



Advanced Slurm job submission



Part I. Workflows with Slurm

Part II: Workflows with GNU tools & Slurm

Part III: Heterogeneous jobs

Part IV: Process placement

wide

workflows



Job arrays

Scripted submissions

deep

workflows

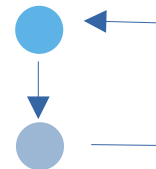


Job dependencies

Packed jobs

cyclic

workflows



Requeuing

In-job submissions



Job arrays

“Job arrays offer a mechanism for submitting and managing collections of similar jobs quickly and easily”

a.k.a. parametrized jobs

Typical use cases:

- parameter sweep
- file collection processing
- line-in-file processing



Job arrays

```
# Submit a job array with index values between 0 and 31  
$ sbatch --array=0-31 -N1 ...
```

```
# Submit a job array with index values of 1, 3, 5 and 7  
$ sbatch --array=1,3,5,7 -N1 ...
```

```
# Submit a job array with index values between 1 and 7  
# with a step size of 2 (i.e. 1, 3, 5 and 7)  
$ sbatch --array=1-7:2 -N1 ...
```



Job arrays IDs

```
SLURM_JOB_ID=36
SLURM_ARRAY_JOB_ID=36
SLURM_ARRAY_TASK_ID=1
SLURM_ARRAY_TASK_COUNT=3

SLURM_JOB_ID=37
SLURM_ARRAY_JOB_ID=36
SLURM_ARRAY_TASK_ID=2
SLURM_ARRAY_TASK_COUNT=3
[...]
```

You can address multiple jobs in the array : for instance
`scancel 36\_[1-2]`



Job array file names

```
#SBATCH --output slurm-%A_%a.out  
#SBATCH --error slurm-%A_%a.err  
  
%A -> SLURM_ARRAY_JOB_ID  
%a -> SLURM_ARRAY_TASK_ID
```



Job array indices

```
#!/bin/bash
#
#SBATCH [...]
#SBATCH --array=0-9

module load [...]

some_program $SLURM_ARRAY_TASK_ID
```

Submits a 10-job array, each job runs `some\_program` with a parameter value from 0 to 9. Jobs are independent but can be managed as a whole.



Caveat: integers only

The parameter must be

- integer,
- non-negative,
- one-dimensional,
- bounded.

The parameter cannot be

- categorical,
- real valued,
- multi-dimensional,
- larger than `MaxArraySize`.



Solution: Bash array



Bash array

```
#!/bin/bash
#
#SBATCH [...]
#SBATCH --array=0-9

module load [...]

PARAMS=( [...])

some_program ${PARAMS[$SLURM_ARRAY_TASK_ID]}
```

Submits a 10-job array, each job runs `some\_program` with a parameter value taken from the `PARAMS` array



Bash array : explicit

```
#!/bin/bash
#
#SBATCH [...]
#SBATCH --array=0-2

module load [...]

PARAMS=(red blue green)

some_program ${PARAMS[$SLURM_ARRAY_TASK_ID]}
```

Submits a 3-job array, each job runs `some\_program`, once with parameter value `red`, the other `blue` and the final one, `green`



Bash array : globbing

```
#!/bin/bash
#
#SBATCH [...]
#SBATCH --array=0-2

module load [...]

PARAMS=(~/data/*.csv) # list of .csv files in data/

some_program "${PARAMS[$SLURM_ARRAY_TASK_ID]}"
```

Submits a 3-job array, each job runs `some\_program` with a file matching the `~/data/\*.csv` pattern, in alphanumerical order.

••• Bash array : brace expansion

```
#!/bin/bash
#
#SBATCH [...]
#SBATCH --array=0-9

module load [...]

PARAMS=(1.{0..9}) # expands to 1.0 1.1 1.2 ... 1.9

some_program ${PARAMS[$SLURM_ARRAY_TASK_ID]}
```

Submits a 10-job array, each job runs `some\_program` with a parameter equal to 1.0, 1.1, 1.2, ... 1.9.

Bash array : brace expansion

```
#!/bin/bash
#
#SBATCH [...]
#SBATCH --array=0-8

module load [...]

PARAMS=({1..3}_{red,green,blue}) # = 1_red 1_green 1_blue 2_red ...

some_program ${PARAMS[$SLURM_ARRAY_TASK_ID]/_/ }
```

Submits a 9-job array, each job runs `some\_program` with two parameters equal to `1 red`, `1 green`, ..., `3 green`, `3 blue`.



Bash array : seq

```
#!/bin/bash
#
#SBATCH [...]
#SBATCH --array=0-9

module load [...]

PARAMS=$(seq --format %.3E 1 0.1 2) #= 1.000E+00 1.100E+00 ...

some_program ${PARAMS[$SLURM_ARRAY_TASK_ID]}
```

Submits a 10-job array, each job runs `some\_program` with a parameter equal to 1.000E+00, 1.100E+00, ..., 1.900+E0



Bash array : mapfile

```
#!/bin/bash
#
#SBATCH [...]
#SBATCH --array=0-9

module load [...]

mapfile PARAMS < /path/to/parameterfile # reads file into variable

some_program ${PARAMS[$SLURM_ARRAY_TASK_ID]}
```

Submits a 9-job array, each job runs `some\_program` with as parameter one line from the `parameterfile` file.



Caveat: nb jobs hardcoded

The `SBATCH` lines are comments in Bash, variable expansion is ignored.

```
`#SBATCH --array=1-$N`
```

↳ sbatch: error: Batch job submission failed: Invalid job array specification

Solutions: CLI option or `stdin` submission



CLI option

```
#!/bin/bash
#
#SBATCH [...]

module load [...]
echo ${PARAMS[@]}

some_program "${PARAMS[$SLURM_ARRAY_TASK_ID]}"
```

```
$ PARAMS=( [...])
$ N=${#PARAMS[@]}
$ sbatch --array=0-$N submit_script.sh
```

Give the argument in the command line rather than in the script



`stdin` submission

```
#!/bin/bash
#SBATCH [...]
#SBATCH --array=1-$N

module load [...]
echo ${PARAMS[@]}

some_program "${PARAMS[$SLURM_ARRAY_TASK_ID]}"
```

```
$ PARAMS=( [...])
$ N=${#PARAMS[@]}
$ cat submit_script.sh | envsubst '${N}' | sbatch
```

Feed the script to `sbatch` through `stdin` rather than as file path



Scripted submissions

Submission scripts are not always necessary ; jobs can be submitted directly on the command line.

Typical use cases:

- jobs too different for job arrays
- submission from within a pre-existing script
- only a few jobs



Inline submissions

```
#!/bin/bash
#
#SBATCH --time=[...]
#SBATCH --ntasks=[...]

module load [...] OpenMPI

mpirun some_program
```

The script can be replaced with a single call to `sbatch`

```
$ module load [...] OpenMPI
$ sbatch --time=[...] --ntasks=[...] --wrap "mpirun some_program"
```



Scripted submissions

```
$ module load OpenMPI
$ for i in 4 8 16 32; do sbatch -n=$i --wrap "mpirun some_program"; done
```

Typical use case: scaling studies

```
#!/bin/bash

for file in ~/data/*.dat
do
compress "$file"
done
```



```
#!/bin/bash

for file in ~/data/*.dat
do
sbatch --wrap "compress \"$file\""
done
```

Typical use case: when you already have a script that works on your laptop without Slurm and want to use it on the cluster.



Job dependencies

“-d, --dependency=<dependency_list>

Defer the start of this job until the specified dependencies have been satisfied completed.”

Typical use cases:

1. pre-processing job
2. processing job
3. post-processing job



Job dependencies

Possible `dependency\_list` items:

```
after:job_id[[+time][:jobid[+time]...]] # after job(s) start (+ time)
afterany:job_id[:jobid...] # after job(s) have terminated
afterburstbuffer:job_id[:jobid...] # after job+bbuffer is done
aftercorr:job_id[:jobid...] # job arrays
afternotok:job_id[:jobid...] # after jobs failed
afterok:job_id[:jobid...] # after jobs completed successfully
singleton
```

Comma-separated list → AND ; ?-separated list → OR



Job dependencies

Example

```
$ sbatch preprocess.sh
submitted batch job 1

$ sbatch -d afterok:1 process1.sh
submitted batch job 2

$ sbatch -d afterok:1 process2.sh
submitted batch job 3

$ sbatch -d afterok:2:3 postprocess.sh

$ sbatch -d afterany:1:2:3 cleanup.sh

$ sbatch -d afternotok:1?afternotok:2?afternotok:3 cancel.sh
```

Jobs whose dependency will never be satisfied must be dealt with



Caveat: IDs unknown until submitted

The dependency jobs must have been submitted before the dependent job and their job IDs be captured.

Solutions: CLI option `--parsable`



Job dependencies

```
JID=9999999 # If jobid does not exist, no dependency is set
for i in {1..4};
do
JID=$(sbatch --parsable --dependency=afterok:$JID submit_script_$i.sh)
done
```

Submits 4 jobs (`submit\_script\_1.sh` -> `submit\_script\_4.sh`)
chained together with N-1 dependency



Job dependencies

chainsubmit.sh

```
#!/bin/bash
ID=$(sbatch --parsable $1)
shift
for script in "$@"; do
    ID=$(sbatch --parsable --dependency=afterok:${ID%*;}* $script)
done
```

/chainsubmit.sh job1.slurm job2.slurm job3.slurm job4.slurm



Packed jobs (serial)

When jobs are small you can pack them into multi-step jobs to ease overhead.

```
#!/bin/bash
#SBATCH [...]

for i in {1..4};
do

srun [...] some_program $i
done
```

Submits 1 job running 4 steps in series (dependency is implicit)
All steps inherit the full allocation.



Packed jobs (parallel)

When jobs are small you can pack them into multi-step jobs to ease overhead.

```
#!/bin/bash
#SBATCH [...]
#SBATCH -n 4 # can be <= 4; srun instances will start when possible
for i in {1..4};
do
srun -n1 -c1 --exact some_program $i &
done
wait
```

Submits 1 job running 4 steps in parallel
Each step gets a subset of the allocation.



Packed jobs (parallel)

When jobs are small you can pack them into multi-step jobs to ease overhead.

```
#!/bin/bash
#SBATCH [...]
#SBATCH -n 4 # can be <= 4; srun instances will start when possible
for i in {1..4};
do
srun -n1 -c1 --exact some_program $i &
done
wait
```

If there are more steps than the allocation allows, they are queued unless `--overlap` is specified.



Packed jobs: --exclusive

```
#SLURM change log
```

```
* Changes in Slurm 20.11.0rc1
```

```
-- Make --exclusive the default with srun as a step adding --overlap to  
reverse behavior.
```

```
-- Add --whole option to srun to allocate all resources on a node in an  
allocation.
```

```
* Changes in Slurm 20.11.3
```

```
-- Partially revert changes made in 20.11.0 to srun step behavior. [...]
```

```
This reverts the behavior such that all resources on a node are assigned  
to the job step by default.
```

```
-- srun - add a new --exact option, and deprecate the --whole option  
(which has been restored as the default behavior).
```

```
# and now --exclusive implies --exact
```

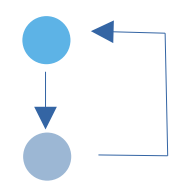
```
* Changes in Slurm 21.08.0rc1
```

```
-- --cpus-per-task and --threads-per-core now imply --exact.
```

```
* Changes in Slurm 21.08.6
```

```
-- Remove implicit --exact when --cpus-per-task is used.
```

Older Slurm versions will prefer `--exclusive`



Job queueing

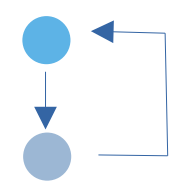
A job can requeue itself if it has not finished working

```
#!/bin/bash
#SBATCH [...]

SBATCH --append

echo "Restart count: ${SLURM_RESTART_COUNT}"
some_program

if some_condition; do
scontrol requeue $SLURM_JOB_ID
fi
```



Job queuing

A job can requeue itself if it has not finished working

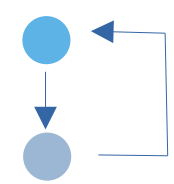
```
#!/bin/bash
#SBATCH [...]

SBATCH --append

echo "Restart count: ${SLURM_RESTART_COUNT}"
some_program

if some_condition; do
exit 99
fi
```

```
[dfr@lemaitre3 ~] (StdEnv) $ scontrol show config | grep RequeueExit
RequeueExit          = 99
RequeueExitHold       = 98
```



In-job submission

A job can submit itself again.

```
#!/bin/bash
#SBATCH [...]

some_program

if some_condition; do
  sbatch $0
fi
```

The same submission script will be used even if modified since.

Part I. Workflows with Slurm

Part II: Workflows with GNU tools & Slurm

Part III: Heterogeneous jobs

Part IV: Process placement

xargs

“xargs reads items from the standard input, [...] and executes the command [...] with items read from standard input”

Typical use cases:

- execute same command on multiple files
- line-in-file processing
- process each line output from another program

xargs

```
$ find . -name \*.csv -print0 |\
  xargs -I{} -0 sbatch [...] --wrap 'some_program "{}"'
```

Submits one **job** for each CSV file, passed as argument to `some\_program`

```
#!/bin/bash
#SBATCH [...]

module load [...]

find . -name \*.csv -print0 |\
  xargs -P $SLURM_NTASKS -I{} -0 srun --exact [...] some_program "{}"
```

Generates one job **step** for each CSV file found (max
\$SLURM_NTASKS at a time)

xargs

```
$ cat parameters.csv |\n  xargs -I{} sbatch [...] --wrap "some_program {}"
```

Submits one **job** for each line in `parameters.csv`

```
#!/bin/bash\n#SBATCH [...]\n\nmodule load [...]\n\ncat parameters.csv |\n  xargs -P $SLURM_NTASKS -I{} srun --exact [...] some_program {}
```

Generates one job **step** for each line in `parameters.csv` (max \$SLURM_NTASKS at a time)

envsubst

“standard input is copied to standard output, with references to environment variables of the form `$VARIABLE` or `${VARIABLE}` being replaced with the corresponding values. “

Typical use case:

- when program parameters are in a file rather than on command line

envsubst

```
$ cat input.txt
nprocs=$NPROCS
tol=$TOL
MaxIter=$MAXITER

$ export NPROCS=4; export TOL=0.01; export MAXITER=10000

$ envsubst < input.txt
nprocs=4
tol=0.01
MaxIter=100090
```

envsubst

```
$ NPROCLIST=(2 4 8)
$ TOLLIST=(0.01 0.001 0.001)
$ MAXITERLIST=(1000 10000 100000)

$ for i in {0..2}; do
> NPROC=${NPROCLIST[$i]} ; TOL=${TOLLIST[$i]} ; MAXITER=${MAXITERLIST[i]}
> envsubst < input.txt > input.job$i.txt
> sbatch [...] --wrap "some_program input.job$i.txt"
> done
```

Submits one job for each triplet (NPROC, TOL, MAXITER), creating the needed input file from the environment variables.



GNU Parallel

“GNU parallel is a shell tool for executing jobs in parallel using one or more computers. “

Typical use cases:

- for generating parameters (scripted submissions)
- for managing parallel processes (job packing)



GNU Parallel and Slurm

```
$ parallel sbatch [...] --wrap "some_program {}" ::: *.csv
```

Submits one **job** for each CSV file found

```
#!/bin/bash
#SBATCH [...]

module load [...]

parallel -j $SLURM_NTASKS srun --exclusive [...] some_program {} ::: *.csv
```

Generates one **job** step for each CSV file found, max
\$SLURM_NTASK at a time



GNU Make

“GNU Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files.”

Typical use case:

- for building software



GNU Make

“GNU Make is a tool which controls the generation of executables and other non-source files of a program from the program’s source files. [...] Make is not limited to building a package. You can also use Make to control installing or deinstalling a package, generate tags tables for it, or anything else you want to do often enough to make it worth while writing down how to do it.”

Highjacked use case:

- for managing processes dependencies (job packing)

<https://www.gnu.org/software/make/>



GNU Make

```
$ cat Makefile
# comment
target1: dependencies1 ... target2
        commands1
        ...

target2: dependencies2 ...
        commands2
        ...
```

Running `make` builds the file `target1` by first building file `target2`



GNU Make

```
$ cat Makefile
# Create archive and compress
archive.tar.gz: archive.tar
    gzip -k archive.tar

archive.tar: file1.txt
    tar cvzf archive.tar directory

file1.txt:
    mkdir -p directory
    touch directory/file1.txt
```



GNU Make

```
$ make
mkdir -p directory
touch directory/file1.txt
tar cvzf archive.tar directory
directory/
directory/file1.txt
gzip -k archive.tar

$ make
make: `archive.tar.gz' is up to date.

$ rm archive.tar.gz
$ make
gzip -k archive.tar
```

Make only builds what is needed (based on timestamps)



GNU Make: shell

```
$ cat Makefile
# default shell is /bin/sh

.ONESHELL:
SHELL = /bin/bash

target1:  dependencies ... target2
          commands1
          ...

target2:  dependencies ...
          commands2
          ...
```

Make will use Bash and run all commands for a given target in the same shell invocation (otherwise, one invocation per line)



GNU Make: //

```
$ make --jobs 4 --output-sync
```

The `-j` or `--jobs` option tells make to execute many recipes simultaneously, while `--output-sync` prevents mingled outputs



GNU Make and Slurm

```
$ cat Makefile
.ONESHELL:
SHELL=srun
.SHELLFLAGS= -n1 -c1 --exact bash -c
[...]
```

```
#!/bin/bash
#SBATCH [...]

module load [...]

make -j $SLURM_NTASKS
```

Will run every command as a Slurm step, in parallel, honoring dependencies.

Part I. Workflows with Slurm

Part II: Workflows with GNU tools & Slurm

Part III: Heterogeneous jobs

Part IV: Process placement

How to submit a master/slave job >

Use `srun --multi-prog`

You want	You ask
N CPUs to launch N processes	<code>--ntasks=N</code>

submit-masterslave.sh

```
#!/usr/bin/env bash
#SBATCH --ntasks=3

cp /CECI/proj/training/slurm/coordinator.sh .
cp /CECI/proj/training/slurm/worker.sh .
cp /CECI/proj/training/slurm/multi.conf .

srun --multi-prog multi.conf
```

multi.conf

```
# multi.conf for --multi-prog
0    ./coordinator.sh
1-2  ./worker.sh
```



Heterogeneous job

A job where all processes do not require the same resources. E.g.

- a master/worker setup
- a coupled simulation model (MPMD)

```
#!/bin/bash
#SBATCH --cpus-per-task=2
#SBATCH --mem-per-cpu=1g
#SBATCH --ntasks=1
#SBATCH --partition=main

#SBATCH packjob

#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=4g
#SBATCH --ntasks=4
#SBATCH --partition=large

[...]
```



Het. MPMD job

A job where all processes do not require the same resources. E.g.

- a master/worker setup

[No `multi.conf` file]

```
[...]
```

```
srun --het-group=0 coordinator.sh &
```

```
srun --het-group=1 worker.sh &
```

```
wait
```

Distinct steps, distinct MPI world



Het. master/worker job

A job where all processes do not require the same resources. E.g.
- a coupled simulation model

```
[...]
```

```
srun model.exe : iomanager.exe
```

```
# or
```

```
mpirun model.exe : iomanager.exe
```

Single step, single MPI world

Part I. Workflows with Slurm

Part II: Workflows with GNU tools & Slurm

Part III: Heterogeneous jobs

Part IV: Process placement



Process placement

Remember:

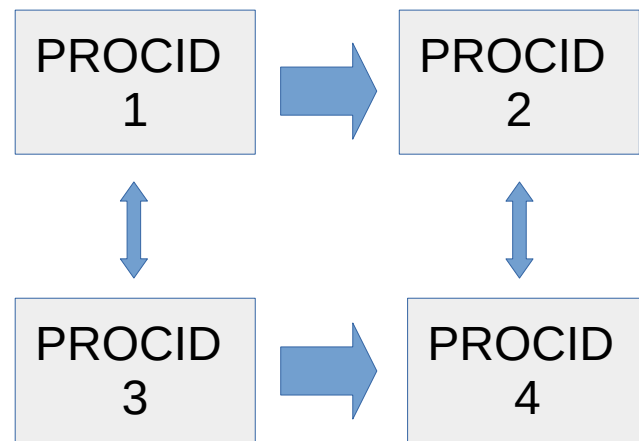
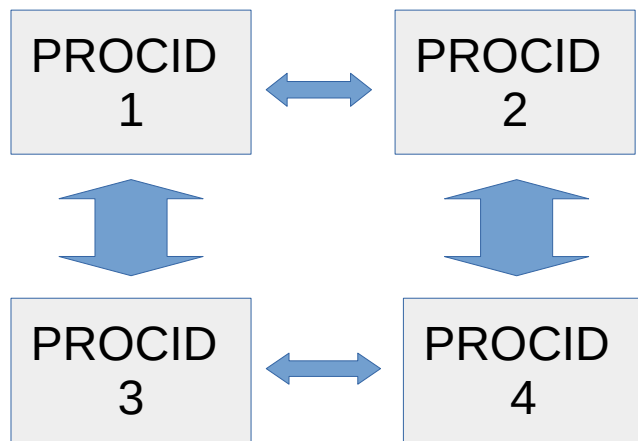
How to submit an MPI job >

Specify a number of “tasks”
and optionally a number of “nodes”

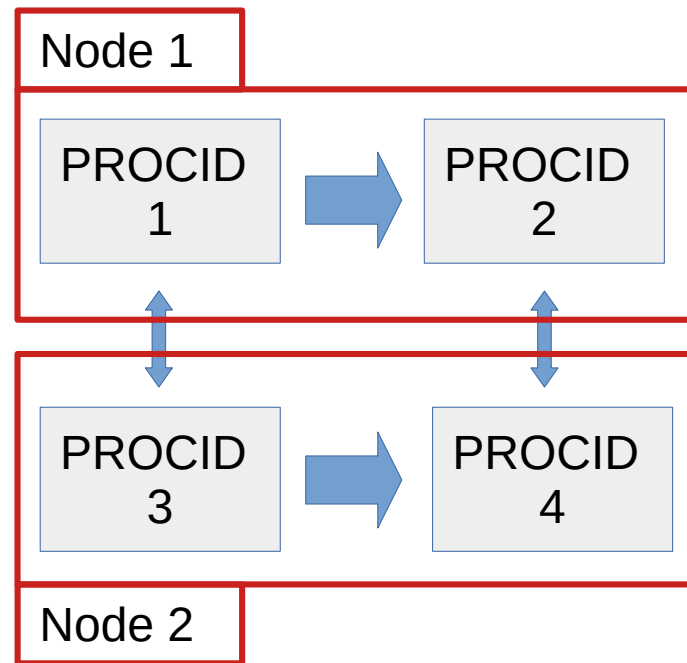
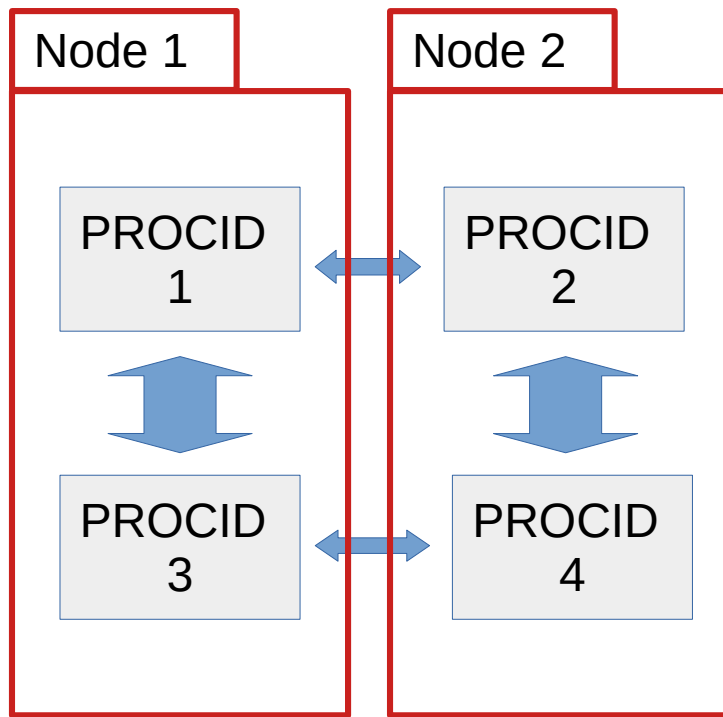
You want	You ask
N CPUs	<code>--ntasks=N</code>
N CPUs spread across distinct nodes	<code>--ntasks=N --nodes=N</code> <i>or</i> <code>--ntasks=N --ntasks-per-node=1</code>
N CPUs spread across distinct nodes and nobody else around	<code>--nodes=N --exclusive</code>
N CPUs spread across $N/2$ nodes	<code>--ntasks=N --ntasks-per-node=2</code>
N CPUs on the same node	<code>--ntasks=N --ntasks-per-node=N</code> <i>or</i> <code>--ntasks=N --nodes=1</code>

How are
ranks
distributed?

Rank distribution across nodes



Rank distribution across nodes





Rank distribution across nodes

The `-m/--distribution` option controls the strategy for placement

```
$ srun -l -n 4 -N 2 -m block --exclusive hostname | sort
```

```
0: lm3-w014.cluster
```

```
1: lm3-w014.cluster
```

```
2: lm3-w018.cluster
```

```
3: lm3-w018.cluster
```

```
$ srun -l -n 4 -N 2 -m cyclic --exclusive hostname | sort
```

```
0: lm3-w014.cluster
```

```
1: lm3-w018.cluster
```

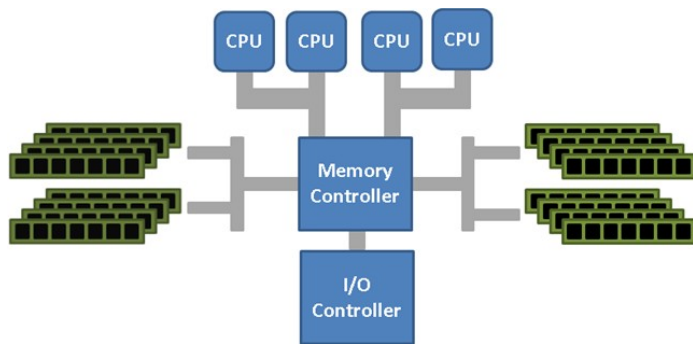
```
2: lm3-w014.cluster
```

```
3: lm3-w018.cluster
```

Other options: arbitrary, plane= n

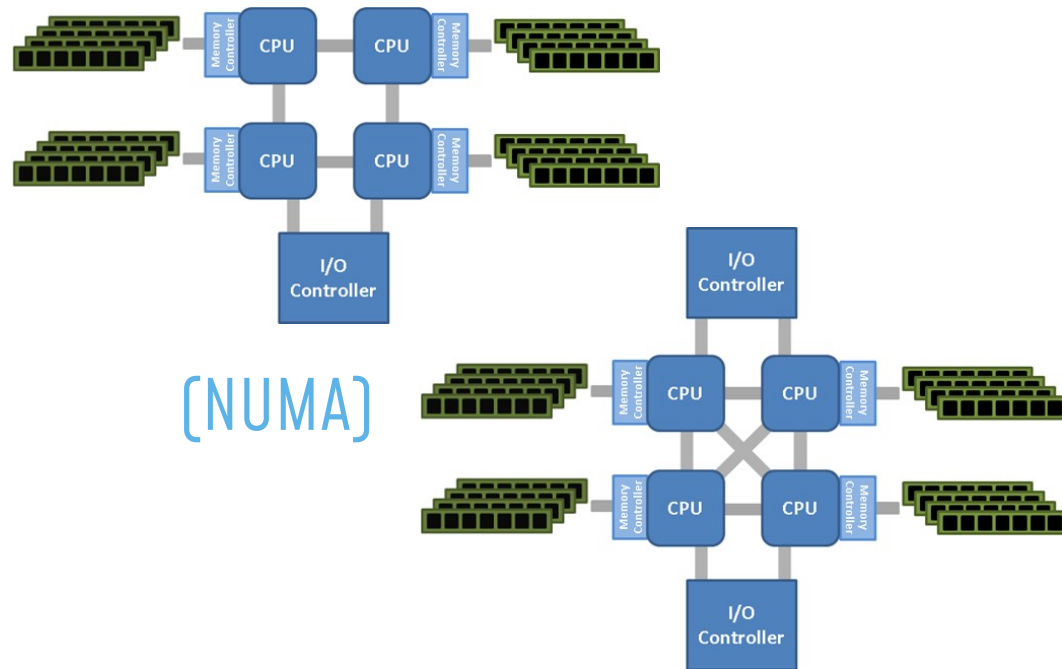
Rank distribution inside the node

[SMP]



VS.

[NUMA]





Rank distribution per socket

The `-m/--distribution` second option controls the strategy for placement on sockets. E.g. on a two-socket node (2x32 cores):

```
$ srun -l -n 4 -c4 -m block:block --exclusive bash -c 'taskset -cp $$'
```

```
0: pid 3156907's current affinity list: 0-3  
1: pid 3156908's current affinity list: 4-7  
2: pid 3156909's current affinity list: 8-11  
3: pid 3156910's current affinity list: 12-15
```

```
$ srun -l -n 4 -c4 -m block:cyclic --exclusive bash -c 'taskset -cp $$'
```

```
0: pid 3156982's current affinity list: 0-3  
1: pid 3156983's current affinity list: 32-35  
2: pid 3156984's current affinity list: 4-7  
3: pid 3156985's current affinity list: 36-39
```



Rank distribution per socket

The `-m/--distribution` second option controls the strategy for placement on sockets. E.g. on a two-socket node (2x32 cores):

```
$ srun -l -n 4 -c4 -m block:cyclic --exclusive bash -c 'taskset -cp $$'
```

```
0: pid 3156982's current affinity list: 0-3  
1: pid 3156983's current affinity list: 32-35  
2: pid 3156984's current affinity list: 4-7  
3: pid 3156985's current affinity list: 36-39
```

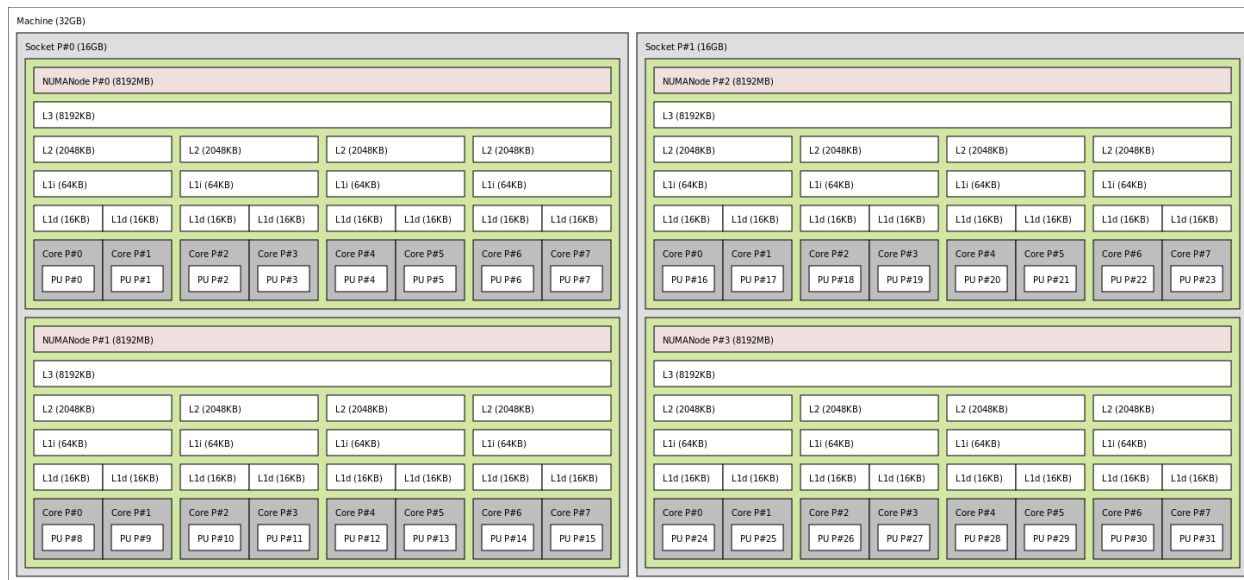
```
$ srun -l -n 4 -c4 -m block:fcyclic --exclusive bash -c 'taskset -cp $$'
```

```
0: pid 3157056's current affinity list: 0,1,32,33  
1: pid 3157057's current affinity list: 2,3,34,35  
2: pid 3157058's current affinity list: 4,5,36,37  
3: pid 3157059's current affinity list: 6,7,38,39
```

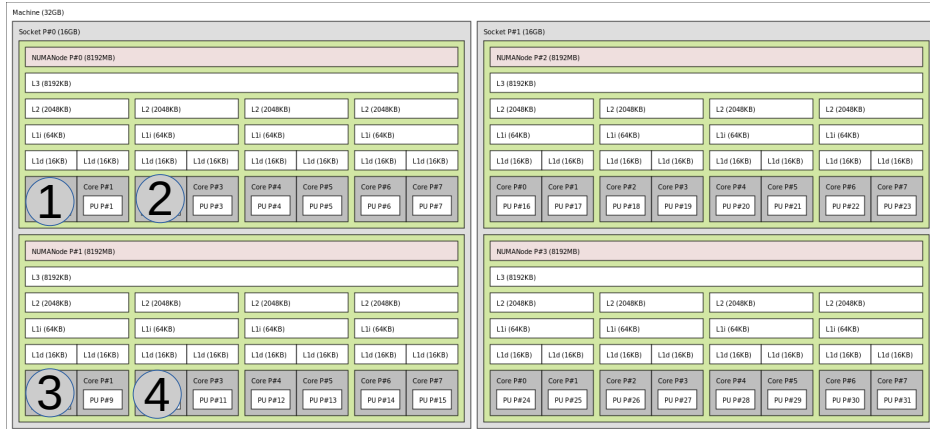
Rank distribution per SMT thread

The `-m/--distribution` third option controls the strategy for placement on cores.

- Can be replaced with `--hint=[no]multithread`
- Unfortunately there are intermediate levels that Slurm does not take into account.



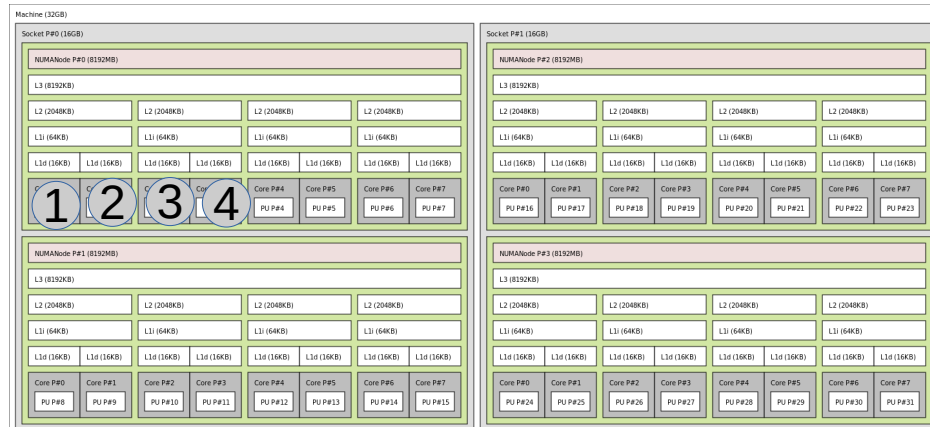
Generic intra-node distribution



Spread Packed →

Cache size available per process
Memory bandwidth available per process
Latency to common shared memory

good
good
bad
bad
good





Arbitrary intra-node distribution

With the `--cpu-bind=map_cpu:...` option you can specify the [ordered] list of CPUs to distribute the processes on.

```
$ srun -l -n 4 --cpu-bind=map_cpu:3,0,22,12 bash -c 'taskset -cp $$'

0: pid 18258's current affinity list: 3
1: pid 18259's current affinity list: 0
2: pid 18260's current affinity list: 22
3: pid 18261's current affinity list: 12
```

Use `--bind-mask` if each tasks must use multiple CPUs



Rank distribution helper tool

OpenMPI's utility tool `hwloc-distrib` can be used to build an optimally-spread placement based on the server hardware configuration.

```
$ D=$(hwloc-distrib --single 16 | xargs hwloc-calc --pulist) #128-CPU node
$ echo $D
0,8,16,24,32,40,48,56,64,72,80,88,96,104,112,120
$ srun -l -n 4 --cpu-bind=map_cpu:$D bash -c 'taskset -cp $$'
0: pid 18258's current affinity list: 0
1: pid 18259's current affinity list: 8
2: pid 18260's current affinity list: 16
3: pid 18261's current affinity list: 24
[...]
```



Rank and binary distribution

In case of MPMD job, you might want to distribute executable binaries as well **with** **--multi-prog** or with a launcher script.

```
# multi.conf for --multi-prog
```

```
0-3    ./model1.exe
```

```
4-7    ./model2.exe
```

```
8-11   ./model1.exe
```

```
12-15  ./model2.exe
```

```
$ hwloc-distrib 4 | xargs -L1 hwloc-calc --pulist \  
                  | cut -d, -f 1-4 | paste -s -d,  
0,1,2,3,32,33,34,35,64,65,66,67,96,97,98,99
```

```
$ srun -l --cpu-bind=map_cpu:0,1,2,3,32,33,... --multi-prog=multi.conf
```



Rank and binary distribution

In case of MPMD job, you might want to distribute executable binaries as well with `--multi-prog` or with a **launcher script**.

```
#!/bin/bash
EXE=(./model1.exe ./model2.exe)
PATTERN=(0 0 0 0 1 1 1 1)

INDEX=$((SLURM_PROCID % ${#PATTERN[@]}))
exec ${EXE[${PATTERN[INDEX]}]}

$ srun -l -cpu-bind=map_cpu:0,1,2,3,32,33,... helper.sh
```

Part I. Workflows with Slurm

Part II: Workflows with GNU tools

Part III: Heterogeneous jobs

Part IV: Process placement