free.

```
#!/bin/bash -l
#SBATCH -p short -c 1 --mem 1G --time 0:05:00
#SBATCH -J arraydemo
#SBATCH -o logs/arraydemo.%A_%a.log
#SBATCH --array=1-5

echo "I am task $SLURM_ARRAY_TASK_ID of array job $SLURM_ARRAY_JOB_ID on $(hostname)"
```

```
mkdir -p logs
sbatch array_demo.sh
squeue -u $USER
```

```
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
       4322001_[3-5]     short arraydem  jsmith0 PD       0:00      1 (Resources)
         4322001_1     short arraydem  jsmith0  R       0:02      1 r21
         4322001_2     short arraydem  jsmith0  R       0:02      1 r25
```

Each task has its own log, `logs/arraydemo.4322001_1.log` to `..._5.log`, because we used `%A` (array job ID) and `%a` (task number) in the name. Without `%a` all five tasks would write to the same file.

Useful forms of `--array` (all from the `sbatch` manual):

Option	Tasks
`--array=1-24`	1, 2, . . . , 24
`--array=1,5,9`	only 1, 5 and 9 (e.g. to rerun the ones that failed)
`--array=1-24%4`	1 to 24, but at most 4 running at a time
`--array=0-99:10`	0, 10, 20, . . . , 90

The `%4` limit is polite when each task reads a lot of data at once or when your class is sharing the group's core limit. `scancel 4322001` cancels the whole array; `scancel 4322001_3` cancels one task.

A sample sheet: picking the Nth line

The usual way to connect task numbers to samples is a **sample sheet**: a text file with one sample per line. Task N reads line N. Say `samples.txt` has:

```
SRR5000101
SRR5000102
SRR5000103
```

In the job script, `sed -n "${N}p"` prints only line N (`-n` = don't print every line, `Np` = print line N):

```
N=${SLURM_ARRAY_TASK_ID}
SAMPLE=$(sed -n "${N}p" samples.txt)
echo "task $N works on $SAMPLE"
```

You can try this without SLURM by setting the variable yourself:

```
SLURM_ARRAY_TASK_ID=2
N=${SLURM_ARRAY_TASK_ID}
SAMPLE=$(sed -n "${N}p" samples.txt)
echo "task $N works on $SAMPLE"
```

```
task 2 works on SRR5000102
```

`awk` works too, and is handy when the sheet has several tab-separated columns or a header line:

```
SAMPLE=$(awk -v n="$N" 'NR == n' samples.txt)     # line N, like sed -n "${N}p"
# skip a header line, and print column 2 of a tab-separated sheet
STRAIN=$(awk -F'\t' -v n="$N" 'NR == n + 1 {print $2}' strains.tsv)
```

Make the array size match the sheet. Rather than counting by hand, give it on the command line (which overrides any `#SBATCH --array` line):

```
sbatch --array="1-$(wc -l < samples.txt)" my_array_job.sh
```

Arrays on the HPCC can have up to 2500 tasks. If you have more samples than that (or thousands of tiny ones), have each task process a block of lines instead, or see Working with many files.

Worked example: all-vs-all BLAST of three bacterial proteomes

The gene networks workshop needs every proteome BLASTed against every other one: 3 x 3 = 9 searches. There it is one job with a double `for` loop. Here we make each search its own array task, so all 9 can run at the same time. The same pattern works for 9 or 900 searches.

The folder will look like this:

```
blast_array/
|-- 00_setup.sh          download, makeblastdb, write pairs.tsv (run once)
|-- 01_blast_array.sh    the array job: one BLAST per line of pairs.tsv
|-- 02_summarize.sh      runs after the array, counts hits
|-- pairs.tsv            the sample sheet
|-- pep/                 proteomes and BLAST databases
|-- results/             one BLAST table per task
`-- logs/                one log per task
```

Step 0: set up. Make the folder in your bigdata space, and save this as `00_setup.sh`:

```
#!/bin/bash -l
# Run once, interactively (srun) or with sbatch: download, index, make the sample sheet
module load ncbi-blast

set -euo pipefail

GENOMES="E_coli_K12 E_coli_O157_H7 S_enterica"
URL=https://github.com/biodataprog/GEN220_data/raw/main/genome
mkdir -p pep results logs

for name in $GENOMES
do
  if [ ! -s "pep/$name.pep" ]; then
    curl -sL "$URL/$name.pep.gz" | gunzip -c > "pep/$name.pep"
  fi
  if [ ! -s "pep/$name.pep.pin" ]; then
    makeblastdb -in "pep/$name.pep" -dbtype prot > /dev/null
  fi
done

# the sample sheet: one line per search, QUERY<tab>DATABASE
for A in $GENOMES
do
  for B in $GENOMES
  do
    printf '%s\t%s\n' "$A" "$B"
  done
done > pairs.tsv
```

```
wc -l pairs.tsv
```

```
mkdir -p ~/bigdata/gen220/blast_array
cd ~/bigdata/gen220/blast_array
# save the three scripts here, then:
srun -p short -c 1 --mem 2G --time 0:30:00 --pty bash -l
bash 00_setup.sh
head -n 4 pairs.tsv
exit
```

```
9 pairs.tsv
E_coli_K12  E_coli_K12
E_coli_K12  E_coli_O157_H7
E_coli_K12  S_enterica
E_coli_O157_H7  E_coli_K12
```

Step 1: the array job, `01_blast_array.sh`. Task N reads line N of `pairs.tsv`, splits it into the query and database names with `cut`, and runs one `blastp`:

```
#!/bin/bash -l
#SBATCH -p short                # quick jobs (2 hour limit)
#SBATCH -c 4                    # 4 CPUs for each task
#SBATCH --mem 4G                # memory for each task
#SBATCH --time 1:00:00          # time for each task
#SBATCH -J blastpairs
#SBATCH -o logs/blastpairs.%A_%a.log
#SBATCH --array=1-9             # one task per line of pairs.tsv

module load ncbi-blast

set -euo pipefail

CPU=${SLURM_CPUS_PER_TASK:-1}
# the line number: from SLURM, or from the command line when testing
N=${SLURM_ARRAY_TASK_ID:-${1:-}}
if [ -z "$N" ]; then
  echo "Usage: sbatch $0   or, to test one line:  bash $0 LINE_NUMBER" >&2
  exit 1
fi

LINE=$(sed -n "${N}p" pairs.tsv)
if [ -z "$LINE" ]; then
  echo "pairs.tsv has no line $N" >&2
  exit 1
fi
QUERY=$(echo "$LINE" | cut -f1)
DB=$(echo "$LINE" | cut -f2)
OUT=results/$QUERY-vs-$DB.BLASTP.tab

echo "Task $N: $QUERY vs $DB, $CPU CPUs, on $(hostname), $(date)"
if [ -s "$OUT" ]; then
  echo "$OUT already exists, nothing to do"
  exit 0
fi

blastp -query "pep/$QUERY.pep" -db "pep/$DB.pep" -outfmt 6 -evalue 1e-5 \
```

```
  -num_threads "$CPU" -out "$OUT.tmp"
mv "$OUT.tmp" "$OUT"      # only a finished search gets the real name

echo "Finished $(date): $(wc -l < "$OUT") hits in $OUT"
```

Things to notice:

- `#SBATCH` resources (`-c`, `--mem`, `--time`) are **per task**: each of the 9 tasks gets 4 CPUs and 4 GB.
- `N=${SLURM_ARRAY_TASK_ID:-${1:-}}` takes the task number from SLURM, or from the first command line argument if SLURM didn't set it. So you can **test one task** without submitting anything: `bash 01_blast_array.sh 1` (in an `srun` session).
- The script checks that the line exists, and skips searches whose output already exists. If a few tasks fail or time out, just resubmit those task numbers.
- BLAST writes to `$OUT.tmp`, which is renamed only when `blastp` finishes. A task that is killed half-way leaves a `.tmp` file, not a truncated table that looks finished.

Test one task, then submit the array:

```
srun -p short -c 4 --mem 4G --time 0:30:00 --pty bash -l
bash 01_blast_array.sh 1
exit

sbatch 01_blast_array.sh
```

```
Task 1: E_coli_K12 vs E_coli_K12, 4 CPUs, on r23, Thu Oct 29 10:20:05 PDT 2026
Finished Thu Oct 29 10:20:24 PDT 2026: 21909 hits in results/E_coli_K12-vs-E_coli_K12.BLASTP.tab
Submitted batch job 4322010
```

(Each search took 15-80 seconds with 4 threads when we tested these scripts with BLAST+ 2.17.) If tasks 3 and 7 failed, look at their logs, fix the problem, and rerun just those:

```
cat logs/blastpairs.4322010_3.log
sbatch --array=3,7 01_blast_array.sh
```

Step 2: a job that waits for the array. `02_summarize.sh` counts the hits in each table:

```
#!/bin/bash -l
#SBATCH -p short -c 1 --mem 1G --time 0:10:00
#SBATCH -J blastsummary
#SBATCH -o logs/blastsummary.%j.log

set -euo pipefail

printf 'search\thits\tqueries_with_hit\n' > summary.tsv
for f in results/*.BLASTP.tab
do
  name=$(basename "$f" .BLASTP.tab)
  hits=$(wc -l < "$f")
  queries=$(cut -f1 "$f" | sort -u | wc -l)
  printf '%s\t%s\t%s\n' "$name" "$hits" "$queries" >> summary.tsv
done
column -t summary.tsv
```

You could wait for the array to finish and then `sbatch 02_summarize.sh`, but SLURM can do the waiting for you with a **dependency**. `sbatch --parsable` prints just the job ID, so we can save it in a variable:

```
JOBID=$(sbatch --parsable 01_blast_array.sh)
echo "array job is $JOBID"
sbatch --dependency=afterok:$JOBID 02_summarize.sh
```

The summary job stays pending with reason `(Dependency)` until **all** tasks of the array have finished successfully (`afterok`); then it runs. If any task fails, it can never start (reason `DependencyNeverSatisfied`) and you should `scancel` it, fix the problem, and resubmit. Use `afterany` instead to run whether or not the earlier job succeeded. Chains of dependencies are how you build a small pipeline: index, then align every sample (an array), then call variants on all of them together - as in the variant calling lecture.

When it's done, `logs/blastsummary.JOBID.log` has:

```
search                              hits   queries_with_hit
E_coli_K12-vs-E_coli_K12            21909  4213
E_coli_K12-vs-E_coli_O157_H7        21583  3913
E_coli_K12-vs-S_enterica            19213  3635
E_coli_O157_H7-vs-E_coli_K12        21679  4139
E_coli_O157_H7-vs-E_coli_O157_H7    28914  5477
E_coli_O157_H7-vs-S_enterica        20210  4043
S_enterica-vs-E_coli_K12            19335  3706
S_enterica-vs-E_coli_O157_H7        20205  3796
S_enterica-vs-S_enterica            20374  4603
```

Every protein finds itself in the self-searches (4213 K-12 proteins, 5477 O157:H7, 4603 *Salmonella*), and 3913 of the 4213 K-12 proteins have a hit in O157:H7. These tables are the input for the networks workshop.

Where to put files: storage on the cluster

Location	Size	Backed up?	Use it for
`/rhome/USERNAME` (`~`)	50 GB per user	daily snapshots, kept 1 week	scripts, config files, small things
`/bigdata/gen220/USERNAME` (`~/bigdata`)	shared lab quota	weekly snapshots, kept 1 month	data, results, conda environments
`/bigdata/gen220/shared`	shared lab quota	same	files shared with the class
`$SCRATCH` (on `/scratch`)	the node's local SSD	no - **deleted when the job ends**	fast temporary files during a job
`/tmp`	whatever is free on that node	no	small temporary files
`/dev/shm`	counts against your job's memory	no	very fast temporary files (advanced)

Check how much you are using with `check_quota home` and `check_quota bigdata`, or on https://dashboard.hpcc.ucr.edu (see UNIX II). Programs fail in confusing ways when your home folder is full (editors can't save, conda and even logging in can break), so keep data and conda environments in bigdata. And bigdata space is paid for by the lab: delete intermediate files you don't need and compress the rest.

Scratch. Each job gets its own folder on the compute node's local disk, and its path is in `$SCRATCH`. It is faster than `/bigdata` for programs that write lots of temporary files, and it is **removed automatically when the job ends** - so copy anything you want to keep back before the script finishes. Many programs take a temporary directory option; point it there:

```
samtools sort -@ "$CPU" -T "$SCRATCH/sorttmp" -o sample.sorted.bam sample.bam
sort -T "$SCRATCH" -k1,1 big_table.tsv > big_table.sorted.tsv
```

or do the whole job in scratch:

```
cp reads_1.fq.gz reads_2.fq.gz "$SCRATCH"/
cd "$SCRATCH"
# ... run the analysis here ...
cp results.tsv "$SLURM_SUBMIT_DIR"/     # the folder you ran sbatch from
```

Accidentally deleted something? Home and bigdata have snapshots: read-only copies from the recent past in `/rhome/.snapshots/` and `/bigdata/.snapshots/` (one folder per snapshot, named with a number; higher is newer). Find your file there and copy it back.

Cleaning up. Find what is taking up space:

```
cd ~/bigdata
du -sh * | sort -h                  # size of each folder, biggest last
find . -type f -size +1G            # files over 1 GB
find . -name '*.sam'                # uncompressed alignments: convert to BAM or delete
```

Keeping sessions alive: tmux

If your laptop sleeps or the Wi-Fi drops, your ssh connection closes and everything running in it - including an `srun` session - is stopped. `tmux` (terminal multiplexer) runs your shell inside a session that stays alive on the login node when you disconnect, and lets you reattach to it later.

```
hostname                # remember which login node you are on, e.g. bluejay
tmux new -s blast       # start a session named "blast"
# ... work as usual, e.g. start an interactive job:
srun -p short -c 4 --mem 8G --time 2:00:00 --pty bash -l
```

- **Detach** (leave it running): press `Ctrl-b`, let go, then press `d`.
- `tmux ls` lists your sessions; `tmux attach -t blast` reattaches.
- Inside tmux, `Ctrl-b c` opens another window and `Ctrl-b n` switches to the next one.
- Type `exit` (in every window) or run `tmux kill-session -t blast` when you're done.

```
blast: 1 windows (created Thu Oct 29 10:31:52 2026)
```

Rules and caveats:

- A tmux session lives on the **login node where you started it**. `cluster.hpcc.ucr.edu` may send your next login to the other login node, where `tmux ls` shows nothing; run `hostname`, and if needed `ssh bluejay` (or `skylark`) from the cluster to get to the right one.
- Start tmux on the login node, not inside a job on a compute node, as the HPCC recommends; from inside tmux you can `srun` onto a compute node as above.
- tmux doesn't change the login node rules: heavy work still goes in `srun` or `sbatch`. An `srun` session inside tmux still ends at its `--time` limit.
- Sessions are lost when a login node is rebooted (e.g. during maintenance). Anything that takes hours belongs in a batch job, which survives your logout without tmux.
- The HPCC's terminal IDE guide shows a customized tmux setup that uses `Ctrl-a` instead of `Ctrl-b`.

Software environments: modules, conda and containers

Modules first

The HPCC installs hundreds of programs as modules (UNIX II). If `module avail NAME` finds what you need, use it: it is already set up and tested, and it doesn't use your disk space. Put `module load` lines **inside** your job scripts, with a version number when you want the results to be reproducible (`module load samtools/1.19`); the default version changes when new ones are installed.

conda: installing software yourself

When a program isn't installed (or you need a different version), install it with **conda**. The Bioconda channel has more than 10,000 bioinformatics packages, and conda-forge has general software (Python and R packages, compilers, utilities).

On the HPCC conda comes from the `miniconda3` module, which is loaded for you when you log in and sets conda up (no need for `conda init`). Before you create any environments, make conda put them in bigdata, not your 50 GB home: environments easily add up to tens of GB. Create a file `~/.condarc` containing:

```
channels:
  - conda-forge
  - bioconda
channel_priority: strict
pkgs_dirs:
  - ~/bigdata/.conda/pkgs
envs_dirs:
  - ~/bigdata/.conda/envs
auto_activate_base: false
```

(The HPCC's example `.condarc` lists the `defaults` channel; the channel lines above are the setup the Bioconda project recommends.) Then install on a compute node, not the login node - the HPCC suggests an interactive job like this:

```
srun -p short -c 4 --mem 10G --time 2:00:00 --pty bash -l
conda create -n gen220 python=3.12 pandas biopython seqkit samtools
conda activate gen220
seqkit version
conda deactivate
```

- `conda activate NAME` switches to the environment: its programs come first in your `$PATH`. `conda deactivate` leaves it.
- `conda install -n gen220 PACKAGE` adds a package later; `conda env list` lists your environments; `conda env remove -n NAME` deletes one.
- Keep one environment per project (or per tool if tools conflict), not one giant environment for everything - big environments get slow to solve and break when updated.
- `mamba` is a faster drop-in replacement for the `conda` command. Recent conda versions (23.10 and later) use the same fast solver by default, so plain `conda` is fine; if `mamba` is available you can type it anywhere you would type `conda`.
- Conda keeps a copy of every package it downloads. `du -sh ~/bigdata/.conda` shows the space used and `conda clean -a` removes the cached downloads.

Using an environment in a job script:

```
#!/bin/bash -l
#SBATCH -p short -c 4 --mem 4G --time 1:00:00
#SBATCH -o logs/seqkit.%j.log

module load miniconda3
conda activate gen220

set -euo pipefail

seqkit stats -j "${SLURM_CPUS_PER_TASK:-1}" pep/*.pep
```

Reproducibility: environment.yml. Write down what an environment contains, so you (or a reviewer, or a classmate) can build the same one later:

```
name: gen220
channels:
  - conda-forge
  - bioconda
dependencies:
  - python=3.12
  - pandas
```

```
  - biopython
  - seqkit
  - samtools
```

```
conda env create -f environment.yml                        # build it anywhere
conda env export --from-history -n gen220 > environment.yml    # write one from an env
```

`--from-history` lists only the packages you asked for (not the hundreds of dependencies with operating-system-specific builds), so the file also works on a Mac. Keep `environment.yml` in your project's Git repository next to your scripts.

Containers: Singularity / Apptainer

A **container** is a whole software environment (an operating system plus the tool and all its dependencies) packed into one file. Docker is the most common format, but it needs administrator rights, so HPC clusters run containers with **Singularity**, now also called **Apptainer** (the HPCC currently provides SingularityCE via `module load singularity`; Apptainer uses the same commands with `apptainer` in place of `singularity`). The BioContainers project makes a container for every Bioconda package:

```
module load singularity
export SINGULARITY_CACHEDIR=~/bigdata/.singularity    # the download cache, out of home
singularity pull docker://quay.io/biocontainers/seqkit:2.13.0--he881be0_0
singularity exec seqkit_2.13.0--he881be0_0.sif seqkit stats pep/E_coli_K12.pep
```

`pull` makes a `.sif` image file; `exec IMAGE COMMAND` runs a command inside it. By default your home and current folders are visible inside the container; add `--bind /bigdata` if files there aren't. Modules don't work inside a container, and you can't build new images on the cluster (build them elsewhere, or use existing ones). Containers are how workflow managers such as nf-core make pipelines give the same results everywhere. See the HPCC Singularity guide.

Working with many files: find, xargs, parallel

find

`ls` and wildcards (UNIX II) look in one folder. `find` searches a whole tree of folders, selecting files by name, type, size or age:

```
cd ~/bigdata/gen220/blast_array
find . -name '*.tab'                 # by name (quote the pattern!)
find results -name '*.tab' -size +2M # larger than 2 MB (k, M, G)
find . -maxdepth 1 -type f -name '*.sh'   # only this folder, only files
find results -mmin -60               # modified in the last 60 minutes
find ~/bigdata -type f -mtime +30    # not modified in more than 30 days
find . -type f -empty                # empty files (often a failed step)
```

```
./results/E_coli_K12-vs-E_coli_K12.BLASTP.tab
./results/E_coli_K12-vs-E_coli_O157_H7.BLASTP.tab
./results/E_coli_K12-vs-S_enterica.BLASTP.tab
...
```

(The first command's output; `find` lists files in the order it meets them, so pipe it to `sort` if you want them sorted.) `find` can also run a command on what it finds. With `-exec COMMAND {} +` the `{}` is replaced by the file names:

```
find results -name '*vs-S_enterica*' -exec wc -l {} +
```

```
  20374 results/S_enterica-vs-S_enterica.BLASTP.tab
  20210 results/E_coli_O157_H7-vs-S_enterica.BLASTP.tab
  19213 results/E_coli_K12-vs-S_enterica.BLASTP.tab
  59797 total
```

`-exec ... {} \;` runs the command once per file instead. `-delete` deletes what it finds, with no undo: **always run the same `find` without `-delete` first** and read the list.

```
find . -name '*.tmp'            # check the list first...
find . -name '*.tmp' -delete    # ...then delete
```

xargs

`xargs` reads a list of names on STDIN and runs a command with them as arguments. With `-P` it runs several at once, e.g. compressing every table with 4 `gzip`s in parallel (inside a job with `-c 4`):

```
find results -name '*.tab' -print0 | xargs -0 -n 1 -P "${SLURM_CPUS_PER_TASK:-1}" gzip
```

`-print0` and `-0` separate the names with a special null character instead of spaces, so file names containing spaces don't break; `-n 1` gives each `gzip` one file.

GNU parallel

GNU `parallel` does the same with more convenient syntax, and keeps the output of each command together. `{}` is the input, `:::` gives the inputs on the command line, `-j` is how many to run at once, `-k` keeps the output in input order, and `--tag` labels each line with its input:

```
parallel -j 4 -k --tag 'zcat {} | cut -f1 | sort -u | wc -l' ::: results/E_coli_K12-*.tab.gz
```

```
results/E_coli_K12-vs-E_coli_K12.BLASTP.tab.gz  4213
results/E_coli_K12-vs-E_coli_O157_H7.BLASTP.tab.gz  3913
results/E_coli_K12-vs-S_enterica.BLASTP.tab.gz  3635
```

(It prints a request to cite it the first time; `parallel --citation` makes that go away. Check `module avail parallel` on the cluster.)

Arrays or parallel? On the cluster, a job array is usually the better choice when each piece takes minutes or more: every task gets its own resources, log file and `sacct` record, and failed tasks can be rerun by number. `xargs -P` or `parallel` inside **one** job is better for thousands of tiny commands (a few seconds each), which would swamp the scheduler as separate tasks - set `-j` / `-P` to `$SLURM_CPUS_PER_TASK`. Never run either of them with many processes on a login node.

Next step: workflow managers

Once an analysis has several steps and many samples - trim, align, sort, call variants, merge - keeping track of what has run, what failed and what needs rerunning by hand gets hard. **Workflow managers** do this for you: you describe each step's inputs, outputs and command, and the tool works out the order, runs only what is missing or out of date, submits the steps to SLURM for you, and can use conda or containers for each step.

- Snakemake - rules written in a Python-like language, like a `Makefile` for bioinformatics; install it with conda. The SLURM executor plugin submits each step as a job.
- Nextflow and nf-core - a large set of peer-reviewed, ready-made pipelines (RNA-seq, variant calling, assembly, amplicons…) that run on SLURM with Singularity containers.

Everything in this lecture - resources, logs, arrays, dependencies, scratch, conda and containers - is what these tools do under the hood, so it is what you'll need to understand when one of their jobs fails.

Keeping your scripts in Git

Your job scripts, sample sheets and `environment.yml` are small text files that record exactly how you ran an analysis - keep them in a Git repository (but not the big data files or results; use a `.gitignore`). Everything about Git and GitHub for this course - setting up SSH keys or `gh`, cloning your homework repository from GitHub Classroom, add/commit/push, `.gitignore`, and fixing mistakes - is in the Git and GitHub guide. The instructions for each homework are on the Assignments pages.

Quick reference

Command	What it does
`sbatch script.sh`	submit a batch job
`sbatch -p epyc -c 8 --mem 16G --time 4:00:00 script.sh`	submit, overriding the `#SBATCH` lines
`sbatch --array=1-24%6 script.sh`	submit an array, at most 6 tasks at once
`sbatch --parsable script.sh`	print only the job ID (to save in a variable)
`sbatch --dependency=afterok:JOBID next.sh`	start after JOBID finishes successfully
`srun -p short -c 4 --mem 8G --time 2:00:00 --pty bash -l`	interactive shell on a compute node
`squeue -u $USER`	your pending and running jobs
`squeue -u $USER --start`	estimated start times
`scontrol show job JOBID`	full details of a pending or running job
`scancel JOBID` / `scancel JOBID_N` / `scancel -u $USER`	cancel a job / one array task / all yours
`sacct -j JOBID --format=JobID,State,Elapsed,MaxRSS,ExitCode`	what a finished job used
`seff JOBID`	CPU and memory efficiency of a finished job
`sinfo -s`	partitions and node counts
`slurm_limits`, `group_cpus`	your limits and your group's current use (HPCC commands)
`sshare -u $USER`, `sprio -u $USER`	fair share score and job priority
`check_quota home`, `check_quota bigdata`	disk usage and quotas (HPCC command)
`tmux new -s NAME`, `Ctrl-b d`, `tmux attach -t NAME`	start, detach, reattach a session

Inside a job	Value
`$SLURM_JOB_ID`	the job ID
`$SLURM_CPUS_PER_TASK`	the `-c` value (set only if you gave `-c`)
`$SLURM_ARRAY_TASK_ID`	this task's number in an array
`$SLURM_ARRAY_JOB_ID`	the array's job ID (`%A` in log names)
`$SLURM_SUBMIT_DIR`	the folder you ran `sbatch` from
`$SCRATCH`	this job's temporary folder on the node's local disk

Exercises

Do these in a folder in your bigdata space, e.g. `~/bigdata/gen220/unix4`.

1. **First job.** Submit `hello_job.sh` from above. While it runs, look at it with `squeue -u $USER`. When it's done, read the log, then run `sacct -j JOBID` and `seff JOBID`. How much memory did it use? Change the script so the log goes to `logs/hello.JOBID.log` and submit it again. What happens if you forget `mkdir -p logs`?
2. **Threads.** Copy the BLAST job script from the BLAST lecture and rewrite its `#SBATCH` lines to use `-c 4` and `$SLURM_CPUS_PER_TASK`, following the template. Run it once with `-c 1` and once with `-c 4` (override on the `sbatch` command line). Compare `Elapsed` and CPU efficiency from `seff`. Did 4 CPUs make it 4 times faster?
3. **Break it on purpose.** Submit a job with `--mem 100M` that loads a large file into memory: `python3 -c "x = bytearray(500_000_000); print('got 500 MB')"`. What is the `State` in `sacct` and what message is in the log? Then submit `sleep 300` with `--time 0:01:00`. What happens this time?

4. **Array basics.** Write `array_demo.sh` with `--array=1-5` and a log per task. Make a `samples.txt` with five made-up sample names, and have each task print its task number and "its" sample name from the sheet. Test task 3 without SLURM by setting `SLURM_ARRAY_TASK_ID=3` first, then submit it and check all five logs.
5. **The BLAST array.** Run the worked example: `00_setup.sh`, test `01_blast_array.sh 1` in an `srun` session, then submit the array and the summary job with a dependency. Delete one of the results files and resubmit only that task number. Look at the `seff` of one task: were 4 CPUs and 4 GB a good request?
6. **Your own sample sheet.** Make a sheet with two tab-separated columns: a sample name and a FASTQ file (use the FASTQ files from the short read aligning lecture, or any `.fastq.gz` files you have). Write an array script that, for each line, counts the reads in the file (`zcat FILE | wc -l`, divided by 4, using `$(( ))`) and writes `NAME<tab>READS` to `counts/NAME.txt`. Add a dependency job that `cat`s them all into one table.
7. **A conda environment.** Set up `~/.condarc` as above. In an `srun` session create an environment containing `seqkit`, and use it in a job script to run `seqkit stats` on the three proteomes. Export it with `conda env export --from-history` and look at the file. How much space does `~/bigdata/.conda` use?
8. **Find and clean up.** Using `find`, list (a) all `.sh` files under your `~/bigdata/gen220` folder, (b) all files bigger than 10 MB, and (c) all log files older than 7 days. Then compress every `.tab` file in `blast_array/results` using `xargs -P` or `parallel` inside an `srun` session with 4 CPUs, and use `du -sh` to see how much space you saved.

Sources

HPCC documentation (checked September 2026):

- Managing jobs: partitions and their defaults, sbatch/srun examples, `seff`, job reason codes, arrays, `-c` vs `-n`
- Queue policies: partition limits, fair share, priority, `slurm_limits`
- Cluster introduction: head nodes and hardware
- Data storage: home, bigdata, scratch, snapshots, `check_quota`
- Package management: conda
- Singularity jobs
- Terminal IDEs: tmux

SLURM manuals: sbatch, squeue, sacct, job arrays. Other: Bioconda, conda documentation, GNU parallel tutorial, tmux wiki.