Understanding HTCF I/O: Storage Types and Use

CGS_SB High Throughput Computing Facility
September 2023

Brian Koebbe - Systems Manager – Room 5312
Eric Martin - Operations & Systems Specialist – Room 5214
Couch Biomedical Research Building

https://htcf-users.slack.com

Storage

- Arguably the most important part of any cluster.
- Many types of storage and many ways to use that storage.
- When used appropriately, I/O bound jobs can be surprisingly fast.
- When not...bad things can happen

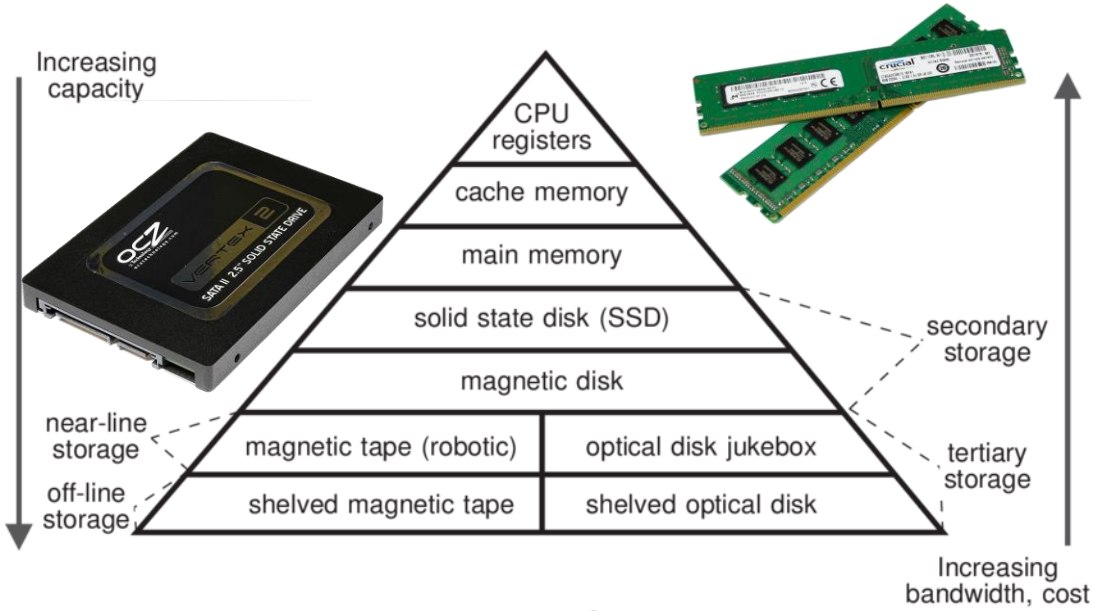
Take aways...

You'll have...

1. A clearer understanding of each type of storage and when/how to use each one
2. A way determine the kind of I/O your programs are using
3. An idea of what object storage is and how it can be used to eliminate the need for excessive back and forth copying of data between /scratch and /lts
4. A glimpse into how workflow tools such as Snakemake and Nextflow, when paired with object storage, can be your new best friends.

Types of Storage

- Local Disk
  - /tmp
- Network File System (NFS)
  - /home
- BeeGFS
  - /scratch
- Ceph
  - /lts (Ceph RBDs)
  - /ref (CephFS)
  - LTOS (Ceph Rados Gateway)

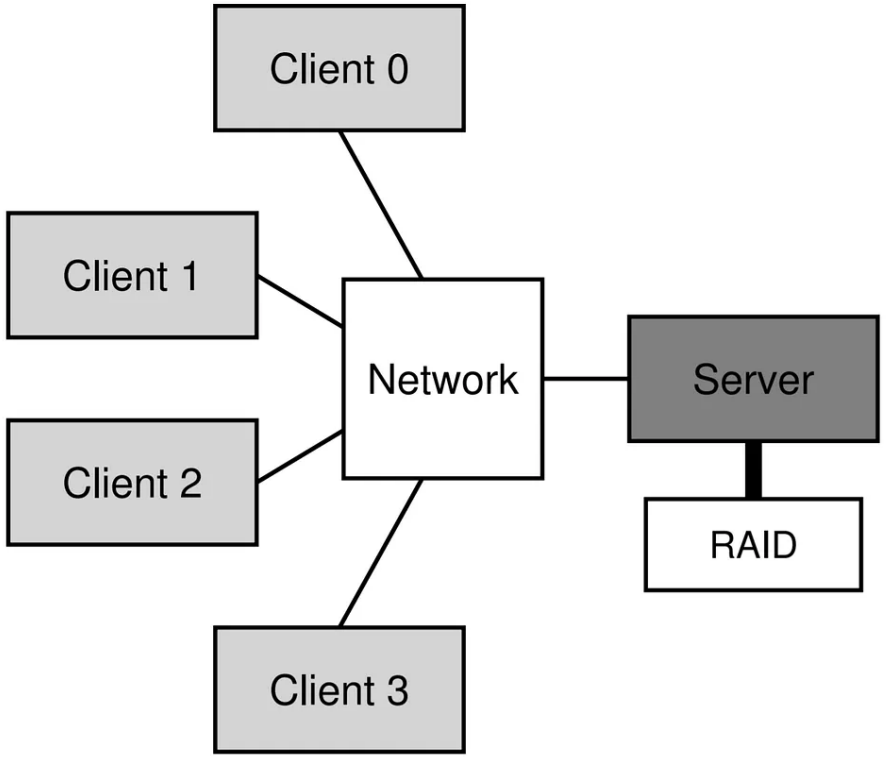


Types of Storage: Local Disk

- **Local Disk**
  - **/tmp**
- Network File System (NFS)
  - /home
- BeeGFS
  - /scratch
- Ceph
  - /lts (Ceph RBDs)
  - /ref (CephFS)
  - LTOS (Ceph Rados Gateway)

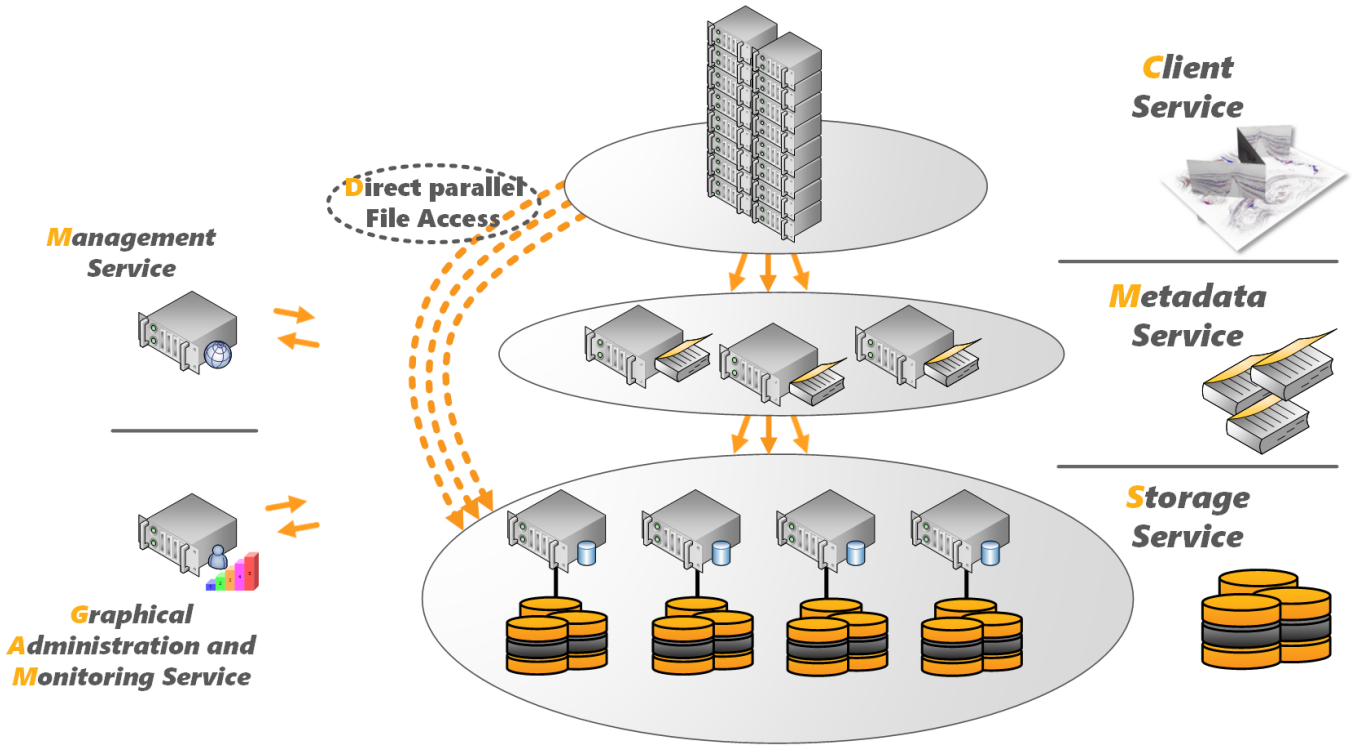
Types of Storage: NFS

- Local Disk
  - /tmp
- **Network File System (NFS)**
  - **/home**
- BeeGFS
  - /scratch
- Ceph
  - /lts (Ceph RBDs)
  - /ref (CephFS)
  - LTOS (Ceph Rados Gateway)



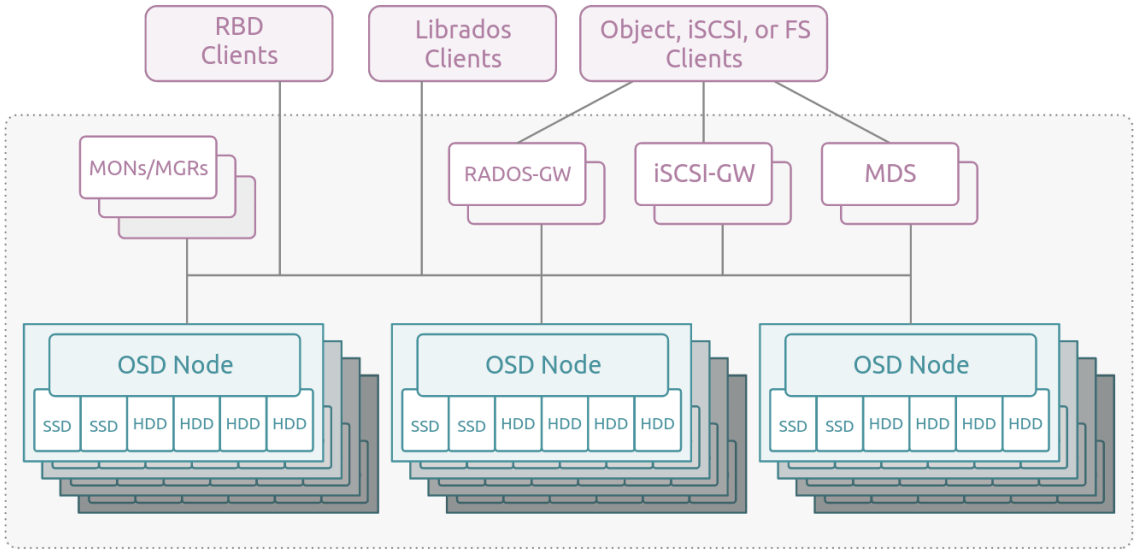
Types of Storage: BeeGFS

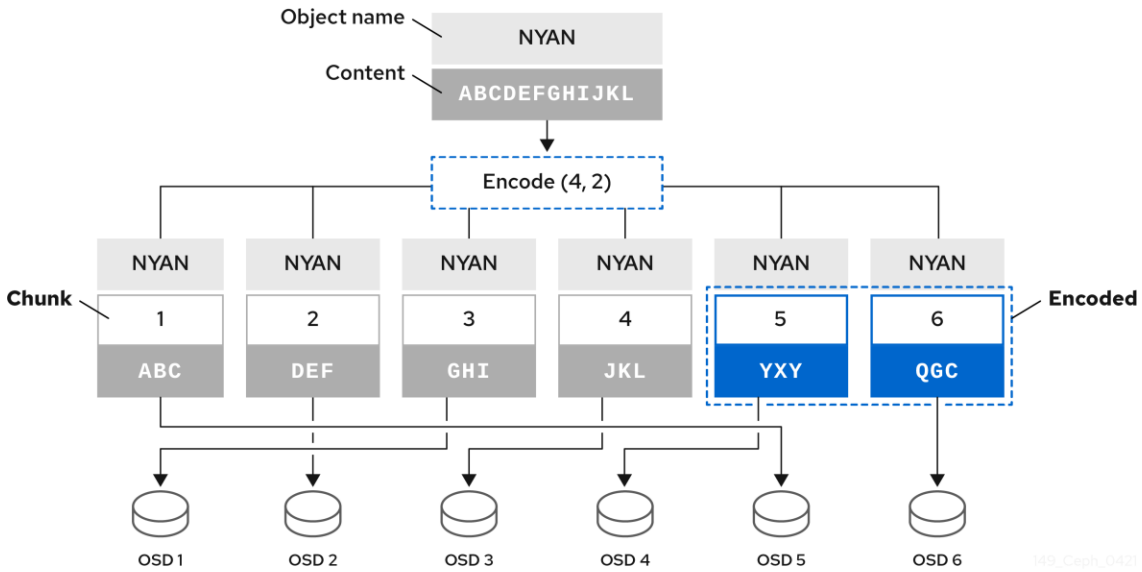
- Local Disk
  - /tmp
- Network File System (NFS)
  - /home
- **BeeGFS**
  - **/scratch**
- Ceph
  - /lts (Ceph RBDs)
  - /ref (CephFS)
  - LTOS (Ceph Rados Gateway)



Types of Storage: Ceph

- Local Disk
  - /tmp
- Network File System (NFS)
  - /home
- BeeGFS
  - /scratch
- **Ceph**
  - **/lts (Ceph RBDs)**
  - **/ref (CephFS)**
  - **LTOS (Ceph Rados Gateway)**





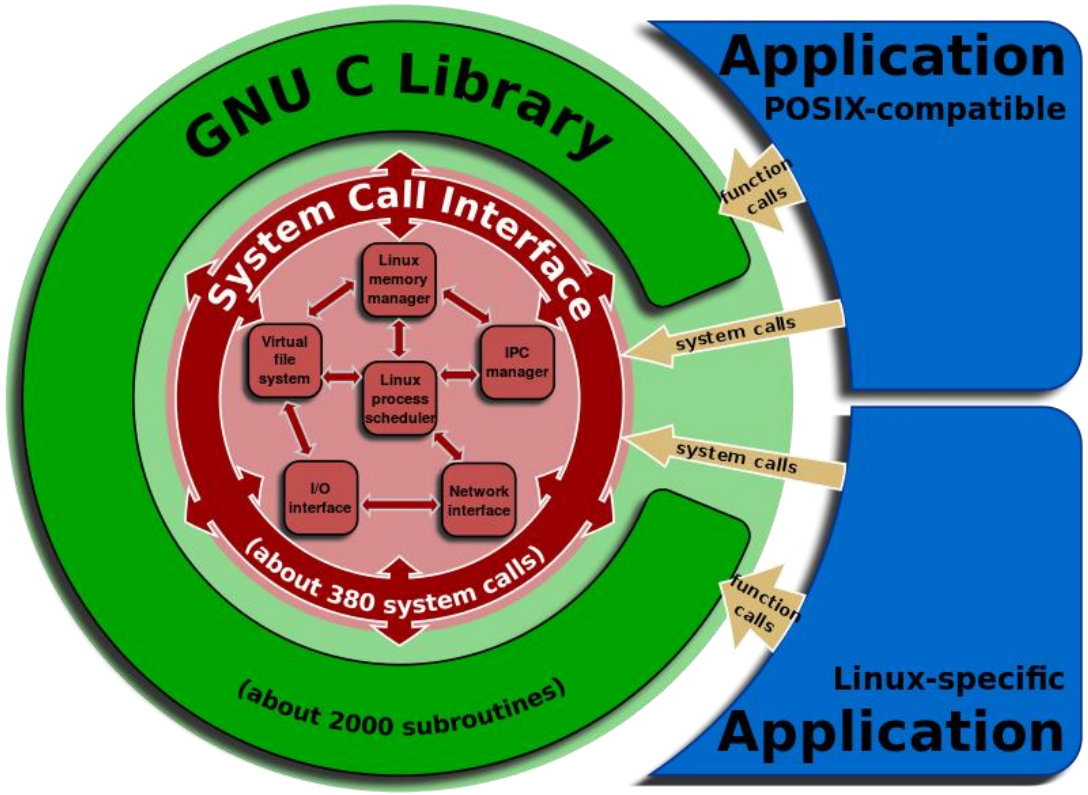
Why so many types?

- Like so many things in life, there's not a one-size-fits-all solution for storage.
- Usually, if storage is fast, it's not robust or it's not affordable
- It comes down to how the storage types handle reads and writes.

https://www.summitracing.com/parts/orb-90158226

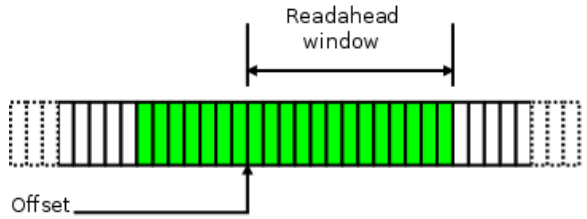
Crash Course: Reading / Writing

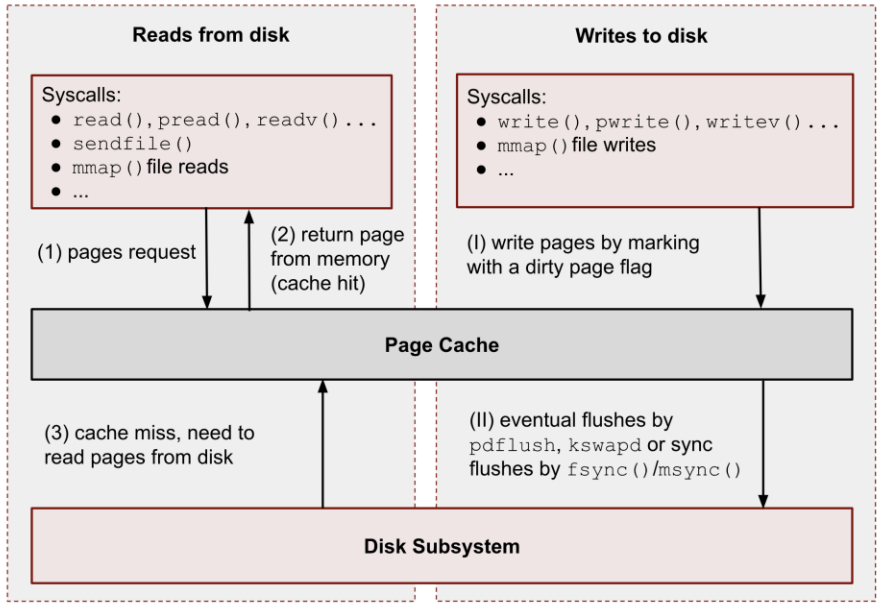
- Storage is all about reads and writes
- User software/applications don't actually do the reading/writing, as they live in "user space"
- The OS (specifically, the "kernel") lives in kernel space and does the reads/writes.
- The software asks the kernel to read/write on its behalf via **system calls**
- System calls are functions provided by the kernel/OS for accessing resources devices/disks/network.



Crash Course: Reading / Writing

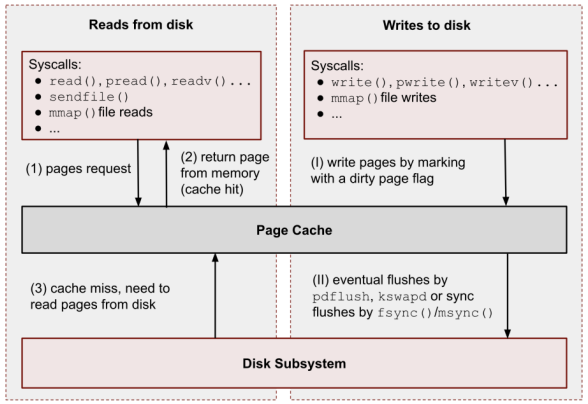
- In return, the kernel can do some behind-the-scenes work that makes our I/O more efficient:
  - Read-ahead
  - Page Caching





https://biriukov.dev/docs/page-cache/2-essential-page-cache-theory/

Crash Course: Page Caching



https://biriukov.dev/docs/page-cache/2-essential-page-cache-theory/

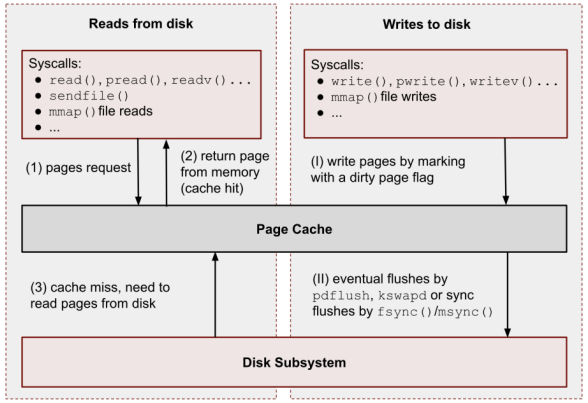
1. copy ~2GB to /tmp
2. "drop caches"
3. Read all files. time it.
4. Immediately read all files again

```
[root@n004 tmp]# pwd
/tmp
[root@n004 tmp]# ls -l
total 1948472
-rw-r--r-- 1 root root 136473378 Sep 18 15:09 chr10.fa
-rw-r--r-- 1 root root 116651622 Sep 18 15:09 chr13.fa
-rw-r--r-- 1 root root 109184600 Sep 18 15:09 chr14.fa
-rw-r--r-- 1 root root  84922597 Sep 18 15:09 chr17.fa
-rw-r--r-- 1 root root 253935557 Sep 18 15:09 chr1.fa
-rw-r--r-- 1 root root  65733058 Sep 18 15:09 chr20.fa
-rw-r--r-- 1 root root  47644190 Sep 18 15:09 chr21.fa
-rw-r--r-- 1 root root  51834845 Sep 18 15:09 chr22.fa
-rw-r--r-- 1 root root 247037406 Sep 18 15:07 chr2.fa
-rw-r--r-- 1 root root 202261477 Sep 18 15:09 chr3.fa
-rw-r--r-- 1 root root 194018853 Sep 18 15:09 chr4.fa
-rw-r--r-- 1 root root 185169031 Sep 18 15:09 chr5.fa
-rw-r--r-- 1 root root 141162618 Sep 18 15:09 chr9.fa
-rw-r--r-- 1 root root     16907 Sep 18 15:09 chrM.fa
-rw-r--r-- 1 root root 159161719 Sep 18 15:09 chrX.fa
[root@n004 tmp]# sudo sysctl vm.drop_caches=1
vm.drop_caches = 1
[root@n004 tmp]# time cat * > /dev/null

real    0m3.667s
user    0m0.004s
sys     0m0.584s
[root@n004 tmp]# time cat * > /dev/null

real    0m0.352s
user    0m0.001s
sys     0m0.350s
[root@n004 tmp]#
```

Crash Course: Page Caching



https://biriukov.dev/docs/page-cache/2-essential-page-cache-theory/

1. copy ~2GB to /scratch
2. "drop caches"
3. Read all files. time it.
4. Immediately read all files again

```
[root@n004 tmp]# mkdir /scratch/htcfadmin/chr
[root@n004 tmp]# cp * /scratch/htcfadmin/chr/
[root@n004 tmp]# cd /scratch/htcfadmin/chr/
[root@n004 chr]# ls -l
total 1948449
-rw-r--r-- 1 root htcfadmin 136473378 Sep 18 15:18 chr10.fa
-rw-r--r-- 1 root htcfadmin 116651622 Sep 18 15:18 chr13.fa
-rw-r--r-- 1 root htcfadmin 109184600 Sep 18 15:18 chr14.fa
-rw-r--r-- 1 root htcfadmin  84922597 Sep 18 15:18 chr17.fa
-rw-r--r-- 1 root htcfadmin 253935557 Sep 18 15:18 chr1.fa
-rw-r--r-- 1 root htcfadmin  65733058 Sep 18 15:18 chr20.fa
-rw-r--r-- 1 root htcfadmin  47644190 Sep 18 15:18 chr21.fa
-rw-r--r-- 1 root htcfadmin  51834845 Sep 18 15:18 chr22.fa
-rw-r--r-- 1 root htcfadmin 247037406 Sep 18 15:18 chr2.fa
-rw-r--r-- 1 root htcfadmin 202261477 Sep 18 15:18 chr3.fa
-rw-r--r-- 1 root htcfadmin 194018853 Sep 18 15:18 chr4.fa
-rw-r--r-- 1 root htcfadmin 185169031 Sep 18 15:18 chr5.fa
-rw-r--r-- 1 root htcfadmin 141162618 Sep 18 15:18 chr9.fa
-rw-r--r-- 1 root htcfadmin     16907 Sep 18 15:18 chrM.fa
-rw-r--r-- 1 root htcfadmin 159161719 Sep 18 15:18 chrX.fa
[root@n004 chr]# sudo sysctl vm.drop_caches=1
vm.drop_caches = 1
[root@n004 chr]# time cat * > /dev/null

real    0m3.284s
user    0m0.003s
sys     0m0.600s
[root@n004 chr]# time cat * > /dev/null

real    0m3.160s
user    0m0.003s
sys     0m0.597s
[root@n004 chr]#
```

Crash course: read-ahead and small reads

Tiny script to demonstrate kernel read-ahead.

1. Create 10MB file
2. Read...
    a) 1 x 10MB chunk
    b) 10 x 1MB chunks
    c) 1280 x 8KB chunks
    d) 2560 x 4KB chunks
    e) 10MB x 1 Byte chunks

```
#!/usr/bin/env python3

import os
import sys
import tempfile
import time

TOTAL = 1024 * 1024 * 10
SIZES = [TOTAL, 1024*1024, 8192, 4096, 1024, 1]
fname = tempfile.mkstemp(dir='.')[1]

def go():
    sys.stdout.write("Writing {} bytes...".format(TOTAL))
    open(fname, 'wb').write(b'\0' * TOTAL)
    sys.stdout.write("Done\n")

    for rs in SIZES:
        with open(fname, 'rb', 0) as f:
            sys.stdout.write("Read size {}: ".format(rs))
            sys.stdout.flush()
            t = time.time()
            while True:
                x = f.read(rs)
                if x == b'': break
            sys.stdout.write('{}\n'.format(time.time() - t))

    os.unlink(fname)

if __name__ == '__main__':
    go()
```

Crash course: read-ahead and small reads

Tiny script to demonstrate kernel read-ahead.

1. Create 10MB file
2. Read…
    a) 1 x 10MB chunk
    b) 10 x 1MB chunks
    c) 1280 x 8KB chunks
    d) 2560 x 4KB chunks
    e) 10MB x 1 Byte chunks

```
[koebbe@n004 tmp]$ read-demo.py
Writing 10485760 bytes...Done
Read size 10485760: 0.003555774688720703
Read size 1048576: 0.0024728775024414062
Read size 8192: 0.002218008041381836
Read size 4096: 0.002351999282836914
Read size 1024: 0.005714893341064453
Read size 1: 4.551784038543701
[koebbe@n004 tmp]$
```

```
[koebbe@n004 ~]$ read-demo.py
Writing 10485760 bytes...Done
Read size 10485760: 0.0037994384765625
Read size 1048576: 0.0024938583374023438
Read size 8192: 0.0023012161254882812
Read size 4096: 0.0025272369384765625
Read size 1024: 0.006322383880615234
Read size 1: 4.890070915222168
[koebbe@n004 ~]$
```

```
[koebbe@n004 ~]$ cd /ref/htcfadmin/data
[koebbe@n004 data]$ read-demo.py
Writing 10485760 bytes...Done
Read size 10485760: 0.0035457611083984375
Read size 1048576: 0.002311229705810547
Read size 8192: 0.0023055076599121094
Read size 4096: 0.0027718544006347656
Read size 1024: 0.007003545761108398 4
Read size 1: 5.990774393081665
[koebbe@n004 data]$
```

```
[koebbe@n004 ~]$ cd /scratch/htcfadmin
[koebbe@n004 htcfadmin]$ read-demo.py
Writing 10485760 bytes...Done
Read size 10485760: 0.013638973236083984
Read size 1048576: 0.01630091667175293
Read size 8192: 0.14689421653747559
Read size 4096: 0.29636263847351074
Read size 1024: 1.041081428527832
Read size 1: 877.302651643753
[koebbe@n004 htcfadmin]$
```

Job bottlenecks

- All jobs have bottlenecks...
  - CPU
  - I/O
  - RAM (CPU or I/O in disguise?)
- Most HTCF jobs, at some point in their lifetime, are I/O bound.
- How do we determine if and how these jobs can be optimized?

I/O Bound:

...a condition in which the time it takes to complete a ***computation*** *is determined principally by the period spent* ***waiting*** *for input/output operations to be completed.*

strace: trace system calls

- monitors all system calls that a process makes.
- Can prefix a command or attach to an already running command.
- Can be used to see how many read/write calls are made and how long those reads/writes take.

```
1  #!/usr/bin/env python3
2
3  import os
4  import sys
5  import tempfile
6
7  fname = sys.argv[1]
8  size = int(sys.argv[2])
9
10 with open(fname, 'rb', 0) as f:
11     while True:
12         x = f.read(size)
13         if x == b'': break
14
```

strace: trace system calls

```
[koebbe@n004 ~]$ cd /scratch/htcfadmin/
[koebbe@n004 htcfadmin]$ strace strace-demo.py file.fastq 2048
```

```
openat(AT_FDCWD, "file.fastq", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0664, st_size=10485760, ...}) = 0
ioctl(3, TCGETS, 0x7ffcf2bba040)        = -1 ENOTTY (Inappropriate ioctl for device)
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 2048) = 2048
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 2048) = 2048
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 2048) = 2048
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 2048) = 2048
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 2048) = 2048
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 2048) = 2048
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 2048) = 2048
read(3, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 2048) = 2048
```

strace: trace system calls

strace -c -f strace-demo.py file.fastq 2048

```
[koebbe@n004 htcfadmin]$ strace -c strace-demo.py file.fastq 2048
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 90.83    0.019275           3      5280           read
  2.11    0.000448           1       268        24 stat
  1.08    0.000230           2        94         2 openat
  0.98    0.000209           2        87           mmap
  0.91    0.000194          32         6         4 execve
  0.81    0.000171           1       145           fstat
  0.77    0.000164           1        95           close
  0.64    0.000136           7        18           getdents64
  0.47    0.000100           2        50           mprotect
  0.44    0.000094           1        91         6 lseek
  0.21    0.000045           3        13           munmap
  0.19    0.000041           1        28           brk
  0.10    0.000022           2         8         2 readlink
  0.10    0.000021           0        68           rt_sigaction
  0.08    0.000016           0        21           lstat
```

strace: trace system calls

strace -c -f -e read strace-demo.py file.fastq 2048

```
[koebbe@n004 htcfadmin]$ strace -c -e read strace-demo.py file.fastq 2048
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
100.00    0.016388           3      5280           read
------ ----------- ----------- --------- --------- ----------------
100.00    0.016388           3      5280           total
```

strace: trace system calls

```
[koebbe@n004 htcfadmin]$ strace -c strace-demo.py file.fastq 10485760
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 67.28    0.004224          26       161           read
  6.42    0.000403          67         6         4 execve
  3.52    0.000221           2        89           mmap
  3.27    0.000205           2        94         2 openat
  3.25    0.000204           0       268        24 stat
  2.20    0.000138           2        50           mprotect
  2.15    0.000135           1        68           rt_sigaction
  1.94    0.000122           0       145           fstat
  1.88    0.000118           1        95           close
  1.88    0.000118           7        15           munmap
  1.62    0.000102           5        18           getdents64
  1.40    0.000088           3        28           brk
  0.83    0.000052           0        91         6 lseek
  0.68    0.000043           2        21           lstat
  0.29    0.000018           6         3         3 access
```

```
[koebbe@n004 htcfadmin]$ strace -c strace-demo.py file.fastq 1
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 99.99   32.968573           3  10485920           read
  0.00    0.000367           1       268        24 stat
  0.00    0.000290           3        94         2 openat
  0.00    0.000276           3        87           mmap
  0.00    0.000209           4        50           mprotect
  0.00    0.000156           1       145           fstat
  0.00    0.000151           1        95           close
  0.00    0.000147          24         6         4 execve
  0.00    0.000098           1        91         6 lseek
  0.00    0.000062           3        18           getdents64
  0.00    0.000061           4        13           munmap
  0.00    0.000044           1        28           brk
  0.00    0.000027           3         9           futex
  0.00    0.000026           8         3           getrandom
```

Data Type Locations

What are the best locations for these various types of data?

/tmp

"Small Read" Data

- STAR, SPAdes, Cellranger

/scratch

Staged Input Data

Intermediate Data

Log Files

/ref (write few / read many / small-read safe)

Reference Data

Annotations, NR/NT

Software & Software Environments

/home

sbatch Scripts?

Configuration Files

/lts

Job scripts

Raw Sequence Data

Finished Results

LTOS

Job scripts

Raw Sequence Data

Finished Results

Using LTOS – Long Term Object Storage

Like LTS, with 2 big differences:

1. Available to jobs (accessible from nodes).
2. Not part of the file system
    1. Can't navigate to it the traditional ways. (ie. using **`cd`** and **`ls`**).
    2. Files (objects) are fetched (get) and placed (put) into LTOS.

Using LTOS – Long Term Object Storage

/lts -> /scratch

- manually walking over to freezer
- Finding and loading up **everything** needed for many experiments
- dumping it all over the bench
- doing the work
- taking everything back to the freezer.

- Messy, disorganized, dangerous
- What happens if there's an error in the middle?
- Cleanup is hard/daunting...what to keep, what to dispose of.

LTOS -> /scratch

- having a robot quickly go grab only enough from the freezer to have a still-organized bench
- Robot doing a batch of work on the bench
- taking batch to freezer
- Getting next batch
- Doing batch of work, taking it back, etc.

- More upfront thought needed
- Organized, easier to recover from an error
- Better error reporting
- Easier to "reuse" components of the process for future experiments.
- More hands-off

Using LTOS: Basic Job

1. Map reads
2. Sort alignments
3. Index read alignments
4. Variant calling

```
[koebbe@n004 example]$ tree data
data
├── genome.fa
├── genome.fa.amb
├── genome.fa.ann
├── genome.fa.bwt
├── genome.fa.fai
├── genome.fa.pac
├── genome.fa.sa
└── samples
    ├── A.fastq
    ├── B.fastq
    └── C.fastq

1 directory, 10 files
```

Using LTOS: Basic Job

Before:

1. On login server, manually copy data from /lts to /scratch
2. Job(s) to process the data
3. On login server, manually copy data from /scratch to /lts.

For large gnome(s) or 1000s of samples:

- Huge and/or messy working space
- more manual copying back/forth

```
[koebbe@n004 example]$ tree data
data
├── genome.fa
├── genome.fa.amb
├── genome.fa.ann
├── genome.fa.bwt
├── genome.fa.fai
├── genome.fa.pac
├── genome.fa.sa
└── samples
    ├── A.fastq
    ├── B.fastq
    └── C.fastq

1 directory, 10 files
```

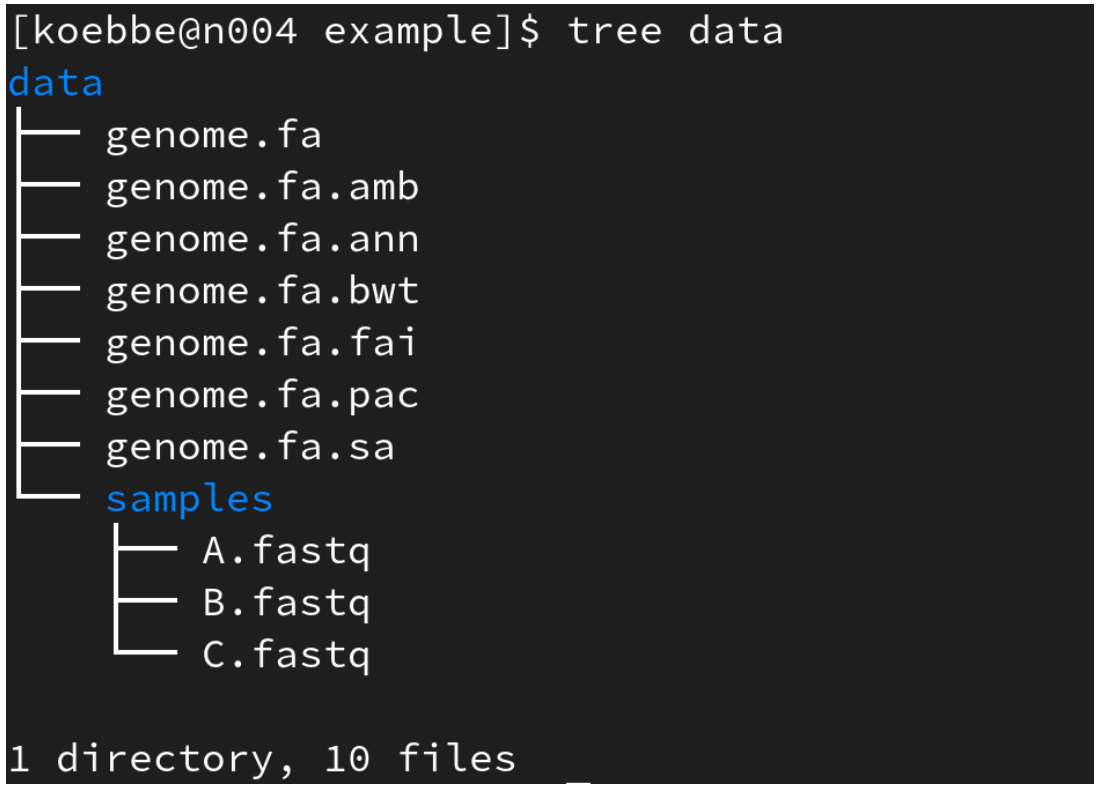
Using LTOS: Basic Job

Using LTOS without workflow:

1. Create job(s) to stage data on /scratch, process the data, save results to LTOS

No manual steps!

Fine for small # samples...still hard for large # samples.



Using LTOS: Basic Job

Using LTOS **with** workflow:

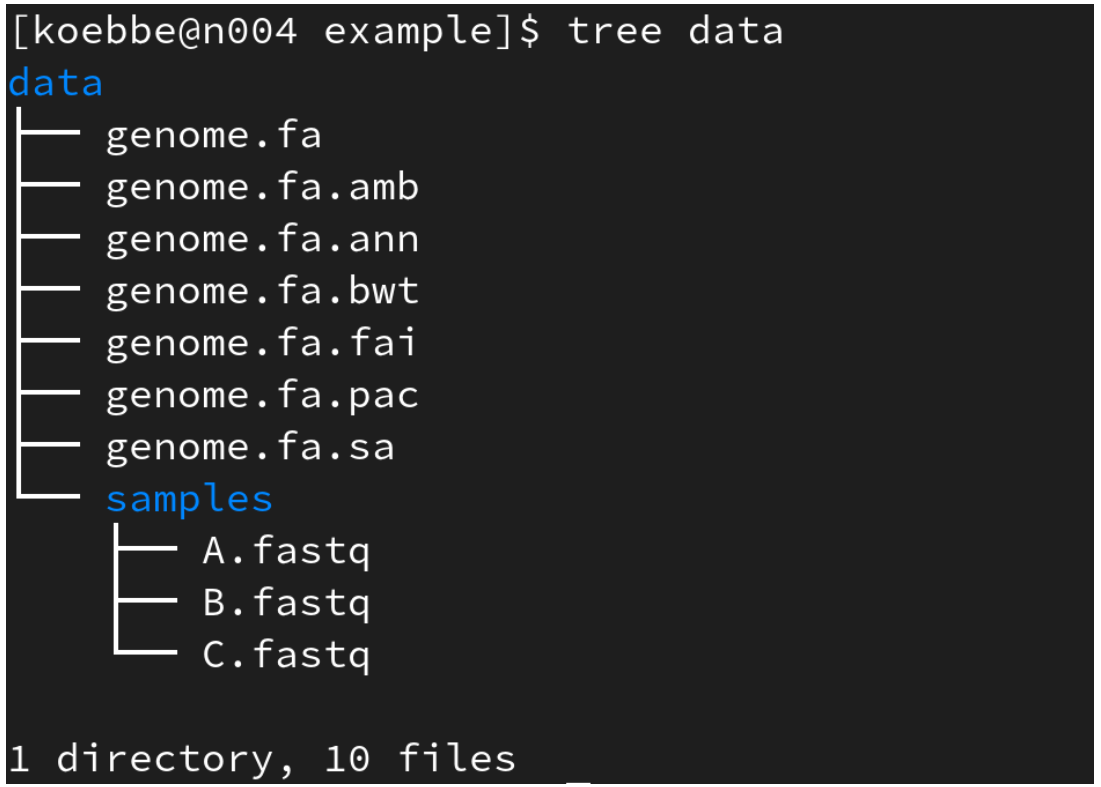
1. Create job(s) to stage data on /scratch, process the data, save results to LTOS

No manual steps!

Great for small or large # samples.

Easy to pick up where left off.

Good logging/reporting.



Using LTOS: Prep Example

Upload data to LTOS:

1. Make bucket (mb)
2. Recursively (-r) put the data in the newly created bucket

```
#!/bin/bash

eval $(spack load --sh py-s3cmd@2.3.0)

export S3CMD_CONFIG=${HOME}/ltos-obs1.conf

s3cmd mb s3://training
s3cmd put -r data s3://training
```

Using LTOS: "Hard Coded"

```
#!/bin/bash

eval $(spack load --sh py-s3cmd@2.3.0 bwa samtools bcftools)
export S3CMD_CONFIG=${HOME}/ltos-obs1.conf

mkdir data
s3cmd get s3://training/data/* data

#
# Map reads
#
s3cmd get s3://training/data/samples/A.fastq .
s3cmd get s3://training/data/samples/B.fastq .
s3cmd get s3://training/data/samples/C.fastq .

bwa mem data/genome.fa A.fastq | samtools view -Sb - > A.bam
bwa mem data/genome.fa B.fastq | samtools view -Sb - > B.bam
bwa mem data/genome.fa C.fastq | samtools view -Sb - > C.bam
```

Using LTOS: "Hard Coded"

```
#
# Sort alignments
#
samtools sort -o A.sorted.bam A.bam
samtools sort -o B.sorted.bam B.bam
samtools sort -o C.sorted.bam C.bam

#
# Call variants
#
bcftools mpileup -f data/genome.fa A.sorted.bam B.sorted.bam C.sorted.bam | bcftools call -mv - > all.vcf

s3cmd put all.vcf s3://training/results/all.vcf

#
# clean up
#
rm A.fastq B.fastq C.fastq
rm A.bam B.bam C.bam
rm A.sorted.bam B.sorted.bam C.sorted.bam
rm all.vcf
rm -rf data
```

Using LTOS: A little less hard coded

```
SAMPLES=(A B C)
for sample in "${SAMPLES[@]}"; do

    #
    # Map reads
    #
    s3cmd get s3://training/data/samples/${sample}.fastq .
    bwa mem data/genome.fa ${sample}.fastq | samtools view -Sb - > ${sample}.bam
    rm ${sample}.fastq

    #
    # Sort alignments
    #

    samtools sort -o ${sample}.sorted.bam ${sample}.bam
    rm ${sample}.bam
done
```

Using LTOS: parallelize

3 jobs

1. **Stage Data**
2. Array job to map and sort 3 samples
3. Call variants and clean up

```
#!/bin/bash

eval $(spack load --sh py-s3cmd@2.3.0)

export S3CMD_CONFIG=${HOME}/ltos-obs1.conf

set -e

mkdir data
s3cmd get s3://training/data/* data
```

Using LTOS: parallelize

3 jobs

1. Stage Data
2. **Array job to map and sort 3 samples**
3. Call variants and clean up

```
#!/bin/bash

#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=1G
#SBATCH --array=0-2

eval $(spack load --sh py-s3cmd@2.3.0 bwa samtools bcftools)
export S3CMD_CONFIG=${HOME}/ltos-obs1.conf

set -e
set -x

SAMPLES=(A B C)
sample=${SAMPLES[${SLURM_ARRAY_TASK_ID}]}

#
# Map reads
#
s3cmd get s3://training/data/samples/${sample}.fastq .
bwa mem data/genome.fa ${sample}.fastq | samtools view -Sb - > ${sample}.bam
#
# Sort alignments
#
samtools sort -o ${sample}.sorted.bam ${sample}.bam
```

Using LTOS: parallelize

3 jobs

1. Stage Data
2. Array job to map and sort 3 samples
3. **Call variants and clean up**

```
#!/bin/bash

eval $(spack load --sh py-s3cmd@2.3.0 bcftools)

set -e
set -x

export S3CMD_CONFIG=${HOME}/p/storage-training/ltos-obs1.conf

SAMPLES=(A B C)

#
# Call variants
#
bcftools mpileup -f data/genome.fa ${SAMPLES[@]/%/.sorted.bam} | bcftools call -mv - > all.vcf

s3cmd put all.vcf s3://training/results/all.vcf

#
# clean up
#
rm ${SAMPLES[@]/%/.fastq}
rm ${SAMPLES[@]/%/.bam}
rm ${SAMPLES[@]/%/.sorted.bam}
rm all.vcf
rm -rf data
```

Using LTOS: parallelize

3 jobs

1. Stage Data
2. Array job to map and sort 3 samples
3. Call variants and clean up
4. **Tie it all together**

```
#!/bin/bash

ID=$(sbatch job1.sbatch | awk '{ print $NF }')

ID=$(sbatch -d afterok:${ID} job2.sbatch |awk '{ print $NF }')

sbatch -d afterok:${ID} job3.sbatch
```

Using LTOS: workflow

1 Snakefile

1. Define rules
   a) **map_reads**
   b) sort_alignments
   c) samtools_index
   d) call_variants
2. Run snakemake
   a) Sequentially
   b) Parallel
   c) Full slurm job

```
rule map_reads:
    input:
        fasta=S3.remote("training/data/genome.fa"),
        genomefiles=[
            S3.remote("training/data/genome.fa.amb"),
            S3.remote("training/data/genome.fa.ann"),
            S3.remote("training/data/genome.fa.bwt"),
            S3.remote("training/data/genome.fa.fai"),
            S3.remote("training/data/genome.fa.pac"),
            S3.remote("training/data/genome.fa.sa")
        ],
        reads=S3.remote("training/data/samples/{sample}.fastq")
    output:
        temporary("mapped_reads/{sample}.bam")
    shell:
        """
        eval $(spack load --sh bwa samtools)
        bwa mem {input.fasta} {input.reads} | samtools view -b - > {output}
        """
```

Using LTOS: workflow

1 Snakefile

1. Define rules
   a) map_reads
   b) sort_alignments
   c) samtools_index
   d) call_variants
2. Run snakemake
   a) Sequentially
   b) Parallel
   c) Full slurm job

```
rule sort_alignments:
    input:
        "mapped_reads/{sample}.bam"
    output:
        temporary("sorted_reads/{sample}.bam")
    shell:
        """
        eval $(spack load --sh samtools)
        samtools sort -T sorted_reads/{wildcards.sample} -O bam {input} > {output}
        """
```

Using LTOS: workflow

1 Snakefile

1. Define rules
   a) map_reads
   b) sort_alignments
   c) samtools_index
   d) call_variants
2. Run snakemake
   a) Sequentially
   b) Parallel
   c) Full slurm job

```
rule samtools_index:
    input:
        "sorted_reads/{sample}.bam"
    output:
        temporary("sorted_reads/{sample}.bam.bai")
    shell:
        """
        eval $(spack load --sh samtools)
        samtools index {input}
        """
```

Using LTOS: workflow

1 Snakefile

1. Define rules
   a) map_reads
   b) sort_alignments
   c) samtools_index
   d) call_variants
2. Run snakemake
   a) Sequentially
   b) Parallel
   c) Full slurm job

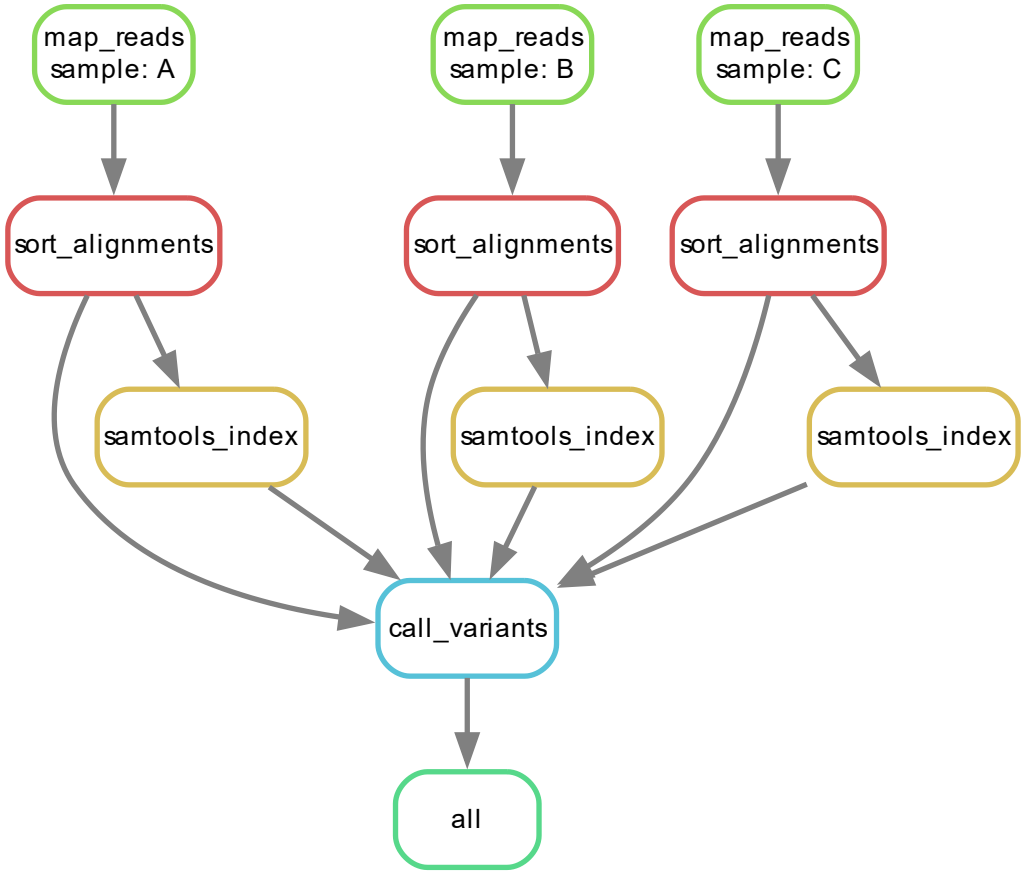
```
rule call_variants:
    input:
        fa=S3.remote("training/data/genome.fa"),
        bam=expand("sorted_reads/{sample}.bam", sample=SAMPLES),
        bai=expand("sorted_reads/{sample}.bam.bai", sample=SAMPLES)
    output:
        all=S3.remote("training/results/all.vcf"),
        foo=temporary("training/data/genome.fa.fai")
    shell:
        """
        eval $(spack load --sh bcftools)
        bcftools mpileup -f {input.fa} {input.bam} | bcftools call -mv - > {output.all}
        """
```

Using LTOS: workflow

```
$ snakemake --dag  | dot -Tsvg > dag.svg
```

1 Snakefile

1. Define rules
   a) map_reads
   b) sort_alignments
   c) samtools_index
   d) call_variants
2. Run snakemake
   a) Sequentially
   b) Parallel
   c) Full slurm job



Using LTOS: workflow

1 Snakefile

1. Define rules
   a) map_reads
   b) sort_alignments
   c) samtools_index
   d) call_variants
2. Run snakemake
   a) Sequentially
   b) Parallel
   c) Full slurm job

```
$ snakemake -c 1
```

```
$ snakemake -c 3
```

```
$ snakemake --slurm --jobs 3
```

Using LTOS: workflow

1 Snakefile

1. Define rules
   a) map_reads
   b) sort_alignments
   c) samtools_index
   d) call_variants
2. Run snakemake
   a) Sequentially
   b) Parallel
   c) Full slurm job

```
[koebbe@n001 job3]$ snakemake --report
Building DAG of jobs...
Creating report...
Downloading resources and rendering HTML.
Report created: report.html.
[koebbe@n001 job3]$
```