Storage and I/O in Large Scale Scientific Projects

Ramses van Zon and Marcelo Ponce

May 17, 2017

Outline of the course

1. Intro to Storage & Performance
2. Performance Tuning
3. File Formats
4. Data Management

Section 1

Intro to Storage & Performance

Data deluge

- The amount of data available and generated is growing at break-neck speed.
- This data has to be processed and stored.
- The capacity and speed of data storage and data access have not kept up with the increase in data volume.
- This can cause the storage and retrieval to become a bottleneck in your projects (even if your data is not necessarily big).

We will show you common bottlenecks in data storage and handling, and how you can alleviate them in many cases.

Let's look at some cases of 'big data'. . .

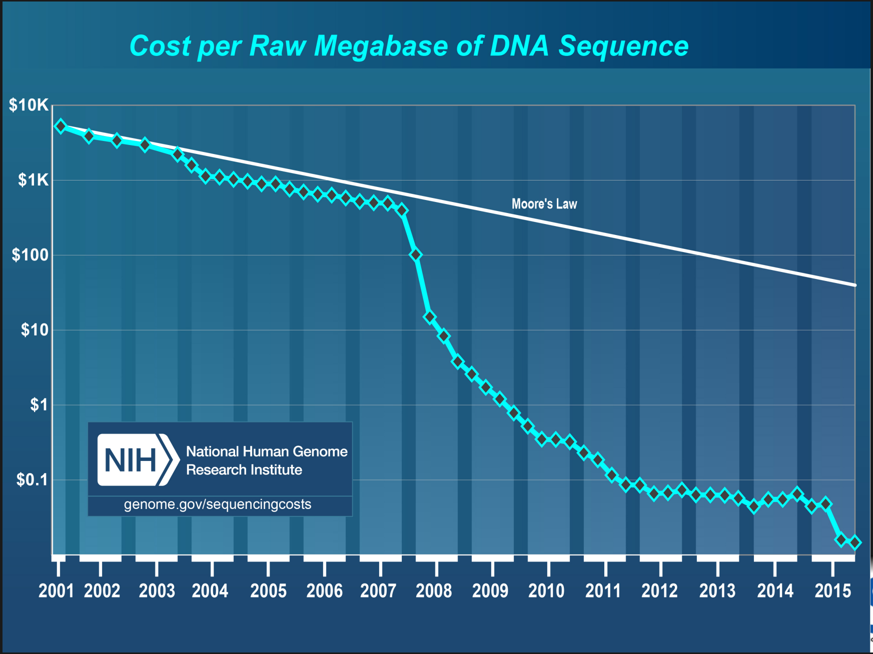
Bioinformatics as an example of Big Data

Bioinformatics is a broad area of research

- Next Gen Sequencing
- Data Analysis
- Alignment
- Assembly
- Simulation

Common features

- Often involves a lot of little bits of data
- Involves a lot of analysis


Cost per Raw Megabase of DNA Sequence
$10K
$1K
$100
$10
$1
$0.1
Moore's Law
NIH National Human Genome Research Institute
genome.gov/sequencingcosts
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015

Bioinformatics & HPC

Suitable for Typical HPC Systems?

Typical HPC cluster:

- Optimized for parallel, floating point calculations.
- Memory per core typically modest (1-3 GB).
- High performance network within cluster.
- Good external transfer rates only if good from end to end.
- Disk storage optimized for large contiguous blocks of data.
- Disk system often the least optimized.
- Shared resource.

For HPC to solve large bioinformatics questions requires some rethinking.

Different applications may have quite different optimal workflows, but the boundary conditions remain the same.

Molecular biophysics

. . . another broad category

- Complex molecules
- Dynamical behaviour to be simulated over time scales that require many millions of time steps (e.g., protein folding, ion translocation)
- Often vast parameter space

The properties holds for other projects too, if they involve complex many-body simulations.

Common features

- Large parameter space
- Lots of output data (trajectories)
- That then requires analysis

Astrophysics

- Telescopes: large quantities of images are produced;
- Gravitational wave detectors (e.g. LIGO): large numbers of 'waveforms' to be produced and analyzed.
- Data is sizable
- Filters may have to be applied on each sample/image
- Images may have to be 'stitched' together (WMAP)
- Until processed, large amount of data to be stored.

Medical physics

Imaging

- Large quantities of images are produced (MRI);
- Data is sizable;
- Filters may have to be applied on each;
- Images may have to be 'stitched' together (maps, 3d imaging, . . .).
- Until processed, large amount of data to be stored.

DNA screening

See bioinformatics.

Digital Humanities and Social Studies

Digital Humanities

- Many books, articles, etc., have been digitized, and could be searchable for a variety of research goals.

- Social media generates a lot of data
- Getting the information out, i.e., the apropriate analysis, is challenging.
- Automation can also generate large sample sizes than before.

Similarity of Symbol Frequency Distributions with Heavy Tails – Phys. Rev. X 6, 021009 (2016), *Gerlach, Font-Clos, and Altmann*

Common Computational Challenges:

Workflow

What gets done, with what data, in what sequence (or in parallel), and what tasks or items can share resources?

Data Management and Data Transport

What goes where, how fast, how big is it, what is the format?

Throughput

Regardless of the speed of workflow components, what matters is how much data we get to process per second (or per Watt).

Of course these concerns are valid for any large scale scientific computation, but the pace at which data gets produced in data-driven fields makes them more pressing.

Computational Infrastructure

- File systems
- Modern computers (CPU, RAM, DISK)
- Supercomputers
- Networks
- Storage (and 'clouds')
- Linux
- Schedulers

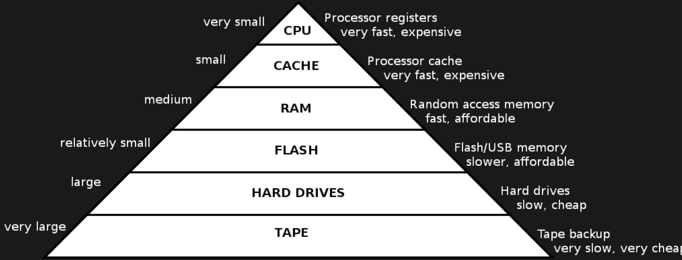
In this workshop, we will focus on I/O, which is often the bottleneck.

Why is I/O a bottleneck?

Your data may live on disk, or tape, or come from the internet/cloud.
Processing occurs on the CPU. For data to get there, it has to go through the Memory/Storage Hierarchy

Location	Latency	Speed
CPU	300 ps	
Cache	5 ns	50 GB/s
RAM	60 ns	10 GB/s
Infiniband	1 μs	6 GB/s
Flash	20 μs	200 MB/s
Ethernet (1Gb)	50 μs	100 MB/s
Hard Drives	5 ms	100 MB/s
Shared File System	10 ms	10 MB/s
Tape	1 h	80 MB/s
Internet	100 ms	100 MB/s

(beware: very approximate numbers!)



100 – 1000 fold difference between how fast your data could be processed by the CPU and how fast the data can get there!

File I/O

File systems

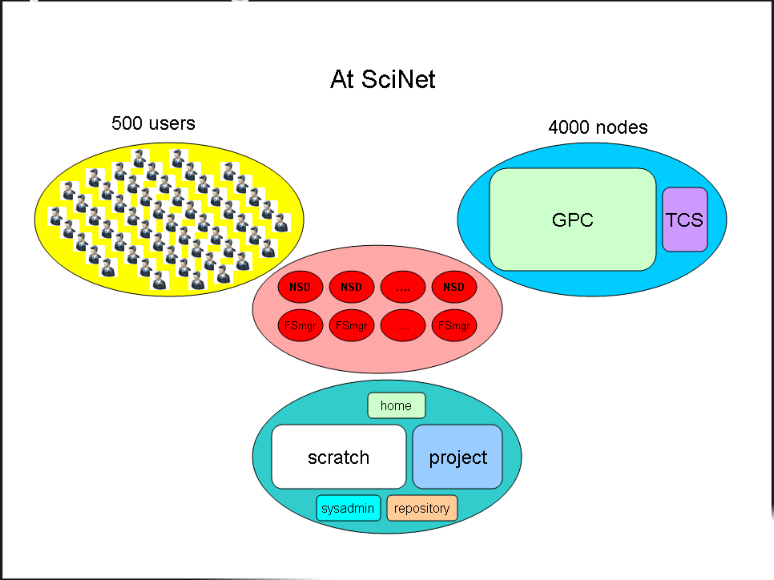
- It's where we keep most data.
- Typically spinning disks
- Logical structure: directories, subdirectories and files.
- On disk, these are just blocks of bytes.
- Each I/O operation (IOPS) gets hit by latency.

File I/O

What are I/O operations, or IOPS?

- Finding a file (ls)
  Check if that file exists, read metadata (file size, date stamp etc.)
- Opening a file:
  Check if that file exists, see if opening the file is allowed, possibly create it, find the block that has the (first part of) the file system.
- Reading a file:
  Position to the right spot, read a block, take out right part
- Writing to a file:
  Check where there is space, position to that spot, write the block.
  Repeated if the data read/written spans multiple blocks.
- Move the file pointer ("seek"):
  File system must check were on disk the data is.
- Close the file.

Parallel file system at a glance



File system at Supercomputer Centres such as SciNet

Taking SciNet as an example:

- 1,790 1TB SATA disk drives, for a total of 1.4PB
- Two DCS9900 couplets, each delivering:
  - 4-5 GB/s read/write access (bandwidth)
  - 30,000 IOP/s max (open, close, seek, . . .)
- Single *GPFS* file system shared on most systems
- I/O goes over the network
- File system is *parallel!*

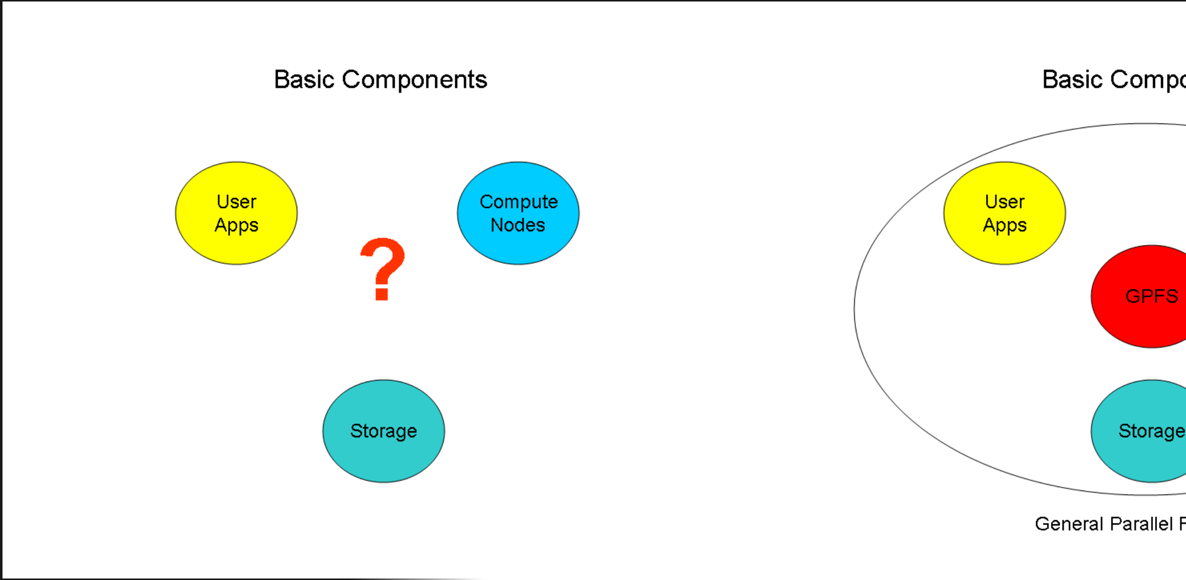
Example: File system limits at SciNet

location	quota	block-size	time-limit	backup	devel	comp
/home	50GB	256kB	unlimited	yes	rw	ro
/scratch	20TB	4MB	3 months	no	rw	rw

- There are quotas
- Home read-only from compute nodes!
- Big block sizes: *small files waste space*
- Issues are common to parallel file systems (Lustre, etc.) present in most modern supercomputers.
- Scratch quota per user oversubscribes disk space, so only for when you *temporarily* really needs a lot of disk space.
- Most users will need much less.

The file system is parallel, what does that mean?

Shared file system



Shared file system

- Optimal for large shared files.
- Behaves poorly under many small reads and writes.
- Your use of it affects everybody!
  (Different from case with CPU and RAM if scheduling is by node.)
- How you read and write, your file format, the number of files in a directory, and how often you `ls`, can all affect every other user!
- File systems are not infinite!
  Bandwidth, metadata, IOPS, number of files, space, . . .

Shared file system

- Think of your laptop/desktop with several people simultaneously doing I/O, doing `ls` on directories with thousands of files . . .
- 2 jobs doing simultaneous I/O can take *much* longer than twice a single job duration due to disk *contention* and directory *locking*.
- E.g. @SciNet: >100 users doing I/O from 4000 nodes.
  That's a lot of sharing and contention!

Some Numbers

SciNet's scratch, which is meant for temporary space:

- 1.8 PB TB on scratch
- $>$ 100 active users, but $>$ 800 users total.
- Would prefer >25% free at any given time.
  (systems can write 0.5 PB per day)
- But experience shows that all space eventually gets filled.
- 100 MB/s: maximum possible read/write speed from a node if there is nothing else running on system

When system is fully utilized:

- 1 MB/s: average expected read/write speed from a node
- 10 IOP/s: average expected iops from a node
  So can't open more than 10 files in a second!

Conclusion:

The I/O part of your project needs as much attention as your algorithms and workflows.

Next: Scripting Review

Why? Because we can often improve our I/O by changing our workflow, using scripts.

Scripting is . . .

Automating tasks. . .

- Reproducable
- Debuggable and scalable

At a high level. . .

- Easy to learn
- Should be easy to read

Using an interpreted language (typically)

- No need to compile, but slow
- Text based: easy to check
- Many choices: bash, zsh, tcsh, perl, python, ruby.

We'll use bash: the same language as that used on the command line and in job scripts (linux-centric? Yes, but so is HPC in general)

A bit of bash: Basic commands

echo [string]	Prints output
time [cmd]	Performs cmd and prints how long it took
man [cmd]	Get the manual for the command cmd
if/then/else/fi	Conditional statement
for/do/done	For loop
function	Define a function
$(cmd)	Run a command and return its output as a string

Example

```
#!/bin/bash
echo Hi!
time echo Hi!
```

Put this in 'ex1.sh'

```
chmod +x ex1.sh
./ex1.sh
```

Example

```
#!/bin/bash
if [ this = that ]
then
   echo this=that
else
   echo this!=that
fi
```

Example

```
#!/bin/bash
a=0
b=5
for ((i=a;i<=b;i++))
do
   echo $i
done
```

A bit of bash: variables

- Declare and initialize a variable:
  `varname="string"`
- To use the variable, you need a dollar sign:
  `echo $varname`
- To read a variable from the command line, use
  `read varname`

Special variables

- **`$1,$2, ...`**: Arguments given to a command or function
- **`$?`**: Error code of the last command

Storing output in a variables

- **`varname=$(cmd)`**
  stores the output of the command **`cmd`** in the variable **`varname`**

A bit of bash: Basic commands

Directories

ls [directory]	Directory LiSting
pwd	Print current directory
cd [directory]	Change directory
mkdir [filename]	Create directory
rmdir [filename]	Remove directory
du	Estimate file space usage

Files

file [file(s)]	Print out file type
cat [file(s)]	Prints content of file(s)
less [file(s)]	Prints out file(s) by page
rm [file(s)]	Delete file(s)
mv [src] [dest]	Move file or directory
cp [src] [dest]	Copy file or directory (-r)

File selection, manipulatation and management

File selection

- You can give a pattern instead of a name.
- ? matches one charcater
- * matches anything
- These get resolved to all filenames that match the pattern

File manipulation

wc [filename]	Line/word/character count of file
grep [text] [file(s)]	Searches files for text

Data management stuff

tar [archive] [file(s)]	Archive multiple files to one file
gzip [file(s)]	Compress files
scp [file(s)] [machine:file(s)]	Copy to different machine
scp -r [dir] [machine:directory]	Copy to different machine
rsync [options] [directory] [machine:directory]	Copy new files

A bit of bash: Redirection

- **[cmd] > [filename]** takes what would have gone to the screen, creates a new file **[filename]**, and redirects output to that file.
- Overwrites previous contents of file if it had existed.
- **[cmd] >> [filename]** appends to **[filename]** if it exists.
- **[cmd] < [filename]** means programs input comes from file, as if you were typing.

Example

```
#!/bin/bash
echo Hello > hello.txt
read s < hello.txt
echo World >> hello.txt
cat hello.txt
```

A bit of bash: Pipelines

- The idea of chaining commands together - the output from one becomes the input of another - is part of what makes the shell (and programming generally) so powerful.
- Instead of

```
$ [cmd1] > [file]
$ [cmd2] < [file]
```

one can say

```
$ [cmd1] | [cmd2]
```

The output of **[cmd1]** becomes the input of **[cmd2]**

- Easier and avoids creating a temporary file

```
echo Hi > somelines.txt
echo An i in this line >> somelines.txt
echo But not here >> somelines.txt
grep i somelines.txt | wc
```

A bit of bash: for loops

- Bash has for loops much like any programming language does.
- Loops are word list based:
  `for [varname] in [list]`
- or range based:
  `for ((varname=start;varname<=finish;varname++))`
- Block of commands in the loop should be between **`do`** and **`done`**.

Example

```
for word in how are you
do
   echo $word
done
```

```
how
are
you
```

A bit of bash: Performance Commands

Runtime performance

time COMMAND	Time execution of a command
top [-u USER]	Display state of current processes
vmstat [INTERVAL]	Report memory, cpu and io of current node,
nodeperf (GPC only)	Current node's memory, cpu usage, and procs
jobperf JOBID (GPC only)	Mem, cpu, and procs used by JOBID

Statistics (GPC only)

diskUsage/quota	How much are you storage using?
scinet usage	How much cpu have you used?
scinet gpc priority	What is your current priority for compute jobs?

(exploratory) hands on: copying files, taring, speeds

```
$ ssh login.scinet.utoronto.ca
$ gpc
$ debugjob
$ cd $SCRATCH
$ time cp -r /scinet/course/data/rotteneggs .
$ source rotteneggs/setup
$ cd rotteneggs/largesamples/
```

Questions:

- How large is the data in the directory?
- How many files does it contain?
- How long does it take to tar all files up?
- How long does it take to copy it to another directory?
- How long does it take to copy it to gpc01?

Use 'time' for timing. While running, log into your node in another window, and monitor with top/nodeperf/vmstat.

(exploratory) hands on: copying files, taring, speeds

- How large is the data in the directory?

```
$ du .
3045952 .
```

- How many files does it contain?

```
$ ls | wc -w
524
```

- How long does it take to tar all files up?

```
$ time tar cf allfiles.tar *
real 0m24.646s
user 0m0.170s
sys 0m4.217s
```

- How long to copy it to another directory?

```
$ mkdir ../newdir
$ time cp allfiles.tar ../newdir/
real 0m2.261s
user 0m0.002s
sys 0m1.847s
```

- How long does it take to copy it to gpc01?

```
$ time scp allfiles.tar gpc01:
real 0m30.831s
user 0m24.480s
sys 0m7.038s
```

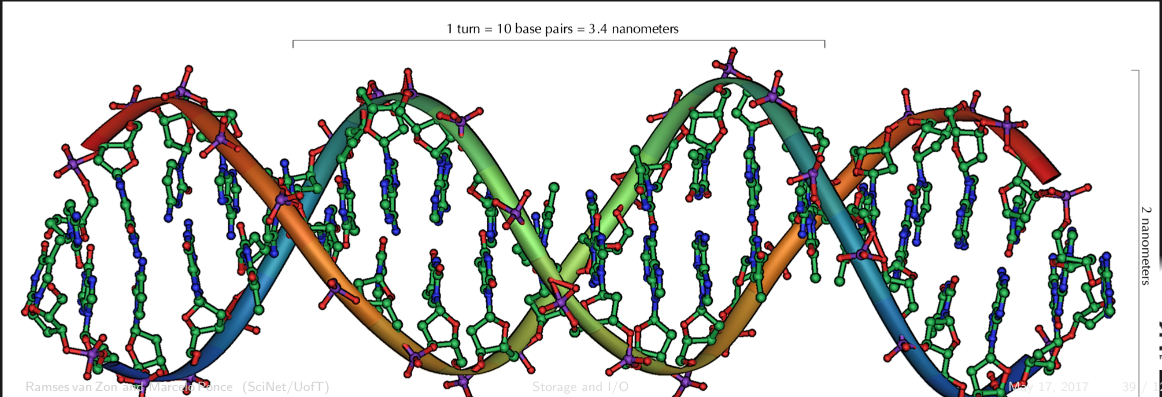
Section 2

Use Case

Use case (based on a true story)

Project

Given a number of batches of DNA fragments taken from 12 eggs, we want to know which batches contain salmonella by comparing ("aligning") against a reference salmonella genome and a reference chicken genome.



Alignment

Alignment in a nutshell

Given some reference sequence, such as **CTGA...AGTTAGTGG...**

There is some query sequence, such as **AGTTCCCTG**

One possible alignment is:

```
CTGA...AGTTAG--TGG...
       ||||    ||
       AGTT*CCCTG
```

Note that gaps are allowed. The alignment depends on the **scoring criterion**. These can be rather sophisticated, but that's beyond scope.

There are some standard tools for align (e.g. `blast`), but we'll use an example program **dalex** written for this workshop **that you probably should not use in real research**. It has a simple scoring metric: the number of exact matches over the extent of the alignment. E.g. the above alignment scores 60%.

The Input Data

Reference genomes

- The salmonella bacterial genome is taken as is, about 4.6MB.
- Of the chicken genome, we've only taken one tenth of one chromosome, giving 19MB, which makes the exercises doable.

Simulated experimental sequences

- We have 12 batches (one per egg) of roughly 40 DNA fragments.
- Each sample is a 150 bases long only with T, G, C and A as well.

Goal: find out which of the eggs is contaminated with salmonella.

Bookkeeping

Bookkeeping and scripting

- Even on an HPC system, you will not 'just run' and get your results.
- There are queuing systems, you will need to split up your work, etc.
- Bookkeeping becomes important, for:
    1. Data management;
    2. Computation management;
    3. Postprocessing and documentation.
- To automate and track all of this, you'll like do some scripting.

Bookkeeping

When dealing with lots of data you'll need to keep track of:

- Where is everything stored?
- What needs or needed data transfers/copying?
- What format was used (conversions)?
- Directory structures, naming conventions.

Bookkeeping

When dealing with lots of analysis you'll need to keep track of:

- What 'jobs' in the batch system were done?
- Were they successful (no errors)?
- Where are the results, do they need to be transfered?
- What remains to be done?

Bookkeeping

Also don't forget:

- Post-processing.
- Take notes of what you are doing.
- If you have scripts, programs, etc, use version control.

Bookkeeping in the example

Even in our simple example, there are lots of things to keep track of, e.g.

- Where are the reference sequences stored, and how?
- Where are the query samples?
- How are they organized?

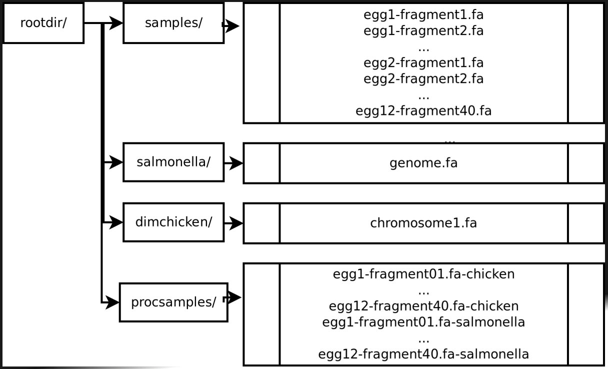
- What queries have been analyzed already?
- Which remain to be done?
- Where are the results?
- How are they organized?

- How far along is the postprocessing (can it start before all's finished)?

Bookkeeping

Simple data structure:

- Queries are in files **`eggA-fragmentB.dat`**, with **`A=1,2,...12`** and **`B=01,02,...40`** (or so) in subdirectory **`samples`**.
- Chromosomes in **`dimchicken/chromosome1.fa`** and **`salmonella/genome.fa`**
- Let's say all results are to go into the subdirectory **`procsamples`**.



Bookkeeping

Simple workflow:

- For each sample in each batch, align with each chromosome.
- Demand a scoring of at least 96%.
- Write result to a file, for post-processing.

Post-processing

- For each output file, count the number of matches.
- For each batch (e.g. egg), add up the numbers from its samples per chromosome.
- The ratio of salmonella matches over chicken matches is an indication of the contamination level.

Scripting

Script for the simple workflow

```
#!/bin/bash
# workflow1a.sh
chicken=dimchicken/chromosome1.fa
salmonella=salmonella/genome.fa
mkdir -p procsamples
for ((i=1;i<=12;i++))
do
   for samplefile in samples/egg$i-fragment*.fa
   do
      outfile=proc$samplefile-salmonella.out
      dalex -w 32 -m 0.96 $salmonella $samplefile > $outfile
      outfile=proc$samplefile-chicken.out
      dalex -w 32 -m 0.96 $chicken $samplefile > $outfile
   done
done
```

Warning: Would take 6 hours to run, won't finish in this workshop.

Scripting

Script for the simple workflow's postprocessing

```
#!/bin/bash
#workflow1b.sh
#!/bin/bash
cz=$(cat dimchicken/chromosome1.fa|wc -c)
sz=$(cat salmonella/genome.fa|wc -c)
for ((i=1;i<=12;i++))
do
   c=$(grep 'Score' procsamples/egg$i-*-chicken|wc -l)
   s=$(grep 'Score' procsamples/egg$i-*-salmonella|wc -l)
   contamination=$((s*cz*100/(c*sz)))
   echo egg $i:  contamination=$contamination
done
```

Section 3

Performance Tuning

Performance tip #1: Try Ramdisk

- The **dalex** program is performing quite a bit of I/O
- Sometimes you are faced with a program that just does that, and cannot be helped.
- In those case, Ramdisk may help

What is ramdisk?

- It is a part of memory that pretends to be a disk.
- With no moving parts (unlike a hard drive), it is very fast and less sensitive to IOPS
- It is private to the computer, unlike a shared file system
- It is gone when the node is reboot
  (or when a new job start, hopefully)
- In linux, usually lives under **/dev/shm** (type **df** to find out).
- The maximum size of the ramdisk is 11GB on most GPC nodes.

Hands-on: HDD vs RAMDISK

- Modify the script `workflow1a.sh` to do one egg, for only 9 samples, to test.
- Try running on GPFS and time!

- Copy data and executable over to ramdisk
- Try running both on HDD and RAMDISK and compare.

Concurrency

- Modern computers have more than one core.
- Modern supercomputers are modern computers linked together by a fast interconnect.
- Modern supercomputers run sophisticated schedulers that can run jobs simultaneously.

Concurrency

Figure out if the tool of your choice can handle shared memory, threaded parallelism, or distributed memory parallelism.
Each has its merits:

- Threaded:
  Pro: Shared memory means some things only need to be loaded once.
  Con: Cannot scale beyond 1 node (but can use all cores of that 1 node).
- Distributed parallelism:
  Con: does not use shared memory.
  Pro: But can (potentially) scale beyond one node.

What if it does not support either (such as **`dalex`**).
I.e. what if you are stuck with a bunch of serial jobs?

Easy case: serial jobs of equal duration

Suppose our node has 8 cores:

```
#!/bin/bash
cd ....
(cd jobdir1; ./dojob1) &
(cd jobdir2; ./dojob2) &
(cd jobdir3; ./dojob3) &
(cd jobdir4; ./dojob4) &
(cd jobdir4; ./dojob5) &
(cd jobdir4; ./dojob6) &
(cd jobdir4; ./dojob7) &
(cd jobdir4; ./dojob8) &
wait # crucial
```

Wait!

Make sure that 8 jobs actually fit in memory, or you will crash the node.
If only 4 fit in memory, and there is no way to reduce that, go ahead.
But there are also about 200 nodes with 32 GB memory.

Hard case #1: serial jobs of unequal duration

What you need is: Load Balancing

- Keep all cores on a node busy.
- GNU Parallel can help you with that!

GNU Parallel

GNU parallel is a really nice tool to run multiple serial jobs in parallel. It allows you to keep the processors on all cores on a node busy, if you provide enough jobs to do.

O. Tange (2011): GNU Parallel - The Command-Line Power Tool, ;login: The USENIX Magazine, February 2011:42-47.

Example

```
#!/bin/bash
cd ...
module load gnu-parallel # (needed on some systems)
echo 'cd jobdir1; ./dojob1' > joblist.txt
echo 'cd jobdir2; ./dojob2' >> joblist.txt
echo 'cd jobdir3; ./dojob3' >> joblist.txt
echo 'cd jobdir4; ./dojob4' >> joblist.txt
echo 'cd jobdir4; ./dojob5' >> joblist.txt
echo 'cd jobdir4; ./dojob6' >> joblist.txt
echo 'cd jobdir4; ./dojob7' >> joblist.txt
echo 'cd jobdir4; ./dojob8' >> joblist.txt
parallel -j 8 < joblist.txt
```

Hard case #2: serial jobs doing lots of I/O

What you need is: I/O Load Balancing

- Nodes have only one “pipe” into the shared file system.
- If all jobs on the node do I/O simultaneously, you have 1/8th the performance.
- This can be really tough.

Strategies

- There is a 1GB disk cache per node. If the data that your jobs use fit inside that, you’re probably fine.
- For this to work, must access ’contiguous’ files: large files, not lots of little ones.
- If jobs read at the start, compute, then write out. Staggering the jobs, so that they do not use I/O at the same time, can help.
  (Strategic ’sleep’ commands can do that).

Performance Tip #2: Get rid of unnecessary files

- As it turns out, dalex writes or reuses an index file for the reference genomes.
- This is an optimization that probably works well for larger genomes.
- But this index has to be read every time, and adds to the I/O overhead.
- How would you know? Always get to know your application!

```
dalex -h
```

- This tells us the option `-i 0` will turn off this option.

Hands-on: Remove intermediate I/O

- Start with your 1 egg, 9 fragments
- Switching off the index saving (`-i 0`) and time!
- We're still producing a lot of intermediate results. Could we not fold the post-processing into the computation and avoid all these files in the procsamples folder?

Performance Tip #3: Bunching your tasks

- In our example, we lauched a new instance of **`dalex`** for every fragment.
- That means reading the executable and reference chromosomes every time.
- This, too, turns out to be unnecessary, as **`dalex`** can take multiple fragments on the command-line.

Hands-on: Bunching

Change the script to use multiple fragments on the dalex command line instead of a loop.

Performance Tip #4: Reduce the number of files

- We still have all our fragments in separate files.
- Wouldn't it be easier if they were all combined (though separated by egg), so we would have just 12 files in the samples directory.
- Save not just at time of computation, but also when trasnfering data.
- This, too, turns out to possible, as **`dalex`** accepts **`.mfa`** files, which are concatenations of **`.fa`** files.

Hands-on: Reduce number of files

Concatenate the sample fragments of each egg together to **`.mfa`** files.
Change the script to have dalex read these files instead of a list of **`.fa`** files.

Section 4

File Formats

In this section, we will discuss the following topics:

- Handling data efficiently; staying organized.
- Storage formats.
- Handling metadata.
- NetCDF.

Data management

Data management

Data is now produced at an amazing rate.

- Increase in computing power makes simulations larger/more frequent.
- Increase in sensor technology makes experiments/observations larger.
  - Large Hadron Collider: $\sim$ 50-100 PB to date.
  - Square Kilometer Array: $\sim$ 1 EB /day !
- Data sizes that used to be measured in MB/GB are now measured in TB/PB.
- It's easier to make big data than to do something useful with it!
- Data access is the now the bottleneck in many situations.

Whether you are producing huge amounts of data or not, you must have a plan for how you are going to deal with your data.

Data considerations

What are the options for dealing with data?

- Deal with it right away:
  - Don't just save everything. It's wasteful to save junk that you'll never use. Storage space is finite.
  - Use on-the-fly analysis; automate your post-processing.
  - Is it worth storing or just recomputing?
- Store it for later analysis:
  - Store for short-term storage, on disk.
  - Store for long-term storage, on tape.

This class is focussed on the later option. How do you create/store/manage your data efficiently?

Terminology

Data creation

Let's start at the beginning. How do you create data efficiently?

- File I/O (Input/Output) is slow! Avoid it as much as you can!
- (CPU operations $\sim$ 1 ns, disk access times $\sim$ 5 ms.)
- Do not create lots of little files! They are an inefficient use of space and time (slow to create).
- Instead, save your data in big files which contain all the information you need.
- Do not have multiple processes writing to files in the same directory (unless you're using parallel I/O). A process will "lock" the directory after it's done writing the file and updating the file metadata. The other processes will have to sit and wait while this is being done.

File I/O is slow, especially with small files, so restrict your I/O to a few big files.

Data management

How should your files be organized on disk?

- Human-interpretable filenames lose their charm after a few dozen files (or after a few months pass). Don't use filenames to store run information.
- Avoid using a flat directory structure (no sub-directories). Organize your data in a sensible directory tree.
- If you're doing many runs with many varied parameters, consider using a database to store the filenames of your runs, with associated run metadata.
- Rigorously maintained meta-data (data about the data) is essential.
- Back up your data, especially your metadata or database.

Take-home message: keep your data well-organized.

ASCII versus binary

Storage formats: ASCII

The storage format we all start with is ASCII: American Standard Code for Information Interchange. It's our old friend, plain text.

- Pros:
  - Human Readable.
  - Portable (architecture independent).
- Cons:
  - Inefficient Storage.
  - Precision is lost for floats.
  - Slow to Read/Write (conversions).
  - (Embarrassing.)

Our old friend has done us well, but there are better storage options we can use.

Storage formats: binary

Native binary is the format in which the data is stored in memory:

- Pros:
  - Efficient Storage (256 x floats @4bytes takes 1024 bytes).
  - Efficient (fast) Read/Write (native, no conversion is needed).
- Cons:
  - Not human readable.
  - Have to know the format it's stored in, or else you'll have to reverse-engineer the file format to read it.
  - Not necessarily portable between systems (Endianness).

In terms of speed and file size, there's no contest.
Writing 128 million doubles to storage:

Format	/scratch (GPFS)	/dev/shm (RAM)	/tmp (disk)
ASCII	173s	174s	260s
Binary	6s	1s !!!	20s

ASCII versus binary

Syntax used:

	Text/ASCII	Binary

C	`f=fopen('test','w');`
	`fprintf(f,...);`
		`f=fopen('test','wb');`
		`fwrite(f,...);`

C++	`std::ofstream f('test');`
	
	`f << ...;`
		`std::ofstream f('test',`
		`   std::ios::binary);`
		`f.write(...);`

Fortran	`open(6,file='test',`
	`   form='formatted')`
	`write(6,*) ...`
		`open(6,file='test',`
		`   form='unformatted')`
		`write(6) ...`

Python	`f=open('test','w')`
	`print >>f, ...`
	`print(...,file=f)`
		`f=open('test','wb')`
		`f.write(bytes(...))`

Hands-on: binary index files with dalex

- Remember switching off those index files for dalex earlier on?
- Well, those were in ascii, but there's an option to do them in binary too.
- Let's switch them back on (`-i 1`), time things, and then, let's switch to binary (`-i 2`).

Metadata

But what about that metadata? What is it?

- Metadata is the data about the data. Meaning information that lets you make sense of the data.
- It can (and should) include just about any and all information about how the data was created:
    - what parameters were used in the run?
    - where it was run, when it was run.
    - the version of the code used to perform the run, compiler used to create the code, compiler flags.
    - and anything else that might or not be useful.
- If you're not sure if that bit information should be kept as metadata, then keep it. You never know what information might be needed in the future.

Standard formats

What's the best way to save our metadata? There are several standard file formats which *combine* the metadata with the data:

- HDF5 (Hierarchical Data Format)
- NetCDF (Network Common Data Form)
- discipline-specific formats

What are the benefits?

- Most are provided as libraries.
- Self-describing (metadata is embedded with the data).
- Many are binary agnostic, so portable.
- Many support Parallel I/O and native FS support.
- Broader tool support (visualization, etc.)

NetCDF

An introduction to netCDF

We are going to focus on netCDF, which is a commonly-used format. What is it?

- Stands for **net**work **C**ommon **D**ata **F**orm.
- Not compatible with NASA's CDF format.
- Used for array-oriented scientific data and metadata format.
- Stores in a binary form, so relatively efficient.
- Though it's in binary, it uses a common output format so different types of machines can share files.
- Self-describing, direct access, appendable.
- Many many wrappers to the API (C, C++, Fortran, Python, ...).

NetCDF classic data model

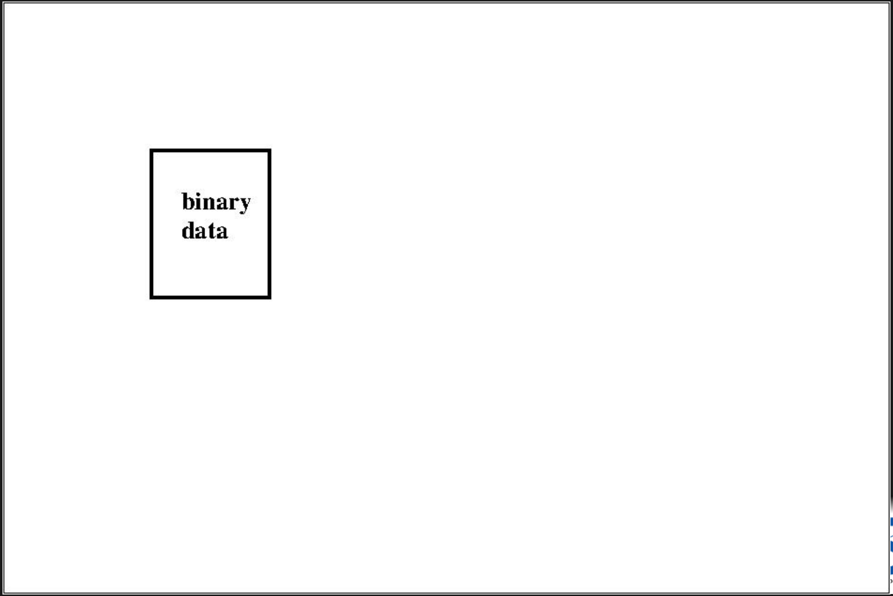
The original netCDF data model contains three entry types:

- Variables: N-dimensional arrays of data, of type char, byte, short, int, float, double.
- Dimensions: these describe the axes of the data arrays. A dimension has a name and a length.
- Attributes: Notes and supplementary information. These are scalar values or 1D arrays.
    - Attributes can be global, or apply to just a dimension or variable.
    - A good place to stick your miscellaneous metadata.
    - Units are a particularly good form of metadata.

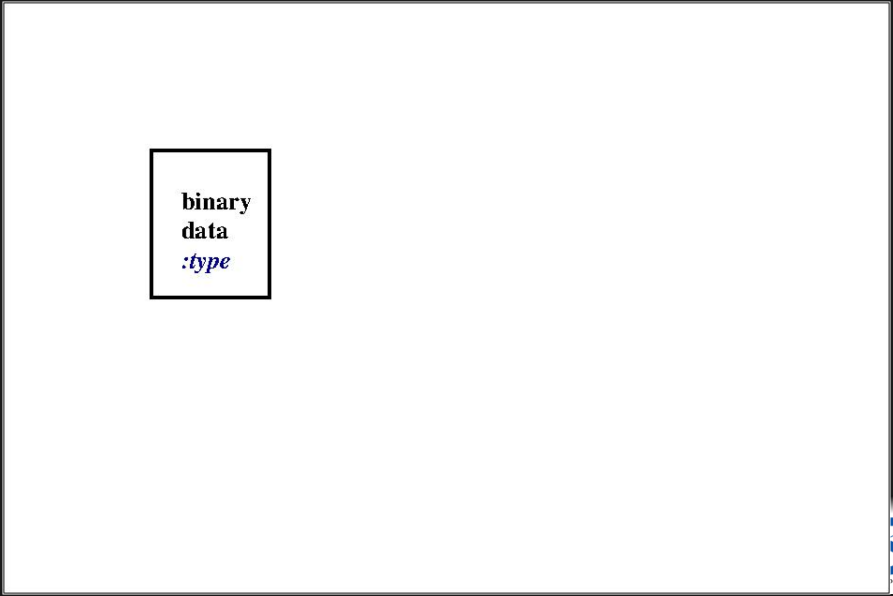
This is enough functionality to get us started. More-advanced features are also available.

NetCDF classic data model

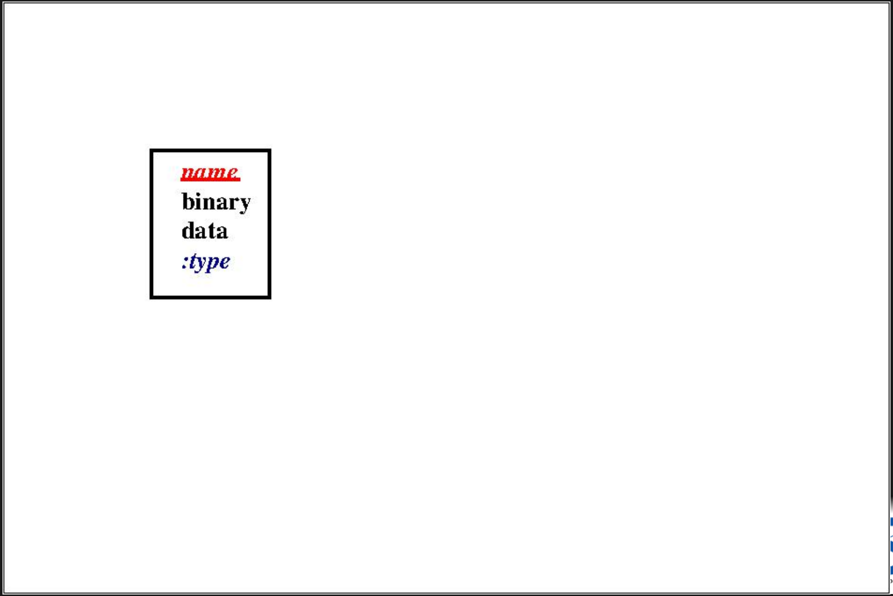
NetCDF classic data model



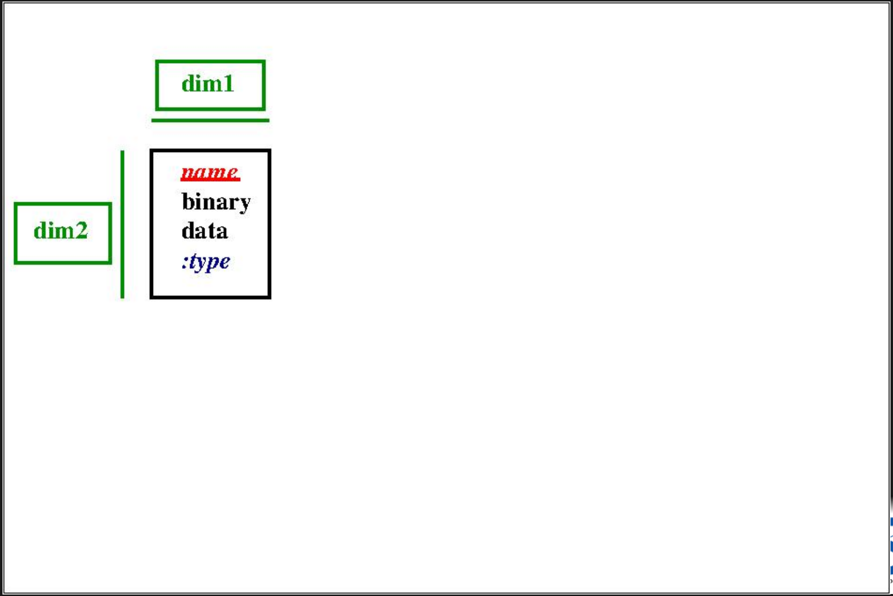
NetCDF classic data model



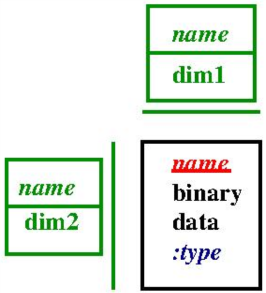
NetCDF classic data model



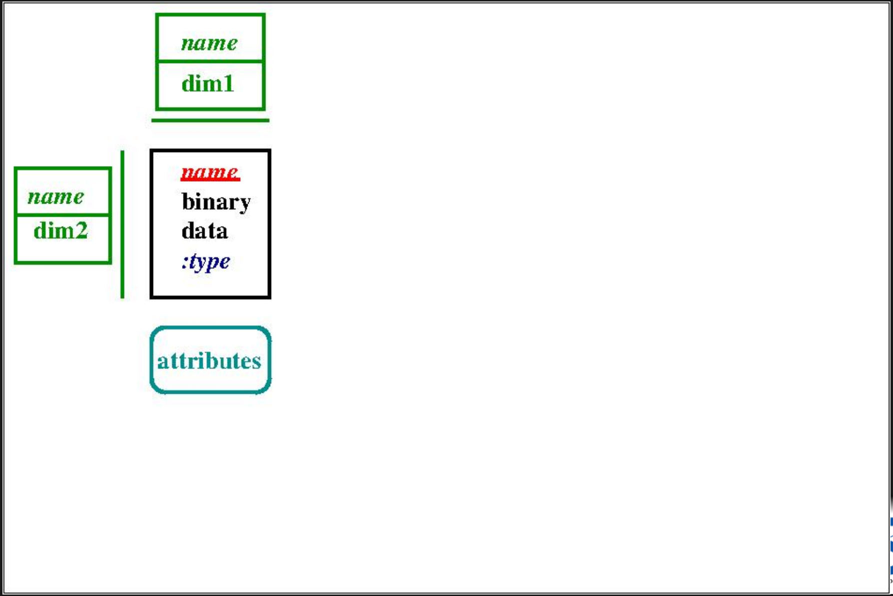
NetCDF classic data model



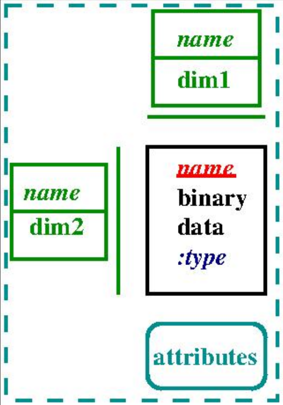
NetCDF classic data model



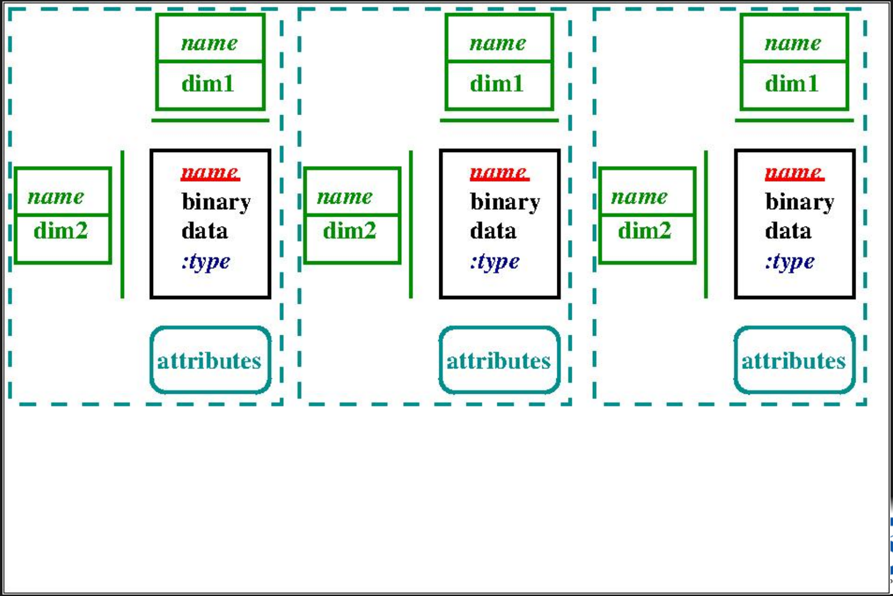
NetCDF classic data model



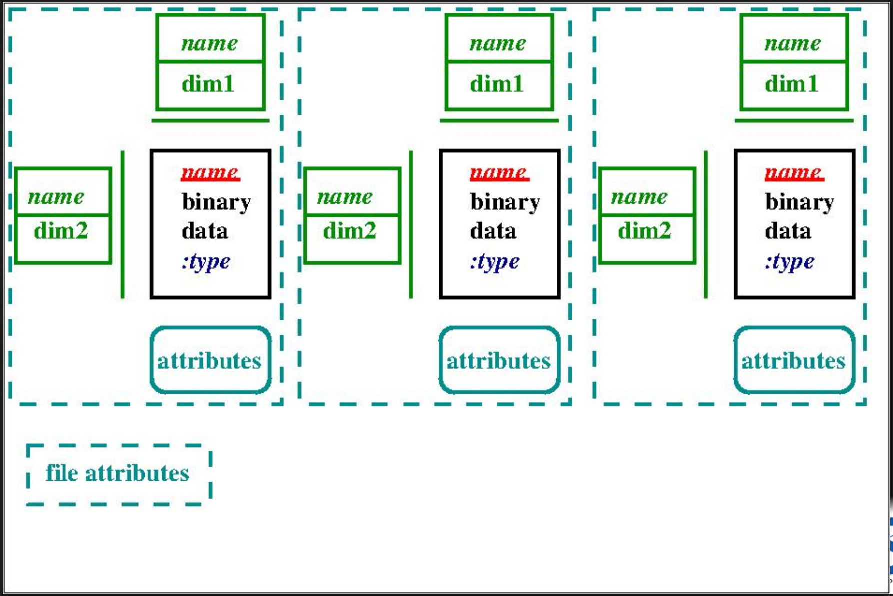
NetCDF classic data model



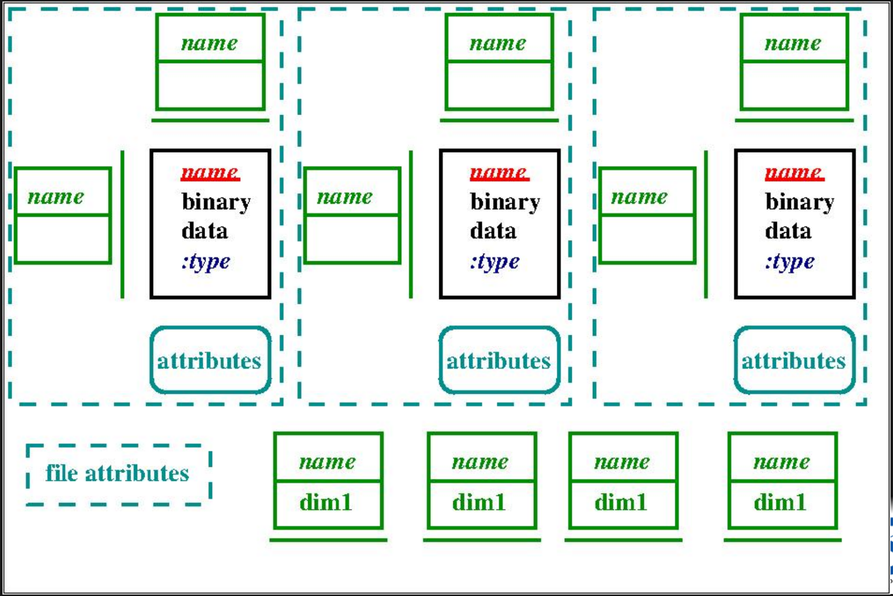
NetCDF classic data model



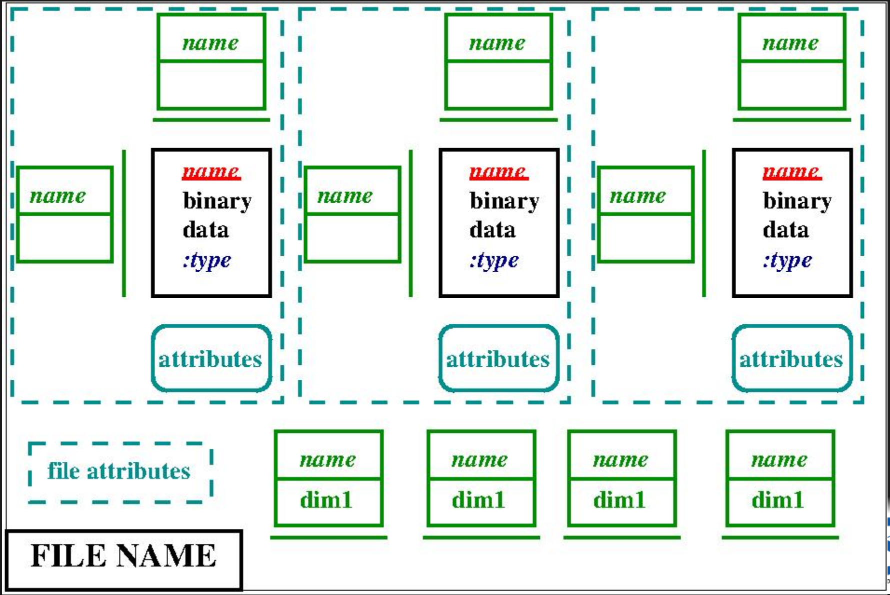
NetCDF classic data model



NetCDF classic data model



NetCDF classic data model



NetCDF-4

In 2008 netCDF-4 was released:

- This extended netCDF to use HDF5 as its data-storage layer, allowing the performance advantages of HDF5:
    - Compression.
    - Chunking.
    - Parallel I/O (many processes can write to the same file simultaneously).
- Completely back-compatible.
- Introduced user-defined types to netCDF.
- Introduced little and big endian support.

This is the version we'll be using in our examples.

NetCDF conventions

A quick note about netCDF conventions:

- There are lists of conventions that you can follow for variable names, unit names ("cm", "centimetre", "centimeter"), *etc.*
- If you are planning for interoperability with other codes, this is the way to go.
- Codes expecting data following, say, CF (Climate and Forcast) conventions for geophysics should use that convention.
- `www.unidata.ucar.edu/software/netcdf/conventions.html`

Make life easier for yourself and your collaborators: use the standard conventions.

NetCDF writing example

```cpp
// netCDF_writing.cpp
#include <vector>
#include <netcdf>
using namespace netCDF;

int main() {
   int nx = 6, ny = 12;
   int dataOut[nx][ny];
   for(int i = 0; i < nx; i++)
      for(int j = 0; j < ny; j++)
         dataOut[i][j] = i * ny + j;
   // Create the netCDF file.
   NcFile dataFile("1st.netCDF.nc",
      NcFile::replace);
   // Create the two dimensions.
   ncdim xDim = dataFile.addDim("x",nx);
   ncdim yDim = dataFile.addDim("y",ny);
   std::vector<ncdim> dims(2);
```

```cpp
   dims[0] = xDim;
   dims[1] = yDim;

   // Create the data variable.
   NcVar data =
      dataFile.addVar("data", ncInt,
      dims);

   // Put the data in the file.
   data.putVar(&dataOut);

   // Add an attribute.
   dataFile.putAtt("Creation date:",
      "12 Dec 2014");

   return 0;

}
```

NetCDF writing example, continued

```
$
$ module load gcc/4.8.1 hdf5/1811-v18-serial-gcc
$ module load netcdf/4.2.1.1_serial-gcc
$
$ g++ -I${SCINET_NETCDF_INC} netCDF_writing.cpp
   -o netCDF_writing -lnetcdf_c++4
$
$ ./netCDF_writing
$
```

NetCDF writing example, continued

```
$
$ ncdump 1st.netCDF.nc
netcdf 1st.netCDF {
dimensions:
   x = 6 ;
   y = 12 ;
variables:
   int data(x, y) ;
// global attributes:
   :Creation date  = "12 Dec 2014" ;
data:
data =
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71 ;
}
$
```

Use the "ncdump" command to see the contents of your netCDF file

NetCDF reading example

```cpp
// nc_reading2.cpp
#include <iostream>
#include <netcdf>
using namespace netCDF;

int main() {
   // Specify the netCDF file.
   NcFile dataFile("1st.netCDF.nc",
      NcFile::read);

   // Read the two dimensions.
   ncdim xDim = dataFile.getDim("x");
   ncdim yDim = dataFile.getDim("y");
   int nx = xDim.getSize();
   int ny = yDim.getSize();
   std::cout << "Our matrix is "
      << nx << " by " << ny <<
      std::endl;
```

```cpp
   int **p = new int *[nx];
   p[0] = new int[nx * ny];
   for(int i = 0; i < nx; i++)
      p[i] = &p[0][i * ny];

   // Create the data variable.
   NcVar data =
   dataFile.getVar("data");
   // Put the data in a var.
   data.getVar(p[0]);

   for(int i = 0; i < nx; i++) {
      for(int j = 0; j < ny; j++)
      {std::cout << p[i][j] << " ";
      }
      std::cout << std::endl;
   }
   return 0;
   }
```

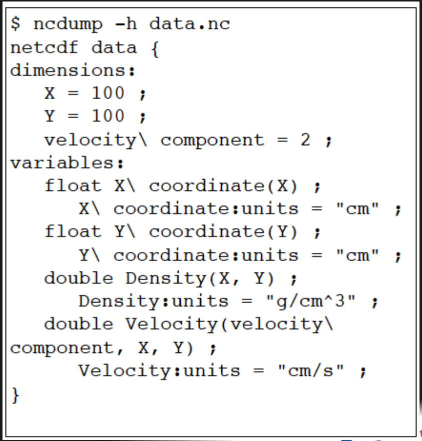
NetCDF reading example 2, continued

```
$
$ module load gcc/4.8.1 hdf5/1811-v18-serial-gcc
$ module load netcdf/4.2.1.1_serial-gcc
$
$ g++ -I${SCINET_NETCDF_INC} nc_reading2.cpp -o nc_reading2 -lnetcdf_c++4
$
$ ./nc_reading2
Our matrix is 6 by 12
0 1 2 3 4 5 6 7 8 9 10 11
12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31 32 33 34 35
36 37 38 39 40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69 70 71
$
$
```

More netCDF goodness

And there are more features:

- Not only can you read in only the variables that you're interested in, it is also possible to access subsections of an array, rather than reading in the entire thing.
- Allows parallel I/O.
- Allows "infinite" arrays, which means the arrays can grow. Good for timestepping, for example.
- Allows you to save custom datatypes (objects, for example).



```
$ ncdump -h data.nc
netcdf data {
dimensions:
   X = 100 ;
   Y = 100 ;
   velocity\ component = 2 ;
variables:
   float X\ coordinate(X) ;
      X\ coordinate:units = "cm" ;
   float Y\ coordinate(Y) ;
      Y\ coordinate:units = "cm" ;
   double Density(X, Y) ;
      Density:units = "g/cm^3" ;
   double Velocity(velocity\
component, X, Y) ;
      Velocity:units = "cm/s" ;
}
```

Writing NetCDF4 files in Python

```
from netCDF4 import Dataset
import numpy as np
root_grp = Dataset('test.nc', 'w', format='NETCDF4')
root_grp.description = 'Example␣temperature␣data'
# dimensions
root_grp.createDimension('time', None)
root_grp.createDimension('lat', 72)
root_grp.createDimension('lon', 144)
# variables
times = root_grp.createVariable('time', 'f8', ('time',))
latitudes = root_grp.createVariable('latitude', 'f4', ('lat',))
longitudes = root_grp.createVariable('longitude', 'f4', ('lon',))
temp = root_grp.createVariable('temp', 'f4', ('time', 'lat', 'lon',))
# data
lats =  np.arange(-90, 90, 2.5)
lons =  np.arange(-180, 180, 2.5)
latitudes[:] = lats
longitudes[:] = lons
for i in range(5):
    temp[i,:,:] = np.random.uniform(size=(len(lats), len(lons)))
# group
root_grp.close()
```

Reading NetCDF4 files in Python

```
from netCDF4 import Dataset
import pylab as pl

root_grp = Dataset('test.nc')

temp = root_grp.variables['temp']

for i in range(len(temp)):
    pl.clf()
    pl.contourf(temp[i])
    pl.show()
    raw_input('Press␣enter.')
```

NetCDF writing example, in R

Writing a NetCDF file can be a bit clunky, but it's worth the effort:

- **ncdim_def** defines a dimension. By default it also creates a "dimension variable".
- ncvar_def defines a variable. The dimensions are in a list.
- nc_create creates the file, variables must be specified.

```
library(ncdf4)

nx <- 10
ny <- 5

# our axes
x <- 1:nx
y <- 1:ny

# our data
a <- matrix(1:(nx * ny), nrow = nx,
     ncol = ny)

# dimensions
xdim = ncdim_def("x", "metres", as.
     double(x), longname = "")
ydim = ncdim_def("y", "metres", as.
     double(y), longname = "")

# create the variable
a.def <- ncvar_def("a", "unitless",
     list(xdim, ydim))
```

NetCDF writing example in R, continued

No seriously, this is a good idea!

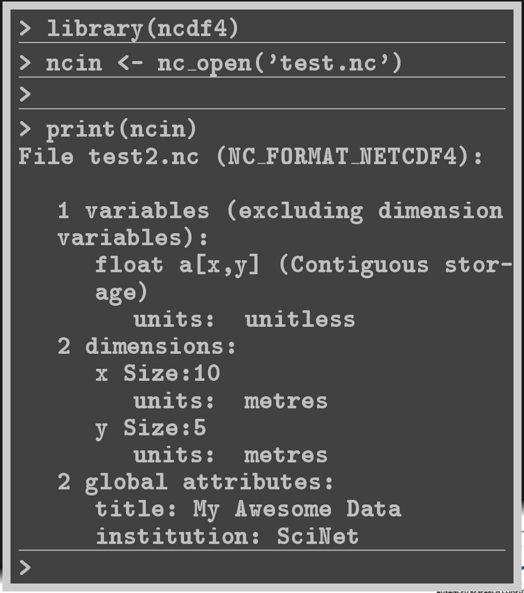
- ncvar_put puts the data ("a") into the variable ("a.def"), which is in the file.
- ncatt_put defines an attribute. This is where you add information about how this data was generated. The second argument begin '0' means it's a global attribute.
- nc_close closes the file.

```
# create the file
ncout <- nc_create("test.nc", a.def
    , force_v4 = T)

# put the variable in the file
ncvar_put(ncout, a.def, a)

# create some global attributes
ncatt_put(ncout, 0, "title", "My␣
    Awesome␣Data")
ncatt_put(ncout, 0, "institution",
    "SciNet")

nc_close(ncout)
```

NetCDF reading example

R can be used to read NetCDF files.

- nc_open opens your NetCDF file.
- The print function gives the output that is similar to 'ncdump'.



```
> library(ncdf4)
> ncin <- nc_open('test.nc')
>
> print(ncin)
File test2.nc (NC_FORMAT_NETCDF4):

   1 variables (excluding dimension
   variables):
      float a[x,y] (Contiguous stor-
      age)
         units:  unitless
   2 dimensions:
      x Size:10
         units:  metres
      y Size:5
         units:  metres
   2 global attributes:
      title: My Awesome Data
      institution: SciNet
>
```

NetCDF reading example, continued

You can also reach into a NetCDF files and just grab the parts that you want.

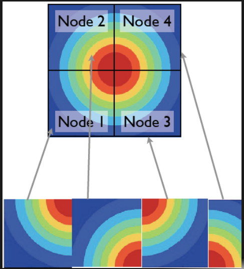
- ncvar_get grabs a variable from the open NetCDF file.
- ncatt_get grabs an attribute. The second argument set to 0 indicates that the attribute is global. The function returns a named list.

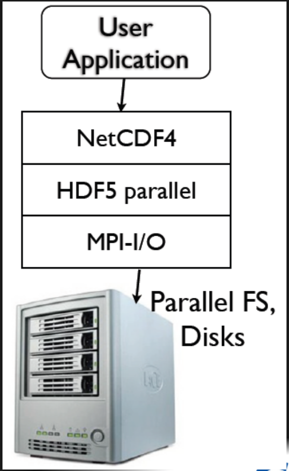
```
>
> x <- ncvar_get(ncin, "x")
> print(x)
[1] 1 2 3 4 5 6 7 8 9 10
>
> a <- ncvar_get(ncin, "a")
>
> dim(a)
[1] 10 5
>
> ncatt_get(ncin, 0, "institu-
tion")
$hassatt
[1] TRUE

$value
[1] "SciNet"
>
```

Data Managament and Parallel I/O

Data files must take advantage of parallel I/O

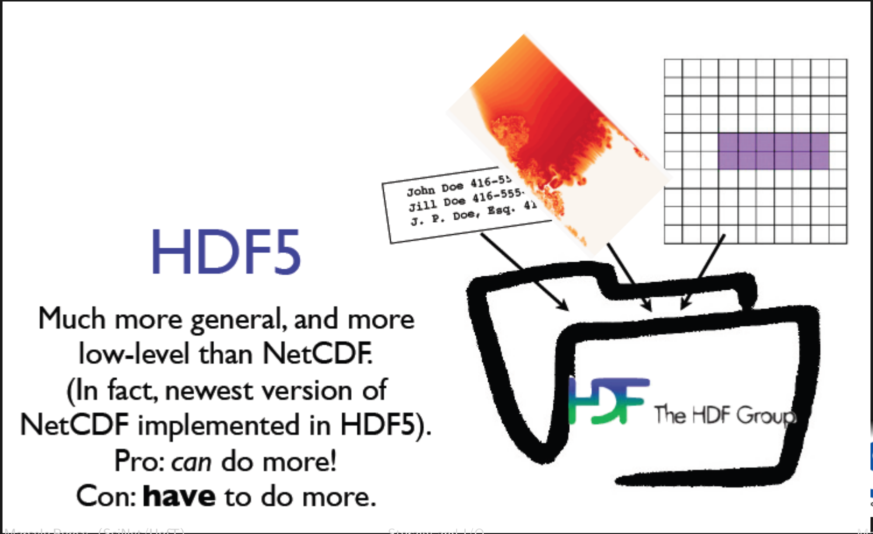
- For parallel operations on single big files, parallel filesystem isn't enough
- Data must be written in such a way that nodes can efficiently access relevant subregions
- HDF5, NetCDF formats typical examples for scientific data





HDF5

- HDF5 is also self-describing file format and set of libraries
- Unlike NetCDF, much more general; can shove almost any type of data in there



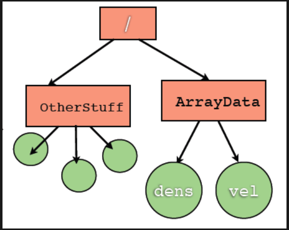
HDF5

Much more general, and more low-level than NetCDF.
(In fact, newest version of NetCDF implemented in HDF5).
Pro: *can* do more!
Con: **have** to do more.

HDF5 Groups

HDF5 has a structure a bit like a linux filesystem:

- "Groups" - directories,
- "Dataset" - files



- NetCDF, HDF are not Databases
- Seem like - lots of information, in key value pairs.
- Relational databases - interrelated tables of small pieces of data
- Very easy/fast to query
- But cant do subarrays, etc..

Summary

Things to remember from this section:

- Use file I/O as little as possible. Keep it to big files, with as few IOPs as possible.
- Use a binary format to store you data, not ASCII.
- It's a good practise to make your data "self-describing", meaning store your metadata with your data in the same file.
- NetCDF is a commonly used format to store data that has many useful features.

Section 5

Data Management

Planning, storing, etc..

- Design/think about the data structure of your problem (simulation, analisys, etc...)
- Optimize your workflow or pipeline
- Consider the *big picture*: "large (big) data" & "long time"
- Keep good records and logs of your data (sims: initial conditions, version of your code, parameters – data analysis: methods, raw data, etc...)
- Version Control: svn, cvs, git, mercurial; bitbucket, github, ...

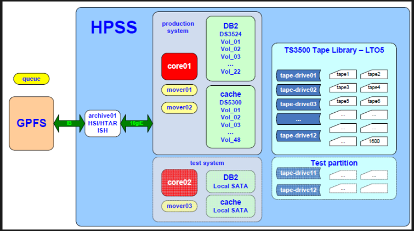
Data Management: monitoring

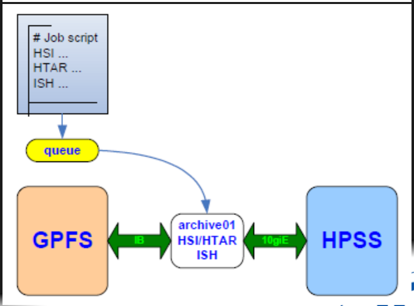
Most HPC systems, have *quotas* in storage resources (SciNet: 1M files, 20TB)

- Minimize use of filesystem commands like `ls -l` and `du`.
- Regularly check your disk usage using /scinet/gpc/bin6/diskUsage.
- Warning signs which should prompt careful consideration:
  - More than 100,000 files in your space
  - Average file size less than 100 MB
- Remember to distinguish: Analysis, Required and By-Product data.

HPSS, tapes & archival resources

- most HPC systems, include a High Performance Storage System
- tape-backed hierarchical storage system that provides a significant storage resource





```
# Job script
HSI ...
HTAR ...
ISH ...
```

How to make the file system work for you rather than against you

Make a Plan!

- Make a plan for your data needs:
  - How much will you generate,
  - How much do you need to save,
  - And where will you keep it?
- Note that in most HPC systems "scratch" is temporary storage for 3 months or less.
- Options?
  1. Save on your departmental/local server/workstation (it is possible to transfer TBs per day on a gigabit link);
  2. Apply for a project space allocation at next RAC call
  3. Change storage format.

Change storage format

- Write binary format files
  Faster I/O and less space than ASCII files.
- Use parallel I/O if writing from many nodes
- Maximize size of files. Large block I/O optimal!
- Minimize number of files. Makes filesystem more responsive!

Don'ts:

- Don't write lots of ASCII files. Lazy, slow, and wastes space!
- Don't write many hundreds of files in a 1 directory.
  Hurts responsiveness!
- Don't write many small files ($<$ 10MB).
  System is optimized for large-block I/O!

Hands-on: compression

- We've focused on processing speed mostly for now, but the way the data is stored in our usecase is itself quite inefficient.
- Let's try to tar and compress all the samples using tar and gzip.
- Compare the difference in size.

Notes:

If you were to run out of ramdisk space, sometimes you could use compression. Other times, the time of compression is not worth it.

For longer term storage, compression and packing are more useful.

General guidelines in restructuring data

1. Identify your unit of computation

- If files bundle naturally (or even mildly forced), put them in single (tar) files if and when you can.

2. Distinguish types of data

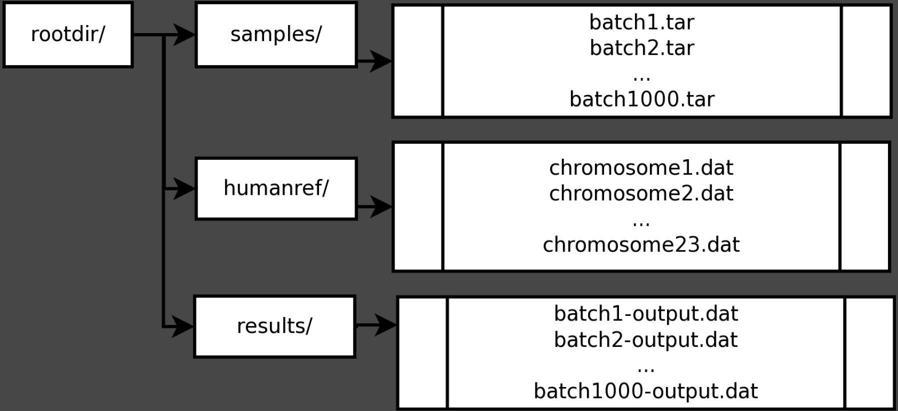
- Analysis: i.e., that which is strictly necessary for later analysis.
- Required: e.g. for restarts, but you might not need this.
- By-product: All the stuff you don't need

3. Take action

- Remove By-product data as soon as possible.
- Bundle data by 'unit of computation'.
- Separately bundle the Analysis and Required data.
- Only keep the Analysis data on hand, store the rest (tarball, HPSS).

tar-ing example

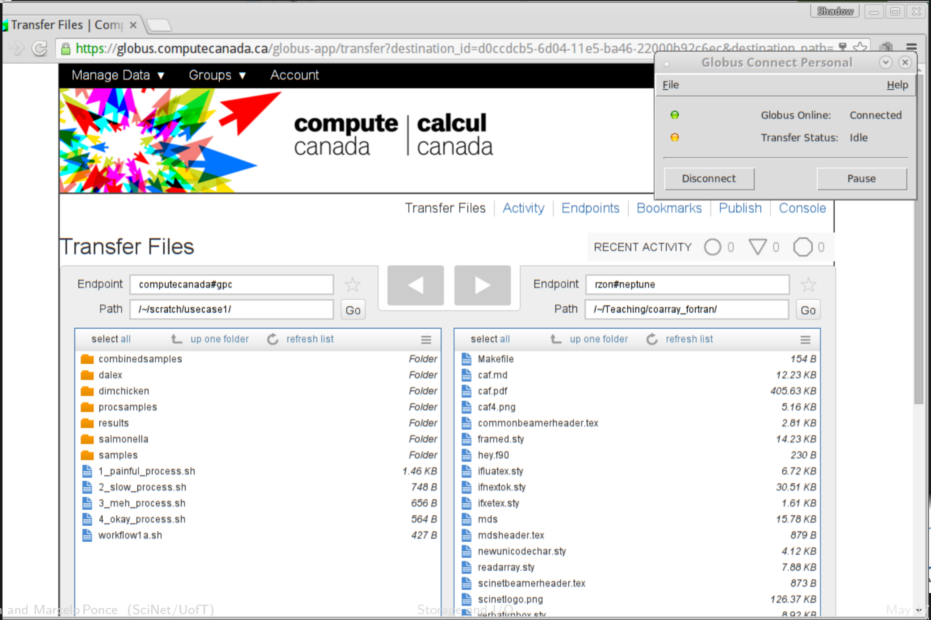
```
$ cd samples
$ for a in $(seq 12); do
$   tar cf batch$a.tar egg$a-*.dat
$ done
```



Moving data between different machines

- To move your data between machines, **`scp`** will do.
- Transfer will be faster if you have compressed and tarred files.
- Still, scp is not always the fastest (single stream), may time out, etc.
- An easier and more robust way is to use Globus.
- See https://www.computecanada.ca/research-portal/globus-portal
  https://globus.computecanada.ca/SignIn

Moving data between different machines: Globus



Transfer Files | Comp
https://globus.computecanada.ca/globus-app/transfer?destination_id=d0ccdcb5-6d04-11e5-ba46-22000b92c6ec&destination_path=
Manage Data
Groups
Account
compute canada | calcul canada
Transfer Files
Activity
Endpoints
Bookmarks
Publish
Console
Globus Connect Personal
File
Help
Disconnect
Pause

Globus Online: Connected
Transfer Status: Idle

Transfer Files

RECENT ACTIVITY 0 0 0

Endpoint: computecanada#gpc
Path: /~/scratch/usecase1/

combinedsamples	Folder
dalex	Folder
dimchicken	Folder
procsamples	Folder
results	Folder
salmonella	Folder
samples	Folder
1_painful_process.sh	1.46 KB
2_slow_process.sh	748 B
3_meh_process.sh	656 B
4_okay_process.sh	564 B
workflow1a.sh	427 B

Endpoint: rzon#neptune
Path: /~/Teaching/coarray_fortran/

Makefile	154 B
caf.md	12.23 KB
caf.pdf	405.63 KB
caf4.png	5.16 KB
commonbeamerheader.tex	2.81 KB
framed.sty	14.23 KB
hey.f90	230 B
ifluatex.sty	6.72 KB
ifnextok.sty	30.51 KB
ifxetex.sty	1.61 KB
mds	15.78 KB
mdsheader.tex	879 B
newunicodechar.sty	4.12 KB
readarray.sty	7.88 KB
scinetbeamerheader.tex	873 B
scinetlogo.png	126.37 KB
verbatimbox.sty	8.92 KB

Section 6

Conclusions

Conclusions

We hope to have conveyed that

- Computing at scale requires careful thought.
- New bottlenecks can arise as one scales up.
- Monitoring and testing is important.

- I/O often the bottleneck.
- Restructuring data can help a lot.
- Using ramdisk can help.
- Many files are bad.