

Workflow Optimization for Large Scale Bioinformatics

Ramses van Zon

SciNet/Compute Canada

September 25, 2012

Outline of the course

- 1 Introduction
- 2 Bookkeeping and scripting
- 3 File I/O
- 4 Running and monitoring
- 5 Concurrency

Part I

Introduction

... is a broad area of research

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

Common features

- Involves a lot of data
- Involves a lot of analysis

⇒ High Performance Computing (HPC)

Tends to hit computational limitations

It is not atypical for bioinformatics applications to require

- Large memory
- Lots of disk space
- High bandwidth to move data
- Fast access to small amounts of that data (“iops”)
- Substantial computation time

often all at the same time!

Suitable for HPC?

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.

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

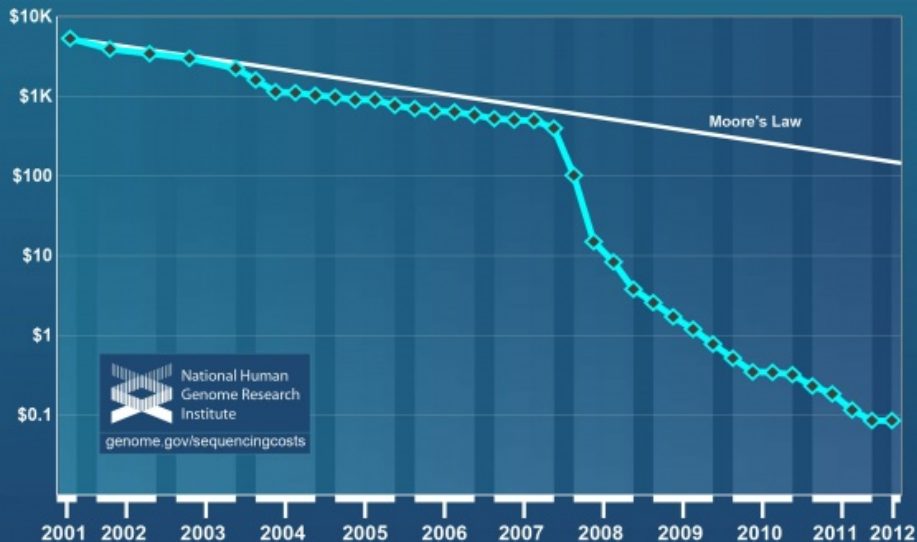
What goes where, how fast, how big is it, what is the format?

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 are important for any large scale scientific computation, but the pace at which data gets produced in bioinformatics is impressive.

Cost per Raw Megabase of DNA Sequence



- File systems
- Modern computers (CPU, RAM, DISK)
- Supercomputers
- Networks
- Linux
- Schedulers

Don't worry...

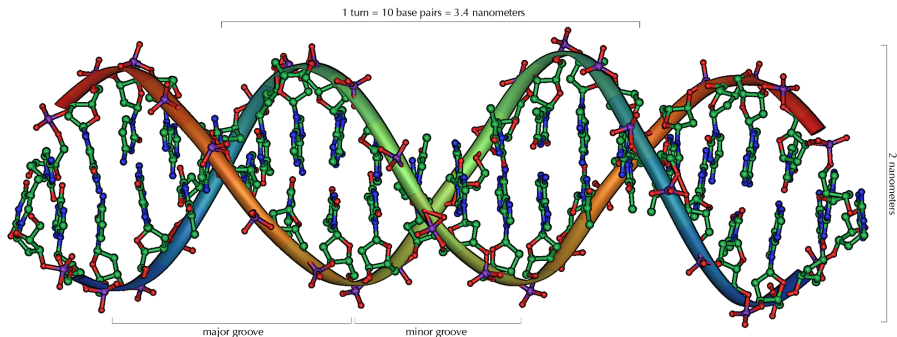
We do not need to understand these in every detail, but we do need to familiarize ourselves with each so we understand the limitations, and can adapt our workflow to maximize throughput.

A running example

A running example

Project

Given a number of batches of DNA fragments, we want to know which batches are most likely human, by comparing (“alignment”) against a reference human 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 this course.

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

Fake Numbers

Fake reference genome

- Our fake human genome is exactly given (T, G, C and A only).
- It is composed of 23 chromosomes of 200M bases each.
- Total bases is thus 4.6G (close to real amount).

Fake experimental sequences

- We have 1000 batches of 500 samples each.
- Each sample is a 100 bases long only with T, G, C and A as well.

Part II

Bookkeeping and scripting

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.

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 transferred?
- What remains to be done?

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.

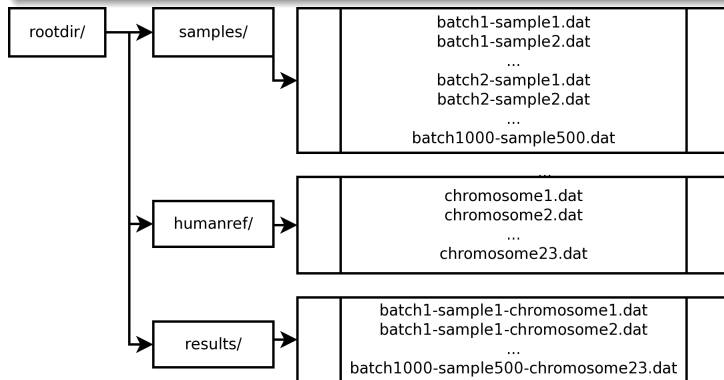
- Where is the reference sequences stored, and how?
 - Where are the query samples?
 - How are they organized?
-
- What queries have been analyzed already?
 - Which remain to be done?
 - Where are the results?
 - How are they organized?
-
- How far along is the postprocessing (can it start before all's finished)?

Simple data structure

- Queries are in file **batchA-sampleB.dat**, with **A=1,2,...1000** and **B=1,2,...500** in subdirectory **samples**.
- Chromosomes in **humanref/chromosomeC.dat**, **C=1,2,...23**.
- Let's say all results are to go into the subdirectory **results**.

Simple data structure

- Queries are in file **batchA-sampleB.dat**, with **A=1,2,...1000** and **B=1,2,...500** in subdirectory **samples**.
- Chromosomes in **humanref/chromosomeC.dat**, **C=1,2,...23**.
- Let's say all results are to go into the subdirectory **results**.



Simple workflow

- For each sample in each batch, align with each chromosome.
- Demand a scoring of at least 95%.
- Write result to a file, for post-processing.

Post-processing

- For each output file, count the number of matches.
- For each batch, add up the numbers from its samples for all chromosomes.
- Order these final numbers: highest numbers are likely human.

Scripting

Scripting is ...

Automating tasks...

- Reproducible
- 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, 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	Prints output
pwd	Print current directory
cd [directory]	Change directory
ls	Directory LiSting
cat [filename]	Dumps out filename(s)
less [filename]	Prints out filename(s) by page
mv [src] [dest]	Move file
cp [src] [dest]	Copy file
rm [filename]	Delete file
mkdir [filename]	Create directory
rmdir [filename]	Remove directory

A bit of bash: Some commands that you may not know

seq [limit]	Print the numbers 1 to limit
wc [filename]	Line/word/char count of file
head [filename]	First lines of file
tail [filename]	Last lines of file
sort [filename]	Sort lines of file
grep	Searches input for text
awk	Column-based processing language
sed	Line-based filter and text transformer
tar	Archive multiple files to one file
gzip	Compress 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.

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
- Avoids creating a temporary file

A bit of bash: variables

- Declare and initialize a variable:
varname="string"
- To use the variable, you need a dollar sign:
echo \$varname
- Passing this variable to other scripts called within the current one:
export varname

Special variables

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

Storing output in a variables

- **varname=\$(cmdline)**
stores the output of the command **cmdline** in the variable **varname**

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]`
- Block of commands in the loop should be between `do` and `done`.
- Example:

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

```
hello
world
how
are
you
```

Script for the simple workflow

```
#!/bin/bash
for a in $(seq 1000)
do
  for b in $(seq 500)
  do
    for c in $(seq 23)
    do
      dalex humanref/chromosome$c.dat \
        samples/batch$a-sample$b.dat -w 32 -m 0.95 \
        > results/batch$a-sample$b-chromosome$c.dat
    done
  done
done
```

If the inner command takes, say, 1 minute on 1 cpu, it would take 8000 days for this computation to complete.

Script for the simple workflow's postprocessing

```
#!/bin/bash
for a in $(seq 1000)
do
  for b in $(seq 500)
  do
    for c in $(seq 23)
    do
      cat results/batch$a-sample$b-chromosome$c.dat \
        | grep '^#' | grep -v '^#0'
    done
  done | wc -l | awk "{print $1,$a}"
done | sort -n -r > results/batchtotals.txt
```

Thoughts on how this would go? Do you see any problems already?

Part III

File I/O

File systems

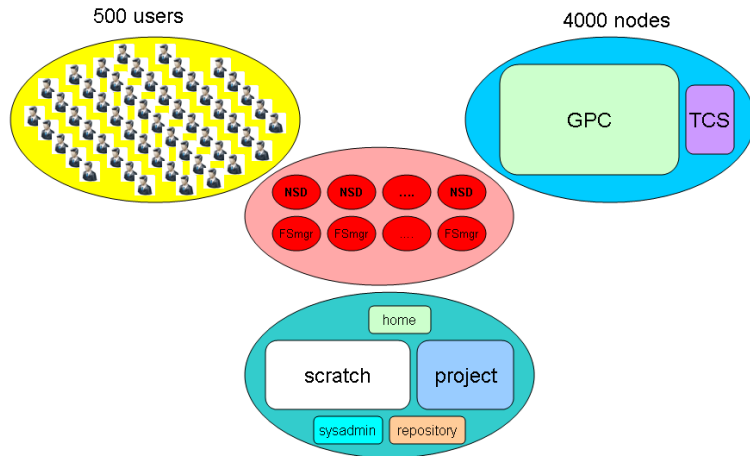
- It's where we keep most data.
- Typically spinning disks
- Logical structure: directories, subdirectories and files.
- On disk, these are just blocks of bytes.

What are I/O operations, or IOPS?

- Finding a file (ls)
Check if that file exists, read metadata (file size, data 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 where on disk the data is.
- Close the file.

Parallel file system at a glance

At SciNet



File system at SciNet



- 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 on TCS and GPC
- I/O goes over the network
- File system is *parallel!*

File system recap

location	quota	block-size	time-limit	backup	devel	comp
/home	10GB	256kB	unlimited	yes	rw	ro
/scratch	20TB	4MB	3 months	no	rw	rw

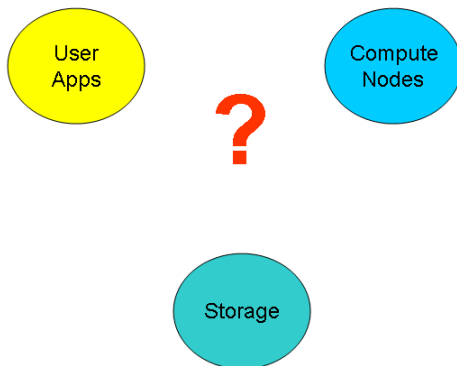
- There are quotas
- Home read-only from compute nodes!
- Big block sizes: *small files waste space*
- Issues are common to parallel file systems (Lustre, etc.) present in most modern supercomputers.
- Scratch quota per user oversubscribes disk space, so only for when you *temporarily* really needs a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

Scratch Policies

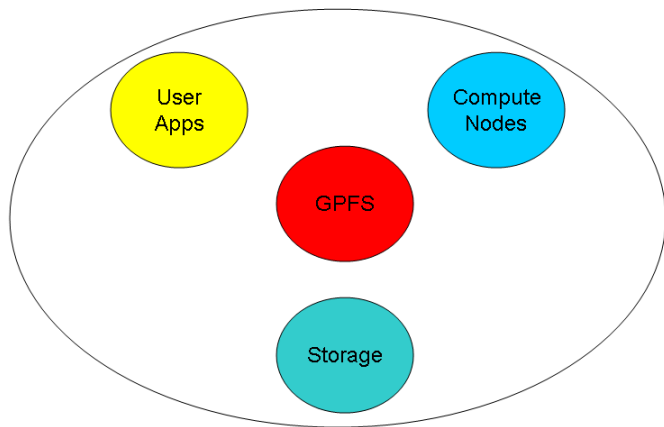
- Scratch is intended for active jobs (e.g. writing checkpoints and data during a run).
- Files are purged after 3 months

The file system is parallel, what does that mean?

Basic Components



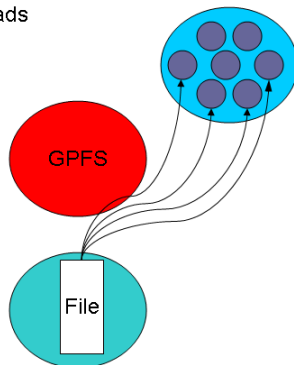
Basic Components



General Parallel File System

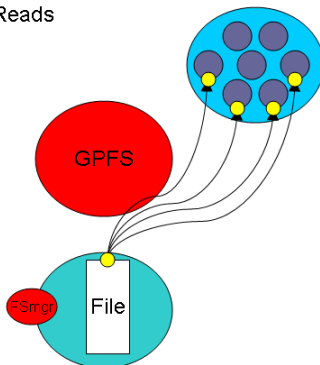
Basic Components

Parallel Reads



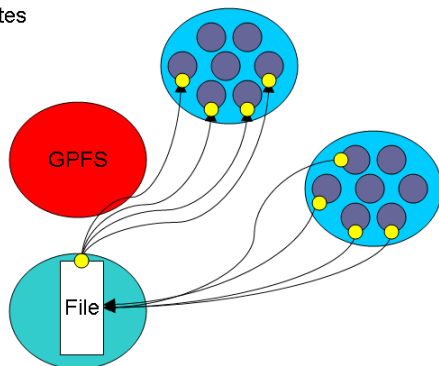
Basic Components

Parallel Reads



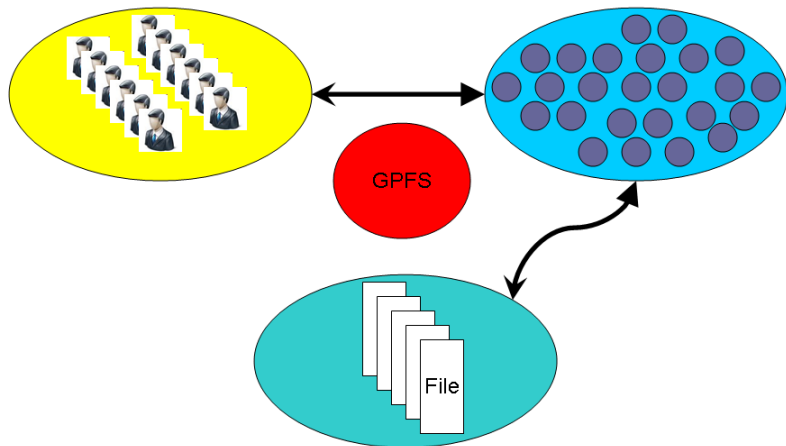
Basic Components

Parallel Writes

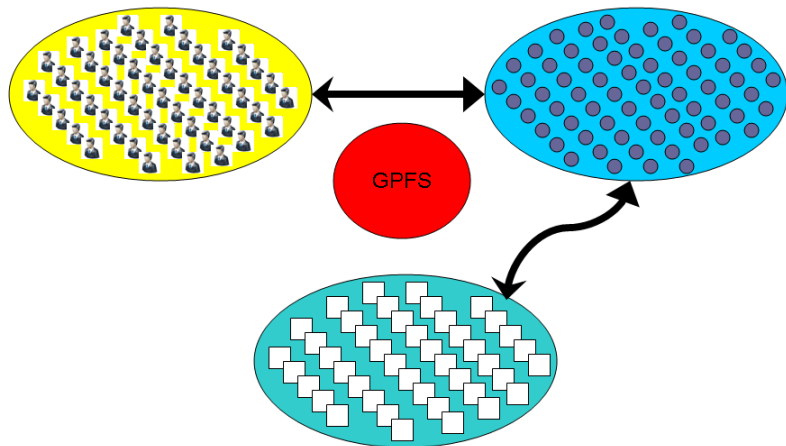


Shared file system

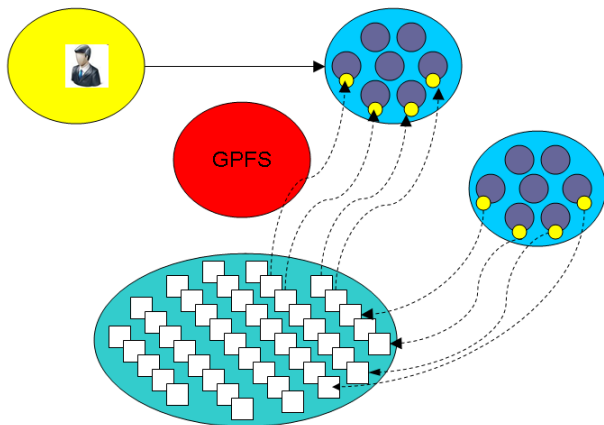
Basic Components (scaled)



How can we push the limit?



How can we BREAK the limit?

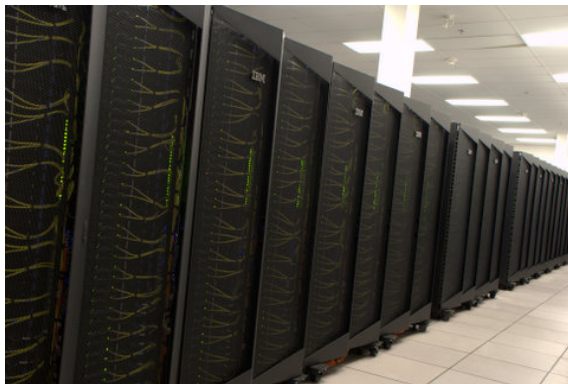


Shared file system

- 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 which are not shared.)
- 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!
- The file system is shared over the ethernet network on GPC:
Hammering the file system can hurt process communications.
- 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*.
- SciNet: ~100 users doing I/O from 4000 nodes.
That's a lot of sharing and contention!



Some Numbers

- 830 TB on scratch
- Over 1000 users - you do the math!
- Want $>25\%$ free at any given time
(systems can write 0.5 PB per day!)
- 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!

How to make the file system work for 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 /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 ($< 10\text{MB}$).
System is optimized for large-block I/O!

Example

Obviously, we need to change our data layout

- $500 \times 1000 = 500,000$ input files, nominally about 6GB, would take:
125 GB on /home: **out of quota**
2 TB on /scratch: **300x larger: insane.**
- Temporary files: $23x \Rightarrow$ **17.5 million files. Out of quota and insane.**

Example

Obviously, we need to change our data layout

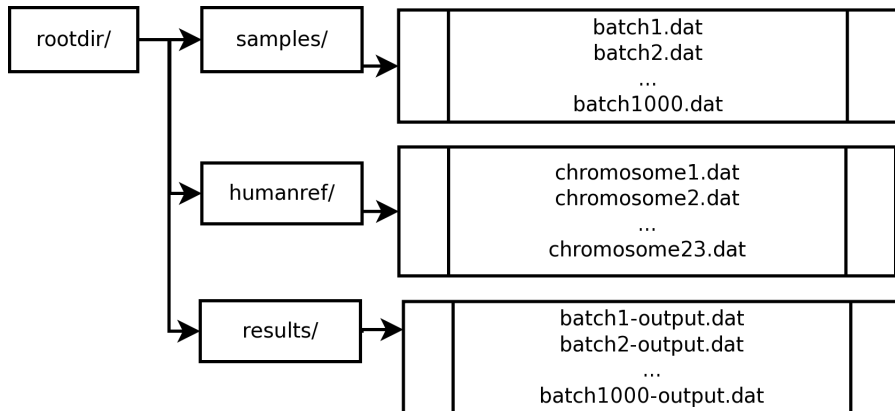
- $500 \times 1000 = 500,000$ input files, nominally about 6GB, would take:
125 GB on /home: **out of quota**
2 TB on /scratch: **300x larger: insane.**
- Temporary files: $23x \Rightarrow$ **17.5 million files. Out of quota and insane.**

What to do?

- **Do we need all these temporary files?**
Certainly not, we don't care which chromosome has the match.
- **Do we need separate files for all the sequences?**
Only need to distinguish by batch: only really need 500 output files.
- **How about the large number of input files?**
Depends on the tool. It turns out the **dalex** can take multiple query sequences from a single file. So we might use one input file per batch.

New data layout for example

```
$ cd samples
$ for a in $(seq 1000); do
$   cat batch$a-*.dat > batch$a.dat
$ done
```



New scripts

```
#!/bin/bash
for a in $(seq 1000)
do
    for c in $(seq 23)
    do
        dalex humanref/chromosome$c.dat samples/batch$a.dat \
            -w 32 -m 0.95
    done > results/batch$a-output.dat
done
```

```
#!/bin/bash
for a in $(seq 1000)
do
    cat results/batch$a-out.dat | grep '^#' | grep -v '^#0' \
    | wc -l | awk "{print $1,$a}"
done | sort -n -r > results/batchtotals.txt
```

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).

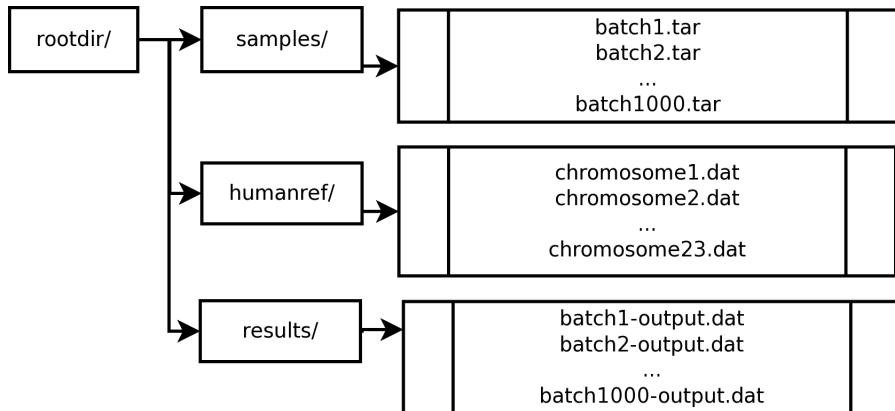
Ramdisk

- Sometimes, the tools you're using is not yours to rewrite, and does bad I/O.
- RAM is always faster than disk; think about using ramdisk.
- If the total size of the files involved in this bad behaviour is not too large, you can put them on ramdisk.
- Ramdisk lives on `/dev/shm` as if it is a regular directory. However, anything put there will actually be stored in memory.
- That memory is then no longer available for you application: plan ahead!
- The maximum size of the ramdisk is 11GB on most GPC nodes.
- Important: results stored to ramdisk are not shared among the cluster!

Suppose dalex did not allow multiple sequences in a file, what then?

Ramdisk example

```
$ cd samples
$ for a in $(seq 1000); do
$   tar cf batch$a.tar batch$a-*.dat
$ done
```



New scripts

```
#!/bin/bash
for a in $(seq 1000)
do
    tar xf samples/batch$a.tar -C /dev/shm/
    for b in $(seq 500)
    do
        for c in $(seq 23)
        do
            dalex humanref/chromosome$c.dat \
                /dev/shm/batch$a-sample$b.dat -w 32 -m 0.95
        done
    done > results/batch$a-output.dat
    rm -rf /dev/shm/batch$a-sample*.dat
done
```

Post-processing remains the same.

Part IV

Running and monitoring

Running and monitoring

- You'll be running on a cluster (GPC)
- Need some understanding of:
 - Clusters
 - Schedulers
- These run fairly “hands-off”, so knowing how to monitor your jobs is important.

Running

Submitting jobs

- To run a job, you must submit to a batch queue.
- You submit jobs from a devel node in the form of a script
- Scheduling is by node. **You need to use all 8 cores on the node!**
- Best to run from the scratch directory (**home=read-only**)
- Copy essential results out after runs have finished.

Submitting jobs - Limits

- Group based limits:
possible for your colleagues to exhaust group limits
- Talk to us first to run massively parallel jobs (> 2048 cores)
- While their resources last, jobs will run at a higher priority than others for groups with an allocation.

GPC queues

queue	min.time	max.time	max jobs	max cores
batch	15m	48h	32/1000	256/8000
debug		2h/30m	1	16/64
largemem	15m	48h	1	16

- Submit to these queues from a GPC devel node with
qsub [options] <script>

- Common options (usually in script):

-l: specifies requested nodes and time, e.g.

-l nodes=2:ppn=8,walltime=1:00:00

-q: specifies the queue, e.g.

-q batch

-q debug

-q largemem

-I specifies that you want an interactive session.

-X specifies that you want X forwarded.

- The largemem queue is exceptional, in that it provides access to two nodes (only) that have 16 processors and 128GB of ram.

GPC job script example

```
#!/bin/bash
#PBS -l nodes=1:ppn=8
#PBS -l walltime=2:00:00
#PBS -N JobName
cd $PBS_O_WORKDIR
dalex -m .95 -w 32 humanref/chromosome1.dat samples/batch1.dat
```

```
$ cd $SCRATCH/...
$ qsub myjob.pbs
2961983.gpc-sched
$ qstat (or checkjob 2961983, or showq -u $USER)
```

Job id	Name	User	Time	Use	S	Queue
13706895.gpc-sched	JobName	rzon		0	Q	batch

```
$ ls
JobName.e13706895 JobName.o13706895 humanref myjob.pbs
results          samples
```


Where's the output? In JobName.o13706895

Unredirected output goes, after the run, to:

```
-----  
Begin PBS Prologue Mon Sep 24 22:16:33 EDT 2012 1348539393  
Job ID: 13706895.gpc-sched  
Username: rzon  
Group: scinet  
Nodes: gpc-f109n001-ib0  
End PBS Prologue Mon Sep 24 22:16:34 EDT 2012 1348539394  
-----  
INFO: read reference from : humanref/chromosome1.dat  
INFO: read in queries from : samples/batch1.dat  
...  
-----  
Begin PBS Epilogue Mon Sep 24 22:17:30 EDT 2012 1348539450  
...  
Limits: neednodes=1:ppn=8,nodes=1:ppn=8,walltime=02:00:00  
Resources: cput=00:00:50,mem=3279680kb,vmem=3386652kb,wallt...
```

Full problem

Just add a for loop over chromosomes:

```
#!/bin/bash
#PBS -l nodes=1:ppn=8
#PBS -l walltime=2:00:00
#PBS -N JobName
cd $PBS_O_WORKDIR
for c in $(seq 23); do
    dalex -m .95 -w 32 humanref/chromosome$c.dat samples/batch1.dat
done
```

and submit 1000 times, right?

We would probably yell at you for that. Why?

Monitoring

SciNet systems are batch compute clusters

- Computing by submitting **batch jobs** to the **scheduler**.
- When you submit a job, it gets placed in a **queue**.
- Job priority is based on **allocation** and **fairshare**.
- When sufficient nodes are free to execute a job, it starts the job on the appropriate compute nodes.
- Jobs remain **'idle'** until resources become available.
- Jobs can be temporarily **'blocked'** if you submit too much.

Components

Torque: Resource manager providing control over batch jobs and distributed compute nodes.

Moab: A policy-based job scheduler and event engine that enables utility-based computing for clusters.

Fairshare: Mechanism using past utilization for prioritization.

Job cycle

Preparation	Monitor	Control	Reports
<ul style="list-style-type: none"> • Compile • Test on devel node • Determine resources • Write job script <p><code>qsub script</code> (returns <i>'jobid'</i>)</p>	<ul style="list-style-type: none"> • Job queued? • When will it run? • What else is queued? • Efficiency? <p><code>qstat -f jobid</code> <code>checkjob jobid</code> <code>showstart jobid</code> <code>showbf</code> <code>showq</code> <code>showq -r -u user</code></p>	<ul style="list-style-type: none"> • Cancel job • Ssh to nodes • Interactive jobs • Debug queue <p><code>canceljob jobid</code> <code>ssh node</code> <code>top</code> <code>qsub -I</code> <code>qsub -q debug</code></p>	<ul style="list-style-type: none"> • Check <code>.o/.e jobid.{o,e}</code> • short-term statistics: <code>showstats -u user</code> • year-to-date usage on: <p>https://portal.scinet.utoronto.ca</p>

Monitoring not-yet-running jobs

qstat and checkjob

- Show torque status right away on GPC: `qstat`
- Show moab status (better): `checkjob jobid`
- See more details of the job: `checkjob -v jobid`
(e.g., why is my job blocked?)

showq

- See all the jobs in the queue: `showq` (from gpc or tcs)
- See your jobs in the queue: `showq -u user`

showstart and showbf

- Estimate when a job may start: `showbf [-f ib]`
- Estimate when a queued job may start: `showstart jobid`

Monitoring running jobs

checkjob

- `checkjob jobid`

output/error files

- `/var/spool/torque/spool/jobid.OU`
- `/var/spool/torque/spool/jobid.ER`

showq

- `showq -r -u user`

ssh

- `ssh node` (node name from checkjob)
- `top`: shows process state, memory and cpu usage

Top example

```
gpc-f103n084-$ ssh gpc-f109n001
gpc-f109n001-$ top
```

```
top - 21:56:45 up 5:56, 1 user, load average: 5.55, 1.73, 0.88
Tasks: 234 total, 1 running, 233 sleeping, 0 stopped, 0 zombie
Cpu(s): 11.4%us, 36.2%sy, 0.0%ni, 52.2%id, 0.0%wa, 0.0%hi, 0.2%si, 0.0%st
Mem: 16410900k total, 1542768k used, 14868132k free, 0k buffers
Swap: 0k total, 0k used, 0k free, 294628k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	P	COMMAND
22479	ljdursi	18	0	108m	4816	3212	S	98.5	0.0	1:04.81	6	gameoflife
22480	ljdursi	18	0	108m	4856	3260	S	98.5	0.0	1:04.85	13	gameoflife
22482	ljdursi	18	0	108m	4868	3276	S	98.5	0.0	1:04.83	2	gameoflife
22483	ljdursi	18	0	108m	4868	3276	S	98.5	0.0	1:04.82	8	gameoflife
22484	ljdursi	18	0	108m	4832	3232	S	98.5	0.0	1:04.80	9	gameoflife
22481	ljdursi	18	0	108m	4856	3256	S	98.2	0.0	1:04.81	3	gameoflife
22485	ljdursi	18	0	108m	4808	3208	S	98.2	0.0	1:04.80	4	gameoflife
22478	ljdursi	18	0	117m	5724	3268	D	69.6	0.0	0:46.07	15	gameoflife
8042	root	0	-20	2235m	1.1g	16m	S	2.3	6.8	0:30.59	8	mmfsd
10735	root	15	0	3702	452	372	S	1.3	0.0	0:16.80	0	cat

Top example

```
gpc-f103n084-$ ssh gpc-f109n001
gpc-f109n001-$ top
```

```
top - 21:56:45 up 5:56, 1 user, load average: 5.55, 1.73, 0.88
Tasks: 234 total, 1 running, 233 sleeping, 0 stopped, 0 zombie
Cpu(s): 11.4%us, 36.2%sy, 0.0%ni, 52.2%id, 0.0%wa, 0.0%hi, 0.2%si, 0.0%st
Mem: 16410900k total, 1542768k used, 14868132k free, 0k buffers
Swap: 0k total, 0k used, 0k free, 294628k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	P	COMMAND
22479	ljdursi	18	0	108m	4816	3212	S	98.5	0.0	1:04.81	6	gameoflife
22480	ljdursi	18	0	108m	4856	3260	S	98.5	0.0	1:04.85	13	gameoflife
22482	ljdursi	18	0	108m	4868	3276	S	98.5	0.0	1:04.83	2	gameoflife
22483	ljdursi	18	0	108m	4868	3276	S	98.5	0.0	1:04.82	8	gameoflife
22484	ljdursi	18	0	108m	4832	3232	S	98.5	0.0	1:04.80	9	gameoflife
22481	ljdursi	18	0	108m	4856	3256	S	98.2	0.0	1:04.81	3	gameoflife
22485	ljdursi	18	0	108m	4808	3208	S	98.2	0.0	1:04.80	4	gameoflife
22478	ljdursi	18	0	117m	5724	3268	D	69.6	0.0	0:46.07	15	gameoflife
8042	root	0	-20	2235m	1.1g	16m	S	2.3	6.8	0:30.59	8	minisid
10735	root	15	0	3702	452	372	S	1.3	0.0	0:16.80	0	cat

canceljob

- If you spot a mistake: `canceljob jobid`

qsub for interactive and debug jobs

- **-I:**
 - Interactive
 - After qsub, waits for jobs to start.
 - Usually combined with:
- **-q debug:**
 - Debug queue has 10 nodes reserved for short jobs.
 - You can get 1 node for 2 hours, but also
 - 8 nodes, for half an hour.

output/error files

- *.e / *.o

In submission directory by default, unless set in *script*.

- If for some reason no .o and .e created, look for

/var/spool/torque/spool/jobid.OU

/var/spool/torque/spool/jobid.ER

Statistics

- Short term: `showstats -u USER`
- Year-to-date: [SciNet Portal](#)

Usage stats for past year, showing a breakdown of TCS, GPC_eth, and GPC_ib usage. Updated every 24 hours.

output/error files

- *.e / *.o

In submission directory by default, unless set in *script*.

```
-----
Begin PBS Prologue Tue Sep 14 17:14:48 EDT 2010 1284498888
Job ID:      3053514.gpc-sched
Username:    ljdursi
Group:       scinet
Nodes:       gpc-f134n009 gpc-f134n010 gpc-f134n011 gpc-f134n012
             gpc-f134n043 gpc-f134n044 gpc-f134n045 gpc-f134n046 gpc-f134n047 gpc-f134n048
[...]
End PBS Prologue Tue Sep 14 17:14:50 EDT 2010 1284498890
-----
[ Your job's output here... ]
-----
Begin PBS Epilogue Tue Sep 14 17:36:07 EDT 2010 1284500167
Job ID:      3053514.gpc-sched
Username:    ljdursi
Group:       scinet
Job Name:    fft_8192_procs_2048
Session:     18758
Limits:      neednodes=256:ib:ppn=8,nodes=256:ib:ppn=8,walltime=01:00:00
Resources:   cput=713:42:30,mem=3463854672kb,vmem=3759656372kb,walltime=00:21:07
Queue:       batch_ib
Account:
Nodes:       gpc-f134n009 gpc-f134n010 gpc-f134n011 gpc-f134n012 gpc-f134n043
[...]
Killing leftovers...
gpc-f141n054:  killing gpc-f141n054 12412

End PBS Epilogue Tue Sep 14 17:36:09 EDT 2010 1284500169
-----
```

Monitoring jobs - recap

From the scheduler's point-of-view

- Once the job is incorporated into the queue (this takes a minute), you can use `showq` to show the queue, and job-specific commands such as `showstart`, `checkjob`, `canceljob`

On the node

- Once the job is running, you can check on what node it is running (`showq -r`)
- This node is yours for the duration of the run, and you can `ssh` into it.
- **top**: tells you about the resources that are used on the node.
- Check and track memory usage.
- Check and track cpu usage.
- Ensure that your program is not stuck in 'D' (waiting for Disk).
- `/var/spool/torque/spool` contains the `.o` and `.e` files so far.

- 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.

Part V

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.

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.

- Distributed parallelism:

Con: does not shared memory even if it can.

Pro: But can (potentially) scalable 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?

There is no queue for serial jobs, so if you have serial jobs, **YOU** will have to bunch together 8 of them to use the node's full power.

Easy case: serial jobs of equal duration

```
#!/bin/bash
#PBS -l nodes=1:ppn=8,walltime=1:00:00
#PBS -N serialx8
cd $PBS_O_WORKDIR
(cd jobdir1; ./dojob1) &
(cd jobdir2; ./dojob2) &
(cd jobdir3; ./dojob3) &
(cd jobdir4; ./dojob4) &
(cd jobdir5; ./dojob5) &
(cd jobdir6; ./dojob6) &
(cd jobdir7; ./dojob7) &
(cd jobdir8; ./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 80 nodes with 32 GB memory.

Hard case: serial jobs of unequal duration

What you need is: Load Balancing

- Keep all 8 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 each 8core node busy, if you provide enough jobs to do.

GNU parallel is accessible on the GPC in the module gnu-parallel, which you can load in your .bashrc.

```
$ module load gnu-parallel/20120622
```

Note that there are currently (Sep 2012) two versions of gnu-parallel installed on the GPC, with the older version, gnu-parallel/2010, as the default, although we'd recommend using the newer version.

GNU Parallel Example

```
gpc-f101n084-$
```

GNU Parallel Example

```
gpc-f101n084-$ module load intel  
gpc-f101n084-$
```

GNU Parallel Example

```
gpc-f101n084-$ module load intel  
gpc-f101n084-$ icpc -O3 -xhost mycode.cc -o mycode  
gpc-f101n084-$
```

GNU Parallel Example

```
gpc-f101n084-$ module load intel
gpc-f101n084-$ icpc -O3 -xhost mycode.cc -o mycode
gpc-f101n084-$ mkdir $SCRATCH/example2
gpc-f101n084-$
```


GNU Parallel Example

```
gpc-f101n084-$ module load intel
gpc-f101n084-$ icpc -O3 -xhost mycode.cc -o mycode
gpc-f101n084-$ mkdir $SCRATCH/example2
gpc-f101n084-$ cp mycode $SCRATCH/example2
gpc-f101n084-$
```

GNU Parallel Example

```
gpc-f101n084-$ module load intel
gpc-f101n084-$ icpc -O3 -xhost mycode.cc -o mycode
gpc-f101n084-$ mkdir $SCRATCH/example2
gpc-f101n084-$ cp mycode $SCRATCH/example2
gpc-f101n084-$ cd $SCRATCH/example2
gpc-f101n084-$
```

GNU Parallel Example

```
gpc-f101n084-$ module load intel
gpc-f101n084-$ icpc -O3 -xhost mycode.cc -o mycode
gpc-f101n084-$ mkdir $SCRATCH/example2
gpc-f101n084-$ cp mycode $SCRATCH/example2
gpc-f101n084-$ cd $SCRATCH/example2
gpc-f101n084-$ cat > joblist.txt
    mkdir run1; cd run1; ../mycode 1 > out
    mkdir run2; cd run2; ../mycode 2 > out
    mkdir run3; cd run3; ../mycode 3 > out
    ...
    mkdir run64; cd run64; ../mycode 64 > out
gpc-f101n084-$
```

GNU Parallel Example

```
gpc-f101n084-$ module load intel
gpc-f101n084-$ icpc -O3 -xhost mycode.cc -o mycode
gpc-f101n084-$ mkdir $SCRATCH/example2
gpc-f101n084-$ cp mycode $SCRATCH/example2
gpc-f101n084-$ cd $SCRATCH/example2
gpc-f101n084-$ cat > joblist.txt
    mkdir run1; cd run1; ../mycode 1 > out
    mkdir run2; cd run2; ../mycode 2 > out
    mkdir run3; cd run3; ../mycode 3 > out
    ...
    mkdir run64; cd run64; ../mycode 64 > out
gpc-f101n084-$ cat > myjob.pbs
    #!/bin/bash
    #PBS -l nodes=1:ppn=8,walltime=24:00:00
    #PBS -N GPJob
    cd $PBS_O_WORKDIR
    module load intel gnu-parallel/20120622
    parallel -j 8 < joblist.txt
gpc-f101n084-$
```

GNU Parallel Example

```
gpc-f101n084-$ module load intel
gpc-f101n084-$ icpc -O3 -xhost mycode.cc -o mycode
gpc-f101n084-$ mkdir $SCRATCH/example2
gpc-f101n084-$ cp mycode $SCRATCH/example2
gpc-f101n084-$ cd $SCRATCH/example2
gpc-f101n084-$ cat > joblist.txt
    mkdir run1; cd run1; ../mycode 1 > out
    mkdir run2; cd run2; ../mycode 2 > out
    mkdir run3; cd run3; ../mycode 3 > out
    ...
    mkdir run64; cd run64; ../mycode 64 > out
gpc-f101n084-$ cat > myjob.pbs
    #!/bin/bash
    #PBS -l nodes=1:ppn=8,walltime=24:00:00
    #PBS -N GPJob
    cd $PBS_O_WORKDIR
    module load intel gnu-parallel/20120622
    parallel -j 8 < joblist.txt
gpc-f101n084-$ qsub myjob.pbs
    2961985.gpc-sched
gpc-f101n084-$
```

GNU Parallel Example

```
gpc-f101n084-$ module load intel
gpc-f101n084-$ icpc -O3 -xhost mycode.cc -o mycode
gpc-f101n084-$ mkdir $SCRATCH/example2
gpc-f101n084-$ cp mycode $SCRATCH/example2
gpc-f101n084-$ cd $SCRATCH/example2
gpc-f101n084-$ cat > joblist.txt
    mkdir run1; cd run1; ../mycode 1 > out
    mkdir run2; cd run2; ../mycode 2 > out
    mkdir run3; cd run3; ../mycode 3 > out
    ...
    mkdir run64; cd run64; ../mycode 64 > out
gpc-f101n084-$ cat > myjob.pbs
    #!/bin/bash
    #PBS -l nodes=1:ppn=8,walltime=24:00:00
    #PBS -N GPJob
    cd $PBS_O_WORKDIR
    module load intel gnu-parallel/20120622
    parallel -j 8 < joblist.txt
gpc-f101n084-$ qsub myjob.pbs
2961985.gpc-sched
gpc-f101n084-$ ls
GPJob.e2961985  GPJob.o2961985  joblist.txt  mycode
myjob.pbs      run1/           run2/
```

- HyperThreading: Appears as if there are 16 processors rather than 8 per node.
 - For OpenMP applications this is the default unless OMP_NUM_THREADS is set.
 - For MPI, try `-np 16`.
 - For gnu parallel, use `-j 16`.
- Always request `ppn=8`, even with hyperthreading.
- *Always test if this is beneficial and feasible!*

GNU parallel on our example problem

Assume only 4 instances of dalex will fit in memory:

```
#!/bin/bash
#PBS -l nodes=1:ppn=8
#PBS -l walltime=48:00:00
#PBS -N JobName
cd $PBS_O_WORKDIR
module load gnu-parallel/20120622
for a in $(seq 1000)
do
    echo "" > results/batch$a-out.dat
done
for c in $(seq 23)
do
    seq 1000 | parallel -j4 \
        "dalex -m .95 -w 32 humanref/chromosome$c.dat \
        samples/batch{}.dat >> results/batch{}-out.dat"
done
```


Can you think of more improvements?

Conclusions

I hope to have conveyed that

- (Bio-)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.