DataLad for Research Data Management

Ian Percel

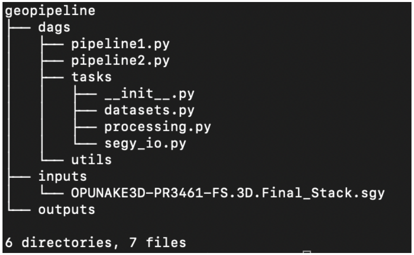
University of Calgary, Research Computing Services

February 14, 2023

Outline

Data Organized by File Systems

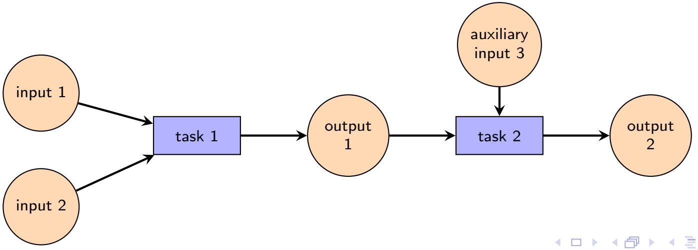
- Some data is collected directly into custom databases
- Most data is organized using files and directories
- This presentation is primarily about imposing order on data captured as files organized in directories



```
geopipeline
├── dags
│   ├── pipeline1.py
│   ├── pipeline2.py
│   ├── tasks
│   │   ├── __init__.py
│   │   ├── datasets.py
│   │   ├── processing.py
│   │   └── segy_io.py
│   └── utils
├── inputs
│   └── OPUNAKE3D-PR3461-FS.3D.Final_Stack.sgy
└── outputs

6 directories, 7 files
```

Tracking Computational Workflows

- A workflow is defined in terms of a network of dependent **tasks**, **task input data**, and **task output data**
- For our purposes, we will consider workflows (like those common in neuroimaging and bioinformatics) that express all variable inputs and outputs as files
- Tracking workflows can be a particularly difficult task as the inputs and outputs vary and the workflow can change over time



DataLad as Distributed Workflow Tracking Utility

- Using a POSIX file system and log files alone make for a clunky, ad hoc tracking system (most WfMS have awkward and custom workflow tracking)
- DataLad is a git annex-based utility that can naturally represent different possible file organization schemes, data parallel organizational units, dataset versioning and workflow versioning using standard functionality
- DataLad also makes it easy to share task sequences for (re)producing data from inputs and/or data itself
- In order to support parallel and distributed workflows, we will need some ideas not in the DataLad Handbook (cf. FAIRly Big Wagner et al 2022)

What is DataLad?

DataLad extends git and git-annex

- Origins in a decentralized Version Control System (leads to a lot of ideas about commits, parallel branches, and included data sources)
- git provides plumbing for tracking changes
- Annexing allows for remote storage of any (or all) files while keeping provenance information local
- Overall very lightweight for what it does

DataLad Interface

- DataLad uses Python to extend existing git and git-annex functionality with simplified, data-oriented wrappers
- Command Line or Python library (no GUI)
- Can think of git/git-annex/DataLad as a Single-File Database like SQLite (but it is a directory and everything under it)
- git commands reference the nearest parent (that includes a .git database) to your current working directory
- Access provided by the `git`, `git-annex`, and `datalad` executables

DataLad Concepts: Dataset Creation with datalad create

```
$ datalad create geodlpipeline

create(ok):  /Users/ian.percel/dlexamples/geophys/geodlpipeline (dataset)

$ ls -a geodlpipeline

.  ..  .datalad .git .gitattributes

$ cd geodlpipeline
$ datalad create -d . inputs


add(ok): inputs (file)
add(ok): .gitmodules (file)
save(ok): . (dataset)
create(ok): inputs (dataset)
action summary:
  add (ok: 2)
  create (ok: 1)
  save (ok: 1)

$ ls -a

.  ..  .datalad .git .gitattributes .gitmodules inputs
```

DataLad Concepts: Commits with datalad save

```
$ cp -R ../geopipeline/dags .
$ ls
dags inputs
$ ls -l dags
```

```
total 24
drwxr-xr-x@ 9 ian.percel  staff   288 31 Jan 15:38 __pycache__
-rw-r--r--@ 1 ian.percel  staff  4157 31 Jan 15:38 pipeline1.py
-rw-r--r--@ 1 ian.percel  staff  3580 31 Jan 15:38 pipeline2.py
drwxr-xr-x@ 7 ian.percel  staff   224 31 Jan 15:38 tasks
drwxr-xr-x@ 2 ian.percel  staff    64 31 Jan 15:38 utils
```

DataLad Concepts: Commits with datalad save

```
$ datalad save


add(ok): dags/__pycache__/FAIRly.cpython-38.pyc (file)
...
add(ok): dags/pipeline1.py (file)
add(ok): dags/pipeline2.py (file)
  [14 similar messages have been suppressed; disable with datalad.ui.suppress-similar-results=off]
save(ok): . (dataset)
action summary:
  add (ok: 24)
  save (ok: 1)

$ cp ../geopipeline/inputs/OPUNAKE3D-PR3461-FS.3D.Final_Stack.sgy inputs
$ cd inputs
$ datalad save


add(ok): OPUNAKE3D-PR3461-FS.3D.Final_Stack.sgy (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)

$ ls -l


total 0
lrwxr-xr-x  1 ian.percel  staff  136 31 Jan 15:40 OPUNAKE3D-PR3461-FS.3D.Final_Stack.sgy
-> .git/annex/objects/F2/0p/MD5E-s10402944680--f5656874eb2da17c08e6c90b58c372ad.sgy/MD5E...
```

DataLad Concepts: Branching with `git checkout -b`

```
$ git checkout -b test1
Switched to a new branch ‘test1’
$ touch example_addition
$ datalad save


add(ok): example_addition (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)

$ ls
dags example_addition inputs outputs
$ git checkout master
Switched to branch ‘master’
$ ls
dags inputs outputs
$ git branch -D test1
Deleted branch test1 (was 575eb96).
```

Branches allow for multiple different versions to coexist. However, changing to another branch in a super dataset does not alter the state of sub datasets.

DataLad Concepts: Editing, Annexes, and configurations

```
$ cd dags
$ touch example_addition
$ datalad save
...
$ echo "some new input" >> example_addition
-bash: example_addition: Permission denied
$ git annex unlock example_addition
unlock example_addition ok
(recording state in git...)
$ echo "some new input" >> example_addition
$ datalad save


add(ok): dags/example\_addition (file)
save(ok): . (dataset)
action summary:
  add (ok: 1)
  save (ok: 1)
```

Passing the text2git configuration to via `datalad create -c text2git` eliminates the need to unlock frequently edited or executed files but precludes the use of remote storage. Generally, text2git is a good option for your code base but data files usually shouldn't be edited once they are finished being written and so it is appropriate to use an annex.

DataLad Concepts: Computational Reproducibility with `datalad run`

```
$ git checkout -b runbranchtest
Switched to a new branch ‘runbranchtest’
$ cd outputs
$ git checkout -b runbranchtest
Switched to a new branch ‘runbranchtest’
$ dataset=/Users/ian.percel/dlexamples/geophys/geodlpipeline
$ input_dl=${dataset}/inputs/OPUNAKE3D-PR3461-FS.3D.Final_Stack.sgy
$ output_dl=${dataset}/outputs/OPUNAKE3D-PR3461-FS.3D.Final_Stack_smoothed.sgy
$ script=${dataset}/src/filter.py
$ sigma=7
$ datalad run -m "filter input segy" -d ${dataset} -i ${input_dl} -o ${output_dl} \
"python3.11 ${script} {inputs} {outputs} ${sigma}"


[INFO   ] Making sure inputs are available (this may take some time)
[INFO   ] == Command start (output follows) =====
[INFO   ] == Command exit (modification check follows) =====
run(ok): /Users/ian.percel/dlexamples/geophys/geodlpipeline (dataset) [python3.11 ...src/filter.py...]
add(ok): OPUNAKE3D-PR3461-FS.3D.Final_Stack_smoothed.sgy (file)
save(ok): outputs (dataset)
add(ok): outputs (dataset)
add(ok): .gitmodules (file)
save(ok): . (dataset)
```

This generates a computationally actionable git log entry and `datalad rerun` can be used to act on that

DataLad Concepts: Merging with git merge

```
$ pip3 uninstall apache-airflow
...
$ pip3 install prefect
$ git checkout -b transition_to_prefect
$ ls
__pycache__ pipeline1.py pipeline1.py pipeline3.py tasks utils
$ datalad remove __pycache__ utils pipeline1.py pipeline2.py pipeline3.py tasks
$ vim dl_util.py
$ vim geo_util.py
$ vim pipeline.py
a couple hours pass
$ datalad save
...many lines of deletes and saves
$ git merge --strategy=ours master
Already up to date.
$ git checkout master
$ git merge transition_to_prefect


Updating 763f678..3230ed0
Fast-forward
 .gitmodules                                                 | 4 ++++
 dags/__pycache__/FAIRly.cpython-38.pyc                      | 1 -
...
 dags/dl_util.py                                             | 1 +
 dags/geo_util.py                                            | 1 +
 dags/pipeline.py                                            | 1 +
 ...
 29 files changed, 8 insertions(+), 24 deletions(-)
```

DataLad Concepts: Saving to remote with `datalad push` and `git push`

`push` sends data to a remote repository. In general, for an annexed dataset

- `git push` sends the git history only
- `datalad push --to <remote> --data nothing` sends the git history and references (not files) to a specified remote repo
- `datalad push --to <remote> --data anything` sends both git history and annexed files to a specified remote repo

After data is stored on a known remote, the actual file content can be dropped locally to reduce local storage while retaining pointers for relationships and the file can be pulled back later.

DataLad and HPC: The Serial(ish) Case

Git and Parallelism

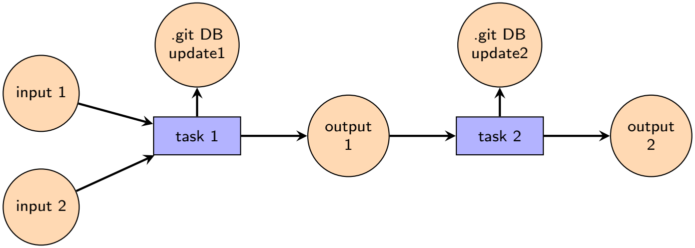
- Git is not generally thread safe for .git database updates
- There is a mechanism for limited buffering when multiple non-conflicting concurrent* updates are committed
- For two concurrently submitted updates that do conflict, one is rejected and has to pull, merge, and push again
- To use DataLad to track parallel processing of disjoint datasets, some care needs to be taken

We will build from the sequential case to a simple parallel one over the following examples

*throughout this presentation we use *concurrent* to mean that a process begins execution before another related process has completed. That is, there is a common time interval where both processes are attempting to do their work and there is potential competition for resources like access to a git database

Workflow Schemes: Strict Serial Pipeline

- One job at a time (e.g. job array with one at a time execution)
- Single DataLad database (on storage accessible to all nodes)
- Sequential calls to DataLad run



With appropriate task serialization and one dataset at a time submission, there will be no overlap here and everything is simple.

Simple Workflow Parallelization with Datalad Tracking: Data Parallel Smoothing in Geophysics

We will begin by examining our geophysics pipeline in prefect:

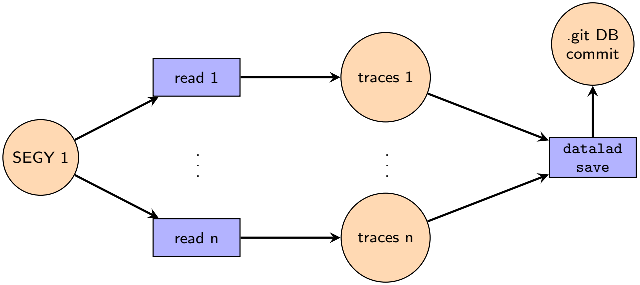
```
from prefect import flow, task

import geo_util as gu
import dl_util as du

@task
def read_data_trcs(dataset_path, filename, l_tr, i_tr):
        return gu.read_segy_trcs(dataset=dataset_path,filename=filename,
                                                              l_tr=l_tr, i_tr=i_tr)

@task
def read_data_extras(filename):
        return gu.read_segy_extra(filename=filename)

@task
def dataset_save(dataset):
        du.dataset_save_recursive(dataset)
...
```

Simple Workflow Parallelization with Datalad Tracking: Data Parallel Smoothing in Geophysics

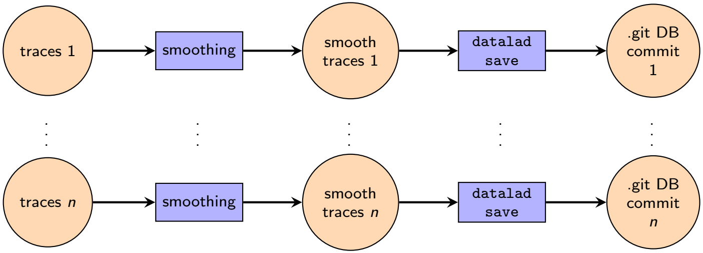
We will begin by examining our geophysics pipeline in prefect:

```
@flow
def trace_processing(dataset_path, input_path, output_path, figure_path, tmp_path,
tmp_out_path, script_path, trace_batch_start_list, trace_batch_length):
        tracedata_futures=[]
        for i_tr in trace_batch_start_list:
                tracedata_futures.append(read_data_trcs.submit(dataset_path, input_path,
                                                                trace_batch_length, i_tr))
        extra_futures=[]
        extra_futures=read_data_extra.submit(input_path)
        #wait on data read and tmp write
        dataset_save.submit(dataset_path,wait_for=tracedata_futures)
        ...

if __name__ == "__main__":
        #set parameters
        ...
        #run pipeline
        trace_processing(dataset, input, output, figure, tmp, tmp_out, script,
                                                        ini_tr, len_tr)
```

Simple Workflow Parallelization with Datalad Tracking: Data Parallel Smoothing in Geophysics



Simple Workflow Parallelization with Datalad Tracking: Data Parallel Smoothing in Geophysics



How do the parallel commits resolve? It depends on how big *n* is.

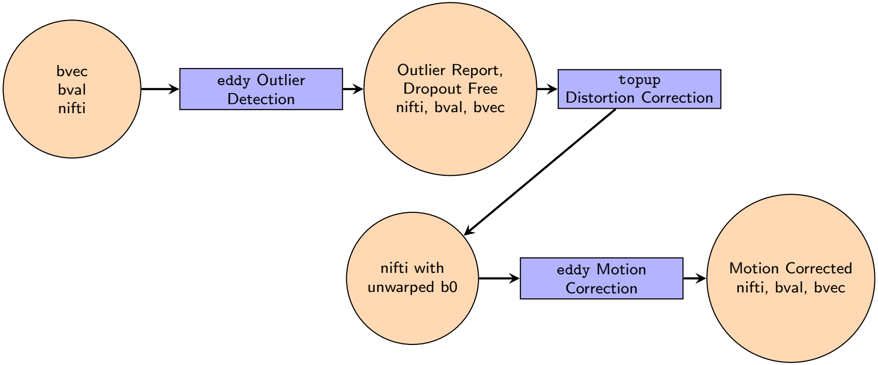
How do the parallel commits resolve?

Git has the ability to enforce interleaving semantics (arbitrary serial ordering) up to some limit. Researchers on FAIRly Big Project have found this limit to be around 1000 concurrent commits when things start to get lost. For small numbers (i.e $n < 10$), we have direct evidence of successful resolution:

```
14:09:49.155 | INFO    | Flow run 'rich-boobook'
- Created task run 'process_trcs_dl-5' for task 'process_trcs_dl'

14:09:49.158 | INFO    | Flow run 'rich-boobook' - Submitted task run 'process_trcs_dl-5' for execution.
processing traces with filter script
input .../geodlpipeline/tmp/OPUNAKE3D-PR3461-FS.3D.Final_Stack.sgy_1500_250.tmp
output(b4) .../geodlpipeline/tmp_out/OPUNAKE3D-PR3461-FS.3D.Final_Stack.sgy_1500_250.tmp.out

... some really scary warnings: "'' failed with exitcode 128",
"fatal: Unable to create '.git/index.lock': File exists.",
'Another git process seems to be running in this repository', ...time passes...

14:09:58.793 | INFO    | Task run 'process_trcs_dl-5' - Finished in state Completed()
outlogs= [['input ../tmp/OPUNAKE3D-PR3461-FS.3D.Final_Stack.sgy_250_250.tmp',
 'run(ok): /Users/ian.percel/dlexamples/geophys/geodlpipeline (dataset) ...',
 'add(ok): tmp_out/OPUNAKE3D-PR3461-FS.3D.Final_Stack.sgy_0_250.tmp.out (file)',
 ... 'save(ok): . (dataset)']]
```

A Scheduled Pipeline Example: FSL for diffusion imaging correction

For this new example, we need to submit tasks as jobs to a SLURM scheduler because the tasks depend on specialized computing resources.


bvec
bval
nifti
eddy Outlier Detection
Outlier Report, Dropout Free nifti, bval, bvec
topup Distortion Correction
nifti with unwarped b0
eddy Motion Correction
Motion Corrected nifti, bval, bvec

A Scheduled Pipeline Example: FSL `eddy` and `topup`

Script #	Task #	Command	Resource	Duration
1	1	`fslmerge`	1 cpu core	$<$ 1 min
1	2	`bet`	1 cpu core	$<$ 1 min
1	**3**	`eddy_cuda11.2`	v100 GPU	**25 min**
2	1	`fslroi`	1 cpu core	$<$ 10 min
2	2	`fslmerge`	1 cpu core	$<$ 10 min
2	**3**	`topup`	1 cpu core	**10 hours**
2	4	`fslmaths`	1 cpu core	$<$ 10 min
2	5	`bet`	1 cpu core	$<$ 10 min
3	**1**	`eddy_cuda11.2`	v100 GPU	**25 min**
3	2	`fslroi`	1 cpu core	$<$ 1 min
3	2	`cut`	1 cpu core	$<$ 1 min

A Scheduled Pipeline Example: FSL Eddy

Two timing estimates:

1. Three scripts run as one job on GPU node $\rightarrow$ 11 hour GPU request per subject
2. GPU jobs run separately $\rightarrow$

11 subjects per 5.5 hour GPU request for script 1
11 single-core job runs submitted in parallel (likely to schedule quickly, run in 10 hours) for script 2
11 subjects per 5.5. hour GPU request for script 3 $\rightarrow$
11 subjects/22hours of computing $\rightarrow$
2 hours of wall-time per subject without accounting for scheduling delays (only 1 hour of which is GPU time)

HPC scheduling with accurate resource requests can maximize throughput and reduce time to schedule.

A Scheduled Pipeline Example: FSL Eddy

For a single subject, `datalad run` structure is trivial and slurm scripts can be submit sequential steps in the pipeline.

```
#SBATCH --time=1:00:00
#SBATCH --mem=10G
#SBATCH -N1 -n1 -c1
#SBATCH --gres=gpu:1

topdir=...
subj=...
dataset=${topdir}/${subj}
coderepo=...
slurmrepo=...
container=...

apptainer exec --nv -B /project/dir ${container} \
datalad run -m "${subj} Outlier Detection" -d ${dataset}  \
"bash ${coderepo}/preprocessing_1.sh"

sbatch ${slurmrepo}/${subj}/preprocessing_2.sh
```

This is no higher throughput than putting all three in one job. Only improvement is better utilization of GPU nodes.

A Scheduled Pipeline Example: FSL Eddy

For parallel processing of subjects in the cpu-only stage (`topup`), `datalad run` structure will get back to our concurrent `datalad save` problem.

```
#SBATCH --time=1:00:00
#SBATCH --mem=10G
#SBATCH -N1 -n1 -c1
#SBATCH --gres=gpu:1

topdir=...
subj_list=...
coderepo=...
slurmrepo=...
ccontainer=...

for subj in ${subj_list}
do
dataset=${topdir}/${subj}
apptainer exec --nv -B /project/dir ${container} \
datalad run -m "${subj} Outlier Detection" -d ${dataset}  \
"bash ${coderepo}/preprocessing_1.sh"

sbatch ${slurmrepo}/${subj}/preprocessing_2.sh
done
```

This improves throughput and the 25 min offset in submit time may save us from concurrency headaches.

DataLad and HPC: The Distributed Case

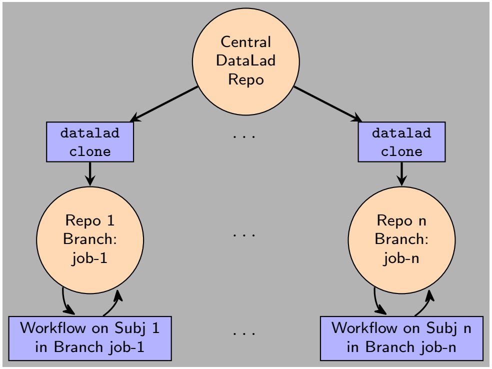
FAIRly Big

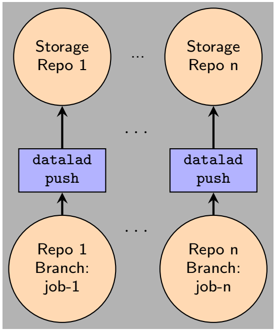
- Important development in using DataLad as part of workflows on HPC and HTC
- `https://www.biorxiv.org/content/10.1101/2021.10.12.464122v1`
- Wagner, A.S., Waite, L.K., Wierzba, M. et al. FAIRly big: A framework for computationally reproducible processing of large-scale data. Sci Data 9, 80 (2022). https://doi.org/10.1038/s41597-022-01163-2
- `https://github.com/psychoinformatics-de/fairly-big-processing-workflow`

The remainder of this talk will outline the main ideas for implementing this strategy as they apply to job scheduled computing.

Workflow Schemes: FAIRly Big Data Parallel WF

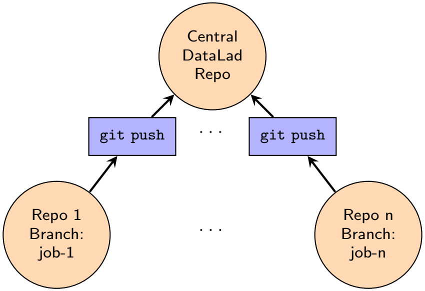
- Several data parallel jobs running (potentially) simultaneously
- Multiple ephemeral DataLad databases (one to track operations in each job)





Workflow Schemes: FAIRly Big Data Parallel WF

- Only concurrency bottleneck left is the git push
- Still face concurrency challenges at this point, but the amount of operations and data transfers to serialize is much smaller
- Wagner et al. tackle this with a global file lock. File locks are notoriously troublesome over NFS
- Other approaches for interleaving are possible



Data Parallel Pipeline Example: FSL Eddy Outlier Removal

```
topdir=...
subj_list=...
source_root=...
target_root=...
coderepo=...
slurmrepo=...
centralrepo=...
container=...

for subj in ${subj_list}
do
dlsource=${source_root}/${subj}
datalad clone ${dlsource} ${topdir}/${subj}/ds
dataset=${topdir}/${subj}/ds

git checkout -b "job-${subj}"

apptainer exec --nv -B /project/dir ${container} \
datalad run -m "${subj} Outlier Detection" -d ${dataset}  \
"bash ${coderepo}/preprocessing_1.sh"

datalad push --to ${target_root}/${subj}

flock $dslockfile git push $centralrepo

sbatch ${slurmrepo}/${subj}/preprocessing_2.sh
done
```

Data Parallel Pipeline: Improvements left out

- `datalad contianers-run`
- Improved batch size optimization
- Improve fine-grained history by recording individual tasks
- automate `git merge` and `git annex fsck`

Q&A

Questions?

Git design features that impact DataLad performance

- Git architecture uses a Content Addressable File System (CAFS) written in C that is layered on top of local storage
- For extremely large collections of files it becomes possible to have hash sum value collisions
- The working directory is expressed by git as symlinks to files stored in a central CAFS
- Versions of your project are expressed through trees of pointers to centrally stored files
- Extremely long (recorded) histories of tiny changes will hurt performance and expand storage utilized to retain each version

Main Idea: choosing appropriate sizes for elements of your data management design can mean the difference between DataLad being a very efficient and flexible utility and DataLad being a clumsy and inefficient one