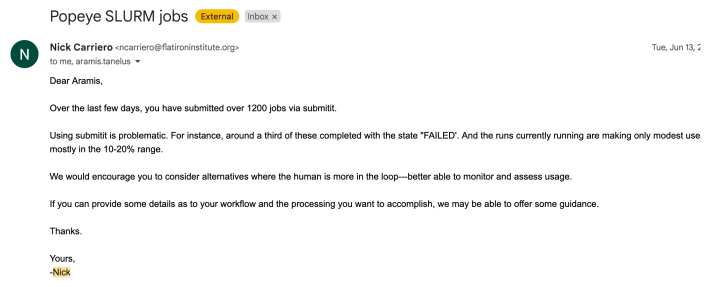
Intro to disBatch

Efficiently schedule and run parallel jobs

2023-08-06

What is disBatch?

- A tool for running many commands across a pool of nodes on the cluster
  - Only requires one resource allocation upfront (saves time in queue)
  - Saves time on initializing shared resources (e.g. copying datasets to local)
  - Saves time spent on reading concerned emails


Popeye SLURM jobs External Inbox ×

Nick Carriero <ncarriero@flatironinstitute.org>
to me, aramis.tanelus
Tue, Jun 13, 2

Dear Aramis,

Over the last few days, you have submitted over 1200 jobs via submitit.

Using submitit is problematic. For instance, around a third of these completed with the state "FAILED'. And the runs currently running are making only modest use mostly in the 10-20% range.

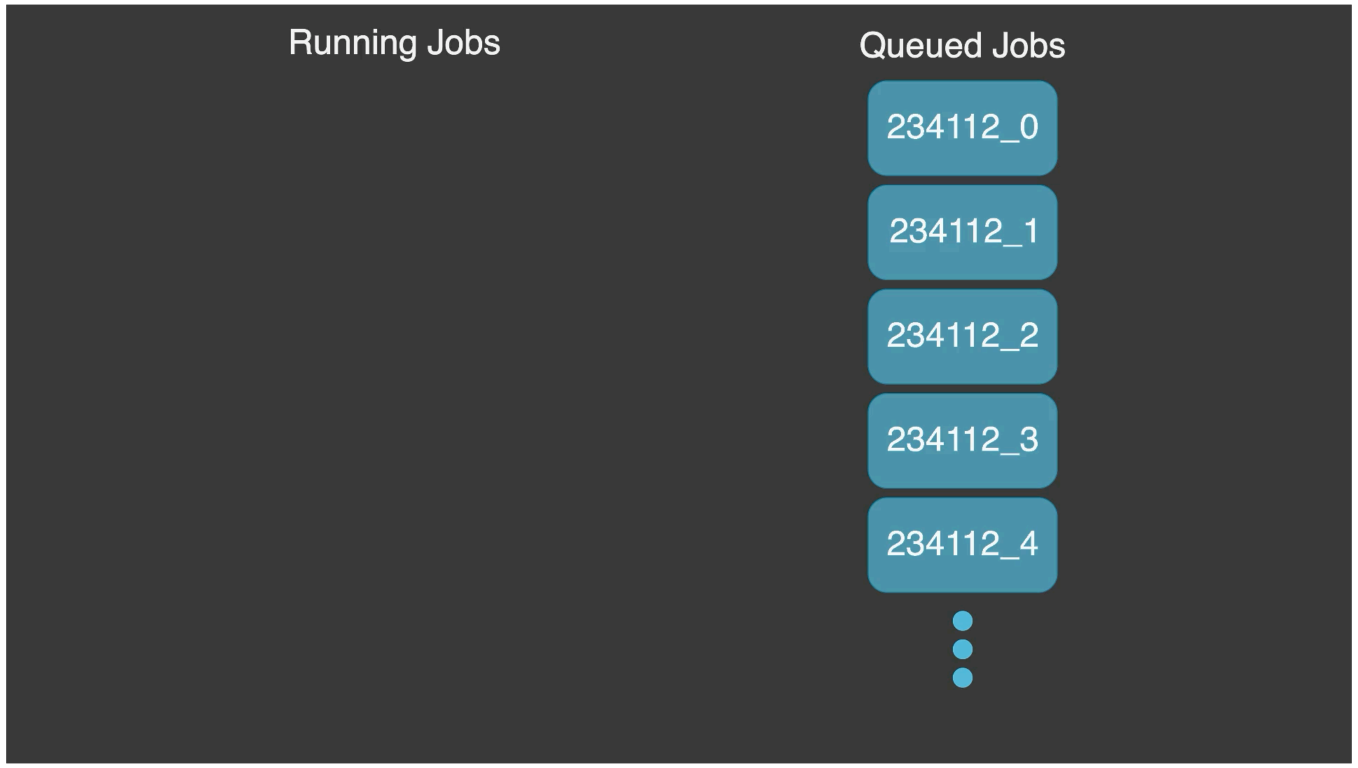
We would encourage you to consider alternatives where the human is more in the loop---better able to monitor and assess usage.

If you can provide some details as to your workflow and the processing you want to accomplish, we may be able to offer some guidance.

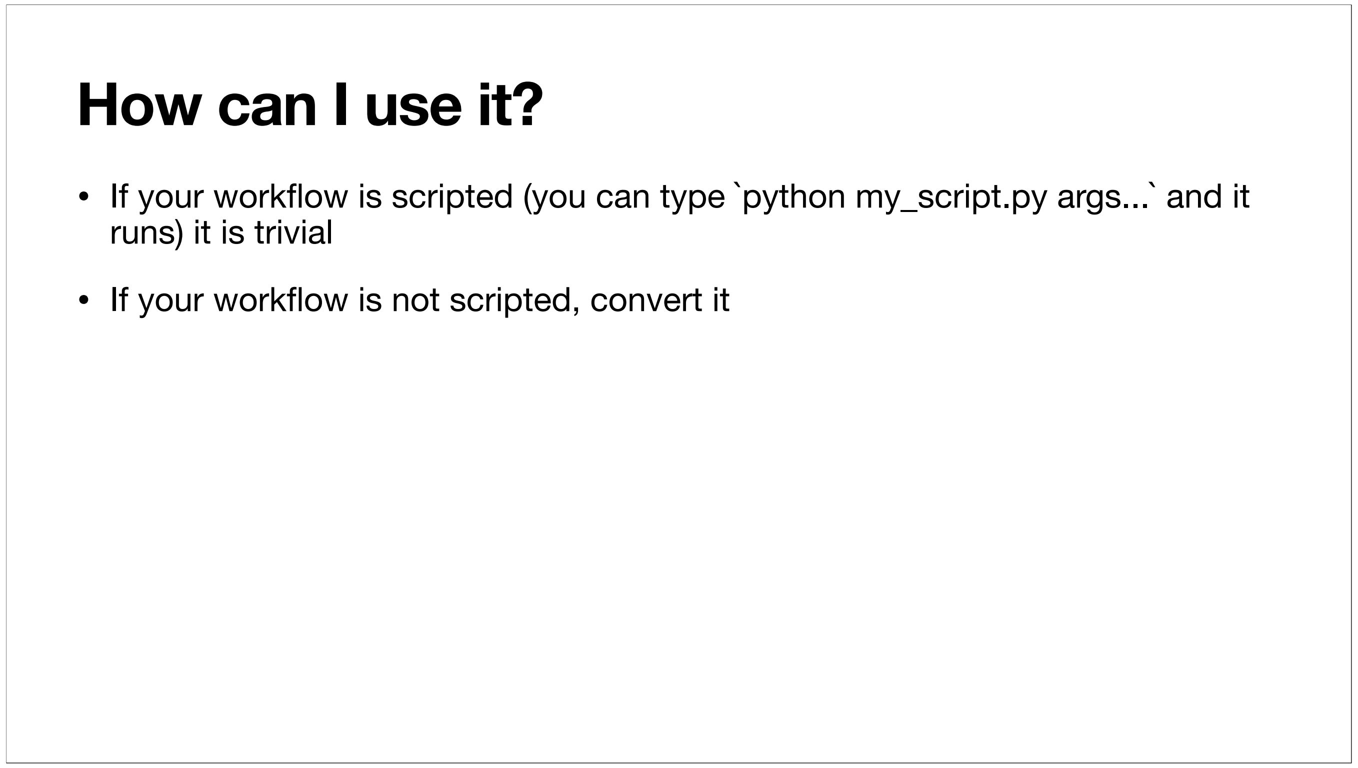
Thanks.

Yours,
-Nick

It only allocates the resources once, when you submit the job. This is cool because you don't need to worry about the queue slowing down half way through your jobs because someone else started using resources.
This is also a huge help for jobs with a large initialization cost, like making a local copy of a large dataset



Notice that you are exposed to risk of the queue slowing down. Additionally, the environment for each node has to be initialized from scratch each time, even if each node is performing the same task as the previous and (coincidentally) ran on the same physical system within the cluster.



How can I use it?

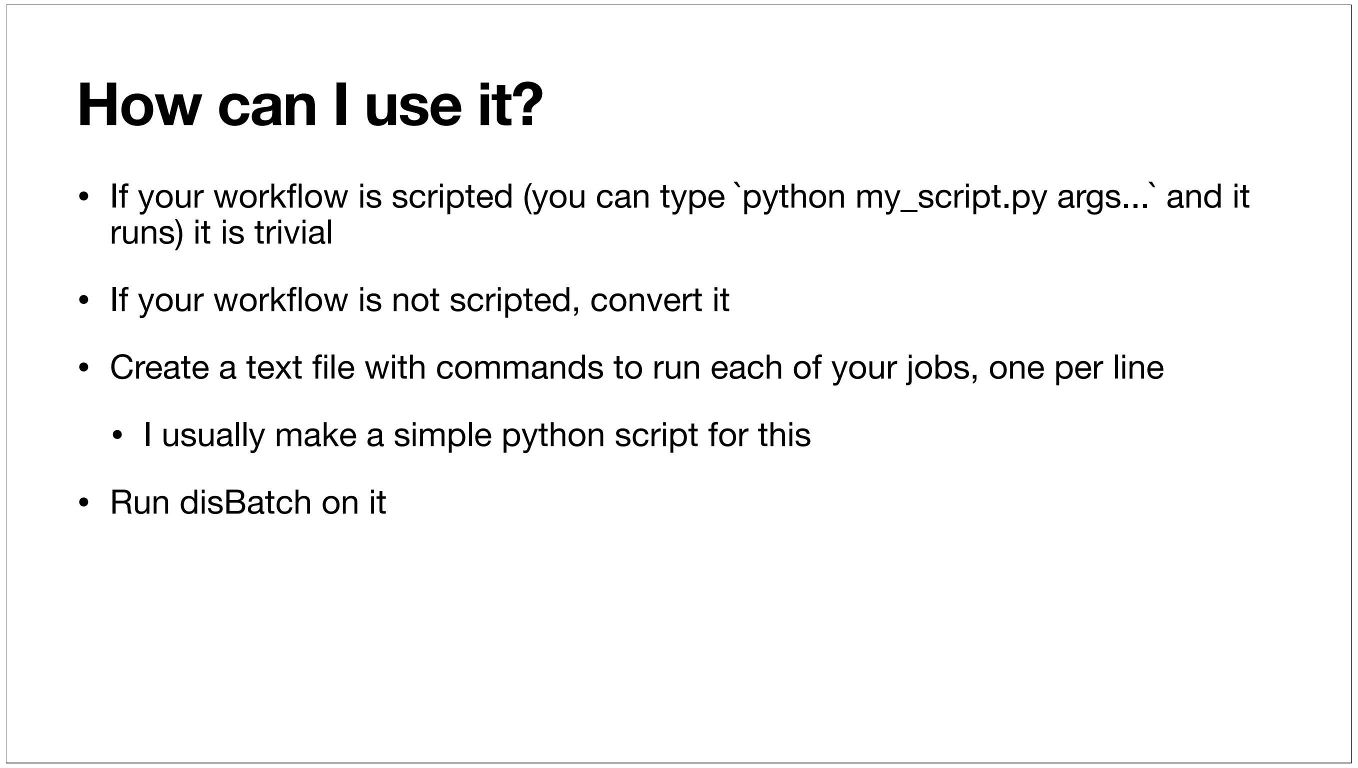
- If your workflow is scripted (you can type `python my_script.py args...` and it runs) it is trivial
- If your workflow is not scripted, convert it

Essentially, disBatch is a tool for running a list of commands in bash across multiple allocated nodes. If you cannot instantiate your job from the command line, start here. Fortunately, argparse is part of the standard python library and provides a straightforward solution to making a python script with arguments runnable from the shell.

Interlude: making Python scripts

- ✨Live demo✨ (what could go wrong?)

Argparse demo



How can I use it?

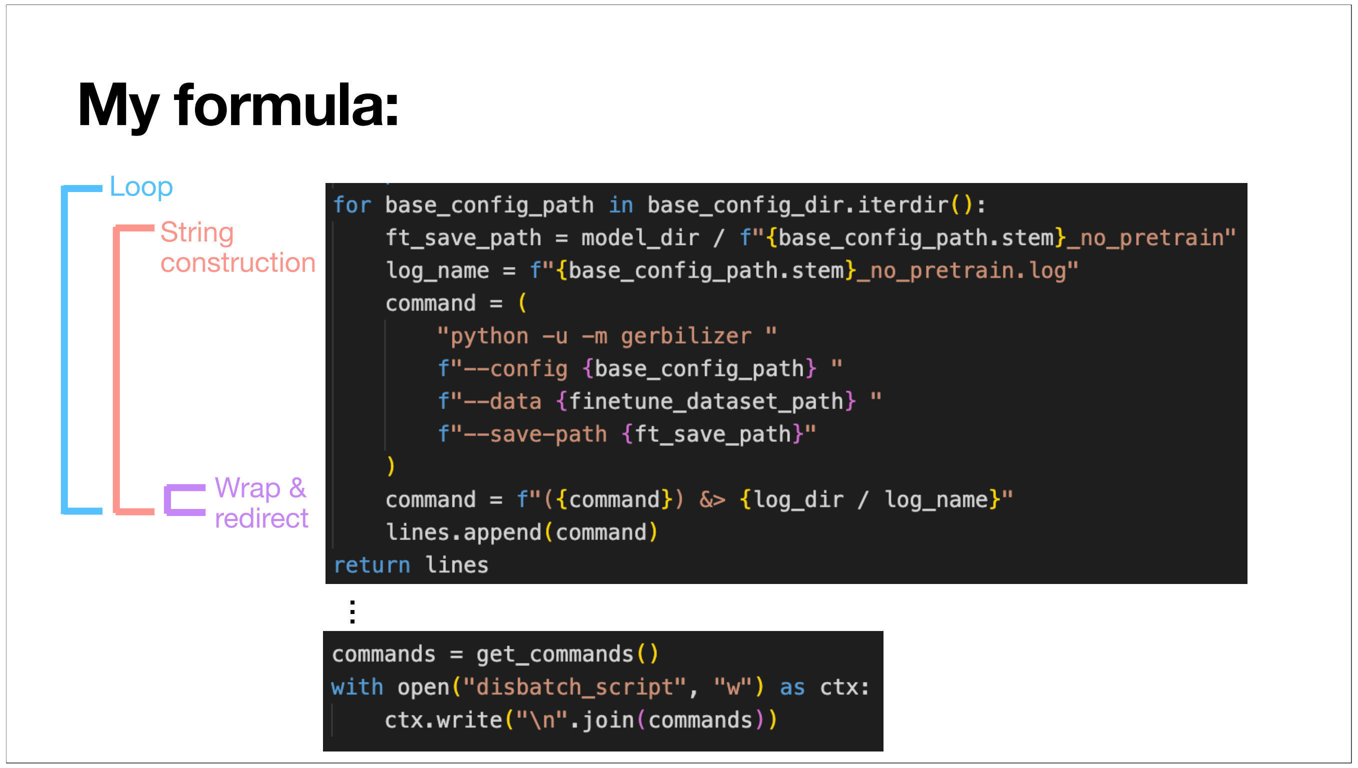
- If your workflow is scripted (you can type `python my_script.py args...` and it runs) it is trivial
- If your workflow is not scripted, convert it
- Create a text file with commands to run each of your jobs, one per line
  - I usually make a simple python script for this
- Run disBatch on it

Once you can launch your jobs from the command line it's really straightforward: put commands to run each of your jobs in a text file, one per line, and run disBatch on it through SLURM

Tips

- You should redirect your logs to make errors and output visible
  - Wrap your commands in `( ... ) &>path_to_log.log`
  - Optionally, separate stdout and stderr: `( ... ) 1>out.log 2>err.log`

Disbatch creates its own logs but they haven't been super useful, in my opinion. It's easier to use your application's own error messages and redirecting the application's stdout and stderr to a unique location for every child job will facilitate this.



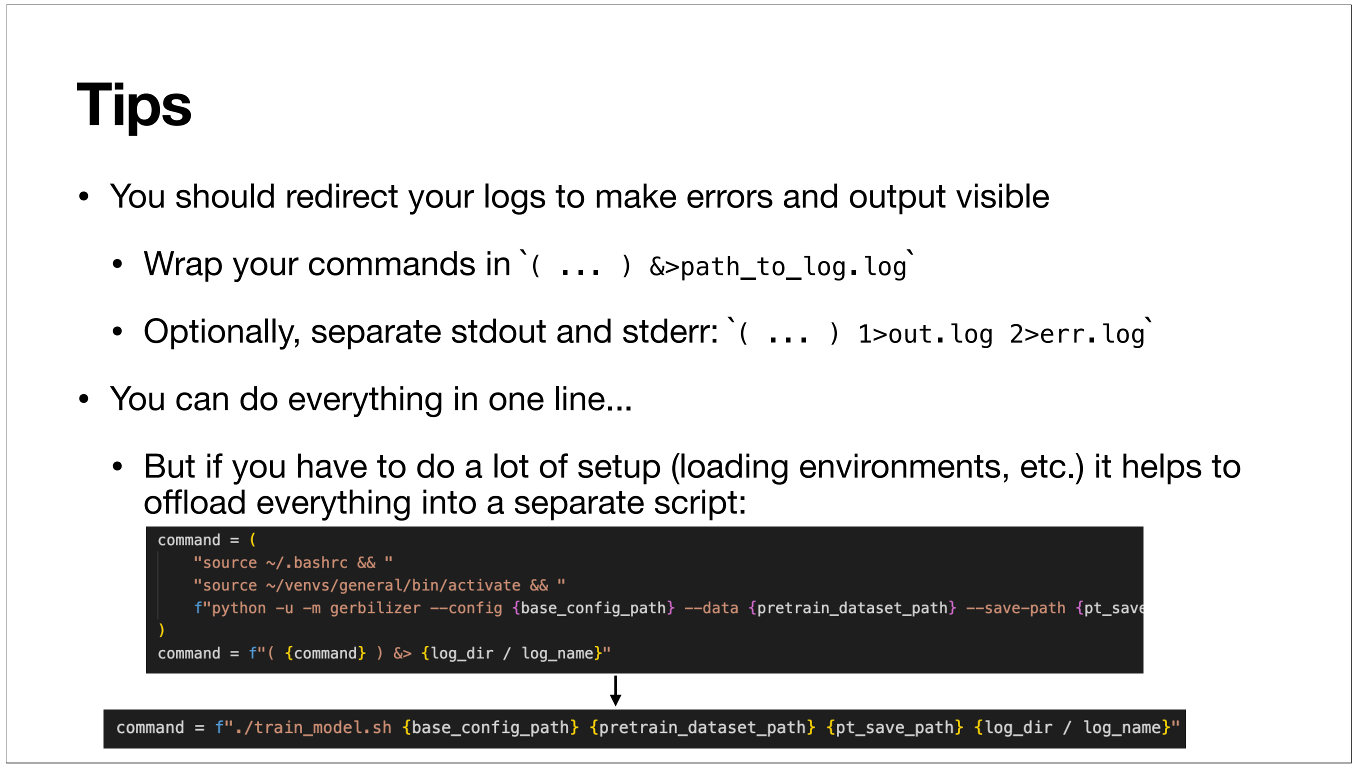
My formula:

```
for base_config_path in base_config_dir.iterdir():
    ft_save_path = model_dir / f"{base_config_path.stem}_no_pretrain"
    log_name = f"{base_config_path.stem}_no_pretrain.log"
    command = (
        "python -u -m gerbilizer "
        f"--config {base_config_path} "
        f"--data {finetune_dataset_path} "
        f"--save-path {ft_save_path}"
    )
    command = f"({command}) &> {log_dir / log_name}"
    lines.append(command)
return lines
```

⋮

```
commands = get_commands()
with open("disbatch_script", "w") as ctx:
    ctx.write("\n".join(commands))
```

Disbatch creates its own logs but they haven't been super useful, in my opinion. It's easier to use your application's own error messages and redirecting the application's stdout and stderr to a unique location for every child job will facilitate this.



Tips

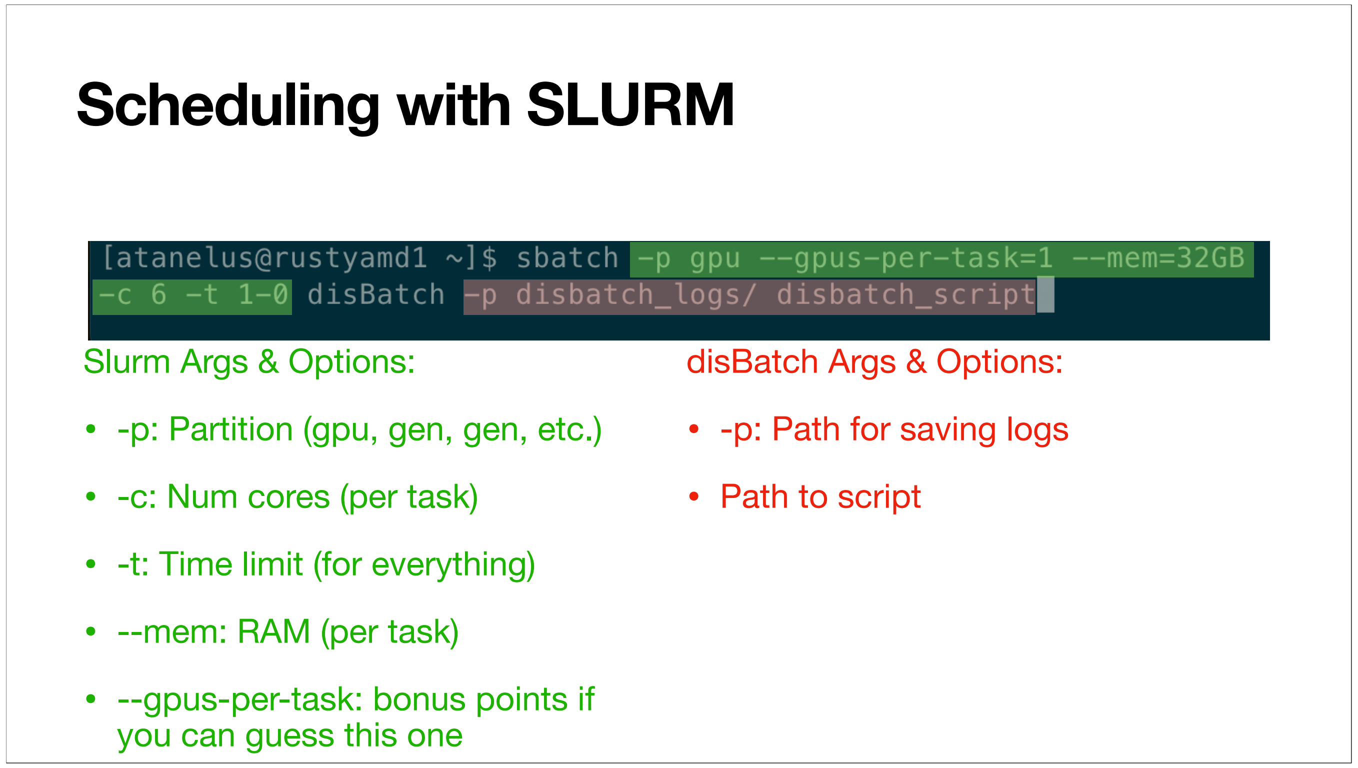
- You should redirect your logs to make errors and output visible
  - Wrap your commands in `` `( ... ) &>path_to_log.log` ``
  - Optionally, separate stdout and stderr: `` `( ... ) 1>out.log 2>err.log` ``
- You can do everything in one line…
  - But if you have to do a lot of setup (loading environments, etc.) it helps to offload everything into a separate script:

```
command = (
    "source ~/.bashrc && "
    "source ~/venvs/general/bin/activate && "
    f"python -u -m gerbilizer --config {base_config_path} --data {pretrain_dataset_path} --save-path {pt_save
)
command = f"( {command} ) &> {log_dir / log_name}"
```

$\downarrow$

```
command = f"./train_model.sh {base_config_path} {pretrain_dataset_path} {pt_save_path} {log_dir / log_name}"
```

Disbatch creates its own logs but they haven't been super useful, in my opinion. It's easier to use your application's own error messages and redirecting the application's stdout and stderr to a unique location for every child job will facilitate this.



Scheduling with SLURM

```
[atanelus@rustyamd1 ~]$ sbatch -p gpu --gpus-per-task=1 --mem=32GB
-c 6 -t 1-0 disBatch -p disbatch_logs/ disbatch_script
```

Slurm Args & Options:

- -p: Partition (gpu, gen, gen, etc.)
- -c: Num cores (per task)
- -t: Time limit (for everything)
- --mem: RAM (per task)
- --gpus-per-task: bonus points if you can guess this one

disBatch Args & Options:

- -p: Path for saving logs
- Path to script

Running disBatch through SLURM gives it the resources of the cluster, which are probably better than what you have locally.