Contents

1 Getting Started

Welcome to CCV's user manual!

This manual is primarily a guide for using Oscar, a large compute cluster maintained by CCV for use by Brown researchers.

1.0.1 Conventions

We use angle brackets to denote command-line options that you should replace with an appropriate value. For example, the placeholders `<user>` and `<group>` should be replaced with your own user name and group name.

1.1 Connecting to Oscar

CCV uses the Secure Shell (SSH) protocol for interactive logins and file transfers. SSH is normally available on Linux, MacOS and other Unix-like systems. A free SSH client application for Windows is available on the CIS software download site. We also recommend putty, another free SSH client for Windows.

CCV systems are accessed through a single login portal called `ssh.ccv.brown.edu`. To login to the Oscar system:

```
$ ssh <user>@ssh.ccv.brown.edu
```

and enter your password. You are now connected to either `login001` or `login002`, the frontend nodes for the cluster.

Note: please do not run computations or simulations on the login nodes, because they are shared with other users. You can use the login nodes to compile your code, manage files, and launch jobs on the compute nodes.

1.1.1 Passwords

To change your Oscar login password, use the command:

```
$ yppasswd
```

You will be asked to enter your old password, then your new password twice.

To change your CIFS password, use the command:

```
$ smbpasswd
```

1.1.2 Shells

A ``shell'' is a program that enables you to interact with a Unix system through a command-line interface. You may wish to change your default shell. To do this:

```
$ ypchsh
```

Enter your password, then enter `/bin/bash` (note: CCV only supports the bash shell).

1.2 Linux

Oscar runs the Linux operating system. General Linux documentation is available from The Linux Documentation Project

Once logged into the Oscar frontend, your home directory will contain a ``README'' file with some quick-start information, and a sample batch script file which can be modified for submitting computing jobs to the batch system. You can view the README with:

```
$ more README
```

(press `q` to quit and `space` to scroll).

The preferred method for transferring files to and from the Oscar system is to use either scp or sftp (the secure shell copy and ftp protocols). Alternatively, you may choose to mount your Oscar home directory on your laptop or workstation via CIFS.

1.3 CIFS

CCV users can access their home, data and scratch directories as a local mount on their own Windows, Mac, or Linux system using the Common Internet File System (CIFS) protocol (also called Samba). There are two requirements for using this service:

- An Oscar account with CIFS access enabled (accounts created since 2010 are automatically enabled).
- Local campus connectivity. Off-campus users can connect after obtaining a campus IP with Brown's Virtual Private Network client, but performance may be degraded.

First, use SSH to connect to Oscar to set your CIFS password. Once logged in, run the command:

```
$ smbpasswd
```

You will first be prompted for your ``old'' password, which is the temporary password you were given by CCV when your account was created. Then, enter a new CIFS password twice. You may choose to use the same password here as for your Oscar account.

Now you are ready to mount your CCV directories locally using the following instructions based on your operating system:

1.3.1 Windows XP

- Right-click ``My Computer'' and select ``Map Network Drive''.
- Select an unassigned drive letter.
- Enter `\\\oscarcifs.ccv.brown.edu\<user>` as the Folder.
- Click ``Connect using a different user name''
- Enter your CCV user name as ``ccv\username'' (no quotes)
- Enter your CCV password and click ``OK''.
- Click ``Finish''

You can now access your home directory through Windows Explorer with the assigned drive letter. Your data and scratch directories are available as the subdirectories (`~/data` and `~/scratch`) of your home directory.

1.3.2 Windows 7

- Right-click ``Computer'' and select ``Map Network Drive''.
- Select an unassigned drive letter.
- Enter `\\\oscarcifs.ccv.brown.edu\<user>` as the Folder.

- Check ``Connect using different credentials''
- Click ``Finish''
- Enter your CCV user name as ``ccv\username'' (no quotes)
- Enter your CCV password and click ``OK''.

You can now access your home directory through Windows Explorer with the assigned drive letter. Your data and scratch directories are available as the subdirectories (`~/data` and `~/scratch`) of your home directory.

1.3.3 Mac OS X

- In the Finder, press ``Command + K'' or select ``Connect to Server…'' from the ``Go'' menu.
- For ``Server Address'', enter `smb://oscarcifs.ccv.brown.edu/<user>` and click ``Connect''.
- Enter your username and password.
- You may choose to add your login credentials to your keychain so you will not need to enter this again.

Optional. If you would like to automatically connect to the share at startup:

- Open ``System Preferences'' (leave the Finder window open).
- Go to ``Accounts'' > ``(your account name)''.
- Select ``Login Items''.
- Drag your data share from the ``Finder'' window to the ``Login Items'' window.

1.3.4 Linux

- Install the `cifs-utils` package:

```
CentOS/RHEL:   $ sudo yum install cifs-utils
Ubuntu:        $ sudo apt-get install cifs-utils
```

- Make a directory to mount the share into:

```
$ sudo mkdir /mnt/rdata
```

- Create a credentials file and add your CCV account information:

  ```
  $ sudo gedit /etc/cifspw
  ```

  ```
  username=<user>
  password=<password>
  ```

- Allow only root access to the credentials files:

  ```
  $ sudo chmod 0600 /etc/cifspw
  ```

- Add an entry to the `fstab`:

  ```
  $ sudo gedit /etc/fstab
  ```

  The `fstab` entry is the single line:

  ```
  //oscarcifs.ccv.brown.edu/<user> /mnt/rdata cifs credentials=/etc/cifspw,nounix,uid=<
  0 0
  ```

  Change `<localUser>` to the login used on your Linux workstation.

- Mount the share:

  ```
  $ mount -a
  ```

2 Managing Files

CCV offers a high-performance storage system for research data called **RData**, which is accessible as the `/gpfs/data` file system on all CCV systems. It can also be mounted from any computer on Brown's campus network using CIFS.

You can transfer files to Oscar and RData through a CIFS mount, or by using command-line tools like scp or rsync.

There are also GUI programs for transfering files using the `scp` protocol, like WinSCP for Windows and Fugu or Cyberduck for Mac.

Note: RData is not designed to store confidential data (information about an individual or entity). If you have confidential data that needs to be stored please contact `support@ccv.brown.edu`.

2.1 File systems

CCV uses IBM's General Parallel File System (GPFS) for users' home directories, data storage, scratch/temporary space, and runtime libraries and executables. A separate GPFS file system exists for each of these uses, in order to provide tuned performance. These file systems are mounted as:

~ → /gpfs/home/<user>

Your **home** directory: optimized for many small files (<1MB) nightly backups 10GB quota

~**/data** → /gpfs/data/<group>

Your **data** directory optimized for reading large files (>1MB) nightly backups quota is by group (usually >=256GB)

~**/scratch** → /gpfs/scratch/<user>

Your **scratch** directory: optimized for reading/writing large files (>1MB) NO BACKUPS purging: files older than 30 days may be deleted 512GB quota: contact us to increase on a temporary basis

A good practice is to configure your application to read any initial input data from `~/data` and write all output into `~/scratch`. Then, when the application has finished, move or copy data you would like to save from `~/scratch` to `~/data`.

Note: class or temporary accounts may not have a `~/data` directory!

2.2 Restoring files

Nightly snapshots of the `/gpfs/home` and `/gpfs/data` file systems are available for the trailing seven days. They are available in the `/gpfs/.snapshots` directory:

```
$ ls /gpfs/.snapshots/home/
daily_0  daily_1  daily_2  daily_3  daily_4  daily_5  daily_6

$ ls /gpfs/.snapshots/data/
daily_0  daily_1  daily_2  daily_3  daily_4  daily_5  daily_6
```

The numbers following the directories indicate the day of the week on which the snapshot was created:

```
0 = Sunday
1 = Monday
2 = Tuesday
```

```
3 = Wednesday
4 = Thursday
5 = Friday
6 = Saturday
```

For example, if it is Tuesday, and you would like to find a 4 day old version of a file in your home directory, you could look in the snapshot from the previous Friday at:

```
/gpfs/.snapshots/home/daily_5/<username>/<path_to_file>
```

2.3 Best Practices for I/O

Efficient I/O is essential for good performance in data-intensive applications. Often, the file system is a substantial bottleneck on HPC systems, because CPU and memory technology has improved much more drastically in the last few decades than I/O technology.

Parallel I/O libraries such as MPI-IO, HDF5 and netCDF can help parallelize, aggregate and efficiently manage I/O operations. HDF5 and netCDF also have the benefit of using self-describing binary file formats that support complex data models and provide system portability. However, some simple guidelines can be used for almost any type of I/O on Oscar:

- Try to aggregate small chunks of data into larger reads and writes. For the GPFS file systems, reads and writes in multiples of 512KB provide the highest bandwidth.
- Avoid using ASCII representations of your data. They will usually require much more space to store, and require conversion to/from binary when reading/writing.
- Avoid creating directory hierarchies with thousands or millions of files in a directory. This causes a significant overhead in managing file metadata.

While it may seem convenient to use a directory hierarchy for managing large sets of very small files, this causes severe performance problems due to the large amount of file metadata. A better approach might be to implement the data hierarchy inside a single HDF5 file using HDF5's grouping and dataset mechanisms. This single data file would exhibit better I/O performance and would also be more portable than the directory approach.

2.4 Revision control with Git

Git is a revision control system for tracking and merging changes to documents from multiple authors. It is most commonly used for collaborative software development, but is also useful for other activities, like writing papers or analyzing data, that rely on collaboration or have a need for detailed tracking of changes (for instance, to recover earlier versions).

Academic users are eligible for free Git repository hosting at Bitbucket (see the application form for more information).

Alternatively, you can easily host your own Git repositories in your home or data directory on Oscar by following the tutorial below. The repos you create in this way will be accessible to any other Oscar user who has read or write permission to the directory containing the repo.

2.4.1 Creating an empty repository

To use Git on Oscar, load its software module:

```
$ module load git
```

Choose a directory on Oscar to contain your repositories. If the repo will contain mostly small files, and less than 1GB, a subdirectory like `~/git` or `~/repos` in your home directory is a good candidate. If you will store large files or more than 1GB, consider using `~/data/git` or `~/data/repos` in your data directory instead (see more information on Oscar's file systems).

Once you have created this directory, created a subdirectory using the convention of your project name followed by ``.git''. For example,

```
$ mkdir ~/git
$ mkdir ~/git/myproject.git
```

In the project directory, execute this command to create a bare repository:

```
$ cd ~/git/myproject.git
$ git --bare init
```

Your repository is now ready to be ``cloned'' (or copied) in another location.

2.4.2 Cloning the new repository

When you clone a Git repo, you create a local copy that contains the full revision history. That is, if you were to lose the original, your clone provides a complete backup of all revisions. You can create a local clone in another directory on Oscar using

```
$ git clone ~/git/myproject.git
```

This will create a directory myproject that contains the clone. So if you ran this command from your home directory, you would have a clone in `~/myproject`.

You can also create a clone on another system by using Git in `ssh` mode:

```
$ git clone ssh://ssh.ccv.brown.edu/~/git/myproject.git
```

2.4.3 Importing the initial content

Once you have a clone of the repo, you can begin populating it by moving files into it and adding them with the ``git add'' command. For example:

```
$ cd ~/myproject
$ mv ~/mycode.c .
$ git add mycode.c
```

You can check the status of your repo with ``git status''.

Once you have populated the repo, you need to commit these new additions with:

```
$ git commit -m "initial version"
```

Every commit requires a comment after the `-m` flag, so that you can later identify what changed between versions.

Finally, to synchronize your cloned repo with the original repo, you need to ``push'' your changes. The first time you push, you have to use the command:

```
$ git push origin master
```

This tells Git that the clone you have (which was originally a ``branch'') is actually the master repo.

2.4.4 Keeping repos in sync

Once you have populated your repo, you can synchronize different clones using the push and pull commands. Above, you pushed your initial changes back to the original repo. If you had multiple clones of this repo, you would then need to use ``git pull'' in the other repos to synchronize them.

When you make changes in a clone, you can push them using the add, commit, and push commands like you did when you first populated the repo. If you haven't added any new files, but have modified existing ones, you can use the shortcut command:

```
$ git commit -a -m "comment"
```

to automatically add any updated files to the commit (instead of issuing an add command for each modified file). Also, you can simply do ``git push'' (without ``origin master'') on all subsequent pushes.

3 Software

Many scientific and HPC software packages are already installed on Oscar, and additional packages can be requested by submitting a ticket to `support@ccv.brown.edu`.

CCV cannot, however, supply funding for the purchase of commercial software. This is normally attributed as a direct cost of research, and should be purchased with research funding. CCV can help in identifying other potential users of the software to potentially share the cost of purchase and maintenance. Several commercial software products that are licensed campus-wie at Brown are available on Oscar, however.

3.1 Software modules

CCV uses the Modules system for managing the software environment on OSCAR. The advantage of the modules approach is that it allows multiple versions of the same software to be installed at the same time. With the modules approach, you can ``load'' and''unload'' modules to dynamically control your environment. You can also customize the default environment that is loaded when you login. Simply put the appropriate module commands in the .modules file in your home directory. For instance, if you edited your .modules file to contain

```
module load matlab
```

then the default module for Matlab will be available every time you log in.

3.1.1 Module commands

module **list**

Lists all modules that are currently loaded in your software environment.

module **avail**

Lists all available modules on the system. Note that a module can multiple version numbers: this allows us to maintain legacy versions of software or to try out beta or preview versions without disrupting the stable versions.

module **help** *package*

Prints additional information about the given package.

module **load** *package*

Adds a module to your current environment. It does so silently, unless there is a problem with the modulefile (in which case you should notify support). If you load the generic name of a module, you will get the default version. To load a specific version, load the module using its full name with the version:

```
$ module load gcc/4.7.2
```

module **unload** *package*

Removes a module from your current environment.

3.2 GUI software

You can run GUI software on CCV systems using two different methods:

3.2.1 X Forwarding

If you have an installation of X11 on your local system, you can access Oscar with X forwarding enabled, so that the windows, menus, cursor, etc. of any X applications running on Oscar are all forwarded to your local X11 server. Here are some resources for setting up X11:

- Mac OS X http://developer.apple.com/opensource/tools/x11.html
- Windows http://software.brown.edu/dist/w-exceed2007.html

Once your X11 server is running locally, open a terminal and use

```
$ ssh -Y <user>@ssh.ccv.brown.edu
```

to establish the X forwarding connection. Then, you can launch GUI applications from the Oscar login node and they will be displayed locally on your X11 server.

Note: the login nodes are shared resources and are provided for debugging, programming, and magaing files. Please do not use them for production runs (for example, executing a long-running script in a GUI instance of Matlab). You can use the batch system to submit production runs if your application can be run without a GUI (for example, with `matlab -nodisplay`).

One limitation of X forwarding is its sensitivity to your network connection's latency. We advise against using X forwarding from a connection outside of the Brown campus network, since you will likely experience lag between your actions and their response in the GUI.

3.2.2 Virtual Network Computing (VNC)

CCV offers a VNC service that allows you to access a complete Linux graphical desktop environment. The VNC protocol will perform better than X forwarding when connecting from off-campus locations.

3.3 MATLAB

MATLAB is available as a software module on Oscar. The default version of MATLAB is loaded automatically when you log in.

The command `matlab` is actually a wrapper that sets up MATLAB to run as a single-threaded, command-line program, which is the optimal way to pack multiple MATLAB scripts onto the Oscar compute nodes.

If you will only be running one MATLAB script per compute node, you can instead run MATLAB in threaded mode with:

```
$ matlab-threaded
```

If you would like to run the MATLAB GUI, for instance in an X-forwarded interactive session, you also need to use the `matlab-threaded` command, which enables the display and JVM.

You can find an example batch script for running Matlab on an Oscar compute node in your home directory:

```
~/batch_scripts/matlab_1node.sh
```

Further reading from Mathworks:

- Speeding Up MATLAB Applications
- Profiling for Improving Performance

3.4 Compiling

By default, the `gcc' software module will load when you login, providing the GNU compiler suite of gcc (C), g++ (C++), and gfortran (Fortran 77/90/95). To compile a simple (single source) program, you can use:

```
$ gcc -g -O2 -o myprogram myprogram.c
$ g++ -g -O2 -o myprogram myprogram.cpp
$ gfortran -g -O2 -o myprogram myprogram.f90
```

The `-g` and `-O2` flags tell the compiler to generate debugging symbols and to use a higher level of optimization than the default.

Optionally, you can load the `pgi' module to access the Portland Group compiler suite, including pgcc (C), pgCC (C++), pgf77 (Fortran 77) and pgf90 (Fortran 90), or the `intel' module to access the Intel compiler suite, with icc (C), ifort (Fortran), and icpc (C++).

3.4.1 OpenMP and pthreads

Both the GNU and PGI compilers provide support for threaded parallelism using either the POSIX threads (pthreads) library or the OpenMP programming model.

To link against pthreads, append the -lpthread flag to your compile or link command. For OpenMP, use -fopenmp with the GNU suite, -mp with PGI, or -openmp with Intel.

3.4.2 MPI

The Message Passing Interface is the most commonly used library and run-time environment for building and executing distributed-memory applications on clusters of computers.

We provide the OpenMPI implementation of MPI. The `openmpi' module loads by default, providing the OpenMPI library compilers for C (mpicc), C++ (mpicxx or mpic++), Fortran 77 (mpif77) and 90 (mpif90). These are wrappers for the GNU compilers that add MPI support. Simply use the MPI wrapper in place of the compiler you would normally use, so for instance:

```
$ mpicc -g -O2 -o mpiprogram mpiprogram.c
```

If you would like to use MPI with the PGI or Intel compilers, you can switch to the appropriate version of OpenMPI with a module swap command:

```
$ module swap openmpi openmpi/1.4.3-pgi
$ module swap openmpi openmpi/1.4.3-intel
```

Unfortunately, you cannot load several versions of MPI at the same time, because their environment variables will conflict.

3.5 Linking

To use external libraries in your code, you must link them against your program after compiling your code. Most compiler frontends, such as gcc, can perform both compiling and linking, depending on the flags you use. A simple example of compiling and linking against the pthreads library in a single command is:

```
$ gcc -o myprogram myprogram.c -lpthread
```

The `-o` flag provides the name of the linked executable. The `-l` flag instructs the compiler to search for a library called `pthread' in the typical library paths, such as `/lib` or `/usr/lib`.

3.5.1 Dynamic vs. static linking

Typically the linker will default to dynamic linking, which means the library will remain as a separate file with a .so extension (for `shared object') that must be available when your program executes. You can see what libraries are dynamically linked against your program using

```
$ ldd myprogram
```

In the example with pthreads above, this may result in output like

```
libpthread.so.0 => /lib64/libpthread.so.0 (0x0000003ebb200000)
libc.so.6 => /lib64/libc.so.6 (0x0000003eba600000)
/lib64/ld-linux-x86-64.so.2 (0x0000003eba200000)
```

which shows that the `pthread' library was found by the linker at the path `/lib64/libpthread.so.0` in the previous compile/link command.

In contrast, static linking means the object code from the library will be copied and duplicated inside your program. The advantage of this approach is that calls to functions within the library may have less overhead (because they do not pass through the dynamic loader) and the executable is self-contained and does not require any external .so files at runtime, which can make it more portable.

To link the pthread example statically, you would use

```
$ gcc -o myprogram myprogram.c -lpthread -static
```

This compiles and links `myprogram' as a static executable, so that using ldd will generate the error

```
not a dynamic executable
```

Instead, you can use the nm tool to inspect what symbol names (functions, global variables, etc.) are contained in a static executable. This tool also works on static libraries, which have the extension .a. For instance, using nm on a statically compiled executable for R (the statistics package/language) outputs

```
...
00000000005ac970 T BZ2_bzclose
00000000005ac8c0 T BZ2_bzdopen
00000000005aca10 T BZ2_bzerror
00000000005ac960 T BZ2_bzflush
00000000005ac830 T BZ2_bzlibVersion
00000000005ac8b0 T BZ2_bzopen
00000000005ac8d0 T BZ2_bzread
00000000005ac920 T BZ2_bzwrite
0000000000614390 T BZ2_compressBlock
...
```

which shows that it was linked statically against the `bzip2' compression library. You can also link a dynamic executable against a static library (e.g., if there is no dynamic version of the library available), but this usually requires that the static library was compiled with `position independent code' (or PIC) using the flag `-fPIC`. Most of the static libraries available on CCV systems have been compiled with this flag.

3.5.2 Finding libraries

Many frequently used libraries are available already compiled and installed on CCV systems through software modules. These modules frequently have an environment variable that provides a shortcut for the include and library flags. You can use:

```
$ module help <package>
```

to find out more. For example, the help for the `gotoblas2` module explains how to use the `$GOTO` shortcut:

```
You can compile and link statically against GotoBLAS2 and
the included LAPACK library using the $GOTO shortcut:
  CC -o blas-app blas-app.c $GOTO
Environment Variables:
    GOTO = -I/gpfs/runtime/opt/gotoblas2/1.13/include
           -L/gpfs/runtime/opt/gotoblas2/1.13/lib
           -lgoto2 -lpthread -lgfortran
GOTO_DIR = /gpfs/runtime/opt/gotoblas2/1.13
```

4 Running Jobs

A ``job'' refers to a program running on the compute nodes of the Oscar cluster. Jobs can be run on Oscar in two different ways:

- An **interactive** job allows you to interact with a program by typing input, using a GUI, etc. But if your connection is interrupted, the job will abort. These are best for small, short-running jobs where you need to test out a program, or where you need to use the program's GUI.
- A **batch** job allows you to submit a script that tells the cluster how to run your program. Your program can run for long periods of time in the background, so you don't need to be connected to Oscar. The output of your program is continuously written to an output file that you can view both during and after your program runs.

Jobs are scheduled to run on the cluster according to your account priority and the resources you request (e.g. cores, memory and runtime). When you submit a job, it is placed in a queue where it waits until the required computes nodes become available.

NOTE: please do not run CPU-intense of long-running programs directly on the login nodes! The login nodes are shared by many users, and you will interrupt other users' work.

Previously, Oscar used Torque/Moab for scheduling and managing jobs. With the migration to a new cluster environment, we are now using the Simple Linux Utility for Resource Management (SLURM) from Lawrence Livermore National Laboratory.

Although the two systems share many similarities, the key benefit of moving to SLURM is that we can treat a **single core** as the basic unit of allocation instead of an **entire node**. As a result, we can more efficiently schedule the diverse types of programs that Brown researchers run on Oscar.

With SLURM, jobs that only need part of a node can share the node with other jobs (this is called ``job packing''). When your program runs through SLURM,

it runs in its own container, similar to a virtual machine, that isolates it from the other jobs running on the same node. By default, this container has 1 core and a portion of the node's memory.

The following two sections have more details on how to run interactive and batch jobs through SLURM, and how to request more resources (either more cores or more memory).

4.1 Interactive jobs

To start an interactive session for running serial or threaded programs on an Oscar compute node, simply run the command `interact` from the login node:

```
$ interact
```

By default, this will create an interactive session that reserves 1 core, 4GB of memory, and 30 minutes of runtime.

You can change these default limits with the following command line options:

```
usage: interact [-n cores] [-t walltime] [-m memory] [-q queue]
                [-o outfile] [-X] [-f featurelist] [-h hostname]

Starts an interactive job by wrapping the SLURM 'salloc' and 'srun' commands.

options:
  -n cores        (default: 1)
  -t walltime     as hh:mm:ss (default: 30:00)
  -m memory       as #[k|m|g] (default: 4g)
  -q queue        (default: 'timeshare')
 -o outfile    save a copy of the session's output to outfile (default: off)
  -X              enable X forwarding (default: no)
  -f featurelist  CCV-defined node features (e.g., 'e5-2600'),
                  combined with '&' and '|' (default: none)
  -h hostname     only run on the specific node 'hostname'
                  (default: none, use any available node)
```

4.1.1 MPI programs

To run an MPI program interactively, first create an allocation from the login nodes using the `salloc` command:

```
$ salloc -N <# nodes> -n <# MPI tasks> -p <partition> -t <minutes>
```

Once the allocation is fulfilled, it will place you in a new shell where you can run MPI programs with the `srun` command:

```
$ srun ./my-mpi-program ...
```

Calling `srun` without any parameters will use all the available MPI tasks in the allocation. Alternatively, you can use a subset of the allocation by specifying the number of nodes and tasks explicitly with:

```
$ srun -N <# nodes> -n <# MPI tasks> ./my-mpi-program ...
```

When you are finished running MPI commands, you can release the allocation by exiting the shell:

```
$ exit
```

Alternatively, if you only need to run a single MPI program, you can skip the `salloc` command and specify the resources in a single `srun` command:

```
$ srun -N <# nodes> -n <# MPI tasks> -p <partition> -t <minutes> ./my-mpi-program
```

This will create the allocation, run the MPI program, and release the allocation.

4.2 Batch jobs

To run a batch job on the Oscar cluster, you first have to write a script that describes what resources you need and how your program will run. Example batch scripts are available in your home directory on Oscar, in the directory:

```
~/batch_scripts
```

To submit a batch job to the queue, use the `sbatch` command:

```
$ sbatch <jobscript>
```

This command will return a number, which is your job ID. You can view the output of your job in the file `slurm-<jobid>.out` in the directory where you ran the `sbatch` command. For instance, you can view the last 10 lines of output with:

```
$ tail -10 slurm-<jobid>.out
```

4.2.1 Batch scripts

A batch script starts by specifing the `bash` shell as its interpreter, with the line:

```
#!/bin/bash
```

Next, a series of lines starting with `#SBATCH` define the resources you need, for example:

```
#SBATCH -n 4
#SBATCH -t 1:00:00
#SBATCH --mem=16G
```

The above lines request 4 cores (`-n`), an hour of runtime (`-t`), and a total of 16GB memory for all cores (`--mem`). By default, a batch job will reserve 1 core and a proportional amount of memory on a single node.

Alternatively, you could set the resources as command-line options to `sbatch`:

```
$ sbatch -n 4 -t 1:00:00 --mem=16G <jobscript>
```

The command-line options will override the resources specified in the script, so this is a handy way to reuse an existing batch script when you just want to change a few of the resource values.

Useful `sbatch` options:

`-J`

Specify the job name that will be displayed when listing the job.

`-n`

Number of cores.

`-t`

Runtime, as HH:MM:SS.

`--mem=`

Number of cores.

`-p`

Request a specific partition.

`-C`

Add a feature constraint (a tag that describes a type of node). You can view the available features on Oscar with the `nodes` command.

`--mail-type=`

Specify the events that you should be notified of by email: BEGIN, END, FAIL, REQUEUE, and ALL.

You can read the full list of options with:

```
$ man sbatch
```

4.3 Managing jobs

4.3.1 Canceling a job

```
$ scancel <jobid>
```

4.3.2 Listing running and queued jobs

The `squeue` command will list all jobs scheduled in the cluster. We have also written wrappers for `squeue` on Oscar that you may find more convenient:

`myq`

List only your own jobs.

`myq\ <user>`

List another user's jobs.

`allq`

List all jobs, but organized by partition, and a summary of the nodes in use in the partition.

`allq\ <partition>`

List all jobs in a single partition.

4.3.3 Listing completed jobs

The `sacct` command will list all of your running, queued and completed jobs since midnight of the previous day. To pick an earlier start date, specify it with the `-S` option:

```
$ sacct -S 2012-01-01
```

To find out more information about a specific job, such as its exit status or the amount of runtime or memory it used, specify the `-l` (``long'' format) and `-j` options with the job ID:

```
$ sacct -lj <jobid>
```

4.4 Partitions

When submitting a job to the Oscar compute cluster, you can choose different partitions depending on the nature of your job. You can specify one of the partitions listed below either in your `sbatch` command:

```
$ sbatch -p <partition> <batch_script>
```

or as an `SBATCH` option at the top of your batch script:

```
#SBATCH -p <partition>
```

Partitions available on Oscar:

default

Default partition with most of the compute nodes: 8-, 12-, or 16-core; 24GB to 48GB of memory; all Intel based.

gpu

Specialized compute nodes (8-core, 24GB, Intel) each with 2 NVIDIA GPU accelerators.

debug

Dedicated nodes for fast turn-around, but with a short time limit of 40 node-minutes.

You can view a list of all the Oscar compute nodes broken down by partition with the command:

```
$ nodes -v
```

4.5 Job priority

The scheduler considers many factors when determining the run order of jobs in the queue. These include the:

- size of the job;
- requested walltime;
- amount of resources you have used recently (e.g., ``fair sharing'');
- priority of your account type.

The account priority has three tiers:

- Low (Exploratory)
- Medium (Premium)
- High (Condo)

Both Exploratory and Premium accounts can be affiliated with a Condo, and the Condo priority only applies to a portion of the cluster equivalent in size to the Condo. Once the Condo affiliates have requested more nodes than available in the Condo, their priority drops down to either medium or low, depending on whether they are a Premium or Exploratory account.

4.5.1 Backfilling

When a large or long-running job is near the top of the queue, the scheduler begins reserving nodes for it. If you queue a smaller job with a walltime shorter than the time required for the scheduler to finish reserving resources, the scheduler can backfill the reserved resources with your job to better utilize the system. Here is an example:

- User1 has a 64-node job with a 24 hour walltime waiting at the top of the queue.
- The scheduler can't reserve all 64 nodes until other currently running jobs finish, but it has already reserved 38 nodes and will need another 10 hours to reserve the final 26 nodes.
- User2 submits a 16-node job with an 8 hour walltime, which is backfilled into the pool of 38 reserved nodes and runs immediately.

By requesting a shorter walltime for your job, you increase its chances of being backfilled and running sooner. In general, the more accurately you can predict the walltime, the sooner your job will run and the better the system will be utilized for all users.

4.6 Condo priority

Users who are affiliated with a Condo group will automatically use that Condo's priority when submitting jobs with `sbatch`.

Users who are Condo members and also have Premium accounts will by default use their Premium priority when submitting jobs. This is because the core limit for a Premium account is per user, while the limit for a Condo is per group. Submitting jobs under the Premium account therefore leaves more cores available to the Condo group.

Since Premium accounts have slightly lower priority, a user in this situation may want to instead use the Condo priority. This can be accomplished with the `--qos` option, which stands for ``Quality of Service'' (the mechanism in SLURM that CCV uses to assign queue priority).

Condo QOS names are typically `<groupname>-condo`, and you can view a full list with the `condos` command on Oscar. The command to submit a job with Condo priority is:

```
$ sbatch --qos=<groupname>-condo ...
```

Alternatively, you could place the following line in your batch script:

```
#SBATCH --qos=<groupname>-condo
```

To be pendantic, you can also select the priority QOS with:

```
$ sbatch --qos=pri-<username> ...
```

although this is unnecessary, since it is the default QOS for all Premium accounts.

4.7 Job arrays

A job array is a collection of jobs that all run the same program, but on different values of a parameter. It is very useful for running parameter sweeps, since you don't have to write a separate batch script for each parameter setting.

To use a job array, add the option:

```
#SARRAY --range=<range-spec>
```

below the `#SBATCH` options in your batch script. The range spec can be a comma separated list of integers, along with ranges separated by a dash. For example:

```
1-20
1-10,12,14,16-20
```

The values in the range will be substituted for the variable `$SLURM_ARRAYID` in the remainder of the script. Here is an example of a script for running a serial Matlab script on 16 different parameters:

```
#!/bin/bash
#SBATCH -J MATLAB
#SBATCH -t 1:00:00
#SARRAY --range=1-16

echo "Starting job $SLURM_ARRAYID on $HOSTNAME"
matlab -r "MyMatlabFunction($SLURM_ARRAYID); quit;"
```

After you have modified your batch script to use job arrays, you must submit it using the `sarray` command instead of `sbatch`:

```
$ sarray <jobscript>
```

This will return a list of job IDs, with one job ID per parameter value. The parameter value will also be appended to the job name (`-J` value).

5 XSEDE

XSEDE is an NSF-funded, nation-wide collection of supercompting systems that are available to researchers through merit-based allocations. It replaces what used to be called the TeraGrid.

Brown is a member of the XSEDE Campus Champions program, which provides startup allocations on XSEDE resources for interested researchers at Brown. If you would like help getting started with XSEDE and obtaining a startup allocation, please contant `support@ccv.brown.edu`.

5.1 Connecting to XSEDE

Once you have created an XSEDE Portal account and have been added to an allocation on an XSEDE system, you can connect to that machine use the Globus Toolkit. You can install this locally on your workstation or laptop using these instructions, or you can login to Oscar where the toolkit is avaiable in the `xsede' module:

```
$ module load xsede
$ myproxy-logon -l <XSEDE username>
```

After running `myproxy-logon`, you will have a certificate checked out to the `/tmp` directory on that node, and you can connect to XSEDE systems using the `gsissh` command For exampe, to connect to the system `blacklight` at PSC, use:

```
$ gsissh blacklight.psc.xsede.org
```

You do not have to enter another password because your Globus certificate automatically handles the authentication.

There is also a `gsiscp` command for copying files to and from XSEDE systems.

6 GPU Computing

Oscar has 44 GPU nodes that are regular compute nodes with two NVIDIA Tesla M2050 GPUs (*Fermi* architecture) added. Each M2050 GPU has 448 CUDA cores and 3GB GDDR5 memory. To gain access to these nodes, please submit a support ticket and ask to be added to the `gpu' group.

6.0.1 Interactive Use

To start an interactive session on a GPU node, use the `interact` command and specify the `gpu` partition:

```
$ interact -q gpu
```

6.0.2 GPU Queue

For production runs with exclusive access to GPU nodes, please submit a batch job to the `gpu` partition using:

```
$ sbatch -q gpu <jobscript>
```

You can view the status of the `gpu` partitionwith:

```
$ allq gpu
```

6.1 Getting started with GPUs

While you can program GPUs directly with CUDA, a language and runtime library from NVIDIA, this can be daunting for programmers who do not have experience with C or with the details of computer architecture.

You may find the easiest way to tap the computation power of GPUs is to link your existing CPU program against numerical libraries that target the GPU:

- CUBLAS is a drop-in replacement for BLAS libraries that runs BLAS routines on the GPU instead of the CPU.
- CULA is a similar library for LAPACK routines.
- CUFFT, CUSPARSE, and CURAND provide FFT, sparse matrix, and randon number generation routines that run on the GPU.
- MAGMA combines custom GPU kernels, CUBLAS, and a CPU BLAS library to use both the GPU and CPU to simaultaneously use both the GPU and CPU; it is available in the `magma' module on Oscar.
- Matlab has a GPUArray feature, available through the Parallel Computing Toolkit, for creating arrays on the GPU and operating on them with many built-in Matlab functions. The PCT toolkit is licensed by CIS and is available to any Matlab session running on Oscar or workstations on the Brown campus network.
- PyCUDA is an interface to CUDA from Python. It also has a GPUArray feature and is available in the `cuda` module on Oscar.

6.2 Introduction to CUDA

CUDA is an extension of the C language, as well as a runtime library, to facilitate general-purpose programming of NVIDIA GPUs. If you already program in C, you will probably find the syntax of CUDA programs familiar. If you are more comfortable with C++, you may consider instead using the higher-level Thrust library, which resembles the Standard Template Library and is included with CUDA.

In either case, you will probably find that because of the differences between GPU and CPU architectures, there are several new concepts you will encounter that do not arise when programming serial or threaded programs for CPUs. These are mainly to do with how CUDA uses threads and how memory is arranged on the GPU, both described in more detail below.

There are several useful documents from NVIDIA that you will want to consult as you become more proficient with CUDA:

- CUDA C Programming Guide
- CUDA C Best Practices Guide
- CUDA Runtime API

There are also many CUDA tutorials available online:

- CUDA Training from NVIDIA

- CUDA, Supercomputing for the Masses from Dr. Dobb's
- CUDA Tutorial from The Supercomputing Blog

6.2.1 Threads in CUDA

CUDA uses a data-parallel programming model, which allows you to program at the level of what operations an individual thread performs on the data that it owns. This model works best for problems that can be expressed as a few operations that all threads apply in parallel to an array of data. CUDA allows you to define a thread-level function, then execute this function by mapping threads to the elements of your data array.

A thread-level function in CUDA is called a **kernel**. To launch a kernel on the GPU, you must specify a **grid**, and a decomposition of the grid into smaller **thread blocks**. A thread block usually has around 32 to 512 threads, and the grid may have many thread blocks totalling thousands of threads. The GPU uses this high thread count to help it hide the latency of memory references, which can take 100s of clock cycles.

Conceptually, it can be useful to map the grid onto the data you are processing in some meaningful way. For instance, if you have a 2D image, you can create a 2D grid where each thread in the grid corresponds to a pixel in the image. For example, you may have a 512x512 pixel image, on which you impose a grid of 512x512 threads that are subdivided into thread blocks with 8x8 threads each, for a total of 64x64 thread blocks. If your data does not allow for a clean mapping like this, you can always use a flat 1D array for the grid.

The CUDA runtime dynamically schedules the thread blocks to run on the **multiprocessors** of the GPU. The M2050 GPUs available on Oscar each have 14 multiprocessors. By adjusting the size of the thread block, you can control how much work is done concurrently on each multiprocessor.

6.2.2 Memory on the GPU

The GPU has a separate memory subsystem from the CPU. The M2050 GPUs have GDDR5 memory, which is a higher bandwidth memory than the DDR2 or DDR3 memory used by the CPU. The M2050 can deliver a peak memory bandwidth of almost 150 GB/sec, while a multi-core Nehalem CPU is limited to more like 25 GB/sec.

The trade-off is that there is usually less memory available on a GPU. For instance, on the Oscar GPU nodes, each M2050 has only 3 GB of memory shared by 14 multiprocessors (219 MB per multiprocessor), while the dual quad-core Nehalem CPUs have 24 GB shared by 8 cores (3 GB per core).

Another bottleneck is transferring data between the GPU and CPU, which happens over the PCI Express bus. For a CUDA program that must process

a large dataset residing in CPU memory, it may take longer to transfer that data to the GPU than to perform the actual computation. The GPU offers the largest benefit over the CPU for programs where the input data is small, or there is a large amount of computation relative to the size of the input data.

CUDA kernels can access memory from three different locations with very different latencies: global GDDR5 memory (100s of cycles), shared memory (1-2 cycles), and constant memory (1 cycle). Global memory is available to all threads across all thread blocks, and can be transferred to and from CPU memory. Shared memory can only be shared by threads within a thread block and is only accessible on the GPU. Constant memory is accessible to all threads and the CPU, but is limited in size (64KB).

6.3 Compiling with CUDA

To compile a CUDA program on Oscar, first load the CUDA module with:

```
$ module load cuda
```

The CUDA compiler is called `nvcc`, and for compiling a simple CUDA program it uses syntax simlar to `gcc`:

```
$ nvcc -o program source.cu
```

6.3.1 Optimizations for Fermi

The Oscar GPU nodes feature NVIDIA M2050 cards with the Fermi architecture, which supports CUDA's ``compute capability'' 2.0. To fully utilize the hardware optimizations available in this architecture, add the `-arch=sm_20` flag to your compile line:

```
$ nvcc -arch=sm_20 -o program source.cu
```

This means that the resulting executable will not be backwards-compatible with earlier GPU architectures, but this should not be a problem since CCV nodes only use the M2050.

6.3.2 Memory caching

The Fermi architecture has two levels of memory cache similar to the L1 and L2 caches of a CPU. The 768KB L2 cache is shared by all multiprocessors, while the L1 cache by default uses only 16KB of the available 64KB shared memory on each multiprocessor.

You can increase the amount of L1 cache to 48KB at compile time by adding the flags `-Xptxas -dlcm=ca` to your compile line:

```
$ nvcc -Xptxas -dlcm=ca -o program source.cu
```

If your kernel primarily accesses global memory and uses less than 16KB of shared memory, you may see a benefit by increasing the L1 cache size.

If your kernel has a simple memory access pattern, you may have better results by explicitly caching global memory into shared memory from within your kernel. You can turn off the L1 cache using the flags `--Xptxas --dlcm=cg`.

6.4 Mixing MPI and CUDA

Mixing MPI (C) and CUDA (C++) code requires some care during linking because of differences between the C and C++ calling conventions and runtimes. A helpful overview of the issues can be found at How to Mix C and C++.

One option is to compile and link all source files with a C++ compiler, which will enforce additional restrictions on C code. Alternatively, if you wish to compile your MPI/C code with a C compiler and call CUDA kernels from within an MPI task, you can wrap the appropriate CUDA-compiled functions with the `extern` keyword, as in the following example.

These two source files can be compiled and linked with both a C and C++ compiler into a single executable on Oscar using:

```
$ module load mvapich2 cuda
$ mpicc -c main.c -o main.o
$ nvcc -c multiply.cu -o multiply.o
$ mpicc main.o multiply.o -lcudart
```

The CUDA/C++ compiler `nvcc` is used only to compile the CUDA source file, and the MPI C compiler `mpicc` is user to compile the C code and to perform the linking.

```
01. /* multiply.cu */
02.
03. #include <cuda.h>
04. #include <cuda_runtime.h>
05.
06. __global__ void __multiply__ (const float *a, float *b)
07. {
08. const int i = threadIdx.x + blockIdx.x * blockDim.x;
09.     b[i] *= a[i];
```

```
10. }
11.
12. extern "C" void launch_multiply(const float *a, const *b)
13. {
14.     /* ... load CPU data into GPU buffers a_gpu and b_gpu */
15.
16.    __multiply__ <<< ...block configuration... >>> (a_gpu, b_gpu);
17.
18.     safecall(cudaThreadSynchronize());
19.     safecall(cudaGetLastError());
20.
21.     /* ... transfer data from GPU to CPU */
```

Note the use of `extern "C"` around the function `launch_multiply`, which instructs the C++ compiler (`nvcc` in this case) to make that function callable from the C runtime. The following C code shows how the function could be called from an MPI task.

```
01. /* main.c */
02.
03. #include <mpi.h>
04.
05. void launch_multiply(const float *a, float *b);
06.
07. int main (int argc, char **argv)
08. {
09.        int rank, nprocs;
10.     MPI_Init (&argc, &argv);
11.     MPI_Comm_rank (MPI_COMM_WORLD, &rank);
12.     MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
13.
14.     /* ... prepare arrays a and b */
15.
16.     launch_multiply (a, b);
17.
18.     MPI_Finalize();
19.        return 1;
20. }
```

6.5 MATLAB

GPU Programming in Matlab