Workflows with basic GNU tools (and Maestrowf)

UCLouvain

Goal of this session:

Learn methods and readily-available tools that can help managing HPC workflows.

Part I. Workflows with Slurm

Part II: Workflows with GNU tools

Part III: Maestrowf

wide

workflows

Job arrays

Scripted submissions

deep

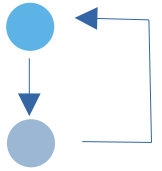
workflows

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

https://slurm.schedmd.com/job_array.html

Job arrays

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



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



# 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 tmp
```

https://slurm.schedmd.com/job_array.html

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
[…]
```

Can address multiple jobs in the array : for instance

`scancel 36_[1-2]`

https://slurm.schedmd.com/job_array.html

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
```

https://slurm.schedmd.com/job_array.html

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

https://slurm.schedmd.com/job_array.html

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`

https://slurm.schedmd.com/job_array.html

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.

https://www.gnu.org/software/bash/manual/bash.html#Filename-Expansion

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.

https://www.gnu.org/software/bash/manual/bash.html#Brace-Expansion

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`.

https://www.gnu.org/software/bash/manual/bash.html#Brace-Expansion

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

https://www.gnu.org/software/bash/manual/bash.html#Brace-Expansion

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.

https://www.gnu.org/software/bash/manual/bash.html

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

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

`stdin` submission

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

module load […]
echo ${PARAMS[@]}

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

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

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

https://www.gnu.org/software/gettext/manual/html_node/envsubst-Invocation.html

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

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

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

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

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

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

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=999999 # 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

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

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 less than 4; srun will start when possible
for i in {1..4};
do
srun -n1 -c1 --exclusive some_program $i &
done
wait
```

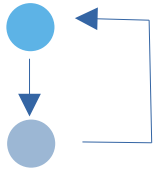
Submits 1 job running 4 steps in parallel

Each step gets a subset of the allocation

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.
```

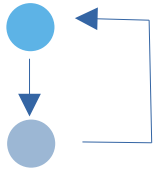
Safe to use `--exclusive` regardless of Slurm version



Job requeuing

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
```



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

Part III: Maestrowf

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

https://en.wikipedia.org/wiki/Xargs

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 --exclusive […] some_program “{}”
```

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

https://en.wikipedia.org/wiki/Xargs

xargs

```
$ cat parameters.csv |\
    xargs -I{} sbatch […] --wrap “some_program {}”
```

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

```
#!/bin/bash
#SBATCH […]

module load […]

cat parameters.csv |\
    xargs -P $SLURM_NTASKS -I{} srun --exclusive […] some_program {}
```

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

https://en.wikipedia.org/wiki/Xargs

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

https://www.gnu.org/software/gettext/manual/html_node/envsubst-Invocation.html

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
```

https://en.wikipedia.org/wiki/Xargs

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=${MAXITERLIST[i]}
> envubst < input.txt > input.job$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.

https://en.wikipedia.org/wiki/Xargs

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)

https://www.gnu.org/software/parallel/

GNU Parallel

```
$ parallel echo ::: 1 2 3
1
2
3
```

Runs the `echo` command three times in parallel (order of output is random) with each parameter listed after `:::`. Equivalent to

```
$ echo 1 &
$ echo 2 &
$ echo 3 &
```

https://www.gnu.org/software/parallel/

(BEWARE of the version installed by default which can be too old. Look for modules or install by yourself)

GNU Parallel

```
$ parallel echo ::: {1..3}
1
2
3
```

Runs the `echo` command three times in parallel (order of output is random) with each parameter listed after `:::`. Equivalent to

```
$ echo 1 &
$ echo 2 &
$ echo 3 &
```

https://www.gnu.org/software/parallel/

(BEWARE of the version installed by default which can be too old. Look for modules or install by yourself)

GNU Parallel

```
$ parallel echo ::: $(seq 1 3)
1
2
3
```

Runs the `echo` command three times in parallel (order of output is random) with each parameter listed after `:::`. Equivalent to

```
$ echo 1 &
$ echo 2 &
$ echo 3 &
```

https://www.gnu.org/software/parallel/

(BEWARE of the version installed by default which can be too old. Look for modules or install by yourself)

GNU Parallel

```
$ parallel echo ::: *csv
file1.csv
file2.csv
file3.csv
```

Runs the `echo` command three times in parallel (order of output is random) with each parameter listed after `:::`. Equivalent to

```
$ echo file1.csv &
$ echo file2.csv &
$ echo file3.csv &
```

https://www.gnu.org/software/parallel/

(BEWARE of the version installed by default which can be too old. Look for modules or install by yourself)

GNU Parallel: {}

```
$ parallel echo file{}.csv ::: 1 2 3
file1.csv
file2.csv
file3.csv
```

The `{}` is a placeholder where the argument must be inserted.

Equivalent to:

```
$ echo file1.csv &
$ echo file2.csv &
$ echo file3.csv &
```

https://www.gnu.org/software/parallel/

(BEWARE of the version installed by default which can be too old. Look for modules or install by yourself)

GNU Parallel: {.}

```
$ parallel cp {} {.}.copy ::: file1.csv file2.csv file3.csv
$
```

The `{.}` is a placeholder where the argument must be inserted and the part after the last dot removed. Equivalent to:

```
$ cp file1.csv file1.copy &
$ cp file2.csv file2.copy &
$ cp file3.csv file3.copy &
```

https://www.gnu.org/software/parallel/

(BEWARE of the version installed by default which can be too old. Look for modules or install by yourself)

GNU Parallel: {/}

```
$ parallel cp {} {/} ::: data/file1.csv data/file2.csv data/file3.csv
$
```

The `{/}` is a placeholder where the argument must be inserted and the part before the last slash removed. Equivalent to:

```
$ cp data/file1.csv file1.csv &
$ cp data/file2.csv file2.csv &
$ cp data/file3.csv file3.csv &
```

https://www.gnu.org/software/parallel/

(BEWARE of the version installed by default which can be too old. Look for modules or install by yourself)

GNU Parallel: ::: :::

```
$ parallel echo {1} {2} ::: 1 2 3 ::: A B C
$ 1 A
$ 1 B
$ 1 C
$ 2 A
[…]
```

Multiple sources can be given → all combinations are generated.

Sources can be referenced in the place holder

https://www.gnu.org/software/parallel/

(BEWARE of the version installed by default which can be too old. Look for modules or install by yourself)

GNU Parallel: :::+

```
$ parallel echo {1} {2} ::: 1 2 3 :::+ A B C
$ 1 A
$ 2 B
$ 3 C
```

With `--link`, or `:::+`, only pairs of correspondent indices.

Sources can be recycled

```
$ parallel --link echo {1} {2} ::: 1 2 3 ::: A B
$ 1 A
$ 2 B
$ 3 A
```

https://www.gnu.org/software/parallel/

(BEWARE of the version installed by default which can be too old. Look for modules or install by yourself)

GNU Parallel: ::::

```
$ cat input.txt
number,letter
1, A
2, B
3, C
$ parallel --header . --colsep , echo {number} {letter} :::: inputs.txt
1 A
2 B
3 C
```

Four ':' → takes sources from file rather than from line.

https://www.gnu.org/software/parallel/

(BEWARE of the version installed by default which can be too old. Look for modules or install by yourself)

GNU Parallel: options

```
--tag # label output lines
--bar # progress bar
--dryrun # tell rather than do
--files # write outputs to files
--delay # do not start all at the same time
--joblog # write progress to a file
--resume # resume interrupted work
--pipe # split large file and process each chunks
--filter # ignore certain combinations of parameters
--embed # 0-install parallel script
--memfree # monitor and cap memory usage
--load # monitor and cap CPU load
[…]
```

https://www.gnu.org/software/parallel/parallel.html#options

(when publishing you will have to cite the author)

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

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

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`

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

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
```

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

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)

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

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)

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

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

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

GNU Make: $()

```
# Variables

objects = file1 file2 file3
fullfile: $(objects)
        cat $(objects) > fullfile
```

Make can use variables (and defer their evaluation if needed) and functions

```
# Functions

$(subst from,to,text)
$(strip string)
$(findstring find,in)
$(sort list)
[…]
```

https://www.gnu.org/software/make/manual/make.html#Text-Functions

GNU Make: options

```
--always-make # Force running all commands again
--makefile=file  # Use another name for the Makefile
--dry-run  # Only tell what will be done, without doing it
--touch    # Pretend all files have been updated
--assume-old=file # Assume file has not been modified recently
--debug  # Print debugging information
[…]
```

https://www.gnu.org/software/make/manual/make.html#Options-Summary

GNU Make and Slurm

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

```
#!/bin/bash
#SBATCH […]

module load […]

make -j $SLURM_NTASKS
```

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

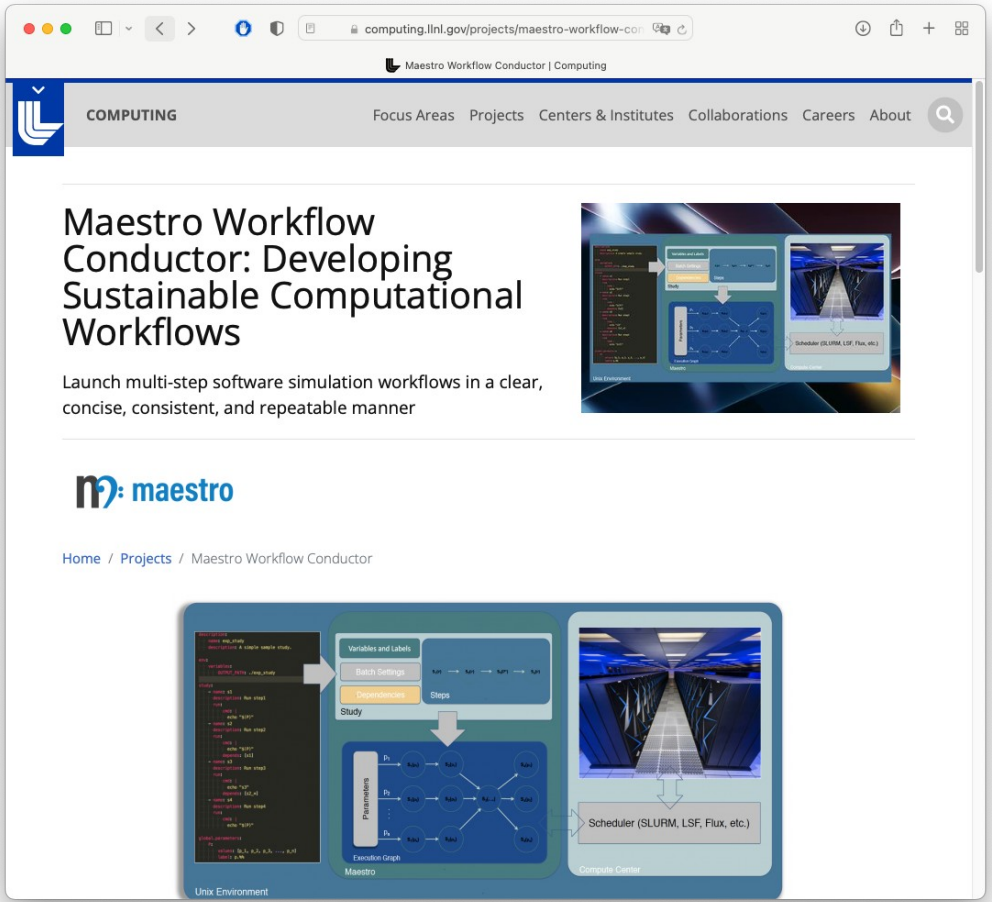
http://plindenbaum.blogspot.com/2014/09/parallelizing-gnu-make-4-in-slurm.html

Part I. Workflows with Slurm

Part II: Workflows with GNU tools

Part III: Maestrowf

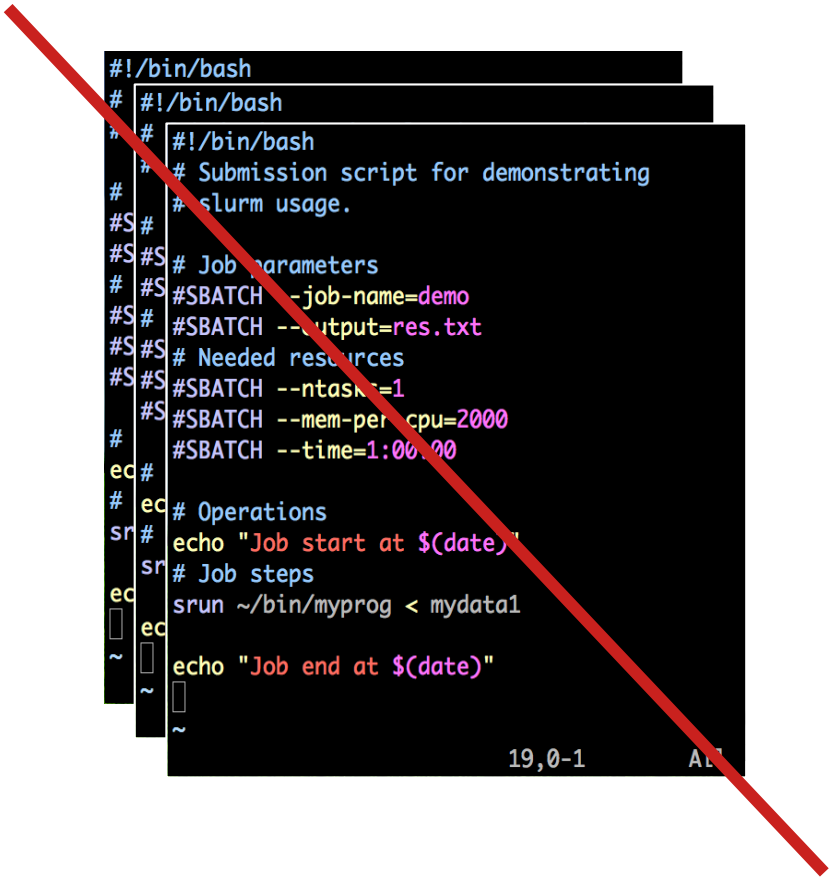
Maestro

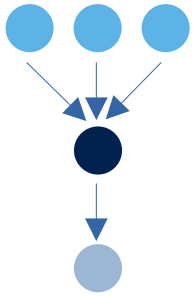


Install with `pip install maestrowf`

https://computing.llnl.gov/projects/maestro-workflow-conductor

Maestro


#!/bin/bash
Submission script for demonstrating
slurm usage.
Job parameters
#SBATCH --job-name=demo
#SBATCH --output=res.txt
Needed resources
#SBATCH --ntasks=1
#SBATCH --mem-per-cpu=2000
#SBATCH --time=1:00:00
Operations
echo "Job start at $(date)"
Job steps
srun ~/bin/myprog < mydata1
echo "Job end at $(date)"
19,0-1



Maestro

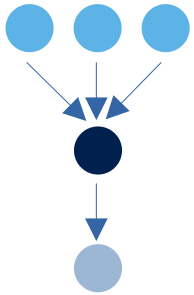
```yaml
description:
    name: Build archive
    description: A simple archive building study

study:
    - name: generate
      description: creates some file
      run:
         cmd: |
           mkdir -p directory
           touch directory/file1.txt
```

Study overview

Study steps
- name
- description
- command(s)

Study: a YAML file with

- information about the study (documentation)
- list of steps and dependencies
- parameters to sweep through
- information about job requirements (scheduler-agnostic)



Maestro

```
description:
    name: Build archive
    description: A simple archive building study

study:
    - name: generate
      description: creates some file
      run:
         cmd: |
           mkdir -p directory
           touch directory/file1.txt
```

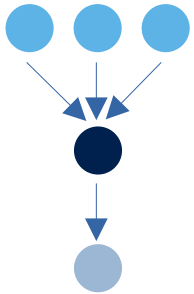
Study overview

Study steps

- name
- description
- command(s)

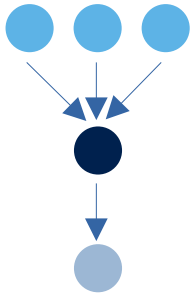
Maestro

- forces you to document your jobs
- organise output directories (workspaces)
- monitors and manage (resubmit, etc.) jobs



Maestro

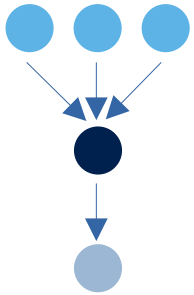
```
$ maestro --help

usage: maestro [-h] [-l LOGPATH] [-d DEBUG_LVL] [-c] [-v] {cancel,run,status} ...

The Maestro Workflow Conductor for specifiying, launching, and managing general
workflows.

positional arguments:  {cancel,run,status}
    cancel              Cancel all running jobs.
    run                 Launch a study based on a specification
    status              Check the status of a running study.

optional arguments:
  -h, --help            show this help message and exit
  -l LOGPATH, --logpath LOGPATH
                        Alternate path to store program logging.
  -d DEBUG_LVL, --debug_lvl DEBUG_LVL
                        Level of logging messages to be output:
                        5 - Critical
                        4 - Error
                        3 - Warning
                        2 - Info (Default)
                        1 - Debug
  -c, --logstdout       Log to stdout in addition to a file. [Default: True]
  -v, --version         show program's version number and exit
```



Maestro

```yaml
description:
    name: Build archive
    description: A simple archive building study

study:
    - name: generate
      description: creates some file
      run:
         cmd: |
           mkdir -p directory
           touch directory/file1.txt
```

```
$ maestro run buildarchive.yml
----------------------------------------
Submission attempts =       1
Submission restart limit =  1
Submission throttle limit = 0
Use temporary directory =   False
Hash workspaces =           False
Dry run enabled =           False
Output path =               /[…]/Build_archive_20220126-111006
----------------------------------------
Would you like to launch the study? [yn] y
Study launched successfully.
$
```

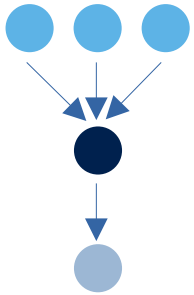


Maestro

```
description:
    name: Build archive
    description: A simple archive building study

study:
    - name: generate
      description: creates some file
      run:
         cmd: |
           mkdir -p directory
           touch directory/file1.txt
```

View progress with `maestro status <directory>`:

```
$ ls
Build_archive_20220126-111006  buildarchive.yml

$ maestro status Build_archive_20220126-111006
Step Name    Workspace    State     Run Time         Elapsed Time
-----------  -----------  --------  ---------------  ------------
 […]
generate     generate     FINISHED  0d:00h:00m:00s   0d:00h:00m:00s
```



Maestro

Study directory:

```
$ find Build_archive_20220126-111006
Build_archive_20220126-111006

Build_archive_20220126-111006/meta
Build_archive_20220126-111006/meta/metadata.yaml
Build_archive_20220126-111006/meta/parameters.yaml
Build_archive_20220126-111006/meta/study
Build_archive_20220126-111006/meta/study/env.pkl
Build_archive_20220126-111006/meta/environment.yaml
Build_archive_20220126-111006/logs
Build_archive_20220126-111006/logs/Build archive.log
Build_archive_20220126-111006/generate
Build_archive_20220126-111006/generate/generate.59117.out
Build_archive_20220126-111006/generate/generate.sh
Build_archive_20220126-111006/generate/generate.59117.err
Build_archive_20220126-111006/generate/directory
Build_archive_20220126-111006/generate/directory/file1.txt
Build_archive_20220126-111006/Build_archive.study.pkl
Build_archive_20220126-111006/status.csv
Build_archive_20220126-111006/batch.info
Build_archive_20220126-111006/Build archive.pkl
Build_archive_20220126-111006/Build_archive.txt
Build_archive_20220126-111006/buildarchive.yml
```

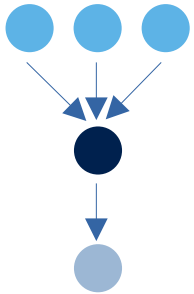
Parameters, environment, etc.

Maestro logs

Workspace

Maestro internal use

Study file



Maestro

```yaml
description:
    name: Build archive
    description: A simple archive building study

study:
    - name: generate
      description: creates some file
      run:
         cmd: |
           mkdir -p directory
           touch directory/file1.txt

    - name: build
      description: creates the tar file
      run:
         cmd: tar cvf archive.tar ../generate/directory
         depends: [generate]

    - name: compress
      description: compress the archive with gzip
      run:
         cmd: |
            ml gzip
            gzip -k $(compressoption) ../../build/archive.tar
         depends: [build]

global.parameters:
    compressoption:
        values: ['--fast', '--best']
        label: COPT.%%
```

Study overview

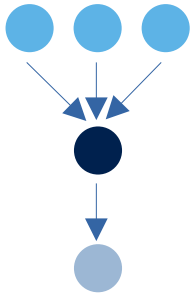
Step 1

Step 2

Step 3

- name
- description
- command(s)
- dependencies

Study parameters



Maestro

```
description:
    name: Build archive
    description: A simple archive building study

study:
    - name: generate
      description: creates the tar file
      run:
         cmd: |
           mkdir -p directory
           touch directory/file1.txt

    - name: build
      description: creates the tar file
      run:
         cmd: tar cvf archive.tar ../generate/directory
         depends: [generate]

    - name: compress
      description: compress the archive with gzip
      run:
         cmd: |
            ml gzip
            gzip -k $(compressoption) ../../build/archive.tar
         depends: [build]

global.parameters:
    compressoption:
        values: ['--fast', '--best']
        label: COPT.%%
```

Study overview

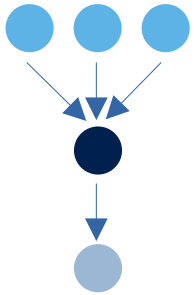
Step 1

Step 2

Step 3

- name
- description
- command(s)
- dependencies

Study parameters



Maestro

```yaml
description:
    name: Build archive
    description: A simple archive building study

study:
    - name: generate
      description: creates the tar file
      run:
         cmd: |
           mkdir -p directory
           touch directory/file1.txt

    - name: build
      description: creates the tar file
      run:
         cmd: tar cvf archive.tar ../generate/directory
         depends: [generate]

    - name: compress
      description: compress the archive with gzip
      run:
         cmd: |
            ml gzip
            gzip -k $(compressoption) ../../build/archive.tar
         depends: [build]

global.parameters:
    compressoption:
        values: ['--fast', '--best']
        label: COPT.%%
```

Study overview

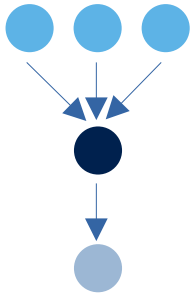
Step 1

Step 2

Step 3

- name
- description
- command(s)
- dependencies

Study parameters



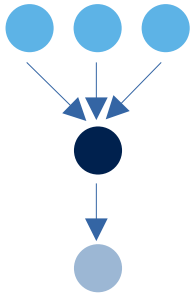
Maestro

Study status:

```
$ maestro status Build_archive_20220126-114700/
Step Name              Workspace    State     Run Time         Elapsed Time
---------------------  -----------  --------  ---------------  -------------
build                  build        FINISHED  0d:00h:00m:00s   0d:00h:00m:00s
compress_COPT.--best   COPT.--best  FAILED    0d:00h:00m:00s   0d:00h:00m:00s
compress_COPT.--fast   COPT.--fast  FINISHED  0d:00h:00m:00s   0d:00h:00m:00s
generate               generate     FINISHED  0d:00h:00m:00s   0d:00h:00m:00s
```

Study directory:

```
$ cd Build_archive_20220126-114700/compress
$ ls
COPT.--best  COPT.--fast
$ cd COPT.--best/
$ ls
compress_COPT.--best.86244.err
compress_COPT.--best.86244.out
compress_COPT.--best.sh
$ cat compress_COPT.--best.86244.err
gzip: ../../build/archive.tar.gz already exists;  not overwritten
```

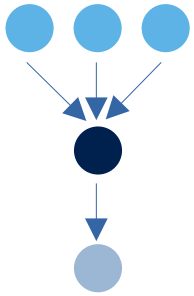


Maestro

```
description:
    name: Build archive
    description: A simple archive building study

batch:
    type:  slurm
    queue: debug
    host: localhost
    bank: ceci

study:
    […]

    - name: compress
      description: compress the archive with gzip
      run:
         cmd: |
            ml gzip
            gzip -k $(compressoption) ../../build/archive.tar
         depends: [build]
           nodes: 1
           procs: 2
           walltime: “10:00”

global.parameters:
    compressoption:
        values: ['--fast', '--best']
        label: COPT.%%
```

Scheduler options

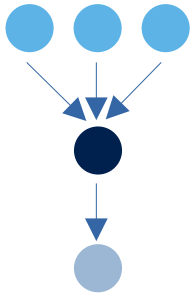
Job options



Maestro

Jobs are submitted

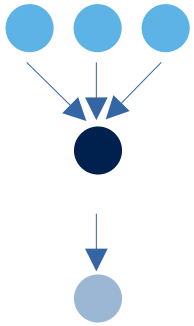
```
$ maestro status Build_archive_20220126-120732
Step Name              Workspace    State     Run Time         Elapsed Time
--------------------  -----------  --------  --------------  --------------
compress_COPT.--best  COPT.--best  FINISHED  --:--:--         0d:00h:01m:00s
build                 build        FINISHED  0d:00h:00m:00s  0d:00h:00m:00s
generate              generate     FINISHED  0d:00h:00m:00s  0d:00h:00m:00s
compress_COPT.--fast  COPT.--fast  FAILED    --:--:--         0d:00h:01m:00s

$ sacct

       JobID    JobName  Partition    Account  AllocCPUS      State
------------ ---------- ---------- ---------- ---------- ----------
70382817     compress_+      debug       ceci          1     FAILED
70382817.ba+      batch                  ceci          1     FAILED
70382817.ex+     extern                  ceci          1  COMPLETED
70382818     compress_+      debug       ceci          1  COMPLETED
70382818.ba+      batch                  ceci          1  COMPLETED
70382818.ex+     extern                  ceci          1  COMPLETED
```



Maestro

Submission scripts automatically generated:

```
$ cd Build_archive_20220126-120732/compress/COPT.--best/

$ cat compress_COPT.--best.slurm.sh
#!/bin/bash
#SBATCH –nodes=1
#SBATCH –partition=debug
#SBATCH –account=ceci
#SBATCH –time=10:00
#SBATCH –job-name="compress_COPT.--best"
#SBATCH –output="compress_COPT.--best.out"
#SBATCH –error="compress_COPT.--best.err"
#SBATCH --comment "compress the archive with gzip"

ml gzip

gzip -k --best ../../build/archive.tar
```

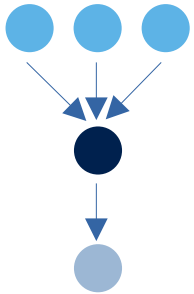


Maestro

```yaml
description:
    name: Build archive
    description: A simple archive building study

env:
    variables: # Static variables
        OUTPUT_PATH: helloworld
        PARAMS: ...



    dependencies:
      git:
        - name: Hello World
          path: ./hello
          url: https://github.com/dfr/helloworld
      paths:
        - name: input data
          path: ./data.txt

study:
    […]
```

Variables

Dependencies (software, cloned automatically) (data, study not launched if non existing)



Maestro

Maestro handles core functions of running a user's workflow

1. Run submission and monitoring

Maestro submits, monitors, and restart jobs. Maestro can also manage the amount of jobs submitted to the scheduler at a given time.

2. Workspace management

Maestro manages the study workspace creating files and ensuring data doesn't overwrite steps/studies.

3. Workflow Provenance

Maestro captures workflow provenance of what is run including the sampled parameters, study spec, and inputs.

https://computing.llnl.gov/sites/default/files/MaestroWF-slides-final.pdf

Workflows with Slurm
Workflows with GNU tools
Workflows with Maestrowf

Workflows with Slurm

Workflows with GNU tools

Workflows with Maestrowf

Job arrays, scripted submissions `--wrap`, `--parsable`, `envsubst`, job dependencies, job packing, job requeuing, job resubmission

Workflows with Slurm

Workflows with GNU tools

Workflows with Maestrowf

```
xargs,
envsubst
(again),
mapfile,
GNU Make,
GNU Parallel,
```

Workflows with Slurm

Workflows with GNU tools

Workflows with Maestrowf

installation,
syntax,
running,
monitoring,
workspaces,
scheduler interaction

Workflows with Slurm

Workflows with GNU tools

Workflows with Maestrowf

Help working smarter not harder