Reading Campus Grid
User guide

This document seeks to give an introduction to using the Campus Grid at Reading. It also provides a brief introduction to the use of Linux.

Version History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Author</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 February 2009</td>
<td>1.0</td>
<td>David Spence</td>
<td>Initial Version</td>
</tr>
<tr>
<td>24 February 2009</td>
<td>1.0.1</td>
<td>David Spence</td>
<td>Added Graphical R (and Cygwin to support)</td>
</tr>
<tr>
<td>05 March 2009</td>
<td>1.0.2</td>
<td>David Spence</td>
<td>Added further tutorial links</td>
</tr>
<tr>
<td>10 March 2009</td>
<td>1.1</td>
<td>David Spence</td>
<td>Added SSHFS documentation</td>
</tr>
<tr>
<td>24 June 2009</td>
<td>1.2</td>
<td>David Spence</td>
<td>Added BLCR/Scratch related things</td>
</tr>
<tr>
<td>13 October 2009</td>
<td>1.3</td>
<td>David Spence</td>
<td>Change order of items</td>
</tr>
<tr>
<td>20 October 2009</td>
<td>1.3.1</td>
<td>David Spence</td>
<td>Notes on Random Numbers</td>
</tr>
</tbody>
</table>

Success with the Campus Grid?

If you have had obtained useful results with the Campus Grid then please do tell us so that we can add your project to the list of successful applications of the Campus Grid. This helps us to tell new users what has already been achieved and report to the University that the Campus Grid is supporting useful research.

Contents

Success with the Campus Grid? 1
Introduction to the Campus Grid 3
Applying for access 5
Logging on to the Campus Grid 5
Transferring Files 6
Graphical Access 8

**Introduction to Linux** 10
Basic introduction to the Linux “shell” 10
Editing text files 12
Downloading files from the Internet 12
Advanced: SSHFS 12
Tutorials 13

**Introduction to Condor** 14
Submitting one job 14
Submitting many jobs 15
Long Running Jobs 16
Useful Condor Commands 16
Tutorials & More Information 19

**Introduction to Applications** 21
Important Note on Randomness - Please read 21
Using Matlab 21
Using the R statistical package 24

**Advanced Topics** 26
Compiling code 26
Shared Scratch Space 26
Checkpointing 27

**Introduction to the National Grid Service** 29
Applying for access 29
Differences to the Campus Grid 30
Using Condor to manage NGS jobs 30
Other Commands 31

**Useful Services in Beta** 32
Matlab: Automatic Process 32
R: Automatic Process 33
BLCR Checkpointing 34
Introduction to the Campus Grid

The Reading Campus Grid is a Condor Pool consisting of about 300 nodes. It is provided by IT Services and the School of Systems Engineering. The aim of the Campus Grid is to provide researchers with a High Throughput Computing resource, these are designed to process tasks that require fairly short processing times (usually minutes or hours) but that need to be run 100’s or 1000’s of times (see http://www.cs.wisc.edu/condor/htc.html for more information).

For example; Say you have a process that takes a sample of data from a data-set, analyses the sample and then stores a result. This process takes 30 minutes but need to be done 1000 times. This requires 500 hours of processing time. On a single computer this task will take about 21 days to complete whereas using the Campus Grid (using around 250 nodes) this can be done in 2 hours!

Tasks that take longer than an hour or so can be run on the Campus Grid but may require some alteration to make the most of the way that the Campus Grid works. This normally involves creating check-points of the tasks progress so that it can restart from where it left-off if interrupted (please see Checkpointing).

To give you more of a feel of what the Campus Grid can be used for, the Campus Grid Users page (http://www.reading.ac.uk/internal/its/escience/its-eresearch-casestudies.asp) gives details of some of the work that is already using the Campus Grid.

Definition of Relevant Terms

- Condor - the software which is used to manage the Campus Grid
- Submit Node - the machine which users make use of to set off jobs
- Head/Master Node - the machine which manages the Campus Grid
- Compute/Worker Nodes - the (300 or so) machines which actually process the Campus Grid jobs
- (Command) Shell - the Linux command line.

![Figure 1: The path that jobs take to be run](image-url)
Machine availability
As it stands up to about 300 machines could be available for running your jobs. The actual number of machines will rise and fall during the day as students use the machines, but overnight nearly all are available.

When a user logs on to a machine in person then the machine becomes no longer available to the Campus Grid, and if there is a Campus Grid job running then it will be stopped. After a while Condor will try restarting stopped jobs on another machine.

Therefore, your code needs to either be one that needs many short runs, or if it runs for a longer time needs to make a record of where it got up to so it can restart from where it left off when it gets restarted on another machine.

Memory
Of the machines approximately 15% have 2GB of memory available, 74% have 1Gb, and 11% have 512Mb. If your code needs more than a certain amount of memory then it is important that you include a requirements line in your ClassAd (see Introduction to Condor).

Account and Access
To access the Campus Grid you need an account on the main server. This will have a different username to your normal Reading account. Access can be through ssh and username and password or using the GSI-SSHTerminal and a e-Science Certificate.

This does mean that you'll need to transfer files to and from this account.

Filesystem
All the nodes in the campus Grid have access to the N drive belonging to your Grid account. This has the advantage that whatever node you write a file it will be visible immediately on the other nodes, although it does mean that you will need to be careful to make sure that each different code run writes its results to a different output file.

The Grid account's N drive appears as your home directory on the nodes. It should be noted that sometimes is not the current directory when Condor runs your application. To find out the full path to your N drive/home directory you should log-on to the head node and type `pwd ~` this will return something like `/home/sufs1/ru6/vx/vxx05163`

Initially your Grid account will have a quota of 2Gb, but we realise that for some this will not be enough and so we can extend this up to a maximum 10Gb if this is required.

There is also a directory on each compute node called `/tmp`, this is a shared space of 20Gb which can be used for temporary files. You will need to delete files when you've finished with them and you will not be able to see them from other nodes.

Operating System
Although the worker machines in the Campus Grid natively run Windows XP, CoLinux (http://www.colinux.org/) is used to provide the Linux environment in which Condor runs. This means that users run their Linux executables on the Campus Grid.

The Campus Grid runs the Linux Operating System, on the login node CentOS release 5.2 (fully compatible with Red Hat Enterprise Linux 5) and on the computational nodes Fedora core 8.
Software

For information on the software available on the Campus Grid please see the Campus Grid Software page. And the later sections on Matlab and R.

Applying for access

In order to access the Campus Grid you will need to visit David Spence in IT Services (d.r.spence@reading.ac.uk) to request an account on the Campus Grid.

Logging on to the Campus Grid

Access to the Campus Grid provides what is termed “shell access” (or a command line in Windows parlance). There is also the possibility of opening graphical interfaces to the Campus Grid, although these are of limited use as most of the tools used are based on the command line. On the website we also describe the use of the GSI-SSHTerm to access the Campus Grid - this is more useful when you wish to use the wider National Grid Service.

Windows

Under Windows you will need to install a tool to allow your computer to talk to the Linux based Campus Grid. An easy to use tool is PuTTY, which can be downloaded from http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html (we recommend using the “Windows installer for everything except PuTTYtel”). You may not have permission to install programs, in this case you will need to contact your local IT staff.

Once installed you should be able to run PuTTY from the start menu. A configuration box is displayed like this:

Type “grid.reading.ac.uk” in the two boxes shown then click “Save”. This will save the configuration for the Grid and “grid.reading.ac.uk” should then be displayed in the central scrolling list. Subsequently double-clicking on this name will start the log in process, do this now. It may ask a question about key fingerprints - answer that it is ok to continue. The following window will be displayed:
Type in the username you were given and press return, and the password when prompted. If successful you should have opened a “shell” session, marked by some text like [vxx05163@vicg2 ~]$.

**Unix/Linux**

Under Unix or Linux you do not need to install any software. Simply open a terminal window (available under “Applications”, “Utilities” or the like) and type the following command:

```
$ ssh <username>@grid.reading.ac.uk
```

It may ask whether you want to continue as the “authenticity of the host can’t be established”. Answer yes and type your password when prompted.

**Transferring Files**

**Windows**

In Windows we recommend the WinSCP tool for transferring files to and from the Campus Grid. This can be downloaded from [http://winscp.net/eng/download.php](http://winscp.net/eng/download.php) and we suggest the “Installation Package” is the easiest way to install this. When run you should get a dialog box like:
Type “grid.reading.ac.uk” into the “Host name” box and your username into the “User name” box then click “Save...” and “OK” in the box that pops up. The dialog box will now look like:

Now and in future you can access the files on the Campus Grid by double-clicking on the highlighted item in the middle box. The first time you may be asked about the key of the server and you should say “Yes” and you will also have a message from the server displayed, which you should also dismiss. You will be prompted for the password. The WinSCP main window looks like:

The left side lets you navigate around your local computer and the right around your Campus Grid home directory. Files can be copied simply by dragging from one side to the other.
Unix/Linux

Under Unix/Linux the most straightforward tool for copying files is **sftp**. This is run by typing:

```bash
$ sftp <username>@grid.reading.ac.uk
```

You will then get a prompt like **sftp>**. This allows shell-like commands to list and copy files, if you are unfamiliar with the Linux shell then see the introduction below. The following are useful commands:

- **ls** - list the current directory on the Campus Grid
- **lls** - list the current directory on the local machine
- **cd** - change the current directory on the Campus Grid
- **lcd** - change the current directory on the local machine
- **get** - copy a file from the Campus Grid to the local machine
- **put** - copy a file from the local machine to the Campus Grid
- **quit** - end the session

There is also the **scp** command that allows files to be copied without opening a sftp session. It is used like a normal "**cp**" command - local files are specified in the same way and remote files by prefixing the filename with `<username>@grid.reading.ac.uk`:

```bash
$ scp file vxx05163@grid.reading.ac.uk:remote_file
```

Graphical Access

Windows

You will first need to obtain and install either Exceed or Cygwin. If you are going to use R then please use Cygwin.

- Details of how to install Exceed this under the Universities site license can be found in the Software Table entry for Exceed ([https://www.reading.ac.uk/closed/its/docs/software/its-software-exceed.asp](https://www.reading.ac.uk/closed/its/docs/software/its-software-exceed.asp)).
- To install Cygwin, perform the following steps:
  - Use your web browser to save the file at [http://www.cygwin.com/setup.exe](http://www.cygwin.com/setup.exe)
  - Run this installer by double-clicking on the downloaded file
    - Click Next
    - Check "Install from Internet" and click Next
    - Click Next
    - Change "Local Package Directory" to C:\cygtmp and click Next
    - Check "Direct Connection" and Click Next
    - Select mirror "http://www.mirrorservice.org" and click Next
    - In the "Select Packages" window:
      - Click on the text saying "Default" next to "All" at the top of the main panel, this will change it (after a brief pause) to "Install", click twice more to change to "Uninstall".
• Scroll down the main panel and click on the "Uninstall" next to X11 until it says "Install"
• Click Next
  - Click Next
  - The installer will download and install the packages.
  - Click Finish

Run PuTTY, and in the "PuTTY Configuration" highlight "grid.reading.ac.uk" in the main panel and click “Load”. Then select the "Connection->SSH->X11" page in the white panel on the left - and in this pane select "Enable X11 Forwarding". For example:

![PuTTY Configuration](image)

You can save these settings by returning to the "Session" page and clicking "Save". You can now load the session at anytime by double-clicking this name in the box in the middle.

Then start up Exceed (using Start->Programs->Hummingbird Connectivity 2008->Exceed) or Cygwin-X (using Start->Programs->CygWin->XWin Server). You then should be able to run graphical programs, for example try:

```
$ xterm &
```

and/or

```
$ xclock &
```

**UNIX/Linux**

If you are running in a UNIX/Linux desktop with a graphical session then it is straightforward to get a graphical connection. Simply use the -Y option to `ssh`:

```
$ ssh -Y <username>@grid.reading.ac.uk
```
Introduction to Linux

Basic introduction to the Linux “shell”

The shell allows you to run commands by typing commands at the “shell prompt”. The shell prompt looks something like [vxx05163@vicg2 ~]$ - which means [$username@$machine name$ <directory> $]. In this example “~” is the shorthand name for your home directory. Commands take the form of:

<command> <argument1> <argument2> ...

In this guide we put a $ in front of commands, to distinguish it from the output, for example:

$ <command> <argument1> <argument2> ...
<OUTPUT>

The shell keeps a record of previous commands that you have typed, which can be scrolled through by the up and down arrow keys. The lines can then be edited with the left and right arrow keys.

When specifying filenames as arguments on the command line, you can use wildcards to match more than one file. * matches any number of characters and ? matches one character. So with files: file0, file01 and file02, file? would only match file0 whereas file* would match all of them.

Commands and filenames can be “completed” by using the TAB key. If you type some of a command name or filename then press TAB the name will be completed. If there are more than one choice pressing TAB twice reveals a list of possibilities.

Files and directories in Linux

In Linux all the files are viewed in one large hierarchical filesystem (not one per hard disk or CD as in Windows). The “root” directory - the directory which contains all the other directories and files is called /. Paths are used to tell the shell where to look for files and directories when running commands. Within paths, directory names are separated with a / as well so /usr/bin/head is the file head in the directory bin in the directory usr in the root directory.

There is also the concept of a “current directory” this is the directory which the shell is currently looking at and files can be specified relative to the current directory to save typing. So if the current directory was /usr then I can refer to the file /usr/bin/head as bin/head and if the current directory was /usr/bin I could refer to it as simply head. Note that if a path does not start with a / then it is taken to be relative.

The two most important directories for users are ~ which is a symbolic name for your home directory (which is normally stored somewhere inside /home) and /tmp which can be used for anyone to store temporary files.

Useful commands

ls - this command will list all the files in the current directory. It can also be used with the argument -l to give more information; -h to display this information in human readable form; and a directory name to list the contents of a directory. Examples:
$ ls          Simple listing of current directory
$ ls -l       Detailed listing of current directory
$ ls -lh      Detailed listing of current directory in a more human-readable form
$ ls docs     Simple listing of the directory docs in the current directory
$ ls /etc     List the /etc directory

cd - this command changes the current directory, the only argument is the name of the new
current directory, which can be specified relative to the current directory. For
example:

    $ cd bin
    $ cd /usr/bin

pwd - prints out the current directory (in case you are lost). For example

    $ pwd

cat - displays the contents of a file to the screen (be careful that the file contains text). For
example:

    $ cat /usr/share/dict/words

head - displays the first few lines (default 10) of a file to the screen (be careful that the file
contains text). If you add the argument -<number> you can change how many lines are
displayed. For example:

    $ head /usr/share/dict/words
    $ head -100 /usr/share/dict/words

tail - displays the last few lines (default 10) of a file to the screen (be careful that the file
contains text). If you add the argument -<number> you can change how many lines are
displayed. In addition the option -f means that the tail command will print the last 10 lines
and then wait for the file to get longer and then print the new lines (until you press Ctrl-C).
This is useful where you have a program which is writing to a file and you want to see the
output. For example:

    $ tail /usr/share/dict/words
    $ tail -100 /usr/share/dict/words
    $ tail -f outputfile

cp - this command copies a file. At its most simple you just follow the command with the
source filename and then the destination filename. If you want to copy a directory then you
will need to use the -r flag. Also the destination can be an existing directory, in this case the
file or directory is copied to this directory and has the same name. For example:

    $ cp a b        copy file a to a new file b
    $ cp -r dir1 dir2 make a copy of dir1 and its contents and call it dir2
copy the file `myfile` to the `/tmp` directory

`mv` - this command moves/renames a file. It is used just like `cp`, except the `-r` flag is not used for directories. For example:

```
$ mv a b
rename file `a` to `b`
$ mv dir1 dir2
rename `dir1` to `dir2`
$ mv myfile /tmp
move the file `myfile` to the `/tmp` directory
```

### Editing text files

The Campus Grid has a few text editors installed. If you have opened a graphical session then type the command:

```
$ gedit &
```

Into your shell window. This will start a normal text editor much like Notepad or Wordpad under Windows.

If you are not using a graphical session then you can use the nano text editor:

```
$ nano <filename>
```

This is a fairly intuitive text editor - the arrow keys move about the text. Press Ctrl-X to exit, it will ask you whether you want to save and what the filename should be. While in the editor Ctrl-G will give you more help.

### Downloading files from the Internet

Often in Linux you will want to download a file, but without using a web browser on your desktop and copying the files to the Campus Grid. The `wget` command allows you to easily download one file. For example:

```
$ wget 'http://www.reading.ac.uk/e-Research'
```

Will download the e-Research homepage - it is best to put the address in single-quotes as this avoids certain issues. In addition there is a text-based internet browser called `elinks` installed which allows very basic internet browsing, using the arrow keys and return to follow links, and download files:

```
$ elinks 'http://www.reading.ac.uk/e-Research'
```

Press ‘Q’ to exit.

There are no graphical browsers installed on the campus Grid to dissuade the use of the Campus Grid for desktop computer purposes.

### Advanced: SSHFS

The Campus Grid includes the ability to use `sshfs` - which allows any user to use the file system on a remote machine as if it were the local machine. We recommend that you only use this if you already have a working knowledge of Linux.

**Important: only mount file systems within `/tmp`.**
The following commands creates a directory in \texttt{/tmp} where the file system will appear; mounts \texttt{user}'s home directory on the server \texttt{server} (it will prompt you for your password); and makes a link to it (called \texttt{mount}) in your local home directory.

\begin{verbatim}
$ mkdir /tmp/$USER \\
$ sshfs user@server: /tmp/$USER \\
$ ln -s /tmp/$USER $HOME/mount
\end{verbatim}

With \texttt{sshfs} only you can see the filesystem. Before you log off be sure to type:

\begin{verbatim}
$ fusermount /tmp/$USER
\end{verbatim}

To un-mount the filesystem.

\section*{Tutorials}

\begin{itemize}
\item There are useful tutorials on UNIX/Linux and various programming languages on the Bristol training site (\url{http://www.acrc.bris.ac.uk/acrc/training.htm}).
\item Meteorology UNIX Help Pages (\url{http://www.met.rdg.ac.uk/it/docs/unix-sys.html}) - aimed at Meteorology users, but there is lots of General bits here.
\item Meteorology System Administration Pages (\url{http://www.met.rdg.ac.uk/~swsellis/system/})
\item Meteorology Applications Help (\url{http://www.metnt.rdg.ac.uk/Scripts/supportdocs/}) - help for many applications - all OSs - some is Meteorology specific but there is many links to further documentations on the packages.
\item The University of Surry has an introduction to Unix/Linux at \url{http://www.ee.surrey.ac.uk/Teaching/Unix/}.
\item At the UNIX/Linux command line typing \begin{verbatim}
$ man <command name>
\end{verbatim}

or

\begin{verbatim}
$ info <command name>
\end{verbatim}
will give help on the command.
\end{itemize}
Introduction to Condor

The Campus Grid uses a program called Condor to manage the programs (or jobs) that are being run on the Campus Grid. The key thing about running a job using Condor is the ClassAd which tells Condor about the program you wish to run, how to run it, about the files it needs, how to find them and what requirements the jobs have. As an introduction try these examples, which all should run from the grid.reading.ac.uk machine.

Submitting one job

A very simple job running a standard command and checking the output

This example runs the `uname` (with the `-a`) argument on a remote machine. The `uname` command returns information about the machine, including its name and its operation system details. Save the following lines as `uname.sub`:

```plaintext
universe = vanilla
executable = /bin/uname
arguments = -a
output = theoutput
error = theerrors
log = thelog
queue 1
```

Then execute the following command:

```
$ condor_submit uname.sub
```

You should be able to see your job in the queue waiting to be run by using the command:

```
$ condor_q
```

For example:

```
$ condor_q
-  Submitter: vigg1.rdg.ac.uk : <134.225.16.30:55918> : vicg2.rdg.ac.uk
 ID  OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
151.0  vxx05163        1/21 17:07   0+00:00:00 I  0   0.0  uname -a

1 jobs; 1 idle, 0 running, 0 held
```

Eventually the job will disappear from the queue (it may happen too quickly for you to see!), meaning it has finished. There is more information on monitoring and managing jobs in the Useful Condor Commands section.

The files `theoutput`, `theerrors` and `thelog` should have been created and `theoutput` should contain the output of the `uname -a` command:

```
Linux cg-241-16.rdg.ac.uk 2.6.10-co-0.6.2 #5 Sat Feb 5 10:19:16 IST 2005 i686 i686 i386 GNU/Linux
```

As you can see the name of the machine that run the actual command was `cg-241-16.rdg.ac.uk` not the machine that you ran `condor_submit` on.
A job with input and output files using the sort program

This example uses the sort program to sort the lines of a file, creating the file sort-output-file. As an input, create a file with some data in, call it sort-input-file. You need to create the ClassAd, sort.sub:

```
universe = vanilla
executable = /bin/sort
arguments = sort-input-file -o sort-output-file
output = theoutput-2
error = theerrors-2
log = thelog-2
queue 1
```

Submit this using:

```
$ condor_submit sort.sub
```

and monitor it using condor_q, in the same way as the first example. When it finishes you should have a file sort-output-file which contains a sorted version of your sort-input-file.

You might want to try this with a real data file, transferring the data to the Campus Grid service using the instructions in the Transferring Files section.

A job requiring lots of memory

Sometimes you will find that you want to run a job that requires more than a certain amount of memory, so you will want to make sure it runs on a machine with enough memory. In this example, we say that we require a machine with 512Mb or more o memory and that we'd prefer machines with 1024Mb or more memory (although the uname command does not actually require this much memory!):

```
universe = vanilla
executable = /bin/uname
arguments = -a
output = theoutput
error = theerrors
log = thelog
requirements= Memory >= 512
rank = Memory
queue 1
```

The requirements= line gives an expression which must be true of a machine, otherwise the job will not be run on a machine. When the Condor system is deciding which machine to run a job on these requirements are evaluated against all the available machines to narrow down the list of candidates.

The rank= line gives an expression that is used to rank all the available machines, the Condor system will favour running your job on those machines for which the rank expression evaluates to a higher value. In this case Rank = Memory will mean Condor will favour machines more memory.

Submitting many jobs

Now say we want to run more than one copy of a job. There are two ways of doing this, the first way is to use two queue commands. To illustrate this say we want to sort two files, this time create sort-input-file-1 and sort-input-file-2 storing the output in the local files sort-output-file-1 and sort-output-file-2. Note the two sorting jobs are run on different servers. Here is the ClassAd:
universe = vanilla
executable = /bin/sort
arguments = sort-input-file-1 sort-output-file-1
output = multi.out-1
error = multi.err1-1
queue

arguments = sort-input-file-2 sort-output-file-2
output = multi.out-2
error = multi.err-2
queue

Note that each time an option is specified it overwrites the previous value, and each time
there is a queue command a new job is submitted. In this case you should see two jobs in the
queue and get two output files, sort-output-file-1 and sort-output-file-2.

The second way to run more than one job is more practical for large number of jobs. In this
method we use the $(Process) macro which gives an incrementing number for each
submitted job. In this example we once again use sort-input-file as the input, this time we
run 10 jobs, the first one extracting the zeroth character from each line, the second the first
character and so on to the ninth character. The command used is the cut command. The
ClassAd is as follows:

universe = vanilla
executable = /bin/cut
arguments = -c $(Process) sort-input-file
output = output.cut.$(Process)
error = error.cut.$(Process)
log = cut.log
queue 10

This time you will see 10 jobs with condor_q and will have 10 output and error files:
output.cut.0 to output.cut.9 and error.cut.0 to error.cut.9. What has happened is Condor
has submitted 10 jobs but with values 0-9 substituted for each $(Process). In this case the cut
command has extracted the 0th, 1st, 2nd...9th character from each line from the sort-input-
file and put the characters in the output.cut.* files. [Note the output.cut.0 file is empty as
there is no 0th character.]

Long Running Jobs

For long running jobs we suggest that you use checkpointing. This allows your jobs to be
resumed from where they left off, rather than being restarted from the beginning, when
your job is moved off a machine for some reason. For more information see Checkpointing.

Useful Condor Commands

This page describes some of the key condor commands and how they are used to monitor
and manage your jobs in the Campus Grid.

Conventions

Both ways of starting jobs will output a cluster id, for example:

$ condor_submit ack.sub
Submitting job(s).
Logging submit event(s).
1 job(s) submitted to cluster 17316.
In this case the cluster id is "17316", this represents a set of jobs that were submitted using the same run of the condor_submit command. Depending on how the jobs were submitted there could be one or more jobs in this cluster. In this case there is only one. The jobs are numbered from 0 within a cluster. So the job will be called: "17316.0". If there were 4 jobs they would be called "17316.0", "17316.1", "17316.2" and "17316.3".

In the following documentation we will use the shorthand XXXXX.Y to represent where you should put your job id, but note that you can actually use a cluster id instead when you want to operate on all the jobs in a cluster.

In addition the shorthand vxxXXXXX is used where your username is required.

I want to....

Use this section to discover which command you need for a particular task - then you can check the command in the Command Reference sections below.

General

- Submit a job - condor_submit
- Track a running job - condor_q
- Find information on a completed job - condor_histroy
- Pause or restart a job - condor_hold and condor_release
- Cancel a job - condor_rm
- Find out information on the machines in the Campus Grid - condor_status

Advanced

- Checkpoint a job - condor_checkpoint
- Compile a program for use with checkpointing - condor_compile

Command Reference: General

condor_submit: The condor_submit command is used to start jobs, it is described in the above sections.

condor_q: The condor_q command is used to list the jobs currently in the queuing system. When you use the command with no argument you get something like.

```
$ condor_q

-- Submitter: vigg1.rdg.ac.uk : <134.225.16.30:49061> : vigg1.rdg.ac.uk
ID   OWNER  SUBMITTED   RUN_TIME ST PRI SIZE   CMD
16100.0 vxx05164 7/4 20:03 0+06:41:41 I 0 0.0  data
```

The columns are: job id, user who submitted the job, the time it was submitted, the time it has been running so far, the job's status, its priority, its memory size and the command that was run. The status is important to understand, the values can be as follows:

- 'I' - Idle: this means it is waiting to be run or is being set up
- 'R' - Running: this means that the job is currently running
- 'C' - Completed: the job has done but it is not yet been deleted from the queue
- 'X' - Removed: the job has been removed/cancelled by the user but it is not yet been deleted from the queue
- 'H' - On Hold: the job is on hold - this means that either the job has been put on hold by a user so it stops running or by Condor because of an error.

Of these the one that needs more explanation is 'H' Hold. You can use the **condor_hold** command (see below) to put a job on pause, but often Condor does this automatically if there is an error. In this case you should use -l option of **condor_q** to find out the full information on the job, for example:

```
$ condor_q -l XXXXY
```

This will print lots of detailed information about the job, but the line you are looking for is "HoldReason" line which will tell you what the problem. If you can resolve this then you can use **condor_release** (see below) to restart the job.

The **condor_q** command can take the -l option to print full information about the jobs in the queue - but this gives far too much information to be useful you do not restrict the jobs that are reported on. Useful restrictions are a particular job:

```
$ condor_q -l XXXXY
```

or to all the jobs of a user:

```
$ condor_q -l vxxXXXXX
```

A third useful flag is **-better-analyze** - this can be used to work out why your job is not running if it is staying in the Idle/'I' state for a long time. For example:

```
$ condor_q -better-analyze 17316.0
```

```
-- Submitter: vigg1.rdg.ac.uk : <134.225.16.30:49061> : vigg1.rdg.ac.uk
ID   OWNER            SUBMITTED     RUN_TIME ST   COMPLETED CMD
---
17316.000:  Run analysis summary.  Of 247 machines,
  247 are rejected by your job's requirements
  0 reject your job because of their own requirements
  0 match, but are serving users with a better priority in the pool
  0 match, match, but reject the job for unknown reasons
  0 match, but will not currently preempt their existing job
  0 are available to run your job
No successful match recorded.
Last failed match: Tue Jul 15 10:31:27 2008
Reason for last match failure: no match found
WARNING:  Be advised:
  No resources matched request's constraints
Check the Requirements expression below:
Requirements = (Memory > 100000) && (Arch == "INTEL") && (OpSys == "LINUX") &&
  ((CkptArch == Arch) || (CkptArch =?= UNDEFINED)) && ((CkptOpSys == OpSys) ||
  (CkptOpSys =?= UNDEFINED)) && (Disk >= DiskUsage)
```

**condor_history**: This command is like **condor_q** execept that it lists the jobs that have completed. There are many thousands of these so it can be slow! Again you can use a -l option to get full information and can restrict to a particular user or job. For example:

```
$ condor_history
ID   OWNER            SUBMITTED     RUN_TIME ST   COMPLETED CMD
---
1.0   vxx05153       11/23 17:17   0+00:00:00 X   ??/home/sufs1/ru6
3.0   vxx05153       11/23 17:33   0+00:00:24 C  11/23 17:34/home/sufs1/ru6
```

Note that there is now a time of completion column and the status is either 'X' for cancelled or 'C' for completed (i.e. finished naturally).
condor_hold and condor_release: These commands allow you to pause (condor_hold) and restart (condor_release) job(s) - again you can use either the Job id of the job to target a single job or your username to target all jobs.

condor_rm: The condor_rm command is used to cancel a job and remove it from the queue - again you can use either the Job id of the job to cancel a single job or your username to cancel all jobs. It will take a little while for the job to be cancelled so it may stay in the condor_q list for a while with a status of ‘X’. For example to kill all my jobs I would say:

$ condor_rm vxxXXXXX

Sometimes jobs can go to hold/H’ while being removed - try using condor_release to un-hold them. If this does not work add the -forcex flag to the command to force the job to be deleted.

condor_status: The condor_status command lists all the machines that are currently in the Campus Grid, for example:

$ condor_status
Name OpSys Arch State Activity LoadAv Mem ActvtyTime
cg-216-125. rd LINUX INTEL Unclaimed Idle 0.000 512 0+02:55:04
cg-216-130. rd LINUX INTEL Unclaimed Idle 0.000 1024 0+03:00:04
cg-216-133. rd LINUX INTEL Unclaimed Idle 0.000 512 0+02:10:07
..............................................
cg-244-62.rdg LINUX INTEL Claimed Busy 1.000 256 6+00:50:59
cg-244-74.rdg LINUX INTEL Unclaimed Idle 0.000 256 0+02:55:04
Machines Owner Claimed Unclaimed Matched Preempting
INTEL/LINUX 246 0 81 165 0 0
Total 246 0 81 165 0 0

Here there are 246 machines, 81 are in use and 165 are free. The LoadAv column shows how busy the machine is and the Mem column details how much memory the machine has. The Name column is truncated - the full name should look something like NAME.rdg.ac.uk.

To find out full information about a machine use:

$ condor_status -1 NAME.rdg.ac.uk

Command Reference: Checkpointing

condor_compile: This command is used to compile a program for use with the automatic check pointing in condor. It is described in more detail in the Using Condor Checkpointing section. You will need to have the source code for your program and then prefix all the commands used to compile and install the software with condor_compile.

condor_checkpoint: This command allows you to force a job to take a “checkpoint”. A “checkpoint” is a record of the current state of a job which is stored on a central machine and allows a job to be restarted from that point if it gets evicted from a machine, rather than having to restart from the start of processing. For more information see the Using Condor Checkpointing section.

Tutorials & More Information

There were some very useful tutorials given at the European Condor Week: Condor User Tutorial and Advanced User Tutorial (please note that some of the features described may not work currently on the Campus Grid - if they would be useful for your work then contact ITS Help and we will try and enable them).
You can also find out much more information in Condor in general at the Condor Website (http://www.cs.wisc.edu/condor/) and in particular in the Online Condor Manual (http://www.cs.wisc.edu/condor/manual/v7.0/).
Introduction to Applications

The Campus Grid has Matlab and R installed, this section describes the basic use of these applications. For both Matlab and R you can either use the basic manual process or you can use the new library function that will automate the process for you, although this is a beta service - please see the Useful Services in Beta section.

Important Note on Randomness - Please read

This note is relevant to everyone using random numbers/stochastic processes in any application. When you run your new application on the Campus Grid you may find that the quality of the random numbers or distributions will decrease. This is especially true if you are doing many runs of a random process - you may find all the runs come up with identical results, rather than giving a distribution of results.

This is due to the fact that there is no such thing as randomness in a computer. The numbers are pseudo-random. A random number generator is started with a “seed” value and from this it derives, deterministically, a sequence of random numbers. If the same seed is used twice the same sequence is generated. Often, for various reasons, it seems that Matlab and R use the same seed for each run of a set of runs on the Grid.

The solution is sketched as follows:

1. Before you use the Campus Grid generated an array of random numbers in Matlab or R (depending on the flavour of your real application). Make sure you have the same number as the number of runs you wish to do.
2. Save this somewhere in your home directory - both Matlab and R have a \texttt{save} function which can be used to save out an array.
3. Run your N tasks, making sure that each has access to which run it is as a number from 1 to N.
4. In the code for each run arrange to load in the pre-saved array of random numbers (using the \texttt{load} function in Matlab and R).
5. Pick the $i^{th}$ random number from the array, where $i$ is the number of the run.
6. Use this to seed the random number generator:
   \begin{itemize}
   \item In Matlab you specify one value, here X:
     \begin{verbatim}
     RandStream.setDefaultStream(RandStream('mt19937ar', 'Seed', X))
     \end{verbatim}
   \item In R you need to set the variable \texttt{.Random.seed} to a 3 element integer vector, the three values can be the same, or you can make three times as many numbers as run and use three for each run (here X1 to X3).
     \begin{verbatim}
     .Random.seed <- c(as.integer(X1), as.integer(X2), as.integer(X3));
     \end{verbatim}
   \end{itemize}

Using Matlab

If you need more control over your Matlab code then you can manually compile and run your Matlab code.
1. Change your code to use functions.

If you have not come across functions in Matlab before then they are pieces of Matlab code that you give a name to and formally describe the inputs and output to. They start with a line like:

```
function output = name(input1, input2)
```

This means that you are declaring a function called "name" that takes two inputs (there can be 0 or more inputs) and then returns an output and that the output will have been stored in the variable "output" by the time all the instructions in the function have been run. Importantly the function must be stored in a file called "name.m" where name is the same as the name of the function and there must be only one function per file.

A whole example of a function is:

```
function result = highlight(a)
    result = strcat('**', a, '**')
```

This would be stored in a file called "highlight.m". (This function takes on string argument and puts two stars at either end to highlight the word, returning this new string). If you create this file you can then try out the function in Matlab itself:

```
$ matlab
>> highlight 'hello'
result =
**hello**
ans =
**hello**
>> quit
```

2. Compile your code

You need to "compile" your code for use on the Campus Grid. This is done by using the mcc command, for example to compile the highlight command use:

```
mcc -m highlight
```

There will probably be some of warning messages, but if it has succeeded then you will end up with a file called "highlight".

You can see if the program has correctly compiled by running it locally. If you just run the program it would fail as it cannot find the Matlab libraries, but you can run it as follows with the matlabrunc command:

```
$ matlabrunc ./highlight hello
result =
**hello**
```

3. Run your code on the Grid

The easiest way to run your Matlab code is using the matlab-submission-script, for example:

```
$ matlab-submission-script -e highlight -a "hello"
```

```
Number of job = 1
The executable is: /home/sufs1/ru6/vx/vxx05163/highlight
These are the command line args: hello
Any output will be in file: temp-1227803499-1.out
Any error output will be in the file: temp-1227803499-1.err
```
The condor log will be in the file: temp-1227803499-1.log
The submission file will be: temp-1227803499-1.sub

condor_submit temp-1227803499-1.sub....
Submitting job(s).
Logging submit event(s).
1 job(s) submitted to cluster 1790.

The name of the program is given after the -e and the arguments after the -a. There is lots of useful information that is printed out. The cluster identifier (here 1790) and the files (ending .out/.err/.log/.sub) are important to note. To see how your job is doing type:

$ condor_q 1790

Where you replace 1790 with the cluster number of your job. If there is a line starting with the cluster identifier then it is still running, for example:

`-- Submitter: vicg2.rdg.ac.uk : <134.225.32.45:64571> : vicg2.rdg.ac.uk
ID      OWNER            SUBMIT
TED     RUN_TIME ST PRI SIZE CMD
1790.0   vxx05163       11/27 16:31   0+00:00:27 R 0 0.2 /opt/local/matla`n

When it disappears (you may actually not be quick enough to see it!) then the job has done and you can see the output printed by the job in the file that is given above, for example using:

$ nano temp-1227803499-1.out

Also have a look at the .err (error messages printed), .log (the log file for the job) and .sub (the file that describes the job) files to get a feel of what is going on.

**Running lots of copies: MagicSquares**

Follow the steps for the above example - you'll need some of the files as a basis of the next example.

Copy the magic squares example to your directory:

```bash
$ cp /opt/local/matlab/extern/examples/compiler/magicsquare.m .
```

Compile the example:

```bash
$ mcc -m magicsquare
```

We are going to calculate the magic squares 1 to 10, to do this we use a for loop in the shell:

```bash
$ for i in $(seq 1 10); do matlab-submission-script -e magicsquare -a $i -n $i; done
```

This uses a variable $i which counts from 1 to 10. We make this variable the argument by putting after the -a and we also us "-n $i" - this means that the temporary files will be numbered with the number of the magic square, for example after I run this I have the files temp-XXXXXXXX-1.out/.sub/.err/.log to temp-XXXXXXXX-10.out/.sub/.err/.log. To view all the magic squares, use the condor_q command to check that the jobs have finished and then use the command:

```bash
$ ls temp-*.*.out
```

**Advanced: Using ClassAds**

You can also run multiple copies of the program by using the raw Condor submission system (see Introduction to Condor for more information). It would be wise to base your submit file on one of the ones produced by matlab-submission-script as there are a few important features:
executable = /opt/local/matlab/bin/matlab-wrapper
Transfer_Executable = True
remote_initialdir = /home/sufs1/ru6/vx/vxx05163/matlab
Requirements = TARGET.OpSys == "LINUX"
match_list_length = 1
Rank = Memory
arguments = magicsquares 1
WhenToTransferOutput = ON_EXIT
universe = vanilla
notification = ERROR
output = temp-1227865962-1.out
error = temp-1227865962-1.err
log = temp-1227865962-1.log
queue

These are highlighted in the above example:

- The executable is actually a wrapper which sets up the Matlab environment
- This wrapper must be transferred from the submit machine
- The remote directory to run in is specified
- The name of the compiled Matlab program must be the first argument.

Using the R statistical package

Limitations

Please note that when running on the main Campus Grid there is currently no interactive graphical support with R. This means that graphical output needs to be saved as a PDF or PS file and viewed on your desktop. Please note that this happens automatically when R is called with the --save argument - a file is created called Rplots.pdf.

Example 1 - single demo

This example looks at running a single R job, for this we create an R script that simply runs one of the demos:

Create the file demo-nlm.R, the contents should be the single line:

demo(nlm)

This simply runs the "nlm" demo. Run the script on the Campus Grid by creating the submit file demo-nlm.sub:

universe=vanilla
transfer_executable=false
executable=/usr/bin/R
arguments=--save -f demo-nlm.R
log=demo-nlm.log
output=demo-nlm.out
error=demo-nlm.err
queue 1

Submit the job using

$ condor_submit demo-nlm.sub

Track your job using the command:
$ condor_q XXXX

Then you can look at the output, the graphs produced will be in Rplots.pdf and the output produced to the screen will be in demo-nlm.out and demo-nlm.err. The Rplots.pdf file will need to be copied back to your desktop machine before you can view the images. See the Transferring Files section for information about copying files.

**Example 2 - Running lots of jobs**

If you want to run R on the Campus Grid then you will probably want to run many copies of the same job with different arguments. If you are unsure about using UNIX command-lines then try the example above first, which goes into more details.

Create the file demo2.R - the R script

```r
commandArgs()
  x<-c(1,2,3,4,5,6,7,8,9,10)
  exp=type.convert(commandArgs()[6])
  y=x^exp+1
  pdf(paste("Rplots",exp,".pdf", sep=""))
  plot(x,y, xlab="x axis", ylab="y axis", main=paste("x^",exp,"+1", sep=""))
  dev.off()
```

This script prints out the all the command line arguments that is given to R and then plots the graph \(x^\text{exp}+1\), where \(y\) is the sixth argument given to R. This time we also tell R which filename to store the graphs as, this is so we can separate the outputs, we use Rplots<N>.pdf.

We want to run this for arguments 0 to 9 so we will need to make a Condor submit file demo2.sub:

```plaintext
executable=/usr/bin/R
Transfer_Executable=false
arguments=--no-save -f demo2.R --args $\{(Process)\}
universe = vanilla
output=demo2-$\{(Process)\}.out
error=demo2-$\{(Process)\}.err
log=demo2.log
queue 10
```

Notice that we use $\{(Process)\}$ which is replaced with number (from 0) of the job - the “queue 10” line says to submit 10 copies.

Run the job:

```
condor_submit demo2.sub
```

You can track the state of your jobs using the returned cluster number.

The output graphs are in Rplots0.pdf to Rplots9.pdf and the console output in demo2-0.out to demo2-9.out and demo2-0.err to demo2-9.err. Within the contents of demo2-0.out there will be some lines that will look something like:

```
> commandArgs()
[1] "/usr/local/R/lib64/R/bin/exec/R" 
  "--save"
[3] "-f" 
  "demo2.R"
[5] "--args" 
  "0"
```

This shows the output of the commandArgs() vector - you can see that all the arguments we gave, including the name of the R program are included, numbered 1-6. The argument we are interested in is the 6th one "0" - here we use this simply as a number, but if we use a file name we could easily load in a different file for each run.
Advanced Topics

Compiling code

The submit node is also meant to be a Compilation node - where you can compile code for use on the Campus Grid - this is especially true of Matlab code (see Using Matlab) and for check-pointing (see Checkpointing). There are times when you just want to compile normal code for the Campus Grid.

The vast majority of code will just compile and install on the Campus Grid, if they support the gcc suite of compilers, but there is two very important caveats:

- The Campus Grid is 32-bit whereas the submit node is 64-bit. We have used aliases and various settings to ensure that the default is to compile 32-bit code.
  
  If your program compiles but when you run the program on the Campus Grid there is the error “Exec format error” in the log file then the program is likely to have been compiled for 64 bit. To check use the readelf command:

  $ readelf -h file

  Where file is an intermediate file or compiled program. Check the line starting “Class:” if it is ELF32 it is ok, if it is ELF64. Then please ensure the flag -m32 is getting to all runs of the compiler and linker.

- If you are installing a nontrivial program, then you should ensure that you are trying to install it into your home directory (not the default path, which you will not be able to write to). This can normally be done using a --prefix argument to the configure script. This assumes the very common configure-make-make install method of installation:

  $ cd prog-dir
  $ ./configure --prefix=$HOME/install
  $ make
  $ make install

  If you also need install support libraries then install these, first, to the same place, because the application then has more chance of finding them. It is likely that you will have to point the configure script at the installation of the support libraries, for a list of options for configure use:

  $ ./configure --help

  If you require any help getting your application to compile under the Campus Grid please contact ITS Help.

Shared Scratch Space

Note: This is a new service, so if you do notice any issues with it then please do email ITS Help.

The Campus Grid has shared area of scratch space (for temporary files) of approximately 600Gb. This can be accessed at /mnt/auto/scratch on the head node and the worker nodes. You must create a directory named the same as your username and use this to store your files (any files found elsewhere will be deleted).
There are no quotas for users, we trust that users will be fair in their use of the space, only using what they require and deleting unused data as soon as it is no longer needed (although if there is abuse of the system we will enable quotaing). There is no quota as checkpointing could require a large percentage of the disk. We will delete files that have not been used for 7 days or more if the scratch drive is more than 75% used. We reserve the right to delete files before they have been unused for 7 days. Scratch space is not backed up.

There are also local `/tmp` file systems on each node - these are large - totalling about 5.9TB. Most nodes have 20Gb (check the “SCRATCH” ClassAd attribute) - if the temporary data does not need to be seen by more than one node then please use `/tmp`.

**Checkpointing**

A *checkpoint* is a record of the current state of a job which is stored on a central machine. Condor can be instructed to regularly store these checkpoints and this allows a job to be restarted from where it had got to at the checkpoint if it gets evicted from a machine, rather than having to restart from the start of processing.

The Campus Grid provides two forms of checkpointing - the checkpointing built into Condor and an operating system based checkpointing system called BLCR. Although the BLCR system is easier and supports a much wider variety of programs it is actually still in development - it is documented in the Beta services section (see: BLCR).

**Using Condor Checkpointing**

This section documents how to use the check-pointing feature in Condor. You should familiarise yourself with the earlier section Introduction to Condor before reading this section.

To make use of the check-pointing features you need to recompile your code using the `condor_compile` command, this means there are a number of restrictions on what can be done:

- You need to have the source code of your program and it has to be a compiled program (C, C++, Fortran...) not a script.
- Multiprocess jobs are not allowed.
- Network communications should be short and inter-process communication is not allowed.
- The signals SIGUSR2 and SIGSTP are reserved for the use of Condor.
- Alarms, timers and sleeping are not allowed.
- Memory-mapped files are not allowed and file locks are not fully supported.
- Files should only be opened for reading or writing (not both) otherwise there will be problems on check-pointing.

Though it sounds like there is lots of restrictions, these are not a problem for most applications. Please see Check-pointing Restrictions on the Condor Website ([link](http://www.cs.wisc.edu/condor/manual/v7.0/2_4Road_map_Running.html#SECTION00341100000000000000000000000000)) for more details on these restrictions.

**Compiling**

Say you normally use the command:
$ gcc -o myprog myprog.c anotherfile.c

to compile your code, you should put `condor_compile` in front of it:

$ condor_compile gcc -m32 -o myprog myprog.c anotherfile.c

Now if you run this from the command line you'll see something like:

```
$ ./myprog
Condor: Notice: Will checkpoint to ./myprog.ckpt
Condor: Notice: Remote system calls disabled.
[Normal output]
```

This shows that the check-pointing is working. The `condor_compile` command is clever enough to deal with makefiles, so if you have a larger package which uses standard packaging, you could try the following:

```
$ cd prog-dir
$ ./configure --prefix=$HOME/install
$ condor_compile make
$ condor_compile make install
```

This is the standard model for installing UNIX applications from source. Here I have asked it to install the application under the install directory of my home directory.

### Running Checkpoint-enabled Jobs

Using check-pointing requires you to write your own ClassAds. The one change is that you should change the job universe, this involves changing the line:

```
universe = vanilla
```

in your ClassAds to:

```
universe = standard
```
Introduction to the National Grid Service

The National Grid Service (NGS) provides the mechanism to join the Grid resources at academic institution across the UK into a single resource. This allows the researcher access to a wide-range of different types of resources and also to greater amounts of resources than available at their institution. The NGS is currently free at the point of use, and is useful for users who require access to High-Performance Computing resources for tasks where the constituent parts are not independent.

It is possible to use the NGS from the Campus Grid, this means that it can be quick to move your code to the wider National Grid, should you need a different type of resource. In addition there are many applications available for immediate use on the NGS (see the NGS Software page: [http://www.grid-support.ac.uk/content/view/261/132](http://www.grid-support.ac.uk/content/view/261/132) for more details).

Resources include:

**Central Core Services:** these four resources (Leeds, Manchester, Oxford and STFC-RAL) have 256 cores with 2GB per core arranged in dual CPU-dual core and quad CPU-dual core. They are High Performance Resources and so are suited to parallel jobs where the different parts of the job must communicate.

**Other cluster compute resources:** at Cardiff (SGI), Lancaster (Linux), Glasgow (Linux x2), Westminster (Linux).

**Condor Pools:** Cardiff's Windows Condor Pool.

Applications include: AMBER, AUTODOCK, BLAST, CASTEP, DL POLY, DOCK, FASTA, GAMESS, GATE, GAUSSIAN, GROMACS, LAMMPS, MRBAYES, NAMD, PCGAMESS, R, ROOT, SIESTA.

For more details of these resources see NGS Resources ([http://www.grid-support.ac.uk/content/view/239/157/](http://www.grid-support.ac.uk/content/view/239/157/)).

The Reading Campus Grid also supports the standard NGS interfaces. Therefore external academics who collaborate with academics from Reading can make use of the Reading Campus Grid through the NGS interface.

Applying for access

You will need a e-Science digital certificate to access the NGS. To start the process visit the following website and fill out the form at [https://ca.grid-support.ac.uk/cgi-bin/pub/pki?cmd=basic_csr&CSR_PROFILE=PERSON&ra=Reading%20ITS](https://ca.grid-support.ac.uk/cgi-bin/pub/pki?cmd=basic_csr&CSR_PROFILE=PERSON&ra=Reading%20ITS).

Once you have submitted the online request you will need to Visit David Spence in IT Services with a valid photo ID (Note: this can be your University ID/Library card) and the PIN you gave to the online form. Within the latter in the day you should receive an email detailing how you can retrieve your new certificate.

Once you have your certificate loaded into your browser you can actually apply for access. Visit the page [http://www.grid-support.ac.uk/content/view/221/171/](http://www.grid-support.ac.uk/content/view/221/171/) and follow the instructions. It can take up to a week for the NGS to process your application.

Setting up your Certificate on the Campus Grid

Once you have your grid certificate and NGS account (see above) then you will need to upload your certificate to the Campus Grid server. The NGS provide instructions for this, first use the instructions at [http://www.grid-support.ac.uk/content/view/354/276/](http://www.grid-support.ac.uk/content/view/354/276/) to back up your
certificate to a file and then those at http://www.grid-support.ac.uk/content/view/67/184/. Note you will need to upload the pfx file to the Campus Grid server between steps one and two.

Then when you to use the NGS you will need to run the command \texttt{grid-proxy-init} (which will ask for the password for your certificate). This creates a temporary certificate (for 12 hours) with no password so you only have to type in your certificate password once every 12 hours.

**Differences to the Campus Grid**

**You must have a certificate:** unlike the Campus Grid the NGS does not support username and password log-in.

**You must apply for an NGS account:** this is separate from your Reading Campus Grid account and is in addition to having a certificate.

**Files must be copied:** As the resources are remote to the Campus Grid the input, program and output files must be explicitly specified.

**You must specify which resource to send the program to,** for example:

- Leeds core resource: \texttt{ngs.leeds.ac.uk/jobmanager-pbs}
- Manchester core resource: \texttt{vidar.ngs.manchester.ac.uk/jobmanager-pbs}
- Oxford core resource: \texttt{ngs.oerc.ox.ac.uk/jobmanager-pbs}
- STFC-RAL core resource: \texttt{ngs.rl.ac.uk/jobmanager-lsf}
- Oxford old resource: \texttt{grid-compute.oesc.ox.ac.uk/jobmanager-pbsox}
- Cardiff (Windows resource): \texttt{condorngs.cf.ac.uk/jobmanager-condor}

The NGS Resources page (http://www.grid-support.ac.uk/content/view/239/157/) has access to more sites.

**Using Condor to manage NGS jobs**

The native Globus mechanism for access the NGS is quite a bit more involved than Condor, but not all is lost as Condor can talk to the Globus interface on the NGS. To use the NGS we use the \texttt{job-submission-script}, which automatically generates the condor submit files for the National Grid (which are a little more complex). Let’s see an example:

First unlock your certificate:

```
$ grid-proxy-init
```

Then use the submission script to run the demo-nlm.R example from Using the R:

```
$ job-submission-script -g -h ngs.rl.ac.uk/jobmanager-lsf i demo-nlm.R -o Rplots.ps -a "R --no-init-file --save -f demo-nlm.R" -b Rexample -e /usr/ngs/R
```

Job host = \texttt{ngs.rl.ac.uk/jobmanager-lsf}
Number of job = 1
The executable is: \texttt{/usr/ngs/R}
These are the command line args: R --save --no-init-file -f demo-nlm.R
These are the input files: demo-nlm.R
These are the output files: Rplots.ps
Submit to wider Grid.
Any output will be in file: Rexample-1233247548-1.out
Any error output will be in the file: Rexample-1233247548-1.err
The condor log will be in the file: Rexample-1233247548-1.log
The submission file will be: Rexample-1233247548-1.sub

condor_submit Rexample-1233247548-1.sub....
Submitting job(s).
Logging submit event(s).
1 job(s) submitted to cluster 5971.

Here you can see the files that will be produced - the output file, error file and log files being most important. Please note that the NGS resources have their own queuing system so your job could wait in the idle state (for days if busy with big jobs) while it waits for a slot on the resource - check the log file for more detailed information on the job status. Going through the command:

- **job-submission-script** - is the command name
- **-g**: means use the wider Grid
- **-h ngs.rl.ac.uk/jobmanager-lsf**: tells the command that you wish to use the service at STFC-RAL
- **-i demo-nlm.R**: says to copy demo-nlm.R over before running the command
- **-o Rplots.ps**: says to copy Rplots.ps back once the command has finished
- **-a "--save --no-init-file -f demo-nlm.R"**: gives the arguments to the command
- **-b Rexample**: causes the output (e.g. logs and console output) files should be prefixed with Rexample
- **-e /usr/ngs/R**: gives the name of the command (on the core NGS nodes there are wrappers for many packages in the /usr/ngs directory

Another useful flag is **-t**, which causes the program to be copied across to the NGS node, if the program is not installed on the NGS.

**Other Commands**

There are two other commands that you might find useful:

**gsissh** - allows you to log onto a terminal on the resource. It should be used like:

```
$ gsissh -p 2222 ngs.rl.ac.uk
```

Note that the '/') and everything that follows it is removed from the name.

**gsiscp** - this allows you to copy files to and from the resources, e.g.

```
$ gsiscp -P 2222 testR ngs.rl.ac.uk:test.R
```

copies the local file **testR** to the file **test.R** on the machine ngs.rl.ac.uk.

```
$ gsiscp -P 2222 ngs.rl.ac.uk:myfile myfile
```

copies the file **myfile** in the home directory on ngs.rl.ac.uk to the current directory to **myfile**
Useful Services in Beta

Note: These are new services which have been developed within IT Services. They are not supported to the same level as the main Campus Grid service, but if you do notice any issues with it or have any suggestions then please do email ITS Help.

Matlab: Automatic Process

As part of the simplification process of the Campus Grid we have developed a support library `pforf` which enables you to access the power of the Campus Grid from within Matlab. The idea behind `pforf` is to define a Parallel for loop - like a standard for loop but with each iteration executing automatically on a different Campus Grid node. Please note this only works where there is no dependencies between the different values for example:

```matlab
for i = 1:100
    x[i] = i * i;
end
```

would be ok, but

```matlab
for i = 1:100
    x[i] =x[i-1] * i;
    counter = counter + 1;
end
```

has two dependencies.

The function is defined as follows:

```matlab
result = pforf(items, function_pointer, argument_structure)
```

The function takes three arguments:

- **items**: is a list of values to iterate over
- **function_pointer**: is the function to run on each of the nodes - please note this has to be a function in an .m file. The name of the function must be preceded with an @.
- **argument_structure**: is a value (most usefully a structure) which is passed to each of the different runs of the function.

The result is a list of the return values from each run in order.

The function should take two values, the first is one of the **items** from the list items and the other is a copy of **argument_structure**. To use `pforf` you must include the following line at the top of your main program, or enter it into the command prompt before using `pforf`:

```matlab
addpath('/usr/local/cg-tools/current/matlab/matlab-functions');
```

For example the following script:

```matlab
addpath('/usr/local/cg-tools/current/matlab/matlab-functions') ;
arg.a = 100;
arg.b = 30;
list = pforf(1:10, @mainbody, arg);
total=0;
for i = 1:length(list)
    total = total + list(i).c;
end
```
total

Needs a function in `mainbody.m`:

```matlab
function res = mainbody(it, arg)
    res = struct
    res.c = arg.a * it + arg.b;
end
```

To use this follow the instructions in Graphical Access to enable graphical access and then run the `matlab` command. Create the `mainbody.m` file and the execute the above script by typing it into the console window. The `pforf` line will take a while as it takes a while to set up the Campus Grid run - normally such a program as this would be run fully locally.

Please also read the note earlier on in the section “Important Note on Randomness - Please read”.

R: Automatic Process

There is a graphical tool for R the Campus Grid login node only. This requires you to have followed the steps in the Graphical Access section to setup graphical access. Please note that under Windows this only works with Cygwin (not Exceed). To run the R Graphical Interface then type:

```
$ Rgui
```

Please note this is not the same graphical interface as with R under Windows.

As part of the simplification process of the Campus Grid we have developed a support library `pforf` which enables you to access the power of the Campus Grid from within R. The idea behind `pforf` is to define a Parallel for loop - like a standard for loop but with each iteration executing automatically on a different Campus Grid node. Please note this only works where there is no dependencies between the different values for example:

```r
for(i in 1:100) {
    x[i] = i * i;
}
```

would be ok, but

```r
for (i in 1:100) {
    x[i] =x[i-1] * i;
    counter = counter + 1;
}
```

has two dependencies.

The function is defined as follows:

```r
result = pforf(items, function_name, argument)
```

The function takes three arguments:

- **items**: is a list of values to iterate over
- **function_name**: is the name of the function to run on each of the nodes - it should be in quotes.
- **argument**: is a value (most usefully a list with named elements) which is passed to each of the different runs of the function.

The result is a list of the return values from each run indexed by the relevant item.
The function should take two values, the first is one of the items from the list items and the other is a copy of argument. To use pforf you must include the following line at the top of your main program, or enter it into the command prompt before using pforf:

```r
source('/usr/local/cg-tools/current/R/r-functions');
```

For example the following script:

```r
source('/usr/local/cg-tools/current/R/r-functions/grid.R');
mainbody <- function(it, arg) {
  list(c= arg$a * it + arg$b)
}
arg<- list(
  a = 100,
  b = 30)
out = pforf(1:10, "mainbody", arg)
total=0
for (i in 1:length(out)) {
  total = total + out[[i]]$c
}
total
```

To use this follow the instructions in Graphical Access to enable graphical access and then run the `matlab` command. Create the `mainbody.m` file and then execute the above script by typing it into the console window. The `pforf` line will take a while as it takes a while to set up the Campus Grid run - normally such a program as this would be run fully locally.

Please also read the note earlier in the section “Important Note on Randomness - Please read”.

**BLCR Checkpointing**

This section documents how to use BLCR check-pointing. You should familiarise yourself with the earlier section Introduction to Condor before reading this section.

There are very few restrictions on what you can checkpoint using BLCR. The main restriction is:

- **Direct** networked communication is not allowed - although you can augment your code to make this possible. You can use the network file system, but you cannot open sockets (although short connections will probably work).

BLCR will checkpoint scripts and does not require your code to be re-compiled to use a checkpointing library.

We provide a wrapper script called `checkpoint-submit` which can be used to submit jobs with checkpointing enabled and hides the complexity of using BLCR. This automatically generates a Condor submission file which includes all the extra options needed for checkpointing. Advanced users will want to use the generated Condor submission file as an example for their own submission files.

The following are the main options to `checkpoint-submit`:
• `-e <executable name>` Specify the program to run
• `-a <arguments>` Specify the arguments to give to the program
• `-m <memory in Mb>` Specify a minimum memory requirement

For example to run the sleep program for 100 seconds:

```
$ checkpoint-submit -e /usr/bin/sleep -a 100
```

Checkpoints are stored in the shared scratch area in: `/mnt/scratch/<username>/chkpts/` when you use `condor_rm` this may not be cleared up properly so it’s worth clearing this out if you cancel all your jobs. For example:

```
$ cd /mnt/scratch/<username>/chkpts/
$ rm *
```

For more information on BLCR see the BLCR website: [https://ftg.lbl.gov/CheckpointRestart/CheckpointRestart.shtml](https://ftg.lbl.gov/CheckpointRestart/CheckpointRestart.shtml).