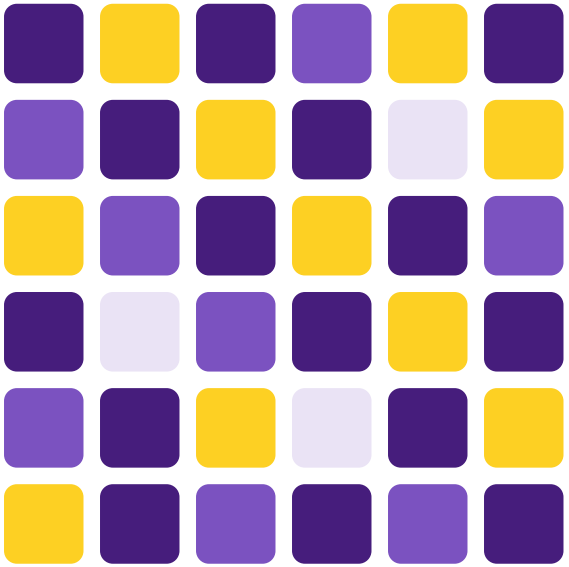
HPC USER SERVICES · LSU & LONI

Parallelizing Massively Independent Tasks

GNU Parallel & SLURM Job Arrays

Siva Prasad Kasetti
HPC User Services, LSU & LONI HPC
sys-help@loni.org
Louisiana State University, Baton Rouge

04 November 2026



one core · one task · all at once

Five parts, three live demos, one break

Part 1

Why parallel?

Many small jobs, and why hardware went multicore

Part 2

GNU Parallel essentials

Syntax, inputs, replacement strings and key options

Part 3

Distributing jobs on Slurm

Serial, threaded and MPI tasks across nodes

Part 4

SLURM job arrays

--array, indexing inputs, choosing a tool

Part 5

Proper usage

Memory, load balancing and task size

Short break after Part 3

```
$ cd ~/part1_why_parallel
```

Why parallel?

- The many-small-jobs problem
- What “massively independent tasks” means
- From one fast core to many cores



PART 1 OF 5

Thousands of small jobs, one bottleneck

Slow & wasteful

Submitting thousands of tiny jobs one by one spends more time queuing and starting than computing.

Impossible to manage

Tracking, resubmitting and cleaning up each job by hand does not scale, and cores sit idle.

It is everywhere

Genome assemblies, sequence alignments, QC checks, MD parameter sweeps: the same analysis, many inputs.

Goal: keep every core you were allocated busy with useful work

Pack the work, or let the scheduler fan it out

GNU Parallel

One Slurm job, many tasks inside it

- Runs a list of commands concurrently on the cores and nodes you already hold
- You decide how many tasks run at once (-j)
- Best for very many short or medium tasks

SLURM Job Arrays

One script, many scheduler jobs

- sbatch --array submits N copies of the same script
- Each copy gets its own index and resources
- Best for fewer, longer, independent runs

Parallelizing Massive Independent Tasks

Parallelize

Split a workload and run the pieces at the same time on many cores or nodes.

Massive

Hundreds to millions of pieces: too many to run one after another.

Independent

No piece waits for, or talks to, another. Order does not matter.

Tasks

The unit of work: one command or script applied to one input.

Also called "embarrassingly parallel" or "high-throughput" computing

Spread independent tasks over the cores of a node

One core: tasks run back-to-back



3 of 4 cores idle the whole time

Four cores: tasks run side by side



Same work, ~4× sooner

When one node is not enough, use several

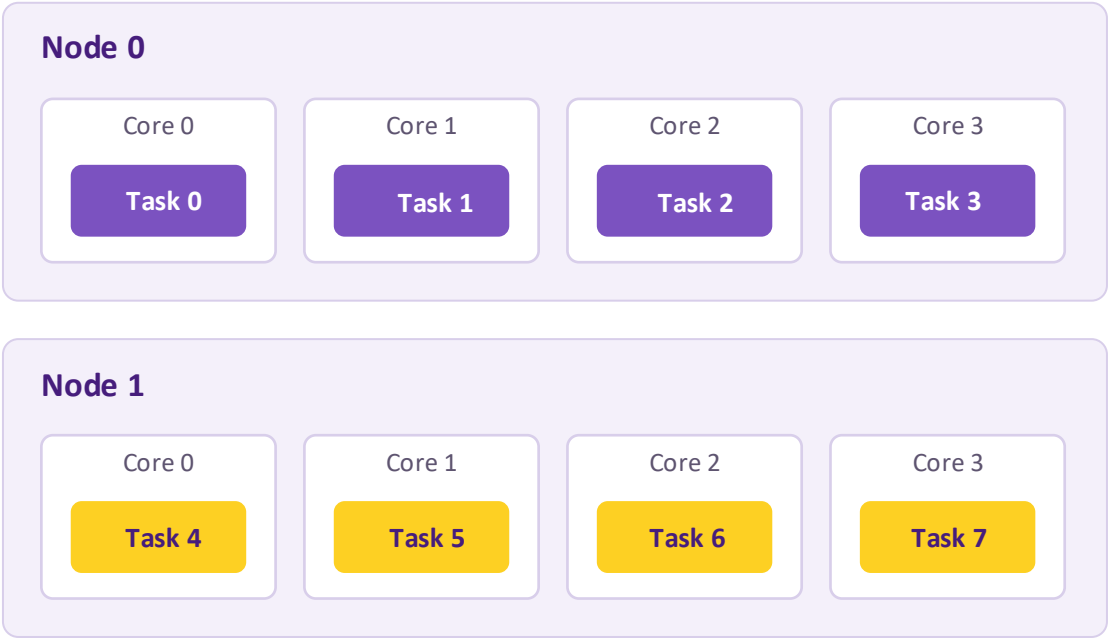
Task list

```
input.lst
01.lj.in
02.lj.in
…
08.lj.in
```

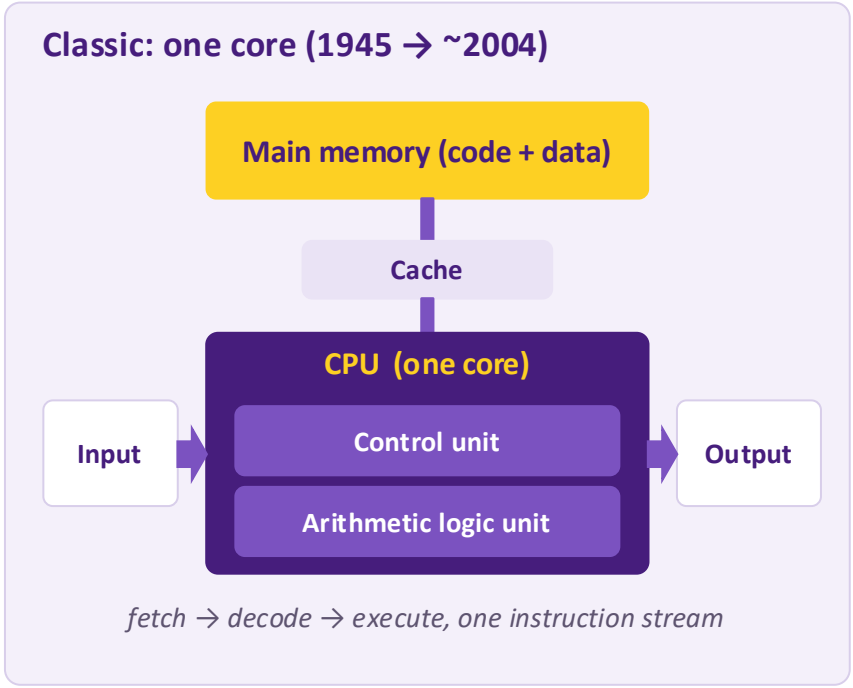
dispatcher hands out the next task as a core frees up



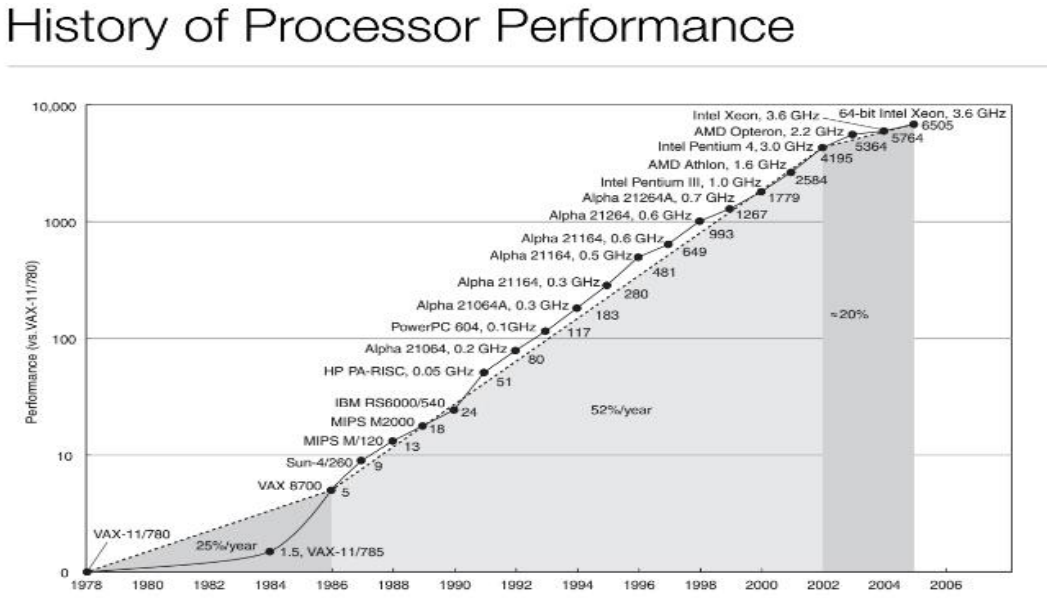




One core became many cores



Single-core speed stopped doubling



Source: Hennessy & Patterson; S. Edwards, Columbia (2012),
http://www.cs.columbia.edu/~sedwards/classes/2012/3827-spring/

~52% per year faster, 1986–2002

~20% per year since then

2× transistors every ~2 years (Moore's law)

The power wall pushed us to multicore

Smaller transistors → Faster clocks → More power → More heat → Unreliable chips



The fix: more cores, not faster ones

Since ~2005 every chip is multicore. A 64-core node only pays off when 64 things run at once.

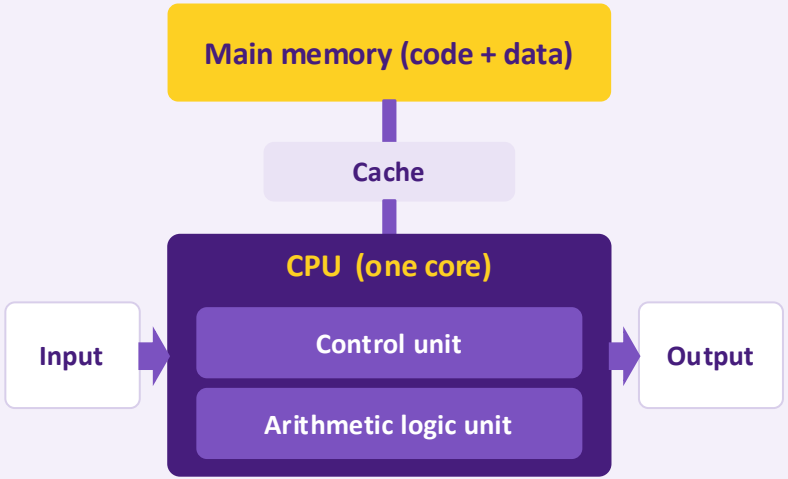


More cores ≠ faster code

Hardware no longer speeds up your program. A serial code uses 1 of 64 cores. You supply the parallelism: rewrite the code (threads, MPI) or run many tasks at once.

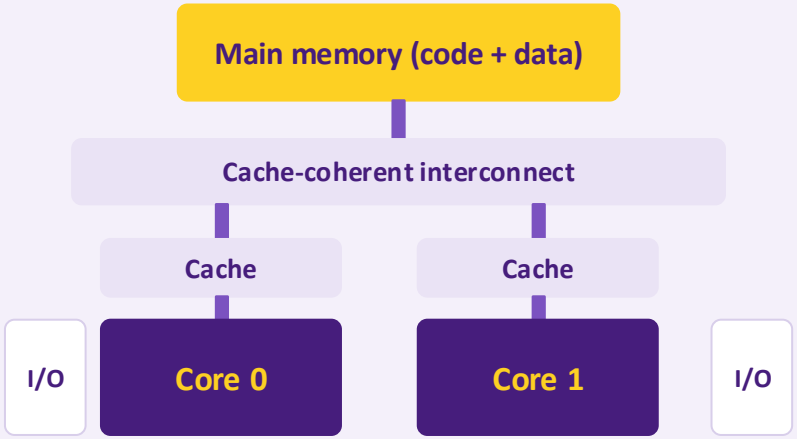
One core became many cores

Classic: one core (1945 → ~2004)



fetch → decode → execute, one instruction stream

Multicore: many cores (~2005 →)



each core is a full CPU with its own instruction stream

```
$ cd ~/part2_gnu_parallel
```

GNU Parallel essentials

- What GNU Parallel is and how to load it
- Feeding inputs: :::, :::: and pipes
- Replacement strings
- The options that matter on a cluster



PART 2 OF 5

A shell tool for embarrassingly parallel work

Give it a command and a list of inputs; it runs the command once per input, several at a time, on one or more machines.

Task

One small, independent piece of work: a single alignment, a single MD run.

Job

The whole list of tasks spread over the nodes and cores you requested.

Typical inputs

files | parameter values | hosts | URLs | tables | sample IDs

gnu.org/software/parallel

Load GNU Parallel into your environment

SuperMike3

```
$ module av parallel
parallel-netcdf/1.12.2/intel-2021.5.0
parallel/20210922/intel-2021.5.0

$ module load parallel/20210922/intel-2021.5.0
$ which parallel
/usr/local/packages/parallel/20210922/.../bin/parallel
$ parallel --version
GNU parallel 20210922
```



Tips

- Pick parallel, not parallel-netcdf
- Use version 20161022 or newer
- Put the module load inside your job script too

Three ways to feed GNU Parallel

1 **A file of full commands** — *each line is a ready-to-run command*

```
parallel [OPTIONS] < CMDFILE
```

2 **Arguments on the command line** — *three colons: values typed inline*

```
parallel [OPTIONS] CMD [ARGS] ::: ARGLIST
```

3 **Arguments from a file** — *four colons: one value per line*

```
parallel [OPTIONS] CMD [ARGS] :::: ARGFILE
```

Or pipe: `cat input.lst | parallel CMD`

Inputs typed on the command line

```
$ parallel echo ::: A B C
A
B
C
$ parallel echo ::: $(seq 1 3)
1
2
3
$ parallel echo ::: {A..Z}
A … Z
```

from a pipe

```
$ ls -1 | parallel echo
2013-06-18.tgz
backups.sh
bigmem_test.pbs
```

Two input sources = every combination

```
parallel echo {1}-{2} ::: A B ::: 1 2
```

→ A-1 A-2 B-1 B-2

Inputs read from a file, one per line

```
/project/<user>/GNU_PARALLEL
$ head -n 3 input.lst
01.lj
02.lj
03.lj

$ head -n 3 input.lst | parallel echo
01.lj
02.lj
03.lj

$ parallel echo :::: input.lst
```



Why a file?

- Thousands of inputs do not fit on a command line
- The list is a record of what you ran
- Generate it once: ls data/*.in > input.lst

Reshape each input inside the command

`Input line:  data/in.lj`

String	What it gives you	Result
`{}`	the whole input line	`data/in.lj`
`{/}`	drop the directory (basename)	`in.lj`
`{//}`	directory only (dirname)	`data`
`{.}`	drop the extension	`data/in`
`{/.}`	basename without extension	`in`

Try: parallel echo {/.} ::: data/in.lj · More in man parallel

Derive the output name from the input name

input

data/lj.in



output

output/lj.out

```
# process data/lj.in and send the result to output/lj.out
$ parallel echo {} output/{/.}.out ::: data/lj.in
data/lj.in output/lj.out
```

Swap echo for your program once the names look right: a free dry run

Any script that takes an argument will do

cmd_ex.sh

```
#!/bin/bash
# cmd_ex.sh: report input, host, directory
echo "Input: $1 on $HOSTNAME:$PWD"
```

- Make it executable: chmod +x cmd_ex.sh
- Works the same for Python, R or Perl scripts
- $1, $2 … receive the values from parallel

```
$ parallel --wd $PWD ./cmd_ex.sh ::: A B C
Input: A on mike139:/project/<user>/GNU_PARALLEL
Input: B on mike139:/project/<user>/GNU_PARALLEL
$ parallel --wd $PWD ./cmd_ex.sh :::: argfile  # same, inputs from a file
```

How many tasks run at once on each node

`-j N`	run N tasks per node at a time
`-j 0`	as many as possible
`-j 100%`	one per CPU core (the default)
`-j +N / -j -N`	cores plus or minus N (never below 1)



On a cluster

N is slots per node. Size it from cores and memory:

```
64 cores ÷ 4 threads
= -j 16
```

Use GNU Parallel ≥ 20161022 to avoid the "max jobs to run" bug

Tell GNU Parallel which nodes it may use

Slurm

```
# interactive job: 2 nodes x 64 cores
$ srun -N2 -n128 -p workq --cpu-bind none --pty bash
srun: job 39474 has been allocated resources
$ echo $SLURM_NODELIST
mike[157-158]
$ scontrol show hostname $SLURM_NODELIST > nodefile
$ cat nodefile
mike157
mike158
```

In your job script

`parallel --slf nodefile …`

one hostname per line; # and blank lines ignored

Why expand?

Slurm gives a compressed list like mike[157-158]. scontrol turns it into real hostnames GNU Parallel can ssh to.

--sshdelay, --wd, --env

Stagger starts

`--sshdelay 0.1`

Pause 0.1 s between ssh launches so the node's sshd is not flooded when 64 tasks start together.

```
parallel --sshdelay 0.1 …
```

Where tasks run

`--wd DIR`

Remote tasks start in your home by default. Point them to the submit directory instead.

```
--wd $SLURM_SUBMIT_DIR
```

Carry variables

`--env VAR`

Remote shells do not inherit your exports. Name each variable that must travel.

```
export OMP_NUM_THREADS=4
parallel --env
OMP_NUM_THREADS …
```

--progress / --bar and --joblog

```
$ parallel --progress echo ::: A B C      # interactive use only
Computers / CPU cores / Max jobs to run
1:local / 64 / 3
```

--joblog: one line per finished task

```
$ parallel --joblog logfile exit ::: 1 2 0 0
$ cat logfile
Seq  Host  Starttime       JobRuntime  Send  Receive  Exitval  Signal  Command
1    :     1477514132.358  0.019       0     0        1        0       exit 1
2    :     1477514132.375  0.003       0     0        2        0       exit 2
3    :     1477514132.376  0.002       0     0        0        0       exit 0
```

Exitval ≠ 0 means that task failed. Keep --progress and --bar out of batch scripts: they flood the output file.

--timeout and --resume-failed



--timeout SECS

Kill any task that runs longer than SECS. Use 200% to mean twice the median runtime: handy when a hung task is a failed task.



--resume-failed

With the same --joblog, rerun only the tasks that failed or never ran. --resume skips anything already logged.

```
$ parallel --joblog run.log --resume-failed --timeout 300% ./task.sh :::: input.lst
```

GNU Parallel on an interactive node



DEMO 1

follow along in your terminal

```
$ srun -N1 -n64 -p workq -t 1:00:00 -A <alloc>
$ module load parallel
$ parallel echo ::: A B C                    # then add -k
$ parallel 'echo "Job {} running on core $(awk "{print \$39}"
/proc/self/stat)"' ::: {A..Z}
$ parallel echo {1}-{2} ::: A B ::: 1 2
$ parallel echo {} output/{/.}.out ::: data/*.in
$ parallel --joblog log exit ::: 1 0 2 0; cat log
$ parallel --joblog log --resume-failed exit ::: 1 0 2 0
```



5-minute break

Stretch, refill, and have your terminal ready. Next: running real LAMMPS tasks across nodes.

```
$ cd ~/part3_distribute
```

Distributing jobs on Slurm

- The LAMMPS example and training files
- Serial tasks across two nodes
- Multi-threaded (OpenMP) tasks
- Small MPI tasks



PART 3 OF 5

Grab the example files

```
$ URL=https://www.hpc.lsu.edu/static/training/weekly-materials/Downloads/gnu_parallel_tut-main-fall2025.tar.gz
$ wget $URL/gnu_parallel_tut-main-fall2025.tar.gz
$ tar -xzf gnu_parallel_tut-main-fall2025.tar.gz
```



data/

LAMMPS input files



input.lst

one input path per line



***.sh**

Slurm scripts used today

No molecular dynamics background needed

LAMMPS is a classical molecular dynamics code (lammps.org). Today it is simply “a program that takes an input file and runs for a while”.

Serial	`lmp_mpi -in in.lammps`
Threads (OpenMP)	`lmp_mpi -sf omp -pk omp 8 -in in.lammps`
MPI (Slurm)	`srun --overlap -n 8 lmp_mpi -in in.lammps`
Hybrid (OpenMP + MPI)	`srun --overlap -n 8 lmp_mpi -sf omp -pk omp 4 -in in.lammps`

A 3D Lennard-Jones melt with a tunable run length

data/01.lj.in (abridged)

```
# 3d Lennard-Jones melt
variable  xx equal 80*$x  # box ~80 cells/side
units     lj
atom_style atomic
lattice   fcc 0.8442
region    box block 0 ${xx} 0 ${yy} 0 ${zz}
create_box 1 box
create_atoms 1 box
pair_style lj/cut 2.5
fix       1 all nve
run       ${nsteps}   # set from command line
```



The one knob

```
run ${nsteps}
```

-var nsteps 200 sets the length of each task from outside. Small value while testing, larger for production.

One Slurm job, 2 nodes, many LAMMPS runs

gp-serial.sh

```
#!/bin/bash
#SBATCH -N 2              # two nodes
#SBATCH -n 128            # 64 cores each
#SBATCH -t 2:00:00
#SBATCH -p checkpt
#SBATCH -A <your_allocation>
#SBATCH -o gp-serial.out

TASKS_PER_NODE=16
SECONDS=0

scontrol show hostname $SLURM_NODELIST > nodefile

parallel --joblog lmp.serial.log \
  -j $TASKS_PER_NODE --slf nodefile \
  --workdir $SLURM_SUBMIT_DIR --sshdelay 0.1 \
  $(which lmp_serial) -in {} -var nsteps 200 :::: input.lst

echo "took $SECONDS sec"
```

1 Allocate
#SBATCH lines reserve 2 nodes

2 List nodes
scontrol writes nodefile

3 Fan out
16 tasks per node, one per input line

4 Audit
joblog + elapsed SECONDS

Move it into a small wrapper script

inside the job script

```
parallel --joblog lmp.serial.log \
  -j $TASKS_PER_NODE \
  --slf nodefile \
  --workdir $SLURM_SUBMIT_DIR
  --sshdelay 0.1 \
  ./call_lmp.sh {} 200 :::: input.lst
```

call_lmp.sh

```
#!/bin/bash
echo "\$1=$1, \$2=$2"
lmp_serial -in $1 -var nsteps $2
```

$1 — **the line from input.lst**

e.g. data/01.lj.in

$2 — **the fixed value 200**

same for every task

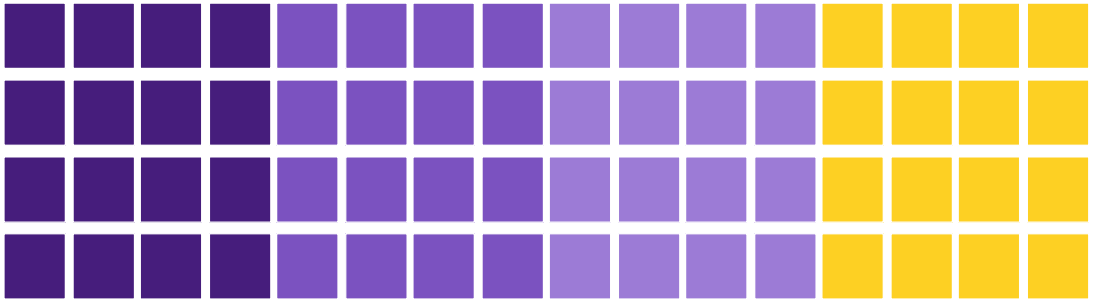
Bonus: module loads and cleanup can live in the wrapper too

Same pattern, fewer tasks per node

`TASKS_PER_NODE = CORES_PER_NODE ÷ THREADS_PER_TASK`

SuperMike3: 64 ÷ 4 = 16 tasks per node

One 64-core node, 16 tasks × 4 threads



Two changes only

- Lower -j to 16
- export OMP_NUM_THREADS=4 and pass it with --env

The serial script with three edits

gp-omp.sh

```
#SBATCH -N 2
#SBATCH -n 128
#SBATCH -t 2:00:00
#SBATCH -p checkpt
#SBATCH -A <your_allocation>
#SBATCH -o gp-omp.out

TASKS_PER_NODE=16               # 64 / 4
export OMP_NUM_THREADS=4
SECONDS=0

scontrol show hostname $SLURM_NODELIST > nodefile

parallel --joblog lmp.omp.log \
  -j $TASKS_PER_NODE --slf nodefile \
  --workdir $SLURM_SUBMIT_DIR --sshdelay 0.1 \
  --env OMP_NUM_THREADS \
  $(which lmp_omp) -sf omp -in {} -var nsteps 200 \
  :::: input.lst
echo "took $SECONDS sec"
```

16 tasks / node

-j sized for 4 threads each

--env OMP_NUM_THREADS

threads setting reaches remote nodes

lmp_omp -sf omp

LAMMPS OpenMP build and switch

Each task is a 4-rank MPI run

gp-mpi.sh

```
#SBATCH -N 2
#SBATCH -n 128
#SBATCH -t 2:00:00
#SBATCH -p checkpt
#SBATCH -A <your_allocation>
#SBATCH -o gp-mpi.out

TASKS_PER_NODE=16
PROC_PER_TASK=4                 # 16 x 4 = 64
SECONDS=0

scontrol show hostname $SLURM_NODELIST > nodefile

parallel --joblog lmp.mpi.log \
  -j $TASKS_PER_NODE --slf nodefile \
  --workdir $SLURM_SUBMIT_DIR --sshdelay 0.1 \
  srun --overlap -n $PROC_PER_TASK $(which lmp) \
  -in {} -var nsteps 200 :::: input.lst

echo "took $SECONDS sec"
```

Same arithmetic

ranks per task × tasks per node = cores per node

Keep it small

Each task stays on one node, 1 thread per rank. Big MPI jobs belong in their own allocation.

Serial vs. threaded LAMMPS tasks



DEMO 2

follow along in your terminal

```
$ cd gnu_parallel_tut-main && head -3 input.lst
$ sbatch run_serial.slurm
$ sbatch run_omp.slurm
$ squeue -u $USER
$ tail -f gp-serial.out          # Ctrl-C to stop
$ awk 'NR>1 && $7!=0' lmp.serial.log    # any failures?
$ grep took gp-serial.out gp-omp.out
```

```
$ cd ~/part4_job_arrays
```

SLURM job arrays

- What a job array is and how to write one
- Array variables, logs and management
- Demo 3, then GNU Parallel vs. job arrays



PART 4 OF 5

One script, one sbatch, many independent jobs

One submission

sbatch --array launches hundreds of jobs in seconds and keeps the scheduler load light.

Same recipe

Every element shares the script and the resource request: CPUs, memory, time.

Unique index

Each element gets $SLURM_ARRAY_TASK_ID to pick its own input and output.

Fault tolerant

Elements succeed or fail on their own; rerun just the failed indices.

Batch jobs only (no interactive arrays). Serial and threaded elements need no node counts; MPI elements still set --nodes / --ntasks.

#SBATCH --array=<index_list> (or -a)

Range	`--array=0-20`	0, 1, 2 … 20
Step	`--array=1-7:2`	1, 3, 5, 7
List	`--array=1,3,5,7`	exactly these
Throttle	`--array=0-15%4`	16 jobs, at most 4 running at once

Index range and maximum array size are capped by MaxArraySize on each cluster (check: scontrol show config | grep -i array)

What each element knows about itself

`SLURM_ARRAY_JOB_ID`	job ID of the whole array
`SLURM_ARRAY_TASK_ID`	this element's index
`SLURM_ARRAY_TASK_COUNT`	number of elements
`SLURM_ARRAY_TASK_MIN / MAX`	lowest / highest index

In #SBATCH -o / -e

`%A` array job ID
`%a` element index

```
slurm_%A_%a.out
→ slurm_12345_7.out
```

Element job IDs look like 12345_7: array ID, underscore, index.

32 LAMMPS runs, one line of input.lst each

job_array.sh

```
#!/bin/bash
#SBATCH --job-name=demo_jobarray
#SBATCH --array=1-32            # or -a 1-32
#SBATCH -o output_%A_%a.out
#SBATCH -e error_%A_%a.err
#SBATCH --time=00:02:00
#SBATCH --cpus-per-task=1

module load lammps/23Jun2022/intel-2021.5.0-intel-mpi-
2021.5.1

# read line N of input.lst, N = array index
INPUT_FILE=$(sed -n "${SLURM_ARRAY_TASK_ID}p" input.lst)

echo "Task $SLURM_ARRAY_TASK_ID on $(hostname): $INPUT_FILE"

$(which lmp_serial) -in "$INPUT_FILE" -var nsteps 100
```

1 Define the array

--array=1-32 makes 32 elements

2 Separate logs

%A_%a gives each element its own .out and .err

3 Pick the input

sed prints line N; lines start at 1, so does the array

The usual Slurm commands, with an index

```
$ sbatch job_array.sh
Submitted batch job 12345
$ squeue -u $USER -r                    # -r lists every element
$ scancel 12345_7                       # cancel one element
$ scancel 12345                         # cancel the whole array
$ scontrol update JobId=12345 ArrayTaskThrottle=8   # change the % limit
$ sacct -j 12345 --format=JobID,State,ExitCode,Elapsed,MaxRSS
$ sbatch --array=5,9,13 job_array.sh # rerun only failed indices
```

Command-line --array overrides the #SBATCH line in the script

A 32-element job array



DEMO 3

follow along in your terminal

```
$ cat job_array_serial.sh
$ sbatch job_array_serial.sh
$ squeue -u $USER -r
$ ls output_*_*.out | head
$ cat output_<jobid>_7.out
$ sacct -j <jobid> --format=JobID,State,ExitCode,Elapsed
$ sbatch --array=1-32%4 job_array.sh    # throttled
```

Pick by task count, length and how you want to recover

	GNU Parallel	Job arrays
Execution scope	Runs tasks inside one existing allocation	Each element is its own scheduler job
Resource control	You set concurrency with -j	Slurm allocates CPUs and memory per element
Visibility	One job in squeue	Every element has its own ID and state
Failure handling	Job ends → all tasks stop; joblog + --resume-failed	Rerun failed indices without touching the rest
Monitoring	joblog, --progress	squeue -r, sacct, scontrol
Sweet spot	Many short tasks: conversions, preprocessing, sweeps	Fewer longer runs: genomes, simulations, big sweeps

$ cd ~/part5_proper_usage

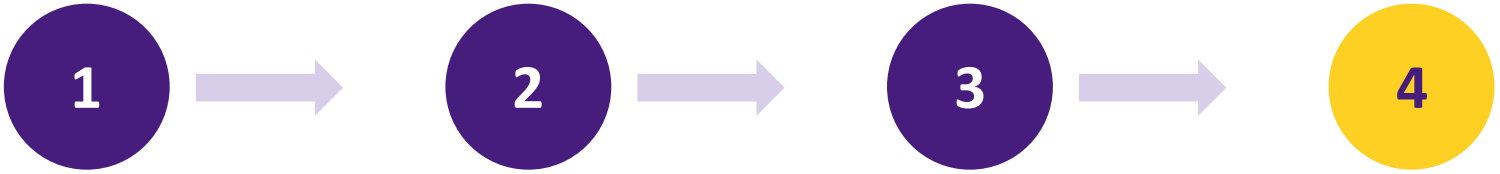
Proper usage

- Common rules: test small, measure, scale
- Memory per node
- Load balancing and task granularity
- Job array checklist



PART 5 OF 5

Grow the run only after it works small



Test

1 node, a few lines of input.lst

Measure

time and memory of one task

Scale

2 nodes next, then more

Cap

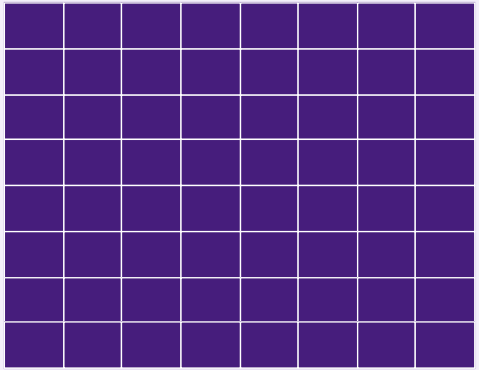
usually ≤ 5 nodes per Parallel job

Measure one task with: /usr/bin/time -v or sacct --format=MaxRSS,Elapsed

Tasks per node is capped by memory as well as cores

4 GB per task

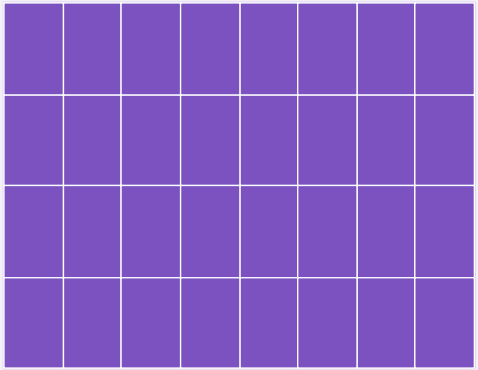
node: 256 GB



≤ 64 tasks fit

8 GB per task

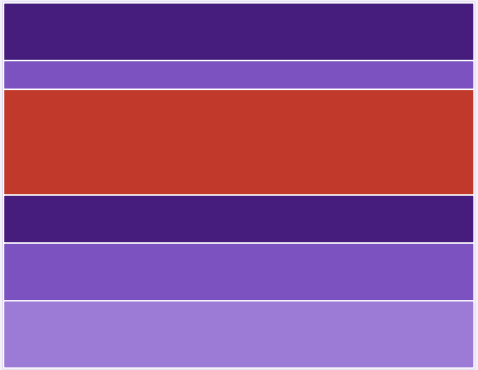
node: 256 GB



≤ 32 tasks fit

Uneven tasks

node: 256 GB

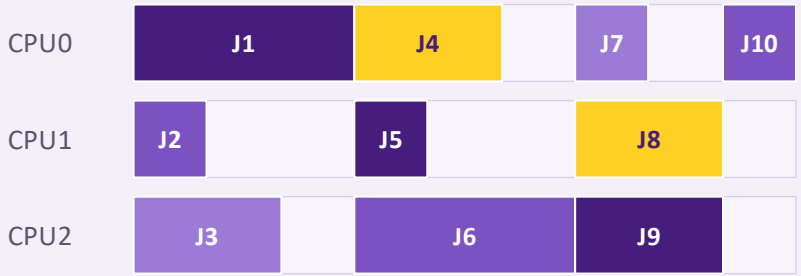


hard to predict: avoid

Leave headroom for the OS: plan on ~90% of node memory

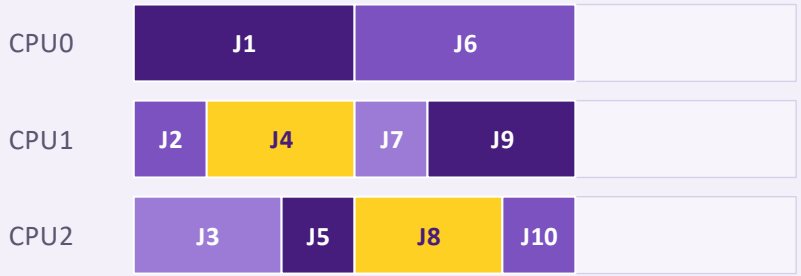
The next task starts the moment a core frees up

Fixed batches: wait for the slowest



finishes at t = 9

GNU Parallel: refill as you go

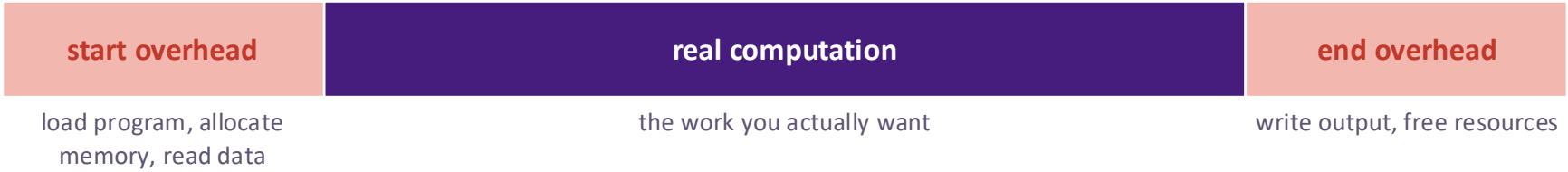


finishes at t = 6

Gaps = idle cores you are still charged for. Dynamic refill closes most of them.

How much work each task carries

Every task pays a fixed cost to start and stop



Fine grain

many tiny tasks

more parallelism, but overhead and scheduling dominate

Coarse grain

few huge tasks

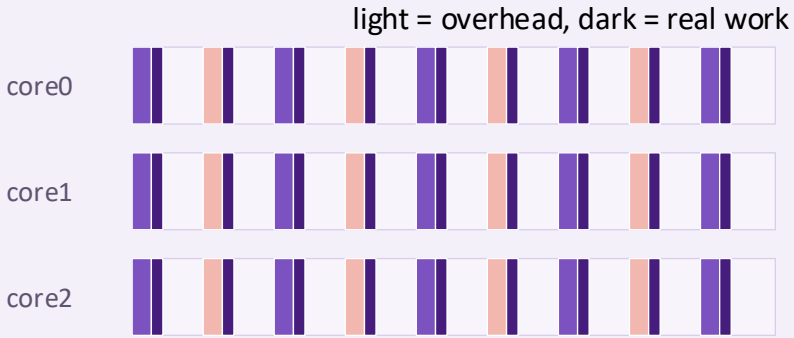
little overhead, but cores go idle waiting on stragglers

Aim between the extremes: tasks of minutes, not milliseconds or days

Both leave the cores you paid for doing nothing

Tiny grains

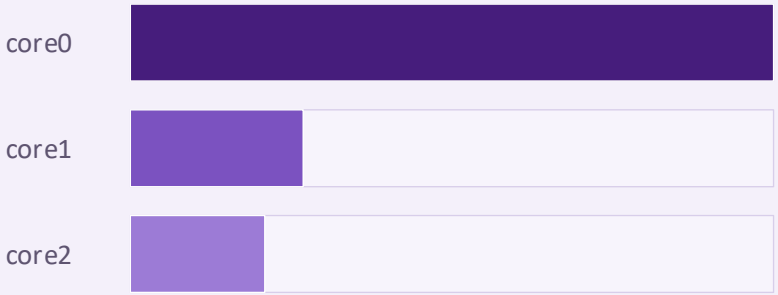
each task under a second



Startup cost swamps the work. Bundle inputs so each task runs for minutes.

Huge, uneven grains

one task far longer than the rest



Two cores finish early and wait. Split long tasks or sort longest first.

Tasks that are too small waste time on overhead, and tasks that are too uneven leave cores waiting.
Aim for tasks of similar size that run for a few minutes.

A checklist before you submit

Fit

many similar, independent runs

Index

$SLURM_ARRAY_TASK_ID picks inputs and outputs

Throttle

--array=1-1000%50 to be kind to the system

Bundle

combine tiny tasks; a 10-second element is mostly overhead

Name

%A_%a in every -o and -e

Recover

--array=5,9,13 reruns only what failed

What you can do now

- Spot embarrassingly parallel work and why it needs packing
- Drive GNU Parallel: :::, ::::, {}, -j, --slf, --joblog
- Fan out serial, threaded and MPI tasks across nodes
- Write, submit and repair SLURM job arrays
- Choose between them and size tasks wisely

Learn more

GNU Parallel tutorial
gnu.org/software/parallel

Slurm job arrays
slurm.schedmd.com/job_array.html

Training archive
hpc.lsu.edu/training

Questions?



Help ticket
sys-help@loni.org



Help desk
+1 (225) 578-0900

Recordings and slides are posted on the LSU HPC training website. Thank you!