Christopher S. Simmons (csim@ices.utexas.edu)

Practical Git, Batch Systems and Compilers

CSE 380: Tools and Techniques of Computational Science

Quick questions from Unix …

- Every thing in Unix is either a ___ or a ___ ?
- What command might you use to show running processes?
- What is a pipe?
- What are the two common output streams for unix processes?
- What environment variable controls where to search for commands in your shell?

Part 1: Practical Git

much of the content taken from:

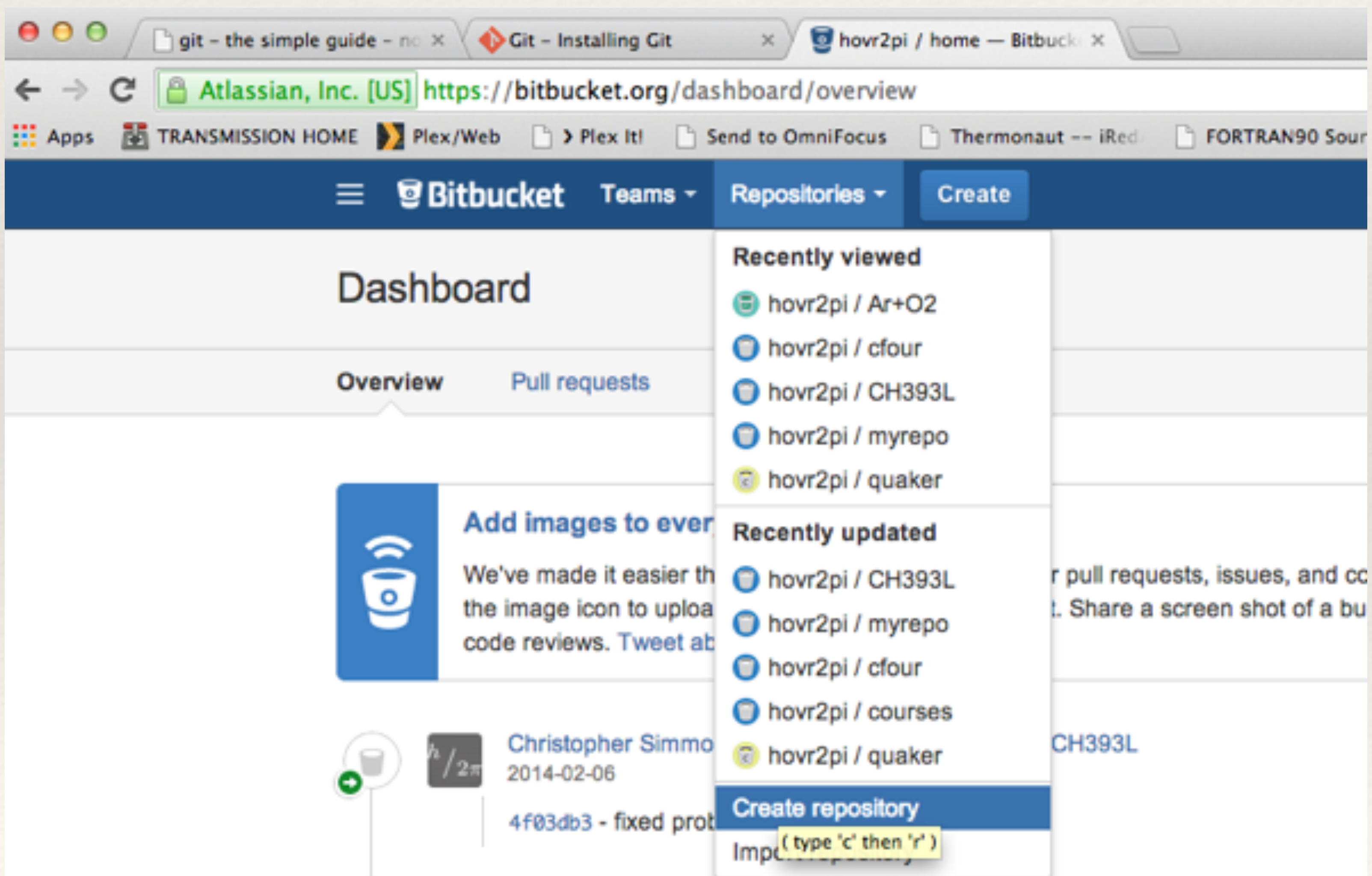
http://rogerdudler.github.io/git-guide/

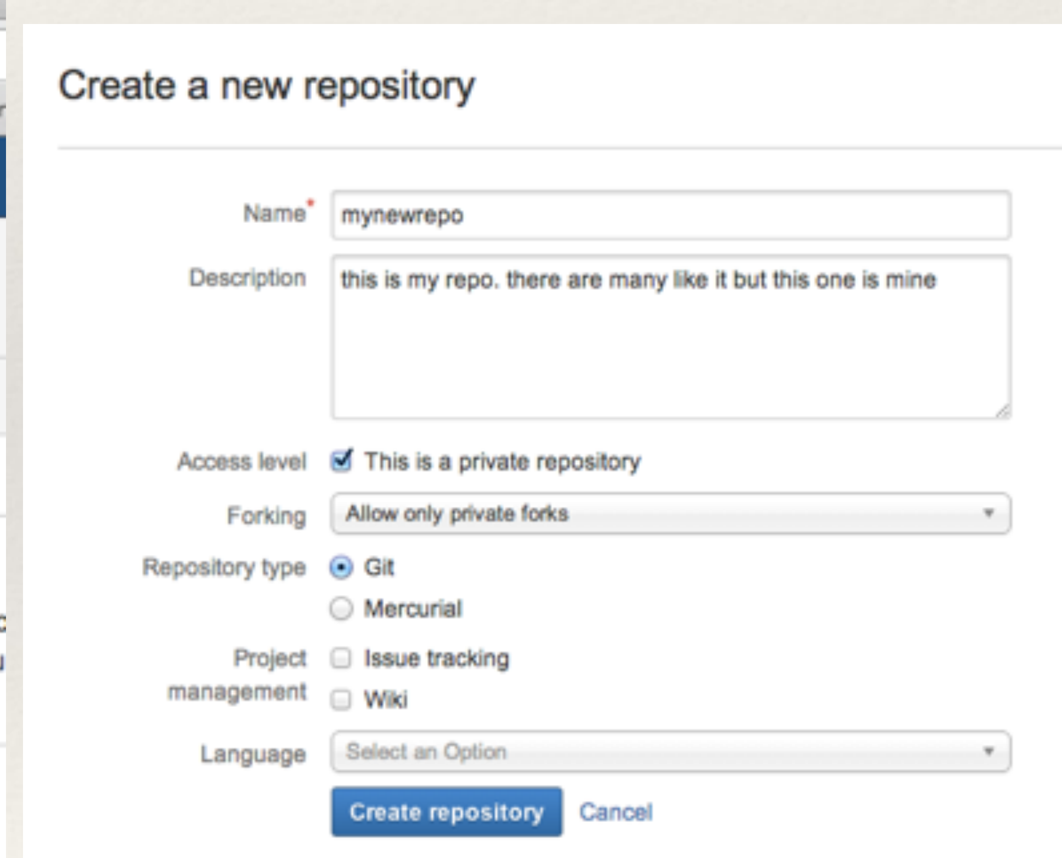
setup

- Download git for OSX — http://code.google.com/p/git-osx-installer/downloads/list?can=3
- Download git for Windows — http://code.google.com/p/msysgit/downloads/list?can=3
- Dowload git for Linux — http://book.git-scm.com/2_installing_git.html
- On OSX and Linux, use your package managers!

Create a new repo

```
csim@gedanken ~/class $ mkdir mynewrepo && cd mynewrepo
csim@gedanken ~/class/mynewrepo $ git init
Initialized empty Git repository in /Users/csim/class/mynewrepo/.git/
csim@gedanken ~/class/mynewrepo (master) $
```





introduce yourself to git

```bash
#!/bin/bash

git config --global user.name "Christopher Simmons"
git config --global user.email csim@hovr2pi.org
case `uname` in

Darwin)
        git config --global credential.helper osxkeychain
        ;;

Linux)
        git config --global credential.helper "cache --timeout=3600"
        echo "caching does no good if git-daemon is not installed"
;;
esac
```

Populate your repo

Push up an existing repository

You already have a Git repository on your computer. Let's push it up to Bitbucket.

```
$ cd /path/to/my/repo
$ git remote add origin https://hovr2pi@bitbucket.org/hovr2pi/mynewrepo2.git
$ git push -u origin --all # pushes up the repo and its refs for the first time
$ git push -u origin --tags # pushes up any tags
```

Do you want to grab a repo from another site? Try our importer!

Next

Clone your new repo

Set up Git on your machine if you haven't already.

```
$ mkdir /path/to/your/project
$ cd /path/to/your/project
$ git init
$ git remote add origin https://hovr2pi@bitbucket.org/hovr2pi/mynewrepo.git
```

Visit Bitbucket 101 for more help getting set up.

Next

checkout a repo

create a working copy of a local repository by running the command

```
git clone /path/to/repository
```

when using a remote server, your command will be

```
git clone username@host:/path/to/repository
```

Basic Git Workflow

your local repository consists of three "trees" maintained by git. the first one is your `Working Directory` which holds the actual files. the second one is the `Index` which acts as a staging area and finally the `HEAD` which points to the last commit you've made.

add & commit

You can propose changes (add it to the **Index**) using

```
git add <filename>
```

```
git add *
```

This is the first step in the basic git workflow. To actually commit these changes use

```
git commit -m "Commit message"
```

Now the file is committed to the **HEAD**, but not in your remote repository yet.

pushing changes

Your changes are now in the **HEAD** of your local working copy. To send those changes to your remote repository, execute

```
git push origin master
```

Change *master* to whatever branch you want to push your changes to.

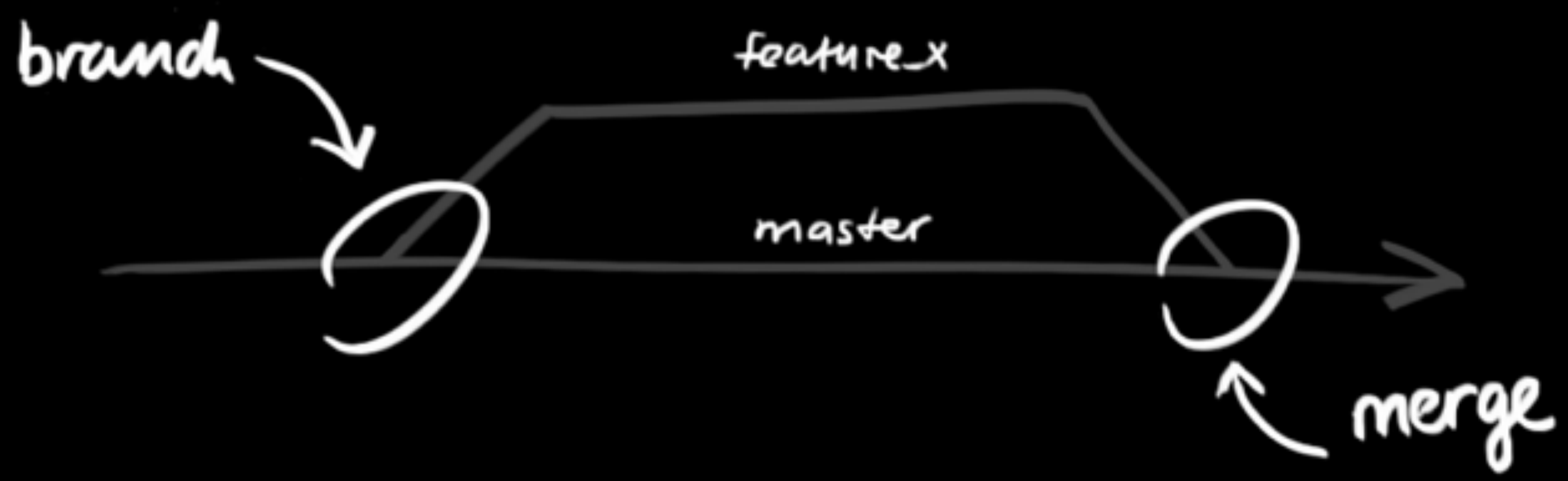
If you have not cloned an existing repository and want to connect your repository to a remote server, you need to add it with

```
git remote add origin <server>
```

Now you are able to push your changes to the selected remote server

branching

Branches are used to develop features isolated from each other. The *master* branch is the "default" branch when you create a repository. Use other branches for development and merge them back to the master branch upon completion.



new branch and switch to it

create a new branch named "feature_x" and switch to it using

```
git checkout -b feature_x
```

switch back to master

```
git checkout master
```

and delete the branch again

```
git branch -d feature_x
```

a branch is *not available to others* unless you push the branch to your remote repository

```
git push origin <branch>
```

update & merge

to update your local repository to the newest commit, execute

```
git pull
```

in your working directory to *fetch* and *merge* remote changes.

to merge another branch into your active branch (e.g. master), use

```
git merge <branch>
```

in both cases git tries to auto-merge changes. Unfortunately, this is not always possible and results in *conflicts*. You are responsible to merge those *conflicts* manually by editing the files shown by git. After changing, you need to mark them as merged with

```
git add <filename>
```

before merging changes, you can also preview them by using

```
git diff <source_branch> <target_branch>
```

tagging

it's recommended to create tags for software releases. this is a known concept, which also exists in SVN. You can create a new tag named *1.0.0* by executing

```
git tag 1.0.0 1b2e1d63ff
```

the *1b2e1d63ff* stands for the first 10 characters of the commit id you want to reference with your tag. You can get the commit id with

```
git log
```

you can also use fewer characters of the commit id, it just has to be unique.

replace local changes

In case you did something wrong (which for sure never happens ;) you can replace local changes using the command

```
git checkout -- <filename>
```

this replaces the changes in your working tree with the last content in HEAD. Changes already added to the index, as well as new files, will be kept.

If you instead want to drop all your local changes and commits, fetch the latest history from the server and point your local master branch at it like this

```
git fetch origin
git reset --hard origin/master
```

useful hints

built-in git GUI

`gitk`

use colorful git output

`git config color.ui true`

show log on just one line per commit

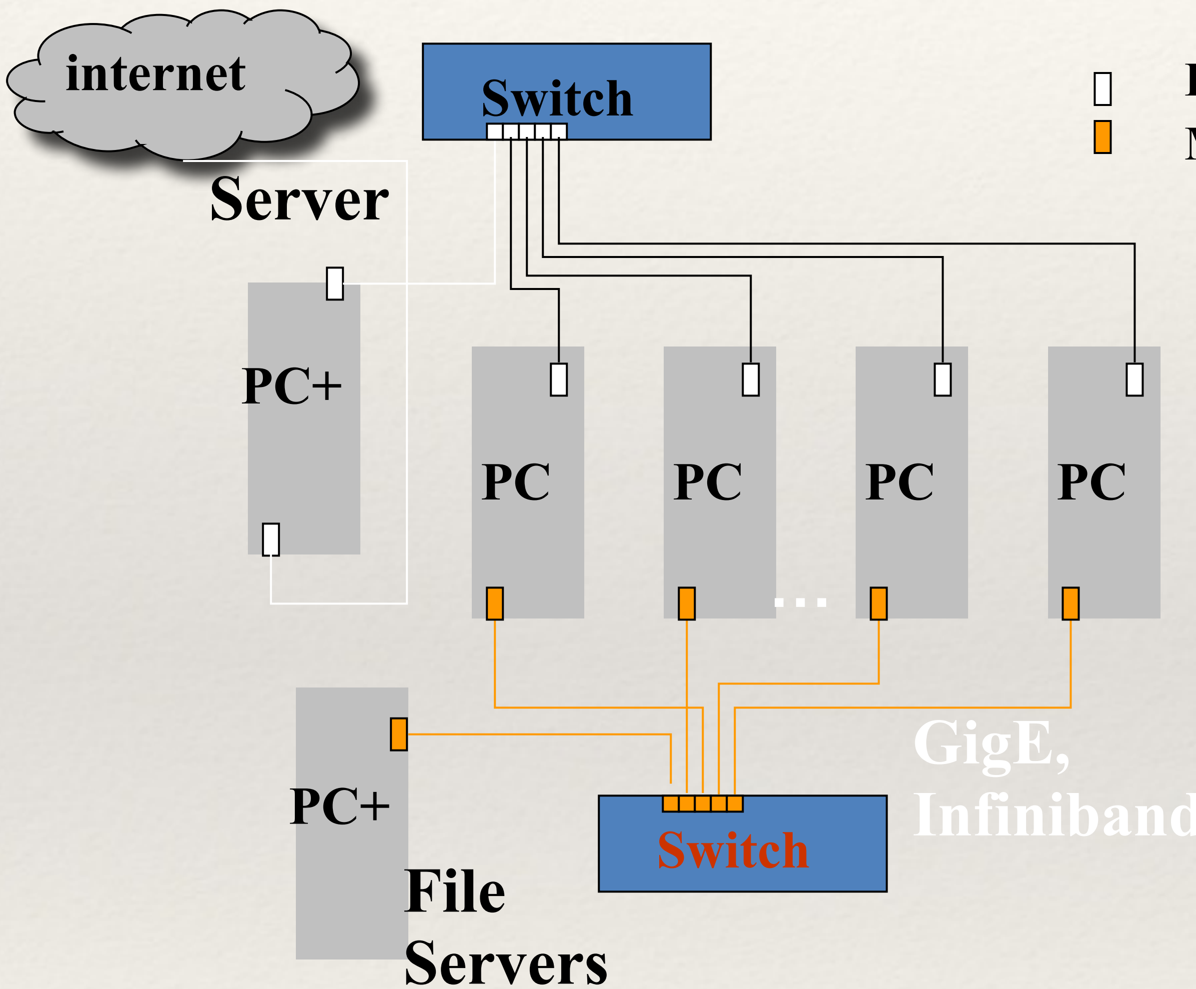
`git config format.pretty oneline`

use interactive adding

`git add -i`

Part 2: batch systems

Generic Cluster Architecture

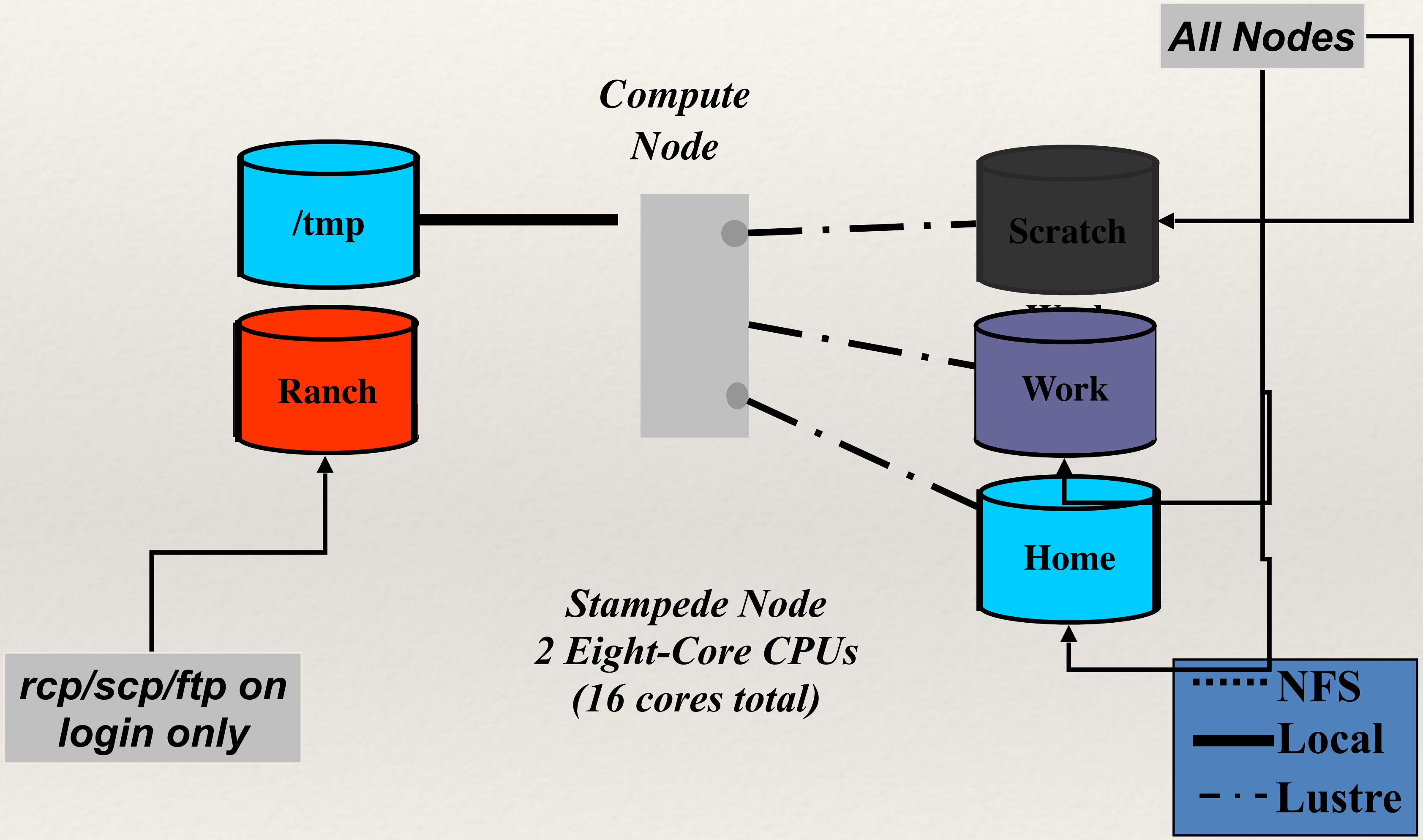


lonestar.tacc.utexas.edu

Quick Question from PRC Visit

- Which network is faster: GigE or InfiniBand?
- Which has the lower latency?
- Why don’t supercomputers use NFS file servers for big data file systems?

Available File Systems (Stampede)



File System Access & Lifetime Table (Stampede)

Environment	User Access Limit	Life Time
$HOME	*5 GB quota (150K files)*	*Project*
$WORK	*~1000 GB quota (3M*	*Project*
$SCRATCH	*~8.5 PB max*	*10 Days*
$ARCHIVE	*Unlimited*	*Project*

Power User Tip: use the aliases cd, cdw, and cds to change directory to $HOME, $WORK and $SCRATCH respectively.

Modules

- Modules are used to setup and remove various environment variables along with PATH, LD_LIBRARY_PATH declarations

- They are used to setup environments for packages & compilers.

  lslogin1% module {lists options}
  lslogin1% module list {lists loaded modules}
  lslogin1% module avail {lists available modules}
  lslogin1% module del <module> {removes a module}
  lslogin1% module add <module> {add a module}
  lslogin1% module switch <mod1> <mod2> {switch modules}

- lslogin1% module swap <mod1> <mod2> {swap modules}

- Example of module avail on Stampede:

Modules

- Modules often define environment variables for convenient access to binaries, libraries, include files, and documentation

- See individual module's help for more information (and suggestions on linking against 3rd party libraries). For example: lslogin1$ module help intel

- Example of module help intel on Stampede

Batch Systems

- In a number of scientific computing environments, multiple users must share a compute resource:
  - research clusters
  - supercomputing centers
- On multi-user HPC clusters, the batch system is a key component for aggregating compute nodes into a single, sharable computing resource
- The batch system becomes the “nerve center” for coordinating the use of resources and controlling the state of the system in a way that must be “fair” to its users
- As current and future expert users of large-scale compute resources, you need to be familiar with the basics of a batch system
  - remember, you look like a rookie when you don’t use the batch system and try to run long-running jobs on a login node

Batch Systems

- The core functionality of all batch systems is essentially the same, regardless of the size or specific configuration of the compute hardware:
  - Multiple Job Queues:
    - queues provide an orderly environment for managing a large number of jobs
    - queues are defined with a variety of limits for maximum run times, memory usage, and processor counts; they are often assigned different priority levels as well
    - may be interactive or non-interactive
  - Job Control:
    - submission of individual jobs to do some work (eg. serial, or parallel HPC applications)
    - simple monitoring and manipulation of individual jobs, and collection of resource usage statistics (e.g., memory usage, CPU usage, and elapsed wall-clock time per job)
  - Job Scheduling
    - policy which decides priority between individual user jobs
    - allocates resources to scheduled jobs

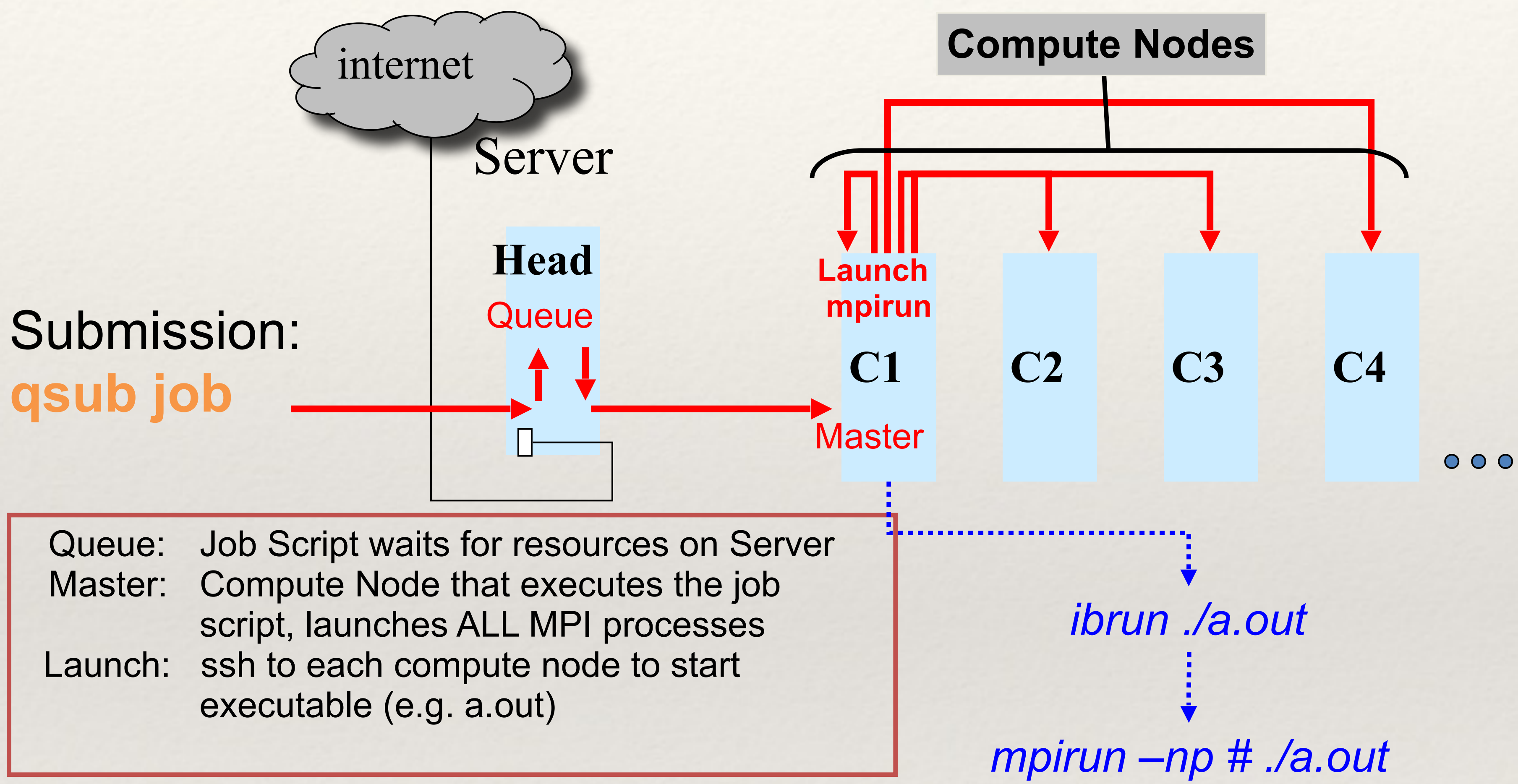
Batch Systems

- Job Scheduling Policies:
  - the scheduler must decide how to prioritize all the jobs on the system and allocate necessary resources for each job (processors, memory, file-systems, etc)
  - scheduling process can be easy or non-trivial depending on the size and desired functionality
    - first in, first out (FIFO) scheduling: jobs are simply scheduled in the order in which they are submitted
    - political scheduling: enables some users to have more priority than others
    - fairshare scheduling, scheduler ensures users have equal access over time
  - Additional features may also impact scheduling order:
    - advanced reservations - resources can be reserved in advance for a particular user or job
    - backfill - can be combined with any of the scheduling paradigms to allow smaller jobs to run while waiting for enough resources to become available for larger jobs
      - back-fill of smaller jobs helps maximize the overall resource utilization
      - back-fill can be your friend for small duration jobs

Batch Systems

- Common batch systems you may encounter in scientific computing:
  - Platform LSF (now owned by IBM)
  - PBS/Torque (seen on Linux and Cray systems)
  - Loadleveler (IBM)
  - SGE
  - SLURM
- All have similar functionality but different syntax
- Reasonably straight forward to convert your job scripts from one system to another
- The above all include specific batch system directives which can be placed in a shell script to request certain resources (processors, queues, etc).
- We will focus on SLURM primarily since it is the system running on Stampede (also a bit of SGE which runs on Lonestar)

Batch Submissions



For Serial Tasks: no launcher is required

./a.out

Stampede Queue Definitions

Queue Name	Max Runtime	Max Nodes/Procs	SU Charge Rate	Purpose
normal	48 hrs	256 / 4K	1	normal production
normal-mic	24 hrs	256 / 4K	1	normal MIC production
large	24 hrs	1024 / 16K	1	large core counts (access by request)
request	24 hrs	--	1	special requests
largemem	24 hrs	4 / 128	2	large memory 32 cores/ node, Teslas, no MICs
development	4 hrs	16 / 256	1	development nodes
serial	12 hrs	1 / 16	1	serial/shared_memory
gpu	24 hrs	32 / 512	1	GPU nodes
gpudev	4 hrs	4 / 64	1	GPU development nodes
vis	8 hrs	32 / 512	1	GPU nodes + VNC service

Fairshare

- A global fairshare mechanism is implemented on Lonestar and Stampede to provide fair access to its substantial compute resources
- Fairshare computes a dynamic priority for each user and uses this priority in making scheduling decisions
- Dynamic priority is based on the following criteria
  - Number of shares assigned
  - Resources used by jobs belonging to the user:
    - Number of job slots reserved
    - Run time of running jobs
    - Cumulative actual CPU time (not normalized), adjusted so that recently used CPU time is weighted more heavily than CPU time used in the distant past

SLURM Batch System: Env. Variables

Variable	Purpose
SLURM_JOB_ID	Batch job id
SLURM_JOB_NAME	User-assigned (-J) name of the job
SLURM_NTASKS	Total number of tasks
SLURM_QUEUE	Name of the queue the job is running in
SLURM_SUBMIT_DIR	Directory of submission
SLURM_NODELIST	List of nodes assigned to job
SLURM_TACC_ACCOUNT	Location of the file where standard output/error is being written

SGE Batch System: Env. Variables

Variable	Purpose
JOB_ID	Batch job id
JOB_NAME	User-assigned (-J) name of the job
NSLOTS	Number of slots/processes for a parallel job
QUEUE	Name of the queue the job is running in
PE	Parallel environment used by the job
SGE_STDOUT_PATH SGE_STDERR_PATH	Location of the file where standard output/error is being written

SLURM: Basic Job Script

```
#!/bin/bash
#SBATCH -N 2
#SBATCH -n 32
#SBATCH -o myjob.%j.out
#SBATCH -J myjob
#SBATCH -P normal
#SBATCH -A CSE380
#SBATCH -t 00:15:00

echo "Master Host = "`hostname`
echo "PWD_DIR: "`pwd`

ibrun ./hello
```

Total number of nodes
Total number of cores
Output file name
Job name
Submission queue
Your Project Name
Max Run Time (15 minutes)

Echo pertinent environment info

Execution command

Parallel application manager and mpirun wrapper script

executable

Job Sizing with SLURM

- Normally, you will want to use a multiple of 16 for the number of cores requested (compute nodes are not shared)
- If you only specify the "-n" option, we will assume you want 16 tasks per node
- When might you want to deviate from this?
  - When you need more memory per task
  - For example, to double the amount of memory available per task, you can request 8 tasks/node:
    - #SBATCH -N 2
    - #SBATCH -n 16
  - Or to maximize the memory for 1 task:
    - #SBATCH -N 2
    - #SBATCH -n 2

Batch System Concerns

- Submission (need to know)
  - Required Resources
  - Run-time Environment
  - Directory of Submission
  - Directory of Execution
  - Files for stdout/stderr Return
  - Email Notification
- Job Monitoring
- Job Deletion
  - Queued Jobs
  - Running Jobs

SLURM: Extended Job Script

```
#!/bin/bash
#SBATCH -N 2                          Total number of nodes
#SBATCH -n 32                         Total number of cores
#SBATCH -o myjob.%j.out               stdout file name
#SBATCH -e error.%j.out               stderr file name
#SBATCH -J myjob                      Job name
#SBATCH -P normal                     Submission queue
#SBATCH -A CSE380                     Your Project Name
#SBATCH -t 00:15:00                   Max Run Time (15 minutes)
#SBATCH --dependency=afterok:1123     dependent on job 1123
#SBATCH --mail-user=user@domain       desired email address
#SBATCH --mail-type=begin             send email at job start

ibrun ./hello
```

Batch Script Suggestions

- Echo issuing commands
  - (“set -x” or “set echo” for ksh and csh).
- Avoid absolute pathnames
  - Use relative path names or environment variables ($HOME, $WORK)
- Abort job when a critical command fails.
- Print environment
  - Include the "env" command if your batch job doesn't execute the same as in an interactive execution.
- Use “./” prefix for executing commands in the current directory
  - The dot means to look for commands in the present working directory. Not all systems include "." in your $PATH variable. (usage: ./a.out).
- Track your CPU time

Job Monitoring (showq utility)

```
lslogin1% showq
ACTIVE JOBS--------------------
JOBID     JOBNAME    USERNAME      STATE  PROC  REMAINING       STARTTIME


11318  1024_90_96x6    vmcalo    Running    64    18:09:19  Fri Jan  9 10:43:53
11352          naf   phaa406    Running    16   17:51:15  Fri Jan  9 10:25:49
11357          24N   phaa406    Running    16   18:19:12  Fri Jan  9 10:53:46
 23 Active jobs     504 of 556 Processors Active (90.65%)


IDLE JOBS----------------------
JOBID     JOBNAME    USERNAME      STATE  PROC    WCLIMIT       QUEUETIME


11169        poroe8      xgai       Idle    128    10:00:00  Thu Jan  8 10:17:06
11645   meshconv019    bbarth       Idle     16    24:00:00  Fri Jan  9 16:24:18
 3 Idle jobs

BLOCKED JOBS-------------------
```

Job Monitoring (showq utility)

- Other tips for showq (TACC has versions for SGE, LSF, and SLURM)
  - "showq –u" will just summarize your jobs
  - "showq –l" will provide a long listing showing submission queues

SLURM Job Manipulation

- To submit a new job request:

    sbatch <job.script>

- Note: you can override the settings in the script via the command-line; handy for changing queue or job size:

    sbatch -p development <job.script>

- To kill a running or queued job:

    scancel <jobID>

- To estimate a job starting time:

    squeue -u <user> --start

    squeue -u jet2016 --start

- JOBID PARTITION NAME USER ST START_TIME NODES NODELIST(REASON)
- 1705331 gpu nona2-s1 jet2016 PD 2013-09-17T10:54:55 4 (Resources)

Interactive Access

- Interactive execution is a method where you request a login shell from the batch system

- When the requested amount of resources are available, you are placed on the master compute host associated with your job
  - you can then run multiple jobs
  - very handy for debugging and development (you can issue ibrun on the command line if you want)
- Note that TACC will not share compute hosts and you are allowed to login to each compute hosts assigned to your job (allows you to attach a debugger to a specific process if necessary)

- There are two ways to request interactive access:
  - via SLURM directly using srun
    - see the motd on the login nodes for an example

      ```
      $ srun -p development -t 0:30:00 -n 32 --pty /bin/bash -l
      ```

    - Note that this command will hang until resources are available
    - Works good for small scale jobs - can hang for larger scale jobs
  - via a locally developed tool called "idev"

Interactive Access: srun Example

```
login1$  srun –p devel --pty /bin/bash –l
c401-102$ cat hello.c
#include<stdio.h>
int main()
{
  printf("Hook 'em Horns!\n");

#ifdef __MIC__
  printf(" --> Ditto from MIC\n");
#endif
}

c401-102$ icc hello.c
c401-102$ ./a.out
Hook 'em Horns!

c401-102$ icc –mmic hello.c
c401-102$ ./a.out
bash: ./a.out: cannot execute binary file

c401-102$ ssh mic0 ./a.out
Hook 'em Horns!
 --> Ditto from MIC
```

Interactive Hello World

- Interactive programming example
  - Request interactive job (srun)
  - Compile on the compute node
  - Using the Intel compiler toolchain
  - Here, we are building a simple hello world…
- First, compile for SNB and run on the host
  - note the __MIC__ macro can be used to isolate MIC only execution, in this case no extra output is generated on the host
- Next, build again and add "-mmic" to ask the compiler to *cross-compile* a binary for native MIC execution
  - note that when we try to run the resulting binary on the host, it throws an error
  - ssh to the MIC (mic0) and run the executable out of $HOME directory
  - this time, we see extra output from within the guarded__MIC__ macro

Interactive Access - idev Example

```
$ idev -A A-ccsc -p normal-mic -N 2 -n 32

Submitted batch job 1705440

 After your idev job begins to run, a command prompt will appear,
 and you can begin your interactive development session.
 We will report the job status every 4 seconds: (PD=pending, R=running).

job status:  PD
job status:  PD
job status:  R
--> Job is now running on masternode= c558-201...OK
--> Creating interactive terminal session (login) on master node c558-201.
TACC Stampede System
LosF 0.40.0 (Top Notch)
Provisioned on 23-Sep-2012 at 15:46
 c558-201$
```

Job Aggregation

- A common need in scientific computing arises when we need to perform the same basic calculation on a variety of data sets
  - e.g. parametric sampling
  - in some cases, this is an embarrassingly parallel workload
  - if it's a serial application, how can we take advantage of a large HPC system?
- Can aggregate the tasks into a single job and use multiple processors
  - can do this yourself, or
  - leverage existing tools
- Stampede has two parametric job launchers that can be used to aggregate serial workloads:
  - $ module help launcher
  - $ module help pylauncher
- We will look at a quick "launcher" example
  - very simple to use
  - requires you to create an input file which defines the tasks to be performed (normally running an executable with a different input)
  - tasks will be run independently
  - be careful managing the input/output of each task (ie. don't allow race conditions)

SLURM/LSF/SGE Batch Systems

SLURM	SGE	LSF
sbatch	*qsub*	*bsub*
squeue	*qstat*	*bjobs*
scontrol hold	*qhold*	*bstop*
scontrol release	*qrls*	*bresume*
scancel	*qdel*	*bkill*

File System Access & Lifetime Table (Stampede)

File System	User Quotas	Life Time	Target Usage
$HOME	***5 GB 150K inodes***	***Project***	Permanent user storage; automatically backed up.
$WORK	***400 GB 3M inodes***	***Project***	Large allocated storage; not backed up.
$SCRATCH	***none***	***10 Days***	Large temporary storage; not backed up, purged periodically.

File Systems Build-Out

Logical Volume	Raw Capacity	Target Usage
$HOME – 4 servers	768 TB	Permanent user storage; automatically backed up, quota enforced
$WORK – 8 servers	1.5 PB	Large allocated storage; not backed up,
$SCRATCH – 58 servers	~11 PB	Large temporary storage; not backed up, purged periodically

Part 3: Compilers

Compilers

- Compilers: What are they good for?
  - what do compilers do, why do we need / want them?
  - what's the difference between interpreted vs compiled languages?

Topic of the Day - Compilers

- Compilers (and assemblers and linkers) turn human-readable programming languages in to machine-executable programs
- As computational scientists, you don't necessarily need to read/write assembly code
- But, you do need to be intimately familiar with compiler front ends, and the various options which control optimization and floating-point accuracy
- Thou shalt consult man pages for the compiler you are using when moving to a new system (and with each new processor architecture)

```
        movl    8(%ebp), %ecx
        movl    $320, %ebx
.L2:
        movl    %ecx, %eax
        movl    $0, %edx
.L3:
        movl    $0, (%eax)
        addl    $1, %edx
        addl    $4, %eax
        cmpl    $200, %edx
        jne     .L3
        addl    $800, %ecx
        subl    $1, %ebx
        jne     .L2
```

Compilers

- Scientific computing tends towards three primary languages:
  - FORTRAN (and its dialects)
    - predates UNIX
    - designed so that compiler can know easily what optimizations it can do
    - and yes, it has multi-dimensional arrays
  - C
    - UNIX/Linux is written in C
    - supercomputers run UNIX
    - has pointers which can be manipulated to provide syntax which looks like multi-dimensional arrays
  - C++
    - We are slowly seeing more C++ usage, particularly in the design of scientific libraries
    - wide array of support tools available
      - Standard Template Library (STL)
      - Boost (www.boost.org)

What should you use?

- be good with at least one of them
- remember, it's easy to write bad code in any language
- high-performance applications can be achieved with any of the above

Compiler Availability

- OS Defaults
  - Linux: GNU
  - AIX: XL
  - OSX: Apple modified GNU
  - Solaris: Oracle Studio
- Vendor
  - Intel: Intel
  - IBM: XL
  - Sun: Sun Studio
- 3rd Party (x86)
  - Portland Group
  - Pathscale

Popular Compiler Families

- GCC: GNU Compiler Collection
  - Supported languages: C, C++, Objective C, Fortran77/95/2003/2008, Java, Ada
  - Highly portable (available for most architectures and some toasters)
  - Free Software
- Intel
  - Supported languages: C, C++, Fortran77/95/2003/2008
  - Available for Linux and Windows for x86 architectures, optimized for Intel chips, works good on all x86_64 hardware (e.g. AMD)
  - Free for non-commercial use on Linux
  - Also available for Mac OS for a price
- Portland Group (recently acquired by NVIDIA)
  - Supported languages: C, C++, Fortran77/95/2003/2008, HPF
  - Available for x86 (Linux and Windows)
  - Includes some accelerator support (e.g. using Fortran on a GPU)
- IBM XL
  - Supported languages: C, C++, Fortran77/95/2003/2008
  - Available for Power and BlueGene systems (AIX and Linux)

Compilers and Linkers

- Traditionally, cc was the compiler and ld was the linker on UNIX systems
- Compiler produced object files
  - one source file --> one object file
  - contains machine code
  - but not executable independently
- These days the compiler knows how to call the linker for you, but it is important to understand the different roles of the compiler and linker

Compilers and Linkers

- Every object file contains
  - executable machine code
  - symbol table
    - functions
    - variables
    - types
- The linker coordinates symbols amongst object files (and system libraries) to create the executable
  - usually no symbol table in the result
    - but, debuggers need the symbols to map variables/functions names from machine code to something you might recognize
    - compiler/linker flags to add them to the executable

Invocation

These are the compiler front-end binaries most commonly encountered for C, C++, and Fortran.

On Stampede, you can use the GNU family or Intel (Intel will be loaded by default)

GNU	gcc, g++, gfortran
Intel	icc, icpc, ifort
IBM XL	xlc, xlC, xlf, xlf95
Portland Group	pgcc, pgCC, pgf77, pgf90, pgf95
Pathscale	pathcc, pathCC, pathf77, pathf95

Invocation

Assumes that you have set CC variable to your C compiler of choice (e.g. export CC=gcc)

- Most basic form:
  - $CC foo.c
  - Creates the executable a.out
  - Links in default libraries only (libc for certain, others vary by compiler/architecture/OS)
- Extra option to name the output:
  - $CC –o foo foo.c
  - creates executable foo
- Option to compile only (no external linking):
  - $CC –c foo.c
  - creates object file foo.o

Invocation

- Compile only, name the output
    - $CC –o bar.o –c foo.c
- Add debugging symbols (important)
    - $CC –g foo.c –o foo

Invocation – include file

- C/C++ and Fortran all have mechanisms to include external source code within a file
  - e.g. #include <stdio.h> in C

- Use "-I" to tell the compiler additional search paths for resolving include paths; often required library linkage as well:

  $CC –I/usr/include –c foo.c

- Use environment variables where possible (e.g. from modules on Stampede)

- The INCLUDE environment variable also influences the compiler search path

Static vs. Dynamic Linking

- Static
  - puts all the external routines into the created executable
  - no dependencies at run time (for static libraries)
  - leads to larger binaries
  - takes longer to build the executable
- Dynamic
  - leaves the routines in the library file
  - loads the routines at run time
  - decreases the the build time and binary size
  - dynamic libraries can be shared by different executables in memory!

Static vs. Dynamic Libraries

- Static
    - usually called ‘libfoo.a’
    - created as an archive of object files
    - no special options needed when building
- Dynamic
    - usually called ‘libfoo.so’
    - more complicated to build than static libraries

Invocation

- Multiple source files
  - $CC foo.c bar.c baz.c –o foo
- Link multiple object files into one executable
  - $CC foo.o fun1.o fun2.o –o foo
- Link in a library by hand (static)
  - $CC foo.c –o foo /home/user/mylib/libmylib.a
  - Works just like object file linking

Library linking

- Link in a library
  - $CC foo.c –o foo –lm
  - looks for libm.a or libm.so in the compiler/linker search path
  - most compilers choose *.so over *.a (unless told otherwise)
  - /lib64 and /usr/lib64 + compiler/linker internal directories included in the default search path
- Link a library in a non-standard path
  - $CC foo.c –o foo –L/home/user/mylib/ –lmylib
  - adds /home/user/mylib/path to the library search path (at link time, more on run time later)
  - looks for libmylib.a or libmylib.so in the search path

Forcing Static Linking

- Dynamic linking preferred on most systems when both ‘libfoo.a’ and ‘libfoo.so’ are available
  - the in memory sharing can be a big win for certain libraries that everyone uses
  - Q: can you think of another reason why dynamic libraries are attractive?
- Most compilers can be forced to link statically
  - GNU and Intel: -static
- Sometimes a static version of the library isn’t available and using –static will cause a error
  - use the “by hand” linking method in these cases

Finding Dynamic Libraries

- At run time, the Linux Loader (ld.so(8)) tries to resolve the shared library dependencies of an executable before it runs it
- It looks in:
  - paths listed in its configuration file: /etc/ld.so.conf
  - LD_LIBRARY_PATH in your environment
    - a colon-separated list of places to search just like PATH and MANPATH
  - the search path built in to the executable
  - Sneaky: note that OSX uses a different variable (DYLD_LIBRARY_PATH)

Power User: Adding to the Executable's Search Path

- You can add to the search path embedded in the executable
    - $CC –o foo –lmylib –L/home/user/mylib/mylib –Wl,-rpath,/home/user/mylib/mylib
- -Wl used to pass command line arguments directly to the linker
- -rpath linker option to add to the executable's search path

ldd - shared library dependencies

- The ldd command can be used to investigate the shared library dependencies of an executable (or other libraries)

- This should be one of your goto tools when debugging missing symbols at runtime

- lslogin1$ ldd foo

```
libm.so.6 => /lib64/tls/libm.so.6 (0x0000003ee3d00000)
libc.so.6 => /lib64/tls/libc.so.6 (0x0000003ee3a00000)
libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x0000003eeb400000)
libdl.so.2 => /lib64/libdl.so.2 (0x0000003ee3f00000)
/lib64/ld-linux-x86-64.so.2 (0x0000003ee3800000)
```

Examining Object Files

- nm can be used to list the symbols in object files, libraries, and executables
  - default shows static function name symbols only
  - can be used to display debugging and dynamic symbols as well
  - most useful for determining actual names of undefined symbols (for inter-language calls especially)

nm Example

foo.c:

```
#include "bar.h"
int c=3;
int d=4;
int main()
{
  int a=2;
  return(bar(a*c*d));
}
```

bar.c:

```
#include "bar.h"
int bar(int a)
{
  int b=10;
  return(b*a);
}
```

bar.h:

```
int bar(int);
```

nm Example

- Compile
  - gcc -g -c -o foo.o foo.c
  - gcc -g -c -o bar.o bar.c

- Link
  - gcc foo.o bar.o -o foo

nm Example

```
lslogin1$ nm foo.o bar.o
foo.o:
         U bar
00000000 D c
00000004 D d
00000000 T main
bar.o:
00000000 T bar
```

- “U” that the symbol “bar” is unknown in foo.o
- “T” means the symbol is listed in the text section of the object file
- “D” means that the symbol defines the location of global, initialized data

nm

- Useful options
    - -a show all symbols
    - -u show only undefined symbols
- Uppercase letters for global symbols, lowercase for local symbols
- Other codes
    - C, uninitialized data
    - N, debugging symbol
    - R, read-only data

Making Static Libraries

- Common code that is useful between programs or that changes infrequently can be put in a library with ar
- ar is more than an object file archiver
  - you can use it for any kind of files
  - nobody does (well, except for Debian)
  - tar is more common for archiving non-object files

Invoking ar

- ar r libfoo.a foo.o bar.o baz.o
  - creates/adds to libfoo.a
  - inserts foo.o, bar.o, and baz.o
  - overwrites members of the same name
- ar s libfoo.a
  - creates or updates the object-file symbol table libfoo.a
  - may be combined with ‘r’ to do it all at once
  - ar rs libfoo.a foo.o bar.o baz.o
  - ranlib libfoo.a is a synonym
- ar t libfoo.a
  - prints the list of files contained in libfoo.a

Compiler Warnings

- The compiler can warn you about certain constructs that aren't syntactically wrong but may cause strange behavior or run-time errors
- With many compilers –W<foo> turns on warnings about foo
- -Wall turns on many (but not always all) warnings
  - $CC –c –Wall foo.c
- -w to disable warning messages all together

Warning Example

```c
int a=1;
int main()
{
   int a;
   int b=1;
   return (a+b);
}
```

```
lslogin1$ gcc -O1 -c foo.c
-Wuninitialized -Wshadow

foo.c: In function 'main':

foo.c:4: warning: declaration of
'a' shadows a global declaration

foo.c:1: warning: shadowed
declaration is here

foo.c:7: warning: 'a' is used
uninitialized in this function
```

Compiler Optimization

- By default compilers try to
  - reduced compilation time
  - execute code faithfully
  - make debugging make sense
- Compilier optimization can
  - increase compilation time (dramatically)
  - reduce run time
  - increase or decrease executable size
  - change the order of operations
  - eliminate some code completely
  - introduce new code

Invoking the Optimizer

- Basic
  - $CC –O –o foo foo.c
- More
  - $CC –O# -o foo foo.c
  - # usually in the range [1-3]
  - each level inclusive of previous levels
  - optimization levels represent a suite of options which may be enabled / disabled individually

Compiler Optimization

- For GCC, level 1 turns on:
  - -fdefer-pop
  - -fdelayed-branch -fguess-branch-probability
  - -fcprop-registers
  - -floop-optimize
  - -fif-conversion -fif-conversion2
  - -ftree-ccp -ftree-dce -ftree-dominator-opts -ftree-dse -ftree-ter -ftree-lrs -ftree-sra -ftree-copyrename -ftree-fre -ftree-ch
  - -fmerge-constants
  - -fomit-frame-pointer (where it doesn't interfere with debugging)

Compiler Optimization

- -floop-optimize
- Perform loop basic optimizations
  - move constant expressions out of loops
  - simplify exit test conditions
  - do strength-reduction
- Q: what is strength reduction?

Strength Reduction

- Replacement of an expensive calculation with a cheaper one:
  - replace x/2.0 by 0.5*x
  - replace y = x * 2.0 with y = x + x
  - simplify array addressing in loops

Constant Folding and Propagation

- Constant folding
  - Simplify expressions containing multiple constants

  *i = 3*4*5;* becomes *i=60;*
- Constant Propagation
  - replace constant-valued variable references with values

int x = 10; *int y = 5*x;*	$\longrightarrow$	*int x = 10;* *int y = 5*10;*

- Multiple passes may be applied

Optimization Example

```
#define M 8000
#define N 9200
void zero_buf(unsigned int *buf)
{
  unsigned int i, j;
  for (j=0; j<N; ++j)
    for(i=0; i<M; ++i)
      buf[j*M+i]=0;
}
int main()
{
  unsigned int buf[M*N];
  zero_buf(buf);
  return(buf[M-1]);
}
```

Look at assembly with Intel

- You can use the compiler to generate assembly with source code annotations:
- icc -g -O0 -S -fsource-asm example.c
- will generate an “example.s” file in this case
- use your favorite editor to examine
- how might optimization settings affect runtime and the amount of assembly?

Look at assembly with Intel

- Compilation time:
  - Compile time for -O0 = 0.286 secs
  - Compile time for -O1 = 0.295 secs
  - Compile time for -O3 = 0.306 secs
  - this difference will widen drastically with larger codes (we will revisit this with our project codes)

- Runtime:
  - with -O0 = 0.59 secs
  - with -O1 = 0.18 secs
  - with -O3 = 0.15 secs

- Even the most trivial of functions sees almost LARGE improvement with optimization; in this case much of the benefit comes from:
  - using register values for counters (as opposed to memory locations)
  - more efficient array index calculation