Multiprocessing, Multithreading, and Vectorization

Expression Learning Group Meeting

Outline

1. Why do we care about these techniques
2. What are processes and threads?
3. Multithreading in python (the Global Interpreter Lock)
4. Multiprocessing in python
5. Multiprocessing in bash
6. Vectorization in python
7. Conclusions and further resources

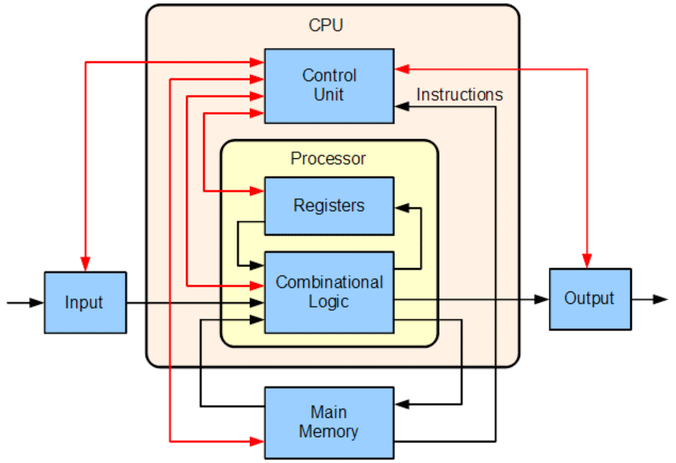
Code for this is located here:
https://github.com/bharris12/practical_progamming_lecture_1

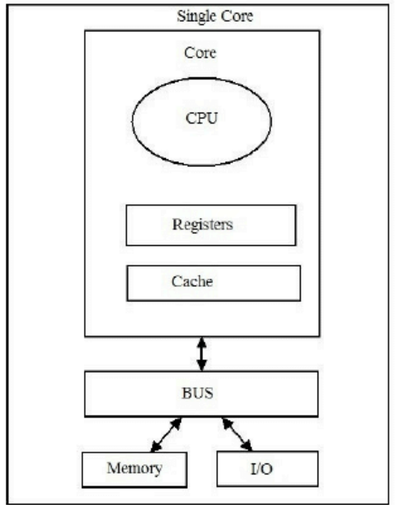
It can be useful to clone the repository now to look along at some of the code during the presentation

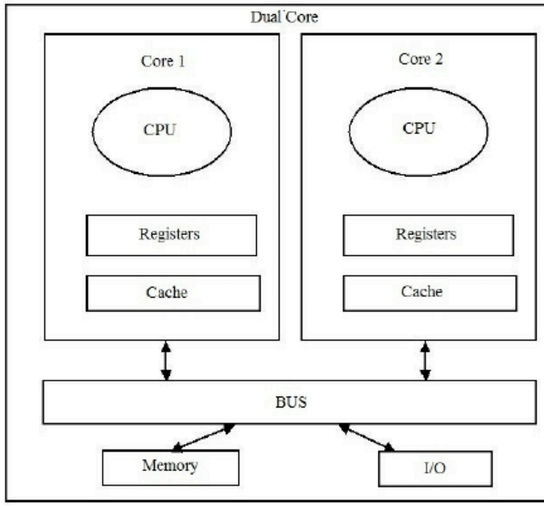
For the example notebook you can either use the binder or google colab buttons on GitHub or clone the repository and work directly on a laptop or server

Why use Multithreading, Multiprocessing and Vectorization?

- Coding to utilize all the resources on a single computer can speed up your code significantly
- A lot of our work can be done concurrently
- You can simplify CPUs as pipes to push your data through
- The way you naturally program will treat your CPU as a singular pipe but in some cases minor tricks will utilize all the resources available

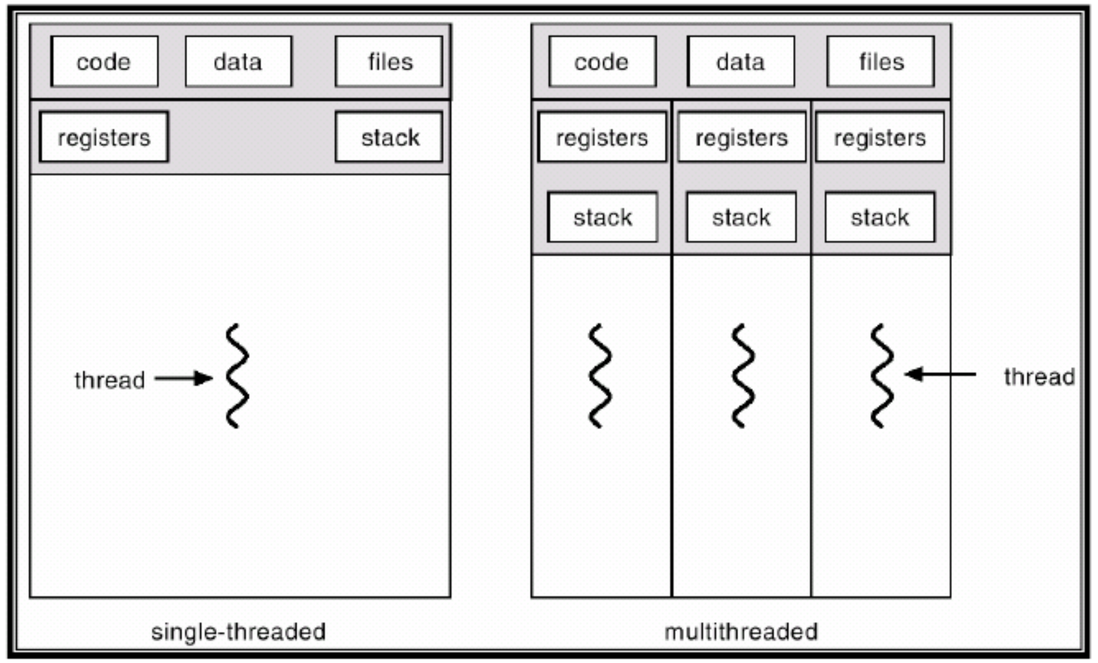






Processes and Threads

- Every process is it's own standalone program
- Process has it's own PID (Process ID)
- Threads share a PID and share memory
- Processes and Threads **both** have overhead to start
- Threads are "lighter", you don't have to create a whole new process, however they still have overhead
- Almost everything that can be done with either -threading or -processing, what you choose is mostly out of convenience to program, but occasionally can have speed differences

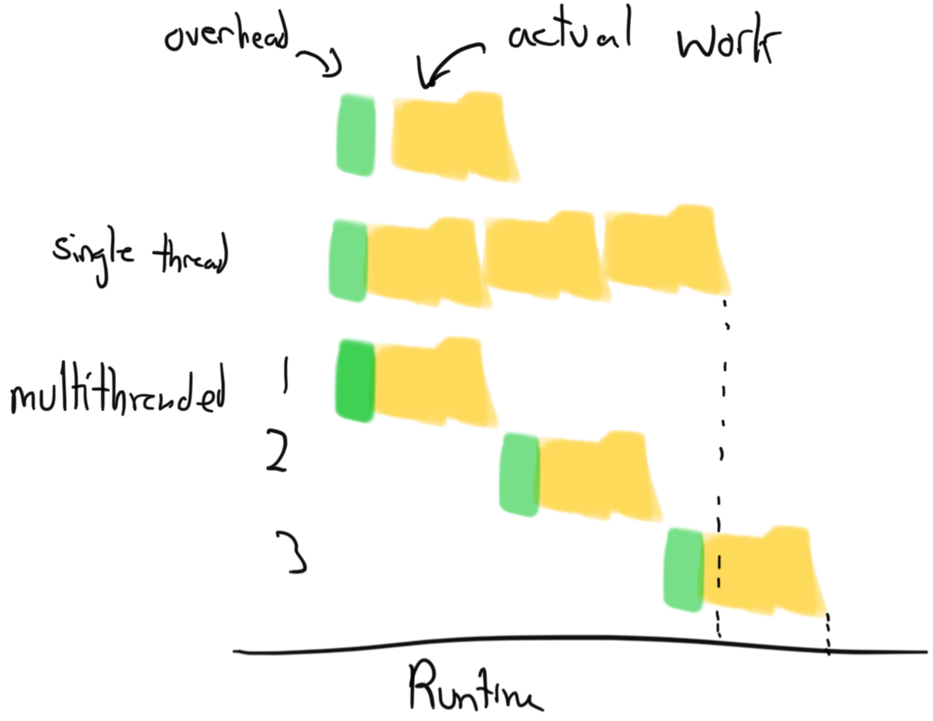


Multithreading and Multiprocessing

- Both are used to concurrently execute code
- **Multithreading:** Utilizing multiple CPU threads to execute a program concurrently. While the memory is shared between threads, each thread has it’s own registers and code stack
- **Multiprocessing:** Creating new processes/subprocesses to execute a program concurrently. This creates a new PID for each process and each process is siloed off from every other concurrent process

Multithreading in python

- Python has a Global Interpreter Lock (GIL) that prevents concurrent execution of operations within a process
    - The GIL does not apply to I/O from disk, so if you have a big I/O bottleneck you could get a speedup from



Linear Algebra Multithreading

- The linear algebra libraries (MKL, Blas, Lapack) by default might use multithreading
- The datatype you use can impact whether you use the multiple threads or are single threaded
- In this matrix multiplication example you get a ~100x speedup by using floats instead of ints

```
a = np.random.randint(low=0, high=100, size=[1000, 1000])
b = np.random.randint(low=0, high=100, size=[1000, 1000])
```

executed in 27ms, finished 12:46:41 2020-10-27

```
a.dtype
```

executed in 10ms, finished 12:46:41 2020-10-27

```
dtype('int64')
```

```
%%timeit
res = a @ b
```

executed in 7.62s, finished 12:46:49 2020-10-27

```
951 ms ± 1.68 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
c = a.astype(float)
d = b.astype(float)
```

executed in 7ms, finished 12:46:49 2020-10-27

```
%%timeit
res = c @ d
```

executed in 8.40s, finished 12:46:57 2020-10-27

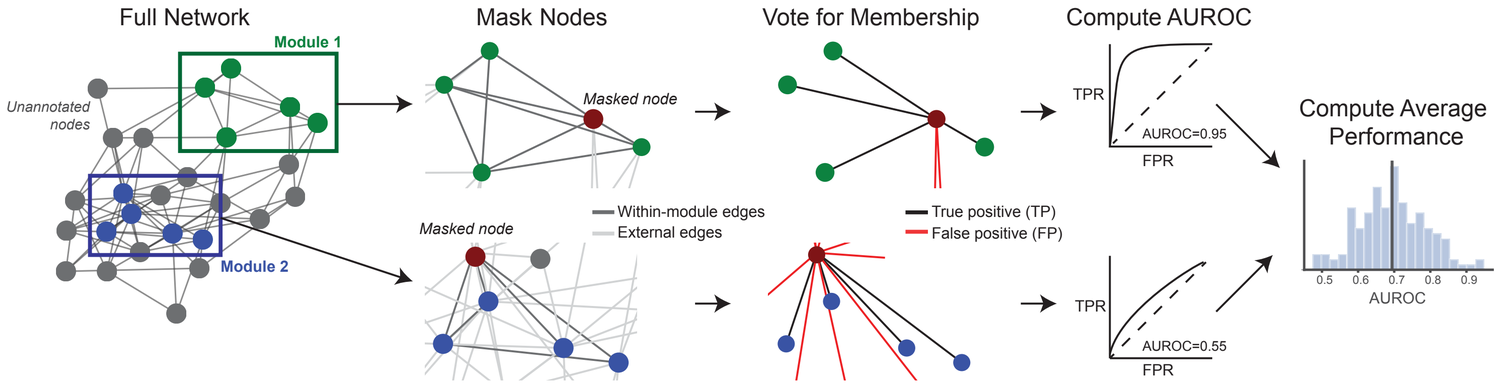
```
9.32 ms ± 600 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Multiprocessing

- When you have lots of modular stuff to run
- Python has many libraries for this
  - Multiprocessing
  - Dask
  - Apache Arrow
  - Ray
- In multiprocessing you are creating subprocesses. These take a lot of overhead but can be really useful if each task has long runtimes

EGAD for measuring Co-expression network performance

- Neighbor voting method for measuring the connectivity of known modules
- Inputs:
  - Gene annotations (like Gene Ontology or KEGG) in the form on Genes x Terms binary encodings
  - Co-expression network, genes x genes matrix of weights of strength of connection [0,1] or binary
- Output: AUROC that shows how well that module can be predicted from the network
- Problem: I have 52 networks and I want to know the average AUROC for all of the Gene ontology for each of them



3 Ways to compute 52 AUROCs

1st Sequential: Just looping over the list of datasets

```
%%time
go = pd.read_hdf('~/GO_data/go_mouse_nw.hdf5', 'go')
res = []
for fn in file_names:
    nw = pd.read_hdf(fn, 'nw')
    res.append(run_egad(go, nw).AUC.mean())
```

executed in 11m 4s, finished 13:58:10 2020-10-27

```
CPU times: user 2h 28min 48s, sys: 56min 19s, total: 3h 25min 7s
Wall time: 11min 3s
```

run_egad() is the function for the algorithm

go is the gene ontology

file_names is a list of filenames to networks

Easiest to program, but slow

3 Ways to compute 52 AUROCs

2nd: Multiprocessing Manager: Explicit control of multiprocessing, but a little cumbersome to code

Import multiprocessing as mp

```
def egad_test(fn,name, return_dict):
    go = pd.read_hdf('~/GO_data/go_mouse_nw.hdf5', 'go')
    nw = pd.read_hdf(fn, 'nw')
    return_dict[name] = run_egad(go, nw).AUC.mean()
```

executed in 5ms, finished 13:58:10 2020-10-27

EGAD test takes the file name, dataset name and dictionary as inputs

Instead of returning a value it stores the result in the return_dict under the key of the dataset name

```
%%time
manager = mp.Manager()
return_dict = manager.dict()
jobs = []
for fn, name in zip(file_names, datasets):
    p = mp.Process(target=egad_test, args=(fn, name, return_dict))
    jobs.append(p)
    p.start()
for proc in jobs:
    proc.join()
```

executed in 2m 52s, finished 14:01:02 2020-10-27

You have to pass positional arguments as a tuple

```
CPU times: user 76 ms, sys: 11.7 s, total: 11.8 s
Wall time: 2min 52s
```

This method is better when you have fewer processes because they all get started at once, there is no scheduler, luckily I have 96 cores and only 52 (mostly single threaded) process so this works fine

3 Ways to compute 52 AUROCs

3rd: Multiprocessing Pool: Explicit control of multiprocessing, but a little cumbersome to code

```
def egad_test_map(fn):
    go = pd.read_hdf('~/GO_data/go_mouse_nw.hdf5', 'go')
    nw = pd.read_hdf(fn, 'nw')
    return run_egad(go, nw).AUC.mean()
```

executed in 3ms, finished 14:19:34 2020-10-27

```
%%time
pool = mp.Pool(10)
map_res = pool.map(egad_test_map, file_names)
pool.close()
pool.join()
```

executed in 2m 51s, finished 14:24:35 2020-10-27

```
CPU times: user 306 ms, sys: 242 ms, total: 548 ms
Wall time: 2min 51s
```

This time I only pass the filename and the average AUROC is returned by this function

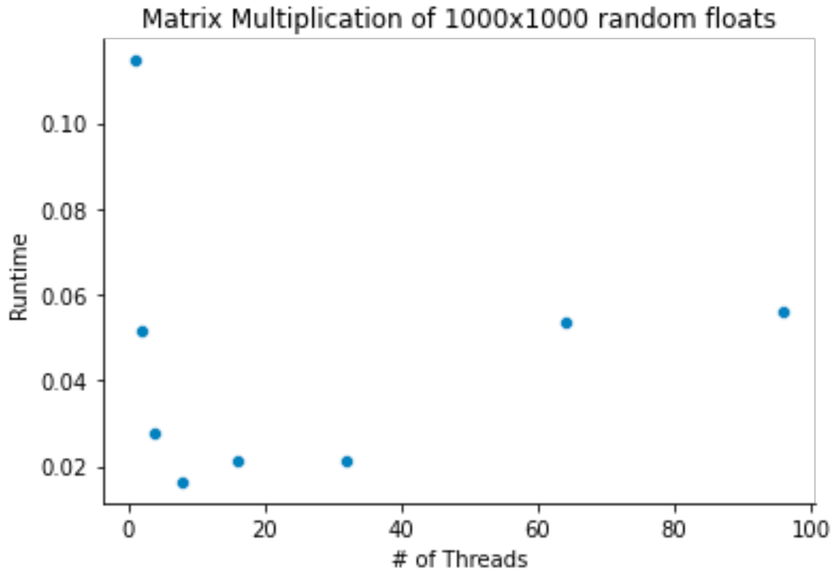
pool.map works just like the native map but uses every worker in the pool to run the function, so in this case you can see that I have 10 workers in the pool

This doesn't get a speedup over the Manager method because the manager method is bottlenecked by too many processes running at the same time

If I increase the pool to 20 I get ~1 minute decrease in runtime because I still have enough available threads during the multithreaded parts of the program

Bottlenecking the # of Workers

- With both multiprocessing and multithreading you can slow yourself down with too many workers
- Overhead or redundant code in parallel tasks reduces the speedup you gain with more workers
- With my EGAD example every worker needs to read in GO
- Python >=3.8 includes shared memory in multiprocessing which can alleviate some of this



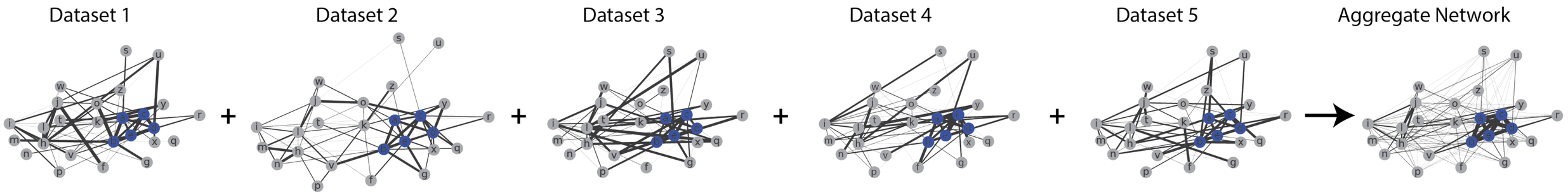

```
def egad_test(fn,name, return_dict):
    go = pd.read_hdf('~/GO_data/go_mouse_nw.hdf5', 'go')
    nw = pd.read_hdf(fn, 'nw')
    return_dict[name] = run_egad(go, nw).AUC.mean()
```

Multiprocessing in Bash

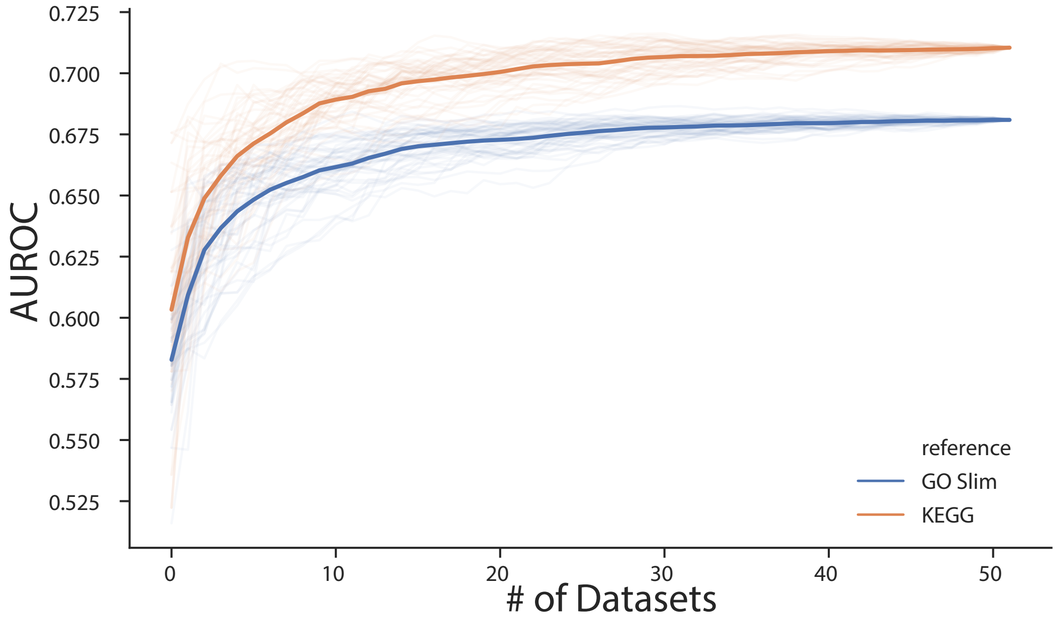
- Fast for running an existing program multiple times with different inputs or pipelines
- More concise that writing for loops in bash
- Common programs for multiprocessing
  - GNU parallel
  - Xargs
- Workflow languages can also work for multiprocessing
  - Snakemake
  - Nextflow
- Sometimes you don’t want to move your data to Elzar and write job submission scripts

Manual: https://www.gnu.org/software/parallel/man.html

Aggregating Co-expression Networks



- Aggregating networks across datasets increases AUROC of known functional modules
- Each thin line on the plot to the right represents a random ordering of the 52 datasets. With them aggregated in that order
- Each step in aggregation requires the new aggregate network to be ranked
- The ranking of genes x genes takes minutes, but is a single threaded process
- Serially creating 100 of these lines would take hours



Using GNU parallel to compute each permutation

```python
import numpy as np
import pandas as pd

from egad import run_egad
from rank import rank

from argparse import ArgumentParser

##Get iteration number from the command line
parser = ArgumentParser()
parser.add_argument('-i', type=int, help='ID for results',required=True)
args = parser.parse_args()


## Load list of datasets and permute the order of them
datasets = np.genfromtxt(
    '/data/bharris/biccn_paper/data/bulk_rna/datasets_used.csv', dtype=str)
np.random.shuffle(datasets) #Shuffle order of datasets

## Compute file paths for aggregation
networks_path = '/data/bharris/biccn_paper/data/bulk_rna/networks/'
file_names = [f'{networks_path}{ds}_pearson_nw.hdf5' for ds in datasets]

## Load in GO annotations
go = pd.read_hdf('~/GO_data/go_mouse_nw.hdf5', 'go')

##Create empty network to store aggregate in
genes = np.genfromtxt(
    '/data/bharris/biccn_paper/data/highly_expressed_7_datasets_75k.csv',
    dtype=str)
agg_nw = np.zeros([genes.shape[0],genes.shape[0]])

##Series to store result
results = pd.Series()
for ds, fn in zip(datasets, file_name):
    agg_nw += pd.read_hdf(fn,'nw').values
    agg_nw_copy = agg_nw.copy()
    rank(agg_nw_copy) #Ranking occurs inplace
    results[ds] = run_egad(go,
                           pd.DataFrame(agg_nw_copy,
                                        index=genes,
                                        columns=genes)).AUC.mean()

## Save output using ID
results.to_csv(f'permute_network_performance_{args.i}.csv')
```

Passing an id to store results as permute_network_performance_{id}.csv

Load in dataset list then randomly shuffle

More loading stuff and getting ready to loop

The bottleneck is rank(agg_nw_copy) this takes ~2 minutes each loop (x52 datasets x 100 permutations) = A LONG TIME

GNU Parallel Dry Run

Highly recommend always using —dry-run first

Input

```
seq 1 10 | parallel --dry-run python3 gnu_parallel_example.py -i {}
```

Output

```
python3 gnu_parallel_example.py -i 1
python3 gnu_parallel_example.py -i 2
python3 gnu_parallel_example.py -i 3
python3 gnu_parallel_example.py -i 4
python3 gnu_parallel_example.py -i 5
python3 gnu_parallel_example.py -i 6
python3 gnu_parallel_example.py -i 7
python3 gnu_parallel_example.py -i 8
python3 gnu_parallel_example.py -i 9
python3 gnu_parallel_example.py -i 10
```

Seq 1 10 creates a list of numbers 1 - 10 that are then piped to parallel

—dry-run outputs the function calls that would occur instead of running it

The {} at the end is where the values for each iteration are placed

GNU parallel control parameters

Input

```
seq 1 10 | parallel -j 10 \
                    --timeout 200% \
                    --joblog parallel_test.log \
                    python3 gnu_parallel_example.py -i {}
```

Example joblog

Seq	Host	Starttime	JobRuntime	Send	Receive	Exitval	Signal	Command
1	:	1603847747.686	0.000	0	2	0	0	echo 1
3	:	1603847747.693	0.003	0	2	0	0	echo 3
4	:	1603847747.696	0.004	0	2	0	0	echo 4
2	:	1603847747.690	0.013	0	2	0	0	echo 2
6	:	1603847747.704	0.005	0	2	0	0	echo 6
7	:	1603847747.708	0.003	0	2	0	0	echo 7
8	:	1603847747.712	0.005	0	2	0	0	echo 8
9	:	1603847747.717	0.003	0	2	0	0	echo 9
5	:	1603847747.700	0.021	0	2	0	0	echo 5
10	:	1603847747.721	0.003	0	3	0	0	echo 10

-j controls the # of workers
—timeout lets you kill processes taking too long
—joblog lets you store a log

These are just some of the MANY parameters that parallel has

Xargs is very similar and can probably do everything that parallel can

GNU parallel passing arguments

Input

```
parallel --dry-run python3 gnu_parallel_example.py -i {} ::: 1 2 3
```

Output

```
python3 gnu_parallel_example.py -i 1
python3 gnu_parallel_example.py -i 2
python3 gnu_parallel_example.py -i 3
```

Instead of piping in data you can pass a typed list using 3 colons

GNU parallel passing arguments cont

Input

```
parallel touch test_{}.txt :::: <(seq 1 10)
```

Can do 4 colons on the right instead of pipe on the left

Output

```
python3 gnu_parallel_example.py -i 1
python3 gnu_parallel_example.py -i 2
python3 gnu_parallel_example.py -i 3
python3 gnu_parallel_example.py -i 4
python3 gnu_parallel_example.py -i 5
python3 gnu_parallel_example.py -i 6
python3 gnu_parallel_example.py -i 7
python3 gnu_parallel_example.py -i 8
python3 gnu_parallel_example.py -i 9
python3 gnu_parallel_example.py -i 10
```

GNU parallel passing arguments cont

```
parallel echo {1} {2} :::: <(seq 1 10) :::: <(seq 10 1
) | head
```

```
1 10
1 8
1 7
1 9
1 5
1 4
1 3
1 6
1 1
```

Can pass multiple parameter lists
Equivalent to nested for loops

For I in a:
    For j in b:
        Echo a b

```
parallel --xapply echo {1} {2} ::: A B C ::: 1 2 3
A 1
B 2
C 3
```

—xapply will match inputs instead of nesting
Equivalent to: for i,j in zip(a,b)

GNU Parallel for file Manipulation

```
bharris@dactyl:~/biccn_paper/data/bulk_rna$ cat datasets_used.csv  | head
GSE65770
GSE75386
GSE54651
GSE77243
GSE104775
GSE70732
GSE63943
GSE108269
GSE95141
GSE100070
```

Datasets used is a list of GEO ids for datasets

```
bharris@dactyl:~/biccn_paper/data/bulk_rna$ cat datasets_used.csv | parallel -j 10 find networks/{}*
_nw.hdf5  | head
networks/GSE65770_pearson_nw.hdf5
networks/GSE65770_proportionality_nw.hdf5
networks/GSE65770_spearman_nw.hdf5
networks/GSE75386_pearson_nw.hdf5
networks/GSE75386_proportionality_nw.hdf5
networks/GSE75386_spearman_nw.hdf5
networks/GSE54651_pearson_nw.hdf5
networks/GSE54651_proportionality_nw.hdf5
networks/GSE54651_spearman_nw.hdf5
networks/GSE77243_pearson_nw.hdf5
```

Find searches for any file in the networks folder starting with each GEO ID and ending with _nw.hdf5

```
bharris@dactyl:~/biccn_paper/data/bulk_rna$ cat datasets_used.csv | parallel -j 10 find networks/{}*
_nw.hdf5  | parallel -j 10 du -h {} | head
135M    networks/GSE65770_pearson_nw.hdf5
135M    networks/GSE65770_proportionality_nw.hdf5
135M    networks/GSE65770_spearman_nw.hdf5
135M    networks/GSE75386_pearson_nw.hdf5
135M    networks/GSE75386_proportionality_nw.hdf5
135M    networks/GSE75386_spearman_nw.hdf5
135M    networks/GSE54651_pearson_nw.hdf5
135M    networks/GSE54651_proportionality_nw.hdf5
135M    networks/GSE54651_spearman_nw.hdf5
135M    networks/GSE77243_pearson_nw.hdf5
```

Du -h gets you the size of each file in human readable numbers and gives you the filename next to it

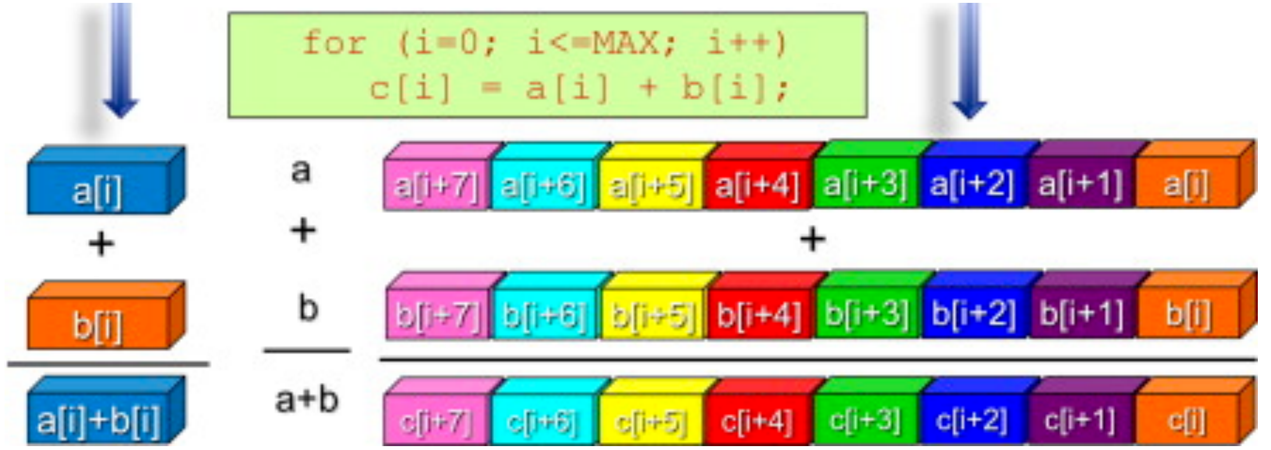
```
bharris@dactyl:~/biccn_paper/data/bulk_rna$ cat datasets_used.csv | parallel -j 10 find network
/{}*_nw.hdf5  | parallel -j 10 du -h {} | cut -f1 | sed 's/[^0-9]*//g' | uniq -c | awk '{print
1 * $2}' | awk '{s+=$1}END{print s}'
21060
```

A bunch of other bash stuff gets you the total amount of space I am using for network files for these GEO #s

See Nathan's advanced bash tutorial in a few months for more endless pipes (or whatever he's gonna teach)

Vectorization

- Different than multithreading, and occurs within a single thread, but still computes stuff concurrently
- Using compiled code with special instructions to use vector components of a CPU
- Most of numpy is already vectorized, although a lot of statistics (in scipy) are not (see example)
- Numpy's vectorize function doesn't necessarily speed up a function you give to it
  - If you want to speed stuff like that up you need to compile it using a JIT, like numba or write it in C/C++ and wrap it




```python
## List Comprehension
listB = [function(a) for a in listA]

## For loop
listB = []
for a in listA:
    listB.append(function(a))


## Vectorization
vector_func = np.vectorize(function)
listB = vector_func(listA)
```

Vectorizing Operations

You get a major speedup ~1000x when using vectorized functions

```
rands = np.random.rand(1000,1000)
```

executed in 12ms, finished 15:26:44 2020-10-27

```
%%timeit
rowSums = []
for row in range(rands.shape[0]):
    rowSums.append(sum(rands[row,:]))
```

executed in 1.77s, finished 15:28:20 2020-10-27

```
221 ms ± 2.24 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%%timeit
rowSums = np.sum(rands, axis=1)
```

executed in 3.88s, finished 15:28:17 2020-10-27

```
477 µs ± 825 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Vectorization and Pandas

- Passing pandas dataframes to vectorized functions will not use the vectorized versions

- There are 4 ways you can write your code to compute the rowSum for a data frame.

```
%%timeit
res = rands_df.sum(axis=1)
```

executed in 5.21s, finished 15:43:21 2020-10-27

```
6.41 ms ± 24.2 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%%timeit
res = np.sum(rands_df,axis=1)
```

executed in 5.78s, finished 15:43:31 2020-10-27

```
7.1 ms ± 3.05 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%%timeit
res = rands_df.values.sum(axis=1)
```

executed in 3.85s, finished 15:43:25 2020-10-27

```
476 µs ± 14.4 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
%%timeit
res = np.sum(rands_df.values, axis=1)
```

executed in 3.93s, finished 15:43:35 2020-10-27

```
482 µs ± 17 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Vectorization Example Problem

Questions?