SLURM for users

Quick start

Simply copy this example.slurm and adapt it to your needs.

SLURM partitions

In this doc we will use an example setup with these 2 cluster names:

- `dev`
- `prod`

To find out the hostname of the nodes and their availability, use:

```
sinfo -p dev
sinfo -p prod
```

Slurm configuration is at `/opt/slurm/etc/slurm.conf`.

Wait time for resource granting

```
squeue -u `whoami` --start
```

will show when any pending jobs are scheduled to start.

They may start sooner if others cancel their reservations before the end of the reservation.

Request allocation via dependency

To schedule a new job when one more of the currently scheduled job ends (regardless of whether it still running or not started yet), use the dependency mechanism, by telling `sbatch` to start the new job once the currently running job succeeds, using:

```
sbatch --dependency=CURRENTLY_RUNNING_JOB_ID tr1-13B-round1.slurm
```

Using `--dependency` may lead to shorter wait times that using `--begin`, since if the time passed to `--begin` allows even for a few minutes of delay since the stopping of the last job, the scheduler may already start some other jobs even if their priority is lower than our job. That's because the scheduler ignores any jobs with `--begin` until the specified time arrives.

Make allocations at a scheduled time

To postpone making the allocation for a given time, use:

```
salloc --begin HH:MM MM/DD/YY
```

Same for `sbatch`.

It will simply put the job into the queue at the requested time, as if you were to execute this command at this time. If resources are available at that time, the allocation will be given right away. Otherwise it'll be queued up.

Sometimes the relative begin time is useful. And other formats can be used. Examples:

```
--begin now+2hours
--begin=16:00
--begin=now+1hour
--begin=now+60  # seconds by default
--begin=2010-01-20T12:34:00
```

the time-units can be `seconds` (default), `minutes`, `hours`, `days`, or `weeks`:

Preallocated node without time 60min limit

This is very useful for running repetitive interactive experiments - so one doesn't need to wait for an allocation to progress. so the strategy is to allocate the resources once for an extended period of time and then running interactive `srun` jobs using this allocation.

set `--time` to the desired window (e.g. 6h):

```
salloc --partition=dev --nodes=1 --ntasks-per-node=1 --cpus-per-task=96 --gres=gpu:8 --time=
salloc: Pending job allocation 1732778
salloc: job 1732778 queued and waiting for resources
salloc: job 1732778 has been allocated resources
salloc: Granted job allocation 1732778
```

now use this reserved node to run a job multiple times, by passing the job id of `salloc`:

```
srun --jobid $SLURM_JOBID --pty bash
```

if run from inside `bash` started via `salloc`. But it can be started from another shell, but then explicitly set `--jobid`.

if this `srun` job timed out or manually exited, you can re-start it again in this same reserved node.

`srun` can, of course, call the real training command directly and not just `bash`.

Important: when allocating a single node, the allocated shell is not on the node (it never is). You have to find out the hostname of the node (reports when giving the allocation or via `squeue` and `ssh` to it.

When finished, to release the resources, either exit the shell started in `salloc` or `scancel JOBID`.

This reserved node will be counted towards hours usage the whole time it's allocated, so release as soon as done with it.

Actually, if this is just one node, then it's even easier to not use `salloc` but to use `srun` in the first place, which will both allocate and give you the shell to use:

```
srun --pty --partition=dev --nodes=1 --ntasks=1 --cpus-per-task=96 --gres=gpu:8 --time=60 ba
```

Hyper-Threads

By default, if the cpu has hyper-threads (HT), SLURM will use it. If you don't want to use HT you have to specify `--hint=nomultithread`.

footnote: HT is Intel-specific naming, the general concept is simultaneous multithreading (SMT)

For example for a cluster with with 2 cpus per node with 24 cores and 2 hyper-threads each, there is a total of 96 hyper-threads or 48 cpu-cores available. Therefore to utilize the node fully you'd need to configure either:

```
#SBATCH --cpus-per-task=96
```

or if you don't want HT:

```
#SBATCH --cpus-per-task=48
#SBATCH --hint=nomultithread
```

This last approach will allocate one thread per core and in this mode there are only 48 cpu cores to use.

Note that depending on your application there can be quite a performance difference between these 2 modes. Therefore try both and see which one gives you a better outcome.

On some setups like AWS the all-reduce throughput degrades dramatically when `--hint=nomultithread` is used! Whereas on some other setups the opposite is true - the throughput is worse without HT!

Reuse allocation

e.g. when wanting to run various jobs on identical node allocation.

In one shell:

```
salloc --partition=prod --nodes=16 --ntasks=16 --cpus-per-task=96 --gres=gpu:8 --time=3:00:
echo $SLURM_JOBID
```

In another shell:

```
export SLURM_JOBID=<JOB ID FROM ABOVE>
srun --jobid=$SLURM_JOBID ...
```

You may need to set `--gres=gpu:0` to run some diagnostics job on the nodes. For example, let's check shared memory of all the hosts:

```
srun --jobid 631078 --gres=gpu:0 bash -c 'echo $(hostname) $(df -h | grep shm)'
```

Specific nodes selection

To exclude specific nodes (useful when you know some nodes are broken, but are still in IDLE state):

```
sbatch --exclude nodeA,nodeB
```

or via: `#SBATCH --exclude ...`

To use specific nodes:

```
sbatch --nodelist= nodeA,nodeB
```

can also use the short `-w` instead of `--nodelist`

The administrator could also define a `feature=example` in `slurm.conf` and then a user could ask for that subset of nodes via `--constraint=example`

Signal the running jobs to finish

Since each SLURM run has a limited time span, it can be configured to send a signal of choice to the program a desired amount of time before the end of the allocated time.

```
--signal=[[R][B]:]<sig_num>[@<sig_time>]
```

TODO: need to experiment with this to help training finish gracefully and not start a new cycle after saving the last checkpoint.

Detailed job info

While most useful information is preset in various `SLURM_*` env vars, sometimes the info is missing. In such cases use:

```
scontrol show -d job $SLURM_JOB_ID
```

and then parse out what's needed.

For a job that finished its run use:

```
sacct -j JOBID
```

e.g. with more details, depending on the partition:

```
sacct -u `whoami` --partition=dev  -ojobid,start,end,state,exitcode --format nodelist%300  -
sacct -u `whoami` --partition=prod -ojobid,start,end,state,exitcode --format nodelist%300  -
```

show jobs

Show only my jobs:

```
squeue -u `whoami`
```

Show jobs by job id:

```
squeue -j JOBID
```

Show jobs of a specific partition:

```
squeue --partition=dev
```

Aliases

Handy aliases

```
alias myjobs='squeue -u `whoami` -o "%.16i %9P %26j %.8T %.10M %.8l %.6D %.20S %R"'
alias groupjobs='squeue -u foo,bar,tar -o "%.16i %u %9P %26j %.8T %.10M %.8l %.6D %.20S %R"'
alias myjobs-pending="squeue -u `whoami` --start"
alias idle-nodes="sinfo -p prod -o '%A'"
```

Zombies

If there are any zombies left behind across nodes, send one command to kill them all.

```
srun pkill python
```

Detailed Access to SLURM Accounting

`sacct` displays accounting data for all jobs and job steps in the Slurm job accounting log or Slurm database.

So this is a great tool for analysing past events.

For example, to see which nodes were used to run recent gpu jobs:

```
sacct -u `whoami` --partition=dev -ojobid,start,end,state,exitcode --format nodelist%300
```

`%300` here tells it to use a 300 char width for the output, so that it's not truncated.

See `man sacct` for more fields and info fields.

Queue

Cancel job

To cancel a job:

```
scancel [jobid]
```

To cancel all of your jobs:

```
scancel -u <userid>
```

To cancel all of your jobs on a specific partition:

```
scancel -u <userid> -p <partition>
```

Tips

- if you see that `salloc`'ed interactive job is scheduled to run much later than you need, try to cancel the job and ask for shorter period - often there might be a closer window for a shorter time allocation.

Logging

If we need to separate logs to different log files per node add `%N` (for short hostname) so that we have:

```
#SBATCH --output=%x-%j-%N.out
```

That way we can tell if a specific node misbehaves - e.g. has a corrupt GPU. This is because currently pytorch doesn't log which node / gpu rank triggered an exception.

Hoping it'll be a built-in feature of pytorch https://github.com/pytorch/pytorch/issues/63174 and then one won't need to make things complicated on the logging side.

Show the state of nodes

```
sinfo -p PARTITION
```

Very useful command is:

```
sinfo -s
```

and look for the main stat, e.g.:

```
NODES(A/I/O/T) "allocated/idle/other/total".
597/0/15/612
```

So here 597 out of 612 nodes are allocated. 0 idle and 15 are not available for whatever other reasons.

```
sinfo -p gpu_p1 -o "%A"
```

gives:

```
NODES(A/I)
236/24
```

so you can see if any nodes are available on the 4x v100-32g partition (`gpu_p1`)

To check a specific partition:

```
sinfo -p gpu_p1 -o "%A"
```

See the table at the top of this document for which partition is which.

sinfo states

- idle: no jobs running
- alloc: nodes are allocated to jobs that are currently executing
- mix: the nodes have some of the CPUs allocated, while others are idle
- drain: the node is unavailable due to an administrative reason
- drng: the node is running a job, but will after completion not be available due to an administrative reason

drained nodes

To see all drained nodes and the reason for drainage (edit `%50E` to make the reason field longer/shorter)

```
% sinfo -R -o "%50E %12U %19H %6t %N"
```

or just `-R` if you want it short:

```
% sinfo -R
```

Job arrays

To run a sequence of jobs, so that the next slurm job is scheduled as soon as the currently running one is over in 20h we use a job array.

Let's start with just 10 such jobs:

```
sbatch --array=1-10%1 array-test.slurm
```

`%1` limits the number of simultaneously running tasks from this job array to 1. Without it it will try to run all the jobs at once, which we may want sometimes (in which case remove %1), but when training we need one job at a time.

Alternatively, as always this param can be part of the script:

```
#SBATCH --array=1-10%1
```

Here is toy slurm script, which can be used to see how it works:

```
#!/bin/bash
#SBATCH --job-name=array-test
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1          # crucial - only 1 task per dist per node!
#SBATCH --cpus-per-task=1            # number of cores per tasks
#SBATCH --time 00:02:00              # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out           # output file name
#SBATCH --error=%x-%j.out            # error file name (same to watch just one file)
#SBATCH --partition=dev

echo $SLURM_JOB_ID
echo "I am job ${SLURM_ARRAY_JOB_ID}_${SLURM_ARRAY_TASK_ID}"
date
sleep 10
date
```

Note `$SLURM_ARRAY_JOB_ID` is the same as `$SLURM_JOB_ID`, and `$SLURM_ARRAY_TASK_ID` is the index of the job.

To see the jobs running:

```
$ squeue -u `whoami` -o "%.10i %9P %26j %.8T %.10M %.6D %.20S %R"
     JOBID PARTITION                       NAME    STATE       TIME  NODES           START_T
591970_[2-   dev             array-test  PENDING       0:00      1  2021-07-28T20:01:06 (Job
```

now job 2 is running.

To cancel the whole array, cancel the job id as normal (the number before _):

```
scancel 591970
```

To cancel a specific job:

```
scancel 591970_2
```

If it's important to have the log-file contain the array id, add `%A_%a`:

```
#SBATCH --output=%x-%j.%A_%a.log
```

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

Job Array Trains and their Suspend and Release

In this recipe we accomplish 2 things:

1. Allow modification to the next job's slurm script
2. Allow suspending and resuming job arrays w/o losing the place in the queue when not being ready to continue running a job

SLURM is a very unforgiving environment where a small mistake can cost days of waiting time. But there are strategies to mitigate some of this harshness.

SLURM jobs have a concept of "age" in the queue which besides project priority governs when a job gets scheduled to run. If your have just scheduled a new job it has no "age" and will normally be put to run last compared to jobs that have entered the queue earlier. Unless of course this new job comes from a high priority project in which case it'll progress faster.

So here is how one can keep the "age" and not lose it when needing to fix something in the running script or for example to switch over to another script.

The idea is this:

1. `sbatch` a long job array, e.g., `-array=1-50%1`
2. inside the slurm script don't have any code other than `source another-script.slurm` - so now you can modify the target script or symlink to another script before the next job starts
3. if you need to stop the job array train - don't cancel it, but suspend it without losing your place in a queue
4. when ready to continue - unsuspend the job array - only the time while it was suspended is not counted towards its age, but all the previous age is retained.

The only limitation of this recipe is that you can't change the number of nodes, time and hardware and partition constraints once the job array was launched.

Here is an example:

Create a job script:

```
$ cat train-64n.slurm
#!/bin/bash
#SBATCH --job-name=tr8-104B
#SBATCH --nodes=64
#SBATCH --ntasks-per-node=1          # crucial - only 1 task per dist per node!
#SBATCH --cpus-per-task=96           # number of cores per tasks
#SBATCH --gres=gpu:8                 # number of gpus
#SBATCH --time 20:00:00              # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out           # output file name
#SBATCH --partition=dev

source tr8-104B-64.slurm
```

Start it as:

```
sbatch --array=1-50%1 train-64.slurm
```

Now you can easily edit `tr8-104B-64.slurm` before the next job run and either let the current job finish if it's desired or if you need to abort it, just kill the currently running job, e.g. `1557903_5` (not job array `1557903`) and have the train pick up where it left, but with the edited script.

The nice thing is that this requires no changes to the original script (`tr8-104B-64.slurm` in this example), and the latter can still be started on its own.

Now, what if something is wrong and you need 10min or 10h to fix something. In this case we suspend the train using:

```
scontrol hold <jobid>
```

with being either a "normal" job, the id of a job array or the id for a job array step

and then when ready to continue release the job:

```
scontrol release <jobid>
```

Troubleshooting

Mismatching nodes number

If the pytorch launcher fails it often means that the number of SLURM nodes and the launcher nodes are mismatching, e.g.:

```
grep -ir nodes= tr123-test.slurm
#SBATCH --nodes=40
NNODES=64
```

This won't work. They have to match.

You can add a sanity check to your script:

```
#!/bin/bash
#SBATCH --job-name=test-mismatch
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=1          # crucial - only 1 task per dist per node!
#SBATCH --cpus-per-task=96           # number of cores per tasks
#SBATCH --gres=gpu:8                 # number of gpus
#SBATCH --time 0:05:00               # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out           # output file name
#SBATCH --partition=prod

[...]

NNODES=2

# sanity check for having NNODES and `#SBATCH --nodes` match, assuming you use NNODES variab
if [ "$NNODES" != "$SLURM_NNODES" ]; then
    echo "Misconfigured script: NNODES=$NNODES != SLURM_NNODES=$SLURM_NNODES"
    exit 1
fi

[...]
```

or you could just do:

```
#SBATCH --nodes=2
[...]
NNODES=$SLURM_NNODES
```

and then it will always be correct

Find faulty nodes and exclude them

Sometimes a node is broken, which prevents one from training, especially since restarting the job often hits the same set of nodes. So one needs to be able to isolate the bad node(s) and exclude it from `sbatch`.

To find a faulty node, write a small script that reports back the status of the desired check.

For example to test if cuda is available on all nodes:

```
python -c 'import torch, socket; print(f"{socket.gethostname()}: {torch.cuda.is_available()]
```

and to only report the nodes that fail:

```
python -c 'import torch, socket; torch.cuda.is_available() or print(f"Broken node: {socket.g
```

Of course, the issue could be different - e.g. gpu can't allocate memory, so change the test script to do a small allocation on cuda. Here is one way:

```
python -c "import torch; torch.ones(1000,1000).cuda()"
```

But since we need to run the test script on all nodes and not just the first node, the slurm script needs to run it via `srun`. So our first diagnostics script can be written as:

```
srun --jobid $SLURM_JOBID bash -c 'python -c "import torch, socket; print(socket.gethostname
```

I slightly changed it, due to an issue with quotes.

You can always convert the one liner into a real script and then there is no issue with quotes.

```
$ cat << EOT >> test-nodes.py
#!/usr/bin/env python
import torch, socket
print(socket.gethostname(), torch.cuda.is_available())
EOT
$ chmod a+x ./test-nodes.py
```

Now let's create a driver slurm script. Use a few minutes time for this test so that SLURM yields it faster:

```
#!/bin/bash
#SBATCH --job-name=test-nodes
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=1          # crucial - only 1 task per dist per node!
#SBATCH --cpus-per-task=96           # number of cores per tasks
#SBATCH --gres=gpu:8                 # number of gpus
#SBATCH --time 0:05:00               # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out           # output file name
#SBATCH --partition=prod

source $six_ALL_CCFRWORK/start-prod
srun --jobid $SLURM_JOBID ./test-nodes.py
```

Once it runs check the logs to see if any reported `False`, those are the nodes you want to exclude.

Now once the faulty node(s) is found, feed it to `sbatch`:

```
sbatch --exclude=hostname1,hostname2 ...
```

and `sbatch` will exclude the bad nodes from the allocation.

Additionally please report the faulty nodes to `#science-support` so that they get replaced

Here are a few more situations and how to find the bad nodes in those cases:

Broken NCCL

If you're testing something that requires distributed setup, it's a bit more complex. Here is a slurm script that tests that NCCL works. It sets up NCCL and checks that barrier works:

```
#!/bin/bash
#SBATCH --job-name=test-nodes-nccl
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=1          # crucial - only 1 task per dist per node!
#SBATCH --cpus-per-task=96           # number of cores per tasks
#SBATCH --gres=gpu:8                 # number of gpus
#SBATCH --time 0:05:00               # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out           # output file name
#SBATCH --partition=prod

source $six_ALL_CCFRWORK/start-prod

NNODES=2

GPUS_PER_NODE=4
MASTER_ADDR=$(scontrol show hostnames $SLURM_JOB_NODELIST | head -n 1)
MASTER_PORT=6000

export LAUNCHER="python -u -m torch.distributed.launch \
    --nproc_per_node $GPUS_PER_NODE \
    --nnodes $NNODES \
    --master_addr $MASTER_ADDR \
    --master_port $MASTER_PORT \
    "

export SCRIPT=test-nodes-nccl.py

cat << EOT > $SCRIPT
#!/usr/bin/env python
import torch.distributed as dist
import torch
import socket
import os
import fcntl

def printflock(*msgs):
    """ print """
    with open(__file__, "r") as fh:
        fcntl.flock(fh, fcntl.LOCK_EX)
        try:
            print(*msgs)
        finally:
            fcntl.flock(fh, fcntl.LOCK_UN)

local_rank = int(os.environ["LOCAL_RANK"])
torch.cuda.set_device(local_rank)
dist.init_process_group("nccl")
header = f"{socket.gethostname()}-{local_rank}"
try:
    dist.barrier()
    printflock(f"{header}: NCCL {torch.cuda.nccl.version()} is OK")
except:
    printflock(f"{header}: NCCL {torch.cuda.nccl.version()} is broken")
    raise
EOT

echo $LAUNCHER --node_rank $SLURM_PROCID $SCRIPT

srun --jobid $SLURM_JOBID bash -c '$LAUNCHER --node_rank $SLURM_PROCID $SCRIPT'
```

The script uses `printflock` to solve the interleaved print outputs issue.

GPU Memory Check

This tests if each GPU on the allocated nodes can successfully allocate 77Gb (e.g. to test 80GB A100s) (have to subtract a few GBs for cuda kernels).

```
import torch, os
import time
import socket
hostname = socket.gethostname()

local_rank = int(os.environ["LOCAL_RANK"]);

gbs = 77
try:
    torch.ones((gbs*2**28)).cuda(local_rank).contiguous() # alloc on cpu, then move to gpu
    print(f"{local_rank} {hostname} is OK")
```

```
except:
    print(f"{local_rank} {hostname} failed to allocate {gbs}GB DRAM")
    pass

time.sleep(5)
```

Broken Network

Yet another issue with a node is when its network is broken and other nodes fail to connect to it.

You're likely to experience it with an error similar to:

```
work = default_pg.barrier(opts=opts)
RuntimeError: NCCL error in: /opt/conda/conda-bld/pytorch_1616554793803/work/torch/lib/c10d/
ncclSystemError: System call (socket, malloc, munmap, etc) failed.
```

Here is how to debug this issue:

1. Add:

   ```
   export NCCL_DEBUG=INFO
   ```

   before the `srun` command and re-run your slurm script.

2. Now study the logs. If you find:

   ```
   r11i6n2:486514:486651 [1] include/socket.h:403 NCCL WARN Connect to 10.148.3.247<56821>
   ```

   Let's see which node refuses to accept connections. We get the IP address from the error above and reverse resolve it to its name:

   ```
   nslookup 10.148.3.247
   247.3.148.10.in-addr.arpa       name = r10i6n5.ib0.xa.idris.fr.
   ```

   Add `--exclude=r10i6n5` to your `sbatch` command and report it to JZ admins.

Run py-spy or any other monitor program across all nodes

When dealing with hanging, here is how to automatically log `py-spy` traces for each process.

Of course, this same process can be used to run some command for all nodes of a given job. i.e. it can be used to run something during the normal run - e.g. dump all the memory usage in each process via `nvidia-smi` or whatever other program is needed to be run.

```
cd ~/prod/code/tr8b-104B/bigscience/train/tr11-200B-ml/

salloc --partition=prod --nodes=40 --ntasks-per-node=1 --cpus-per-task=96 --gres=gpu:8 --ti
```

```
bash 200B-n40-bf16-mono.slurm
```

In another shell get the JOBID for the above `salloc`:

```
squeue -u `whoami` -o "%.16i %9P %26j %.8T %.10M %.8l %.6D %.20S %R"
```

adjust jobid per above and the nodes count (XXX: probably can remove `--nodes=40` altogether and rely on `salloc` config):

```
srun --jobid=2180718 --gres=gpu:0 --nodes=40 --tasks-per-node=1 --output=trace-%N.out sh -c
```

now all `py-spy` traces go into the `trace-$nodename.out` files under `cwd`.

The key is to use `--gres=gpu:0` or otherwise the 2nd `srun` will block waiting for the first one to release the gpus.

Also the assumption is that some conda env that has `py-spy` installed got activated in `~/.bashrc`. If yours doesn't already do that, add the instruction to load the env to the above command, before the `py-spy` command - it'll fail to find it otherwise.

Don't forget to manually release the allocation when this process is done.

Convert SLURM_JOB_NODELIST into a hostfile

Some multi-node launchers require a `hostfile` - here is how to generate one:

```
# autogenerate the hostfile for deepspeed
# 1. deals with: SLURM_JOB_NODELIST in either of 2 formats:
# r10i1n8,r10i2n0
# r10i1n[7-8]
# 2. and relies on SLURM_STEP_GPUS=0,1,2... to get how many gpu slots per node
#
# usage:
# makehostfile > hostfile
function makehostfile() {
perl -le '$slots=split /,/, $ENV{"SLURM_STEP_GPUS"}; $_=$ENV{"SLURM_JOB_NODELIST"}; if (/^(
}
```

Environment variables

You can always do:

```
export SOMEKEY=value
```

from the slurm script to get a desired environment variable passed to the program launched from it.

And you can also add to the top of the slurm script:

```
#SBATCH --export=ALL
```

The launched program will see all the environment variables visible in the shell where it was launched from.

Crontab Emulation

One of the most important Unix tools is the crontab, which is essential for being able to schedule various jobs. It however usually is absent from SLURM environment. Therefore one must emulate it. Here is how.

For this presentation we are going to use `$WORK/cron/` as the base directory. And that you have an exported environment variable `WORK` pointing to some location on your filesystem - if you use Bash you can set it up in your `~/.bash_profile` or if a different shell is used use whatever startup equivalent file is.

1. A self-perpetuating scheduler job

We will use `$WORK/cron/scheduler` dir for scheduler jobs, `$WORK/cron/cron.daily` for daily jobs and `$WORK/cron/cron.hourly` for hourly jobs:

```
$ mkdir -p $WORK/cron/scheduler
$ mkdir -p $WORK/cron/cron.daily
$ mkdir -p $WORK/cron/cron.hourly
```

Now copy these two slurm script in `$WORK/cron/scheduler`: - cron-daily.slurm - cron-hourly.slurm

after editing those to fit your specific environment's account and partition information.

Now you can launch the crontab scheduler jobs:

```
$ cd $WORK/cron/scheduler
$ sbatch cron-hourly.slurm
$ sbatch cron-daily.slurm
```

This is it, these jobs will now self-perpetuate and usually you don't need to think about it again unless there is an even that makes SLURM lose all its jobs.

2. Daily and Hourly Cronjobs

Now whenever you want some job to run once a day, you simply create a slurm job and put it into the `$WORK/cron/cron.daily` dir.

Here is an example job that runs daily to update the `mlocate` file index:

```
$ cat $WORK/cron/cron.daily/mlocate-update.slurm
#!/bin/bash
#SBATCH --job-name=mlocate-update    # job name
#SBATCH --ntasks=1                   # number of MP tasks
#SBATCH --nodes=1
#SBATCH --hint=nomultithread         # we get physical cores not logical
#SBATCH --time=1:00:00               # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out           # output file name
#SBATCH --partition=PARTITION     # edit me
#SBATCH --account=GROUP@PARTITION # edit me

set -e
date
echo "updating mlocate db"
/usr/bin/updatedb -o $WORK/lib/mlocate/work.db -U $WORK --require-visibility 0
```

This builds an index of the files under `$WORK` which you can then quickly query with:

```
/usr/bin/locate -d $WORK/lib/mlocate/work.db pattern
```

To stop running this job, just move it out of the `$WORK/cron/cron.daily` dir.

The same principle applies to jobs placed into the `$WORK/cron/cron.hourly` dir. These are useful for running something every hour.

Please note that this crontab implementation is approximate timing-wise, due to various delays in SLURM scheduling they will run approximately every hour and every day. You can recode these to ask SLURM to start something at a more precise time if you have to, but most of the time the just presented method works fine.

Additionally, you can code your own variations to meet specific needs of your project, e.g., every-30min or every-12h jobs.

3. Cleanup

Finally, since every cron launcher job will leave behind a log file (which is useful if for some reason things don't work), you want to create a cronjob to clean up these logs. Otherwise you may run out of inodes - these logs files are tiny, but there could be tens of thousands of those.

You could use something like this in a daily job.

```
find $WORK/cron -name "*.out" -mtime +7 -exec rm -f {} +
```

Please note that it's set to only delete files that are older than 7 days, in case you need the latest logs for diagnostics.

Nuances

The scheduler runs with Unix permissions of the person who launched the SLRUM cron scheduler job and so all other SLURM scripts launched by that cron job.

Self-perpetuating SLURM jobs

The same approach used in building a scheduler can be used for creating stand-alone self-perpetuating jobs.

For example:

```
#!/bin/bash
#SBATCH --job-name=watchdog          # job name
#SBATCH --ntasks=1                   # number of MP tasks
#SBATCH --nodes=1
#SBATCH --time=0:30:00               # maximum execution time (HH:MM:SS)
#SBATCH --output=%x-%j.out           # output file name
#SBATCH --partition=PARTITION        # edit me

# ensure to restart self first 1h from now
RUN_FREQUENCY_IN_HOURS=1
sbatch --begin=now+${RUN_FREQUENCY_IN_HOURS}hour watchdog.slurm

... do the watchdog work here ...
```

and you launch it once with:

```
sbatch watchdog.slurm
```

This then will immediately schedule itself to be run 1 hour from the launch time and then the normal job work will be done. Regardless of whether the rest of the job will succeed or fail, this job will continue relaunching itself approximately once an hour. This is imprecise due to scheduler job starting overhead and node availability issues. But if there is a least one spare node available and the job itself is quick to finish the requirement to run at an approximate frequency should be sufficient.

As the majority of SLURM environment in addition to the expensive GPU nodes also provide much cheaper CPU-only nodes, you should choose a CPU-only SLURM partition for any jobs that don't require GPUs to run.

Getting information about the job

From within the slurm file one can access information about the current job's allocations.

Getting allocated hostnames and useful derivations based on that:

```
export HOSTNAMES=$(scontrol show hostnames "$SLURM_JOB_NODELIST")
export NUM_NODES=$(scontrol show hostnames "$SLURM_JOB_NODELIST" | wc -l)
export MASTER_ADDR=$(scontrol show hostnames "$SLURM_JOB_NODELIST" | head -n 1)
```

Convert compact node list to expanded node list

Sometimes you get SLURM tools give you a string like: `node-[42,49-51]` which will require some coding to expand it into `node-42,node-49,node-50,node-51`, but there is a special tool to deal with that:

```
$ scontrol show hostnames node-[42,49-51]
node-42
node-49
node-50
node-51
```

Voila!

case study: this is for example useful if you want get a list of nodes that were drained because the job was too slow to exit, but really there is no real problem with the nodes. So this one-liner will give you the list of such nodes in an expanded format which you can then script to loop over this list to undrain these nodes after perhaps checking that the processes have died by this time:

```
sinfo -R | grep "Kill task failed" | perl -lne '/(node-.*[\d\]]+)/ && print $1' | xargs -n1
```

Overcoming the lack of group SLURM job ownership

SLURM runs on Unix, but surprisingly its designers haven't adopted the concept of group ownership with regards to SLURM jobs. So if a member of your team started an array of 10 jobs 20h each, and went on vacation - unless you have `sudo` access you now can't do anything to stop those jobs if something is wrong.

I'm yet to find why this is so, but so far we have been using a kill switch workaround. You have to code it in your framework. For example, see how it was implemented in Megatron-Deepspeed (Meg-DS). The program polls for a pre-configured at start up path on the filesystem and if it finds a file there, it exits.

So if we start Meg-DS with `--kill-switch-path $WORK/tmp/training17-kill-switch` and then at any point we need to kill the SLURM job, we simply do:

```
touch $WORK/tmp/training17-kill-switch
```

and the next time the program gets to check for this file it'll detect the event and will exit voluntarily. If you have a job array, well, you will have to wait until each job starts, detects the kill switch and exits.

Of course, don't forget to remove it when you're done stopping the jobs.

```
rm $WORK/tmp/training17-kill-switch
```

Now, this doesn't always work. If the job is hanging, it'll never come to the point of checking for kill-switch and the only solution here is to contact the sysadmins to kill the job for you. Sometimes if the hanging is a simple case pytorch's distributed setup will typically auto-exit after 30min of preset timeout time, but it doesn't always work.