



# Submission script debugging and best practice





Goal of this session:

Learn some tips for debugging  
Slurm submission scripts



Part I. Debugging SBATCH options

Part II: Debugging script logic

# Reminders

- ! No variable expansion in #SBATCH directives
- ! Slurm stops reading directives after first command
- ! Parameters can be set in 3 places
  - script directives
  - command line arguments
  - environment variables

# Check Slurm's interpretation

```
$ sbatch --verbose --test-only submit.sh
```

- Makes visible options set in the environment
- Emphasizes mistakes in formatting or units
- Can help tracking typos in #SBATCH

# Check Slurm's interpretation

```
$ sbatch --verbose --test-only submit.sh
```

```
sbatch: defined options
```

```
sbatch: -----
```

```
sbatch: clusters           : lemaitre3
```

```
sbatch: mem                : 4G
```

```
sbatch: ntasks             : 1
```

```
sbatch: test-only          : set
```

```
sbatch: time                : 06:00:00
```

```
sbatch: verbose             : 1
```

```
sbatch: -----
```

```
sbatch: end of defined options
```

```
[...]
```

```
sbatch: Job 72819422 to start at 2023-09-14T21:52:05 using 1  
processors on nodes lm3-w071 in partition batch
```

# Use Vim syntax highlighting

```
1 #!/usr/bin/env bash
2
3 #SBATCH --job-name="test"
4 #SBATCH --error=error.out
5 #SBATCH --time="10:100:10"
6
7 module load ...
8
9 srun ....
```



Part I. Debugging SBATCH options

**Part II: Debugging script logic**



# Good practice: “strict mode”

```
#!/bin/bash

set -euo pipefail
IFS=$'\n\t'

# -e immediately exit if any command [1] has a non-zero exit status.
# -o pipefail as above for commands in pipe sequences
# -u a reference to an undefined variable is an error (prevent typos)
# IFS define word splitting characters (prevent common errors with
  spaces in filenames)
```

Make sure the script stops when any command returns an error, and make the job status “FAILED” rather than “COMPLETED”.

# Good practice: “strict mode”

```
#!/bin/bash

set -euo pipefail
IFS=$'\n\t'

cd $LOCASCRATCH
rm -rf *
[...]
```

Disaster prevented.

# Good practice: trapping errors

```
$ cat test.sh
#!/bin/bash
set -euo pipefail
trap 'echo "error: Script $0 failed line ${LINENO}: \"$BASH_COMMAND\": see above"' ERR
ls /nonexistent
```

Print script name, line, and command that failed.

Useful when scripts call other scripts...

# Good practice: trapping errors

```
$ cat test.sh
#!/bin/bash
set -euo pipefail
trap 'echo "error: Script $0 failed line ${LINENO}: \"$BASH_COMMAND\": see above"' ERR
ls /nonexistent

$ ./test.sh

ls: cannot access /nonexistent: No such file or directory
error: Script ./test.sh failed line 3: "ls /nonexistent": with error message above
```

Print script name, line, and command that failed.

Useful when scripts call other scripts...

# Good practice: errors to stderr

```
$ cat test.sh
#!/bin/bash

#SBATCH --output=out.txt
#SBATCH --error=err.txt

echo "some comment"
echo >&2 "Some error"

$ cat out.txt

some comment

$ cat err.txt

some error
```

Write errors to `stderr` so they are filtered to the `--error` file.



# Bash verbosity

---

```
$ cat test.sh
#!/bin/bash -vx
A=(/usr/l*)
ls -d ${A[@]}

$ ./test.sh

A=(/usr/l*)
+ A=(/usr/l*)
ls -d ${A[@]}
+ ls -d /usr/lib /usr/libexec /usr/local

/usr/lib /usr/libexec /usr/local
```

Enable “trace” mode with `-vx` to see details of script execution.



# Dry run (syntax check)

```
$ cat test.sh
#!/bin/bash -vn
A=(/usr/l*)
ls -d ${A[@]}

$ ./test.sh

A=(/usr/l*)
ls -d ${A[@]}
```

Check syntax with `-vn` or use Shellcheck

# ShellCheck

finds bugs in your shell scripts.

You can `cabal`, `apt`, `dnf`, `pkg` OR `brew` install it locally right now.

Paste a script to try it out:

```
Your Editor (Ace)
Load random example
1 #!/bin/bash
2
3 A=(usr/l*)
4 ls -d ${A[@]}
5
```

```
ShellCheck Output
$ shellcheck myscript
Line 4:
ls -d ${A[@]}
    ^-- SC2068 (error): Double quote array expansions to avoid re-splitting elements.
$
```





# Line by line execution

```
$ cat test.sh
#!/bin/bash -v
trap read debug
A=(/usr/l*)
ls -d ${A[@]}

$ ./test.sh

A=(/usr/l*)
read
<ENTER>
ls -d ${A[@]}
read
<ENTER>
/usr/lib      /usr/libexec  /usr/local
```



# Overall strategy

- Instrument your submission script with tracing and error catching
- If job array : submit a 2-jobs array first
- Submit to debug partition
- Run `salloc` with the same parameters then source the submission script line by line in trace mode