1 Running jobs with LSF

1.1 Introduction

A single laptop or desktop PC may be used to process, analyse and visualise biological datasets. However, these datasets can be quite large, making computational analysis on a single machine painfully slow and sometimes impossible. In these situations, a compute farm can be used to run the analysis more efficiently. In this tutorial, we will look at what a compute farm is and how we can use a job manager, such as LSF, to perform computational tasks more efficiently.

1.2 Learning outcomes

On completion of the tutorial, you can expect to be able to:

- view cluster structure using LSF
- view queue information using LSF
- submit and manage jobs using LSF
- submit job arrays using LSF
- submit dependent jobs using LSF
- understand priority and fairshare in LSF
- troubleshoot LSF issues

1.3 Tutorial sections

This tutorial comprises the following sections:

1. Introduction
2. Job queues
3. Submitting jobs
4. Managing jobs
5. Job arrays and dependencies
6. Priority and fairshare
7. Troubleshooting
8. LSF cheat sheet

1.4 Authors

This tutorial was written by Victoria Offord.

1.5 Running the commands from this tutorial

You can run the commands in this tutorial either directly from the Jupyter notebook (if using Jupyter), or by typing the commands in your terminal window.

1.5.1 Running commands on Jupyter

If you are using Jupyter, command cells (like the one below) can be run by selecting the cell and clicking *Cell -> Run* from the menu above or using *ctrl Enter* to run the command. Let's give this a try by printing our working directory using the *pwd* command and listing the files within it. Run the commands in the two cells below.

```
pwd
```

```
ls -l
```

1.5.2 Running commands in the terminal

You can also follow this tutorial by typing all the commands you see into a terminal window. This is similar to the "Command Prompt" window on MS Windows systems, which allows the user to type DOS commands to manage files.

To get started, select the cell below with the mouse and then either press control and enter or choose Cell -> Run in the menu at the top of the page.

```
echo cd $PWD
```

Now open a new terminal on your computer and type the command that was output by the previous cell followed by the enter key. The command will look similar to this:

```
cd /home/manager/pathogen-informatics-training/Notebooks/LSF/
```

Now you can follow the instructions in the tutorial from here.

1.6 Let's get started!

This tutorial assumes that you are connected to a compute farm with LSF installed. To check that you have installed these correctly, you can run the following commands:

```
bqueues -h
```

```
bsub -h
```

```
bjobs -h
```

This should return the help message for `bqueues`, `bsub` and `bjobs` respectively.

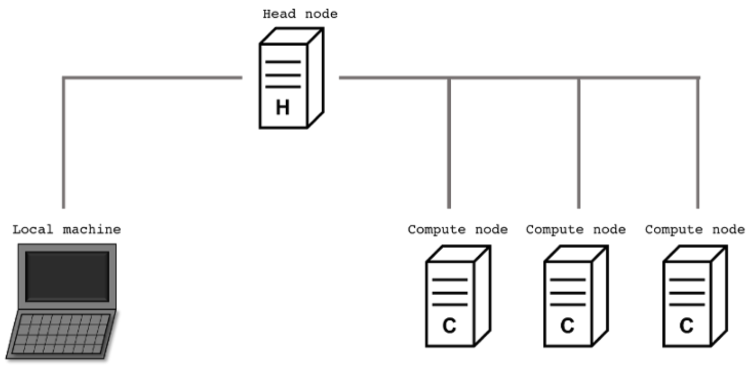
To get started with the tutorial, head to the first section: Introduction

2 Introduction

This part of the tutorial describes the basic structure of a compute cluster and introduces the key concepts behind the workload manager, LSF. After completing this section, you should be able to: describe the basic structure of a cluster, define key terms and find information about cluster structure using LSF.

2.1 What is a cluster?

A cluster is a set of connected computers (**nodes** or **hosts**) which work together. When you log in to the cluster from your local machine, you will most likely be connecting to the **head node**. The head node handles the submission of the computational tasks you want to perform. These tasks are then passed on to the **compute nodes** where they will subsequently be run.



Cluster structure

You don't have to log into the head node, you can also log in to a compute node. When you log into the head node, you can use it to submit your jobs, migrate data between file systems and housekeeping. However, you should ***not*** be running computationally intensive jobs on the head node outside of LSF.

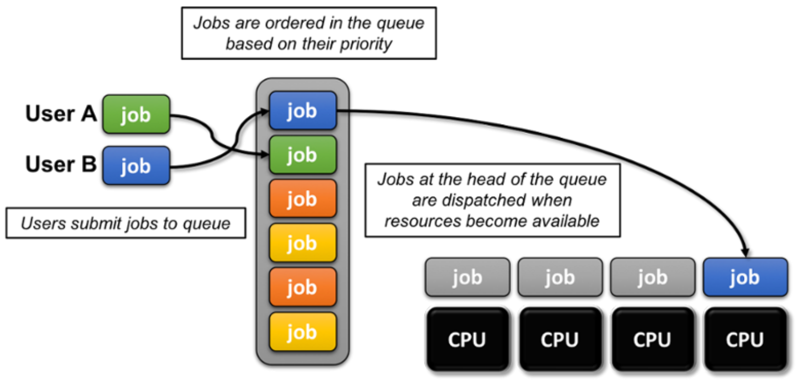
2.2 What is LSF?

When analysing data on a single machine, such as a laptop, commands or scripts are run in the terminal and the results are given back via the terminal. On a cluster, we need to run these commands or scripts as **jobs**.

The resources required to run the jobs may not always be available straight away, so the jobs get submitted into a **queue**. A queue is a list of jobs which are waiting for resources (pending) or being executed (running). As jobs in the queue finish executing, the resources they were using become available again and the next job in the queue will start running.

Job scheduling and execution is controlled by the platform load sharing facility (**LSF**) which manages the workload.

For more information, please see the about LSF section of the LSF user guide.


Jobs are ordered in the queue based on their priority
User A
job
User B
job
Users submit jobs to queue
job
job
job
job
job
job
Jobs at the head of the queue are dispatched when resources become available
job
job
job
job
CPU
CPU
CPU
CPU

Job processing

This tutorial assumes that you are currently connected to a cluster which has LSF installed (e.g. pcs or farm for Sanger users).

Let's start by getting some general information about the cluster.

```
lsid
```

```
IBM Platform LSF Standard 9.1.3.0, Jul 04 2014
Copyright IBM Corp. 1992, 2014. All rights reserved.
US Government Users Restricted Rights...

My cluster name is pcs5
My master name is pcs5a
```

This should tell you the name of the cluster you're connected to (e.g. pcs5) and the version of LSF it's using (e.g. 9.1.3.0).

Next, let's take a look at how the cluster is structured.

```
lshosts
```

```
HOST_NAME      type    model  cpuf ncpus maxmem maxswp server RESOURCES
pcs5a         X86_64 BL465c_G  8.0     32 255.9G  31.9G    Yes (mg linux lustre)
pcs5b         X86_64 BL465c_G  8.0     32 255.9G  31.9G    Yes (mg linux lustre)
pcs5c         X86_64 BL465c_G  8.0     32 255.9G  31.9G    Yes (linux lustre avx)
pcs5d         X86_64 BL465c_G  8.0     32 255.9G  31.9G    Yes (linux lustre avx)
pcs5e         X86_64 BL465c_G  8.0     32 255.9G  31.9G    Yes (linux lustre avx)
```

This should tell you which hosts (nodes) are part of the cluster. In this example, there are five hosts called pcs5a-e.

Finally, let's take get some information about the hosts.

```
bhosts
```

```
HOST_NAME            STATUS          JL/U    MAX  NJOBS    RUN  SSUSP  USUSP    RSV
pcs5a                ok                 -     16      8      8      0      0      0
pcs5b                ok                 -     16      0      0      0      0      0
pcs5c                closed             -     32     32     11      0      0     21
pcs5d                ok                 -     32     26     26      0      0      0
pcs5e                ok                 -     32     24     20      0      0      4
```

For each host, this gives us the host name, host status, job state statistics, and job slot limits. The host status tells us whether the host is available and ready to accept new jobs.

There are four possible host status states:

- **ok** - host is available to accept and run new batch jobs
- **unavail** - host is down, or load and job management controls are unreachable
- **unreach** - load management controls are running but job management controls are unreachable
- **closed** - host not accepting new jobs

To find out why a host is closed you can run `bhosts` again with the `-w` option which returns more detailed information about the host.

```
bhosts -w
```

```
HOST_NAME            STATUS             JL/U    MAX  NJOBS    RUN  SSUSP  USUSP    RSV
pcs5a                ok                 -     16      0      0      0      0      0
pcs5b                ok                 -     16      0      0      0      0      0
pcs5c                closed_Full        -     32     32     11      0      0     21
pcs5d                ok                 -     32     30     26      0      0      4
pcs5e                ok                 -     32     28     28      0      0      0
```

This tells us that the maximum number of jobs which can be run on that host has been reached (see values for STATUS, MAX and NJOBS). Once those jobs have started to complete, the host will be ready to accept new jobs.

2.3 What's next?

For an overview of what this tutorial covers, head to the index. Otherwise, let's take a closer look at queues.

3 Job queues

When you submit a job, the job scheduler will place that job into a queue. Queues are just lists of submitted jobs which share scheduling and resource requirements.

To take a look at which queues are available, you can use the command: `bqueues`.

```
bqueues
```

```
QUEUE_NAME      PRIO STATUS          MAX  JL/U JL/P JL/H NJOBS  PEND   RUN  SUSP
system          1000 Open:Active       -    -    -    -      0     0     0     0
yesterday       500  Open:Active      20    8    -    -      0     0     0     0
small            31  Open:Active       -    -    -    -      0     0     0     0
normal           30  Open:Active       -    -    -    -     35    13     1     0
long              3  Open:Active      50    -    -    - 31686 31636    46     0
basement          1  Open:Active      20   10    -    -    180   170    10     0
```

This will return information about the queues which are available and how busy they are. Here, we can see information about six queues into which jobs can be submitted on the cluster.

By default, `bqueues` will give you the following information:

- **QUEUE_NAME** - the name of the queue
- **PRIO** - the priority of the queue
- **STATUS** - the status of the queue
- **MAX** - the maximum number of job slots available
- **JL/U, JL/P and JL/H** - the job slot limit for users, processors and hosts respectively
- **NJOBS** - the total number of tasks for all jobs in the queue
- **PEND** - the number of pending jobs in the queue
- **RUN** - the number of running jobs in the queue
- **SUSP** - the number of suspended jobs in the queue

3.1 How busy is the cluster?

For each queue, you can see the total number of tasks scheduled (**NJOBS**) and a breakdown of how many of those jobs are waiting to be dispatched (**PEND**), are running (**RUN**) or are suspended (**SUSP**).

3.2 Queue priority

You may have some jobs which are more urgent than others and that you would like to be run sooner. In these instances, the priority of the queue is important.

Jobs submitted to higher priority queues are run first. You can check the queue priority by looking at the **PRIO** column. The larger the priority value of the queue, the higher the priority of the queue. In this example, we can see that the **yesterday** queue has a much higher priority than the

normal queue and so a job submitted to the yesterday queue will often be run before a job on the normal queue if the resources that were requested for that job are available.

For more information on priority and how this works, please see priority and fairshare.

3.3 Queue status

Sometimes a queue might not be available. You can check the status of the queue by looking at the **STATUS** column.

- **Open** - the queue *is* able to accept jobs
- **Closed** - the queue *is not* able to accept jobs
- **Active** - jobs in the queue will be allowed to start when resources are available
- **Inactive** - jobs in the queue won't be started for the time being

---

3.4 Getting more information about a particular queue

You can get more detailed information by using the `-l` option with `bqueues`.

Let's try getting some information about the queues on our cluster.

```
bqueues -l
```

This will give you the requirements and limits for all of the queues on the cluster. You can also get the this information for a specific queue by specifying the name of the queue.

```
bqueues -l <queue_name>
```

In the example command below, we are asking for detailed information about a queue called yesterday.

```
bqueues -l yesterday
```

The `-l` option will give us a lot more information, such as the resource limits for the yesterday queue (e.g. maximum memory usage or run time).

```
QUEUE: yesterday
  -- As in I needed it yesterday highest priority (all nodes)

PARAMETERS/STATISTICS
PRIO NICE STATUS          MAX JL/U JL/P JL/H NJOBS  PEND   RUN SSUSP USUSP  RSV
500   20  Open:Active      20    8    -    -     0     0     0     0     0    0
Interval for a host to accept two jobs is 0 seconds

DEFAULT LIMITS:
 MEMLIMIT
    100 M
```

```
MAXIMUM LIMITS:
 RUNLIMIT
     2880.0 min of BL465c_G8

 CORELIMIT MEMLIMIT
      0 M     250 G

SCHEDULING PARAMETERS
           r15s   r1m  r15m   ut      pg    io   ls    it    tmp    swp    mem
 loadSched   -     -     -     -       -     -    -     -     -      -      -
 loadStop    -     -     -     -       -     -    -     -     -      -      -

              poe nrt_windows adapter_windows ntbl_windows  uptime
 loadSched     -           -               -             -       -
 loadStop      -           -               -             -       -

SCHEDULING POLICIES:  FAIRSHARE
USER_SHARES:  [default, 1]

SHARE_INFO_FOR: yesterday/
USER/GROUP   SHARES  PRIORITY  STARTED  RESERVED  CPU_TIME  RUN_TIME   ADJUST
user1            1       0.302      0         0        47.0     1590       0.000
user2            1       0.301      0         0       590.3     1634       0.000

USERS: all
HOSTS:  pcs5a pcs5b+1 others+2
RES_REQ:  select[type==any]
Maximum slot reservation time: 14400 seconds
```

Below is an example for three queues which have different resource limits. Here, jobs in the normal queue will automatically be terminated or killed by LSF if they try to run for more than 12 hours (RUNLIMIT = 720.0 min), in the long queue after 2 days (RUNLIMIT = 2880.0 min) and in the hugemem queue after 15 days (RUNLIMIT = 21600.0 min). The hugemem also has a much larger memory limit (727.5G) than the normal or long queues (250G).

normal:

```
 RUNLIMIT
 720.0 min of BL465c_G8

 CORELIMIT MEMLIMIT
      0 M     250 G
```

long:

```
 RUNLIMIT
 2880.0 min of BL465c_G8

 CORELIMIT MEMLIMIT
      0 M     250 G
```

hugemem:

```
RUNLIMIT
21600.0 min of HS21_E5450_8

CORELIMIT MEMLIMIT
     0 M   727.5 G
```

For more information, please see the working with queues section of the LSF user guide.

3.5 What's next?

For an overview of the key concepts, you can go back to the introduction. Otherwise, let's take a look at submitting jobs.

4 Submitting jobs

When analysing data on a single machine, such as a laptop, commands or scripts are run in the terminal and the results are given back to us in that terminal. For example, the `sleep` command tells your machine to suspend the execution of a command for a defined period of time (in seconds).

Let's tell our machine to pause for 60 seconds.

```
sleep 60
```

When using a cluster, these commands or scripts need to be submitted as jobs. To submit a job with LSF, we use the `bsub` command.

Now, let's submit the previous command as a job using `bsub`.

```
bsub "sleep 60"
```

```
Returning output by mail is not supported on this cluster.
Please use the -o option to write output to disk.
Job <4015755> is submitted to default queue <normal>.
```

When you submit a job, it will be given a unique identifier (e.g. 4015755) which will help us with getting updates on what's happening with our job. To find out what jobs are scheduled and running, we can use another command, `bjobs`.

We can see how our job is getting on by running `bjobs`.

```
bjobs
```

```
JOBID   USER    STAT  QUEUE      FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
4015755 abc     PEND  normal     pcs5b                   sleep 10   Jan 15 14:06
```

This will give us the following information:

- **JOBID** - unique numerical job identifier used to keep track of the job
- **USER** - username of the person who submitted the job
- **STAT** - job state
- **QUEUE** - which queue the job was submitted to
- **FROM_HOST** - which host the job was submitted from
- **EXEC_HOST** - which host the job is running on (blank if job is pending)
- **JOB_NAME** - name of the job
- **SUBMIT_TIME** - when the job was submitted

Most jobs will have one of three states (STAT):

- **PEND** - the job is waiting in the queue to be scheduled and dispatched
- **RUN** - the job has been dispatched to a host (node) and is running
- **DONE** - the jobs finished normally (has an exit value of 0)

Occasionally, you may also see suspended job states:

- **PSUSP** - job was suspended by the owner or administrator while pending
- **USUSP** - job was suspended by the owner or administrator after being dispatched
- **SSUSP** - job was suspended by LSF after being dispatched

In this example, we can see that our job was submitted to the *normal* queue by default and that it hasn't started yet (*PEND*).

Let's check again.

```
bjobs
```

```
JOBID   USER    STAT  QUEUE      FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
4015755 abc     RUN   normal     pcs5b       pcs5c       sleep 10   Jan 15 14:06
```

Here we can see that our job has now started running (*RUN*) and is being executed on *pcs5c* (EXEC_HOST).

Let's wait a little longer and check one more time.

```
bjobs
```

```
No unfinished job found
```

Now we can't see any jobs. Why? That's because our job has finished running and we have no more jobs scheduled.

Did you notice the message that was returned when you submitted your job?

```
Returning output by mail is not supported on this cluster.
Please use the -o option to write output to disk.
```

This is because we used the default options and didn't specify an output file or an error file. We'll be looking at why printing the job outputs to files is good practice (and very useful!) in the next section.

You can find more information on job submission in general by looking through the job submission and job information sections of the LSF user manual.

4.1 What's next?

For another look at queues, you can go back to queues. Otherwise, let's take a closer look at managing jobs.

5 Managing jobs

In this section, we'll be taking a closer look at submitting jobs and how you can manage them once they've been submitted.

5.1 Writing job outputs to file

In the previous section, we looked at submitting a job using the default options using `bsub`.

```
bsub "sleep 60"
```

That submission returned a message:

```
Returning output by mail is not supported on this cluster.
Please use the -o option to write output to disk.
```

This message is saying that no matter whether the job succeeds or fails, we don't know what resources it used or if there were any errors because we didn't store that information anywhere. Not tracking information about your job and the outputs and errors it produces makes it difficult to troubleshoot any issues with the job execution.

What we can do instead is supply an output file using the `-o` option and an error file using the `-e` option.

```
bsub -o <output_file> -e <error_file> "command"
```

Let's give this a try. We'll call our output and error files *myjob.o* and *myjob.e*.

```
bsub -o myjob.o -e myjob.e "sleep 60"
```

You can check the progress of your job using `bjobs`.

```
bjobs
```

When the job has finished, print the contents of the output file to terminal using `cat`.

```
cat myjob.o
```

```
------------------------------------------------------------
Sender: LSF System <lsfadmin@pcs5a>
Subject: Job 4018040: <sleep 60> in cluster <pcs5> Done

Job <sleep 60> was submitted from host <pcs5a> by user <userA> in cluster <pcs5>.
Job was executed on host(s) <pcs5a>, queue <normal>, user <userA> cluster <pcs5>.
</nfs/users/nfs_u/userA> was used as the home directory.
</nfs/users/nfs_u/userA> was used as the working directory.
Started at Thu Jan 15 12:26:45 2019
Results reported on Thu Jan 15 12:27:45 2019
```

```
Your job looked like:

------------------------------------------------------------
# LSBATCH: User input
sleep 60
------------------------------------------------------------

Successfully completed.

Resource usage summary:

    CPU time :                                   0.82 sec.
    Max Memory :                                 5 MB
    Average Memory :                             5.00 MB
    Total Requested Memory :                     -
    Delta Memory :                               -
    Max Swap :                                   44 MB
    Max Processes :                              3
    Max Threads :                                4

The output (if any) is above this job summary.



PS:

Read file <myjob.e> for stderr output of this job.
```

Now, look at your error file using `cat`.

```
cat myjob.e
```

Printing our error file won't return anything as the error file was empty. This is because our job didn't have any errors. If it did, they would be logged in the error file and we could use it to try and trace what went wrong.

You can also incorporate your JOBID into the filename using a special variable **%J**.

```
bsub -o %J.o -e %J.e "sleep 60"
```

Let's say that the JOBID returned when you submitted the job was 4018041. Your output and error files would be called *4018041.o* and *4018041.e*.

You should always try to have different output and error files for each job your submitting. If you submit two jobs writing to *myjob.o* and *myjob.e* then it can get confusing as they are both writing to the same file.

5.2 Giving your job a name

You may have noticed that when you submitted your job earlier and checked its progress with `bjobs` that the JOB_NAME was *sleep 60*. This is because, by default, if no job name is given when you submit the job, the job name will become the command that you submitted.

```
JOBID   USER    STAT  QUEUE      FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
4018040 UserA   PEND  normal     pcs5a       pcs5b       sleep 60   Jan 15 13:26
```

You can give your job a different name using the `-J` option with `bsub`. This can be useful when you're running multiple jobs and want to be able to tell them apart in the queue with `bjobs`.

Let's try submitting a job called "newjob" which writes outputs and errors to *newjob.o* and *newjob.e*.

```
bsub -o newjob.o -e newjob.e -J newjob "sleep 60"
```

Let's look at the progress of our job using `bjobs`.

```
bjobs
```

```
JOBID   USER    STAT  QUEUE      FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
4018077 UserA   RUN   normal     pcs5a       pcs5b       newjob     Jan 15 13:34
```

Notice that the JOB_NAME is now *newjob*.

5.3 Submitting your job to a particular queue

If we don't specify a queue when we submit a job, the job will be submitted to the default queue. To find out which one of the queues is used by default, we can use the command `bparams`.

```
bparams
```

```
Default Queues:  normal
Default Host Specification:  BL465c_G8
MBD_SLEEP_TIME used for calculations:  10 seconds
Job Checking Interval:  15 seconds
Job Accepting Interval:  0 seconds
```

Running `bparams` will display information about the system parameters, such as the default queue name. In this example, the default queue is the *normal* queue. To find out which other queues are available, we can use `bqueues`.

```
bqueues
```

```
QUEUE_NAME       PRIO STATUS          MAX JL/U JL/P JL/H NJOBS  PEND   RUN  SUSP
system           1000 Open:Active       -    -    -    -     0     0     0     0
yesterday        500  Open:Active      20    8    -    -     0     0     0     0
small             31  Open:Active       -    -    -    -     0     0     0     0
normal            30  Open:Active       -    -    -    -   103    43    60     0
long               3  Open:Active      50    -    -    -   241   224    15     0
basement           1  Open:Active      20   10    -    -   182   164    18     0
```

Now, let's say that we want to submit a job into the *yesterday* queue because it's fairly urgent. To do this, we can use the `-q` option with `bsub` followed by the name of the queue which we want to use (e.g. *yesterday*).

Let's try submitting a job into the yesterday queue called "newjob1" which writes outputs and errors to newjob1.o and newjob1.e.

```
bsub -o newjob1.o -e newjob1.e -J newjob1 -q yesterday "sleep 60"
```

When you check on the progress with `bjobs` you will see that job is in the *yesterday* queue.

```
JOBID   USER    STAT  QUEUE      FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
4018119 UserA   RUN   yesterday  pcs5a       pcs5c       newjob1    Jan 15 13:51
```

5.4 Job resources

When we want to reserve more resources for our jobs, we can use the `-R`, `-M` and `-n` options with the `bsub` command. The `-M` option sets LSF the memory limit, `-n` sets the thread limit and the `-R` option tells LSF that the job needs to run on a host which matches the requirements which follow it.

Let's try submitting a job which has a limit of 2GB memory and 4 threads.

```
bsub -n 4 -R "span[hosts=1] select[mem>2000] rusage[mem=2000]" \
       -M 2000 "sleep 60"
```

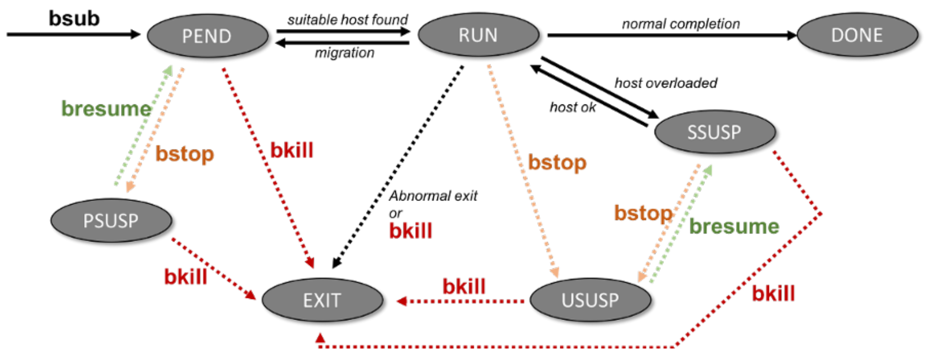
Here we can see that the 4 threads are reserved using the `-n` option. Typically when we ask for multiple threads with `-n`, we also add **span[hosts=1]** to the `-R` option. This indicates that all the processors which are allocated to this job must be on the same host.

We reserve our 2GB of memory using the `-M` options and `-R` option. Notice that the memory requirement is given in MB (2GB ~ 2000MB). With the `-R` option, we use a select string, **select[mem>2000]**, and a usage string, **rusage[mem=2000]**. The selection string specifies the characteristics that a host must have to match the resource requirement. In this case, more than 2GB memory. The usage string defines the expected resource usage of the job. By default, no resources are reserved.

For more information on resource requirements, please see the resource requirement section of the LSF user manual.

5.5 Job workflow

Once you have submitted your job, there are several different ways in which you can manage it. Below is a diagram which shows the typical job workflows and related commands.



LSF job workflow

Along the top of the diagram is the simplest job workflow. Here, a job is submitted using `bsub` and will have the status *PEND* until it gets dispatched.

```
JOBID   USER    STAT  QUEUE     FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
1000    userA   PEND  normal    pcs5b                   job1       Jan 15 14:06
```

Once the job is dispatched, it will start running and get the status *RUN*.

```
JOBID   USER    STAT  QUEUE     FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
1000    userA   RUN   normal    pcs5b       pcs5c       job1       Jan 15 14:06
```

If all goes well and there are no errors (normal completion) then the job finishes and has the status *DONE*.

```
JOBID   USER    STAT  QUEUE     FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
1000    userA   DONE  normal    pcs5b       pcs5c       job1       Jan 15 14:06
```

If there is a problem with a running job, this will trigger an abnormal exit and the status will become *EXIT*.

```
JOBID   USER    STAT  QUEUE     FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
1000    userA   EXIT  normal    pcs5b       pcs5c       job1       Jan 15 14:06
```

5.5.1 Cancelling jobs

Now, let's consider some deviations from this workflow. First, how do we cancel a job once it's been submitted?

```
JOBID   USER    STAT  QUEUE     FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
1000    userA   PEND  normal    pcs5b                   job1       Jan 15 14:06
```

We can cancel or kill a job 1000 using the command `bkill` followed by the JOBID of the job that you want to kill.

```
bkill 1000
```

If you have used a valid JOBID, the `bkill` command should return a message that tells you the job is being terminated.

```
Job <1000> is being terminated
```

Your job status will now get updated from *RUN* or *PEND* to *EXIT*.

```
JOBID   USER    STAT  QUEUE      FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
1000    userA   EXIT  normal     pcs5b                   job1       Jan 15 14:06
```

5.5.2 Suspending and resuming jobs

Let's say you are running a series of commands and you realise there's an error in the input file for one of those commands. When you're running a long job, you probably don't want to have to cancel it and start all over again. If the job hasn't reached that command, you can pause or suspend the job while you fix the input file and resume the job once you're done.

To suspend and resume a job you can use the `bstop` and `bresume` commands. First let's look at suspending a pending job (*PEND*).

```
JOBID   USER    STAT  QUEUE      FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
1000    userA   PEND  normal     pcs5b                   job1       Jan 15 14:06
```

To suspend this job, we use `bstop` followed by the JOBID.

```
bstop 1000
```

The job status will now become *PSUSP* as the job was suspended by a user while it was pending.

```
JOBID   USER    STAT  QUEUE      FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
1000    userA   PSUSP normal     pcs5b                   job1       Jan 15 14:06
```

To allow the job to be dispatched again, we can use the command `bresume` followed by the JOBID.

```
bresume 1000
```

The job status will now return back to *PEND* while the job waits to be dispatched.

```
JOBID   USER    STAT  QUEUE      FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
1000    userA   PEND  normal     pcs5b                   job1       Jan 15 14:06
```

You can also suspend a running job using `bstop`. In this case, the status will be updated to *USUSP* instead of *PSUSP*.

```
JOBID   USER    STAT  QUEUE      FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
1000    userA   USUSP normal     pcs5b                   job1       Jan 15 14:06
```

When you resume a previously running job that has been suspended with `bresume`, there may be an interim status of *SSUSP* before the job starts running again (*RUN*).

For more information on suspending, resuming and cancelling jobs, please see the controlling job execution section in the LSF user guide.

5.6 Moving a job to a different queue

Let's use as an example, a job which has been submitted to the *normal* queue and is waiting to be dispatched (PEND).

```
JOBID   USER    STAT  QUEUE      FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
1000    userA   PEND  normal     pcs5b                   job1       Jan 15 14:06
```

Now, let's say that we've made a mistake and that we know this submitted job will run for longer than the *normal* queue will allow. What can we do?

Well, you could kill the job with `bkill` followed by the JOBID of the job you want to cancel. You can then submit the job again specifying a different queue using the `bsub` option `-q` and the name of the queue you want to use. This would create a new job in a different queue (e.g. *long*).

```
bkill <JOBID>
bsub -q <queue_name> <command>
```

Alternatively, you can move the pending job to a different queue using `bswitch`.

```
bswitch <destination_queue> <JOBID>
```

So, to move our job (JOBID = 1000) from the *normal* queue (jobs killed after 12 hours) to the *long* queue (jobs killed after 48 hours) we would run:

```
bswitch long 1000
```

And if we looked again using `bjobs` we can see that the job has been moved into the `long` queue.

```
JOBID   USER    STAT  QUEUE      FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
1000    userA   PEND  long       pcs5b                   job1       Jan 15 14:06
```

For more information on moving jobs between queues, please see the switching queues section of the LSF user guide.

5.7 What's next?

For an overview of basic job submission, you can go back to job submission. Otherwise, let's take a look at job arrays and dependencies.

6 Job arrays and dependencies

In this section, we're going to take a look at how to submit a script instead of a command. We'll then use a script to help us submit a large number of identical tasks as a single job with a single JOBID. Finally, we'll take a look at how to submit a job that's dependent on another job.

6.1 Writing and submitting scripts

Earlier on, we were submitting a single command as a job.

Let's run that `bsub` command again.

```
bsub "sleep 60"
```

But, what if want wanted to submit several commands in a single job? For this, we can use a shell script. We can write a series of commands in a file (our script) which will read in a DNA sequence and substitutes the letters/bases to return the complimentary sequence (A -> T, T -> A, C -> G and G -> C).

Here, we've written the into a file called `myscript.sh`. We can use `cat` to print the contents of our script `myscript.sh` to the terminal.

Use `cat` to see the contents of *myscript.sh*.

```
cat myscript.sh
```

Notice that at the top we've added the line "#!/bin/bash" which tells the system the script should always be run with bash, rather than another shell.

```
#!/bin/bash

input=$(<data/sequence.txt)
echo "Input sequence: $input"

output=$(echo $input | tr 'ATGC' 'TACG')
echo "Complementary sequence: $output"
```

There's one thing that we need to do before submitting our script. Before we can submit our script, we need to make sure the system recognises our file as a script. We can do this using `chmod` to make our script executable.

Make *myscript.sh* executable.

```
chmod u+x myscript.sh
```

Try running *myscript.sh* directly in the terminal.

```
./myscript.sh
```

In your terminal there should now be a message with the original sequence, read from an input file by the script, and the corresponding complementary sequence.

```
Input sequence: AAAGGTTC
Complementary sequence: TTTCCAAG
```

To submit this script as a job, use `bsub`.

```
bsub -o myscript.o -e myscript.e "./myscript.sh"
```

Check the status of your job using `bjobs`.

```
bjobs
```

Once the job has finished, take a look at the output file which was generated.

```
cat myscript.o
```

At the top you should see the input and complementary sequences which means that your script executed as we'd expect.

```
Input sequence: AAAGGTTC
Complementary sequence: TTTCCAAG

------------------------------------------------------------
Sender: LSF System <lsfadmin@pcs5e>
Subject: Job 4019973: <./myscript.sh> in cluster <pcs5> Done

...
```

6.2 Submitting an array of jobs

Sometimes, you may want to run the same commands across lots of files. This is common in pipelines where your jobs will be identical except for the input data they are working on.

For example, let's say we want to find complementary sequences from in 5 different sequence files. We've named these sequence files so that a number from 1 to 5 is the suffix(sequence.txt.[1-5]).

Take a look at the files in the data directory using `ls`.

```
ls data
```

You should see files called *sequence.txt* followed by a number between 1 and 5.

```
sequence.txt    sequence.txt.2  sequence.txt.4
sequence.txt.1  sequence.txt.3  sequence.txt.5
```

We've provided a second script, *myarrayscript.sh*, which is almost the same as *myscript.sh* except that it includes an environment variable, **$LSB_JOBINDEX**, to tell the script which file to use as input.

Look at contents of *myarrayscript.sh* using `cat`.

```
cat myarrayscript.sh
```

Here, we can see the use of **$LSB_JOBINDEX** at the end of the input filename (infile). As LSF iterates through the job array, this number will increase so that the first job in the array will read in *sequence.txt.1* and the second job, _sequence.txt.2 ... and so on.

```
#!/bin/bash

infile=data/sequence.txt.$LSB_JOBINDEX
echo "Reading $infile"

input=$(<$infile)
echo "Input sequence: $input"

output=$(echo $input | tr 'ATGC' 'TACG')
echo "Complementary sequence: $output"

echo "Done"
```

Don't forget to update the permissions on the script so that we can use it.

```
chmod u+x myarrayscript.sh
```

The submission command below uses two special variables, **%J** and **%I**, in the output file names which represent the JOBID and the JOBINDEX. The JOBID will be the same for all of the jobs in this array and the JOBINDEX, which identifies which job was run from the array, will correspond to the number at the end of our filenames ($LSF_JOBINDEX).

Let's submit our job array.

```
bsub -J"sequence[1-5]" -o sequence.out.%J-%I ./myarrayscript.sh
```

Check the progress of the jobs.

```
bjobs
```

When you check on the progress using `bjobs` you will see five jobs running. Here is an example output from `bjobs`.

```
JOBID   USER   STAT  QUEUE     FROM_HOST  EXEC_HOST  JOB_NAME   SUBMIT_TIME
4019986 userA  RUN   normal    pcs5a      pcs5e      *quence[1] Jan 15 13:46
4019986 userA  RUN   normal    pcs5a      pcs5e      *quence[2] Jan 15 13:46
```

```
4019986 userA   RUN   normal     pcs5a       pcs5e       *quence[3] Jan 15 13:46
4019986 userA   RUN   normal     pcs5a       pcs5e       *quence[4] Jan 15 13:46
4019986 userA   RUN   normal     pcs5a       pcs5e       *quence[5] Jan 15 13:46
```

Notice here that there are five jobs, one for each of our input files, and that the JOBID is the same for all of the jobs (4019986). If we look at the JOB_NAME, LSF has added the JOBINDEX to the end of each of the job names (e.g. [1]).

Once your jobs have finished you can look for the output files using `ls`.

```
ls sequence.out*
```

You should see five files. Each of the filenames will begin with *sequence.out*. This is followed by the JOBID (e.g. 4019986), a hyphen and then the JOBINDEX. Remember, the JOBINDEX will be a number from 1 to 5 which corresponds to the number at the end of each of the input files. So, s_equence.out.4019986-1_ will correspond with *sequence.txt.1*.

Let's take a look at the first three lines of each of the output files using `head`.

```
head -3 sequence.out.*
```

```
==> sequence.out.4019986-1 <==
Reading data/sequence.txt.1
Input sequence: AGGCTA
Complementary sequence: TCCGAT

==> sequence.out.4019986-2 <==
Reading data/sequence.txt.2
Input sequence: TTGGCA
Complementary sequence: AACCGT

==> sequence.out.4019986-3 <==
Reading data/sequence.txt.3
Input sequence: CGCAAT
Complementary sequence: GCGTTA

==> sequence.out.4019986-4 <==
Reading data/sequence.txt.4
Input sequence: TTGCAA
Complementary sequence: AACGTT

==> sequence.out.4019986-5 <==
Reading data/sequence.txt.5
Input sequence: GGCCAA
Complementary sequence: CCGGTT
```

We can see that the same script (*myarrayscript.sh*) has been run on all five of our sequence files and we now have our complimentary sequences.

6.3 Job dependencies

Sometimes, we may have a job which uses the output from another job i.e. job B must only start after job A has finished successfully.

First, we submit jobA using `bsub`.

```
bsub -J jobA -o jobA.o -e jobA.e "sleep 180"
```

Check on jobA using `bjobs`.

```
bjobs
```

In this example, jobA has the JOBID 4020418 and has started running.

```
JOBID   USER    STAT  QUEUE      FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
4020418 UserA   RUN   normal     pcs5a       pcs5e       jobA       Jan 15 15:44
```

Now, let's create jobB which requires jobA to have finished before jobB is allowed to start running. To do this, we use the `-w` option with `bsub`. To say that we want jobB to start running only once jobA has finished, we use `-w 'ended(jobA)'`. Don't forget to put you dependency requirement inside quotes.

Let's submit jobB.

```
bsub -w 'ended(jobA) ' -J jobB -o jobB.o -e jobB.e "sleep 60"
```

Use `bjobs` to check on jobB.

Now, we can see that jobB is waiting (PEND), but how do we know it is waiting for jobA to finish?

```
JOBID   USER    STAT  QUEUE      FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
4020418 UserA   RUN   normal     pcs5a       pcs5e       jobA       Jan 15 15:44
4020421 UserA   PEND  normal     pcs5a                   jobB       Jan 15 15:45
```

You can take a look at the longer output with `bjobs -l`. Make sure to substitute the JOBID in the example below with the JOBID of your jobB.

```
bjobs -l 4020421
```

In the output, we can see the section *PENDING REASONS* which tells us that the job dependency isn't satisfied (i.e. jobA hasn't finished yet).

```
...

 PENDING REASONS:
 Job dependency condition not satisfied;

...
```

We can take a closer look at the dependency conditions using `bjdepinfo -l`.

```
bjdepinfo -l 4020421
```

This tells us that jobB (4020421) depends on jobA (4020418) being finished (ended). It shows the relationship between the two jobs, i.e jobA is the parent job and jobB is the child job.

```
The dependency condition of job <4020421> is not satisfied: ended(jobA)
JOBID          PARENT         PARENT_STATUS  PARENT_NAME  LEVEL
4020421        4020418        RUN            jobA         1
```

6.4 What's next?

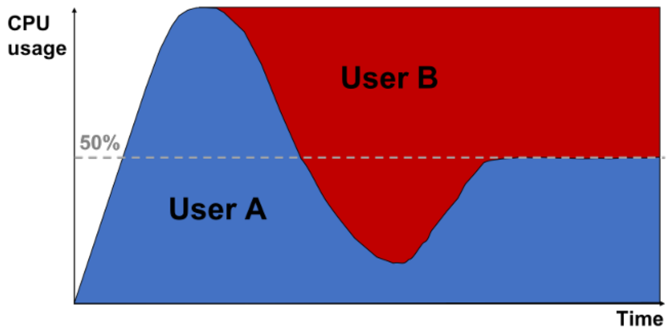
For an overview of managing jobs, you can go back to the managing jobs. Otherwise, let's take a look at priority_and_fairshare.

6.5 Priority and fairshare

Priority and fairshare make sure that everyone can get at least some jobs running and that everyone gets an equal share of the cluster to use over time.

Using an example from the Sanger Farm training course, we'll consider a "first come, first served" situation where user A submits 1000 jobs and then some time shortly after, user B submits another 1000 jobs. In this situation, user B would have to wait for all of the jobs submitted by user A to finish before the first of user B's jobs can begin. That just wouldn't be very fair.

However, with fairshare, the user's priority changes over time based on their recent usage. This means that users which have used the cluster recently (i.e. in the last 48 hours) will have a lower priority than new users when they submit their next job.



Fairshare

Consider two users, User A and User B, who start off with the same priority. User A starts running some jobs and gets the full share of available CPUs. User B now starts submitting some jobs. Initially, User B gets a smaller proportion of CPUs than User A. As the jobs submitted by User A start to finish, User B starts to get a larger share of the CPUs. This is because User A now has a lower priority. Eventually, over a period of 48 hours or so, User A and User B will have equal priorities again and will have used an equal proportion of the cluster resources.

At Sanger we use **hierarchical fairshare** which means that the initial priority for each user will also depend on their group membership. This is because the system needs to accommodate the needs of both the pipelines and the users.

Priority will only affect the decision on which jobs run next. It will *not* affect jobs that are already running. Sanger users can find more information about fairshare here.

You can check your priority in a particular queue using `bqueues` and the `-r` option followed by the name of the queue that you want to look at.

```
bqueues -r <queue_name>
```

```
USER/GROUP    SHARES   PRIORITY  STARTED  RESERVED  CPU_TIME  RUN_TIME    ADJUST
userA            1        0.333        0         0       0.0         0     0.000
userB            1        0.211        0         0    7450.8      8924     0.000
userC            1        0.104        0         0   11218.3     34168     0.000
userD            1        0.044        0         0   72918.9    100394     0.000
userE            1        0.003       16         0  205565.6   1369196     0.000
```

6.6 What's next?

For an overview of jobs arrays and dependencies, you can go back to job arrays and dependencies. Otherwise, let's take a look at troubleshooting.

7 Troubleshooting

7.1 My job failed. How do I find out what went wrong?

There are many different reasons why a job might have failed. The first place to check are the output and error files.

Let's say we submitted the wrong command, writing `slep 10` instead of 'sleep 10.

```
bsub -o bad_cmd.o -e bad_cmd.e "slep 10"
```

Use `bjobs` to see when your job has finished running. The jobs status (STAT) will be *EXIT*. This means that your job had an abnormal exit and there was an issue. Below is an example.

```
JOBID   USER    STAT  QUEUE      FROM_HOST    EXEC_HOST    JOB_NAME    SUBMIT_TIME
1000    userA   EXIT  normal     pcs5a        pcs5c        slep 10     Jan 15 10:48
```

Now, print the contents of the output file to the terminal using `cat` (probably best to use `less` if the file is larger).

```
cat bad_cmd.o
```

```
------------------------------------------------------------
Sender: LSF System <lsfadmin@pcs5c>
Subject: Job 4017581: <slep 10> in cluster <pcs5> Exited

Job <slep 10> was submitted from host <pcs5a> by user <userA> in cluster <pcs5>.
Job was executed on host(s) <pcs5c>, queue <normal>, user <userA> cluster <pcs5>.
</nfs/users/nfs_u/userA> was used as the home directory.
</nfs/users/nfs_u/userA> was used as the working directory.
Started at Thu Jan 15 10:48:46 2019
Results reported on Thu Jan 15 10:48:47 2019

Your job looked like:

------------------------------------------------------------
# LSBATCH: User input
slep 10
------------------------------------------------------------

Exited with exit code 127.

Resource usage summary:

    CPU time :                                   0.09 sec.
    Total Requested Memory :                     -
    Delta Memory :                               -
```

```
The output (if any) is above this job summary.



PS:

Read file <bad_cmd.e> for stderr output of this job.
```

This tells us that the job `Exited with exit code 127`. Any exit code > 0 means there was an error. In this case, it was error code 127 which means that the command we tried to run, `slep`, could not be found.

Here is an overview of the possible exit codes:

- **less than 127** - there was an exit code from your script or command
- **127** - the command was not found / doesn't exist
- **more than 127** - the job was killed by a signal

When a job gets killed by a signal (error code > 127), you can subtract 128 from the error code to get the signal number. You can then run `man 7 signal` to find the meaning of that signal. Error codes 130 and 140 will typically mean you your job exceeded the resources you requested. Try submitting the job again, requesting more memory or to a queue with a longer time limit.

For more information about exit codes, please see the job exit codes, job exception and exit information sections in the LSF user guide.

We can see the error that was generated by printing the contents of the error file to the terminal with `cat`.

```
cat bad_cmd.e
```

```
/tmp/1547722126.4017581: line 8: slep: command not found
```

Here we can see this confirms that the system couldn't find the command `slep`. Whenever you have an issue with an LSF job and need to contact your support team for help, it is always a good idea to give them the JOBID and/or the location of the output and error files. This makes tracking down the issue much quicker!

7.2 Why can't I submit any more jobs to a particular queue?

Some queues have limits, such as the yesterday queue, while others, like the normal queue, have no limit. You can check the queue limits using `bqueues`.

```
bqueues
```

Each queue has a maximum number of **job slots** which can be used by scheduled jobs in the queue. Job slots are reserved by jobs which are pending and used by those which have already started running but are not yet finished. In this example we can see the maximum number of job slots for each queue by looking at the **MAX** column.

```
QUEUE_NAME      PRIO STATUS          MAX  JL/U JL/P JL/H NJOBS  PEND   RUN  SUSP
system          1000 Open:Active       -    -    -    -     0     0     0     0
yesterday       500  Open:Active      20    8    -    -     0     0     0     0
small            31  Open:Active       -    -    -    -     0     0     0     0
normal           30  Open:Active       -    -    -    -    35    13     1     0
long              3  Open:Active      50    -    -    - 31686 31636    46     0
basement          1  Open:Active      20   10    -    -   180   170    10     0
```

Sometimes there are also limits on the job slots that are available to users. You can check this by looking at the **JL/U** column. In this example, the yesterday queue is limited to a maximum of 8 job slots per user and the basement to 10 job slots per user.

For more information on queues, please see queues.

7.3 I've lots of submitted jobs, how can I get my high priority job running first?

There are two things that you can do to influence the order in which your pending jobs will be run. First, you can move your job using **`bswitch`**.

```
JOBID   USER    STAT  QUEUE      FROM_HOST  EXEC_HOST   JOB_NAME   SUBMIT_TIME
1000    userA   PEND  normal     pcs5b                  job1       Jan 15 14:06
```

We can take a look at which queues are available using `bqueues`.

```
QUEUE_NAME      PRIO STATUS          MAX  JL/U JL/P JL/H NJOBS  PEND   RUN  SUSP
system          1000 Open:Active       -    -    -    -     0     0     0     0
yesterday       500  Open:Active      20    8    -    -     0     0     0     0
small            31  Open:Active       -    -    -    -     0     0     0     0
normal           30  Open:Active       -    -    -    -    35    13     1     0
long              3  Open:Active      50    -    -    - 31686 31636    46     0
basement          1  Open:Active      20   10    -    -   180   170    10     0
```

Look at the priorities of the queues (PRIO) to try and find a queue with a higher priority. Here, the *yesterday* queue has a higher priority (500) compared to the normal queue (30). So, we can try moving our job from the normal queue into the yesterday queue.

```
bswitch 1000 yesterday
```

If we then ran `bjobs` we would see our job has been moved to the yesterday queue.

```
JOBID   USER    STAT  QUEUE      FROM_HOST  EXEC_HOST   JOB_NAME   SUBMIT_TIME
1000    userA   PEND  yesterday  pcs5b                  job1       Jan 15 14:06
```

Alternatively, you can use `btop` to move a pending job to the top of your job list. In the example below, we have used `bjobs` which showed us that we have 4 jobs scheduled: 1 running and 3 pending.

```
JOBID   USER    STAT  QUEUE      FROM_HOST  EXEC_HOST   JOB_NAME   SUBMIT_TIME
1000    userA   RUN   normal     pcs5b      pcs5c       job1       Jan 15 14:06
1001    userA   PEND  normal     pcs5b                  job2       Jan 15 14:07
1002    userA   PEND  normal     pcs5b                  job3       Jan 15 14:08
```

```
1003     userA    PEND   normal     pcs5b                        job4        Jan 15 14:09
```

All of these jobs are identical, so *job2* will be the first of our pending jobs to be executed when the necessary resources become available. But, what if we needed *job4* to be executed first?

```
btop 1003
```

Using `btop` followed by the JOBID will move the job (e.g. job4) to the top of the list of jobs waiting to be executed.

```
JOBID    USER     STAT   QUEUE      FROM_HOST   EXEC_HOST    JOB_NAME    SUBMIT_TIME
1000     userA    RUN    normal     pcs5b       pcs5c        job1        Jan 15 14:06
1003     userA    PEND   normal     pcs5b                    job4        Jan 15 14:09
1001     userA    PEND   normal     pcs5b                    job2        Jan 15 14:07
1002     userA    PEND   normal     pcs5b                    job3        Jan 15 14:08
```

For more information on job management, please see job management.

7.4 All of my jobs are pending. Why can't I get anything running?

The first thing to do here is check how busy the cluster is using **bqueues**.

```
bqueues
```

Most commonly, your jobs will be pending because the cluster is busy and it may just take some time for things to get running. Once other people's jobs start to finish, resources will become available and your jobs should start running.

If this keeps happening, there are two things to consider. The first is whether you've requested more resources than your job is likely to require. Let's use the `sleep` command as an example. Well reserve 2GB (2000MB) of memory for this job.

```
bsub -o sleep.o -e sleep.e -R 'select[mem>2000] rusage[mem=2000]' \
       -M 2000 "sleep 10"
```

Once that has finished, take a look at the output file (sleep.o).

```
cat sleep.o
```

We want to look at the amount of resources our job used.

```
Resource usage summary:

    CPU time :                                   0.23 sec.
    Max Memory :                                 6 MB
    Average Memory :                             6.00 MB
    Total Requested Memory :                     2000.00 MB
    Delta Memory :                              1994.00 MB
    Max Swap :                                  44 MB
    Max Processes :                             3
    Max Threads :                               4
```

Here we can see that this job used only used 6MB of memory (Max Memory). We requested 2GB (2000MB) which was 1994MB more than our job required (Delta Memory).

You should always try to request only the resources that you need. If you're running a large analysis, try running the analysis on a small subset of the data and scale up to estimate the resources you'll require.

Alternatively, it could be because your priority is low. Perhaps you or other members of your group have been running a lot of jobs or using a lot of resources in the last 48 hours. This can decrease your priority and chances of getting jobs running. For more information, please see priority and fairshare.

7.5 I made a mistake when I submitted my job, can I update it?

If the job is already running, it's probably best to cancel it with `bkill`, update the command or script and then submit it again as a new job.

```
bkill <JOBID>
```

However, if your job is pending, you can use **`bmod`** to update your job.

```
bmod <options_or_command_to_update> <JOBID>
```

To update the command, you can use the `-Z` option. For example, if we made a mistake an spelt the `sleep` command wrong and ran `slep`:

```
bsub "slep 60"
```

Now let's say it submitted the job, we now have a JOBID of 1000 and that the job is pending.

```
JOBID   USER    STAT  QUEUE      FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
1000    userA   PEND  normal     pcs5b                   slep 60    Jan 15 14:06
```

We can update the command using `bmod` with the `-Z` option, followed by the JOBID (1000), to update the job to run the correct command.

```
bmod -Z "sleep 60" 1000
```

When the job is dispatched and executed, it will now run `sleep 60` instead of `slep 60`. You can change many other job parameters using `bmod` such as the job name.

Let's call our job something more useful like "sleepyjob" using the `-J` option.

```
bmod -J sleepyjob 1000
```

We can add an output and error file too using the `-o` and `-e` options.

```
bmod -o sleepyjob.o -e sleepyjob.e 1000
```

Or, we can update the resources it's requesting using the `-M`, `-R` and `-n` options.

```
bmod -n 2 -R "span[hosts=1] select[mem>2000] rusage[mem=2000]" -M 2000 1000
```

This would be updating job 1000 and asking LSF to reserve 2 cores/threads and 2GB (2000MB) memory.

7.6 What's next?

For an overview of priority and fairshare, you can go back to the priority_and_fairshare. Otherwise, you can take a look at our LSF cheat sheet.

8 Cheat sheet

Below is a list of commands and their meaning taken from the LSF user guide here.

8.1 Cluster overview

- **lshosts** - Displays information about hosts in the cluster
- **bhosts** - Displays information about hosts and their resources
- **lsid** - Displays the cluster name
- **lsclusters** - Displays cluster status and size

8.2 Queue information

- **bqueues** - Displays information about queues

8.3 Job execution management

- **bjobs** - Displays information about scheduled jobs
- **bsub** - Submits job(s) for execution
- **bstop** - Suspends job(s)
- **bresume** - Resumes suspended job(s)
- **bkill** - Cancels scheduled job(s)
- **bswitch** - Moves job(s) to a different queue
- **bmod** - Modifies the properties of pending job(s)
- **btop** - Moves job(s) relative to your first job in the queue
- **bbot** - Moves job(s) relative to your last job in the queue

8.4 General information

- **bacct** - Displays accounting statistics about finished jobs

8.5 bqueues

Get more information about a particular queue:

```
bqueues -r <queue_name>
```

8.6 bsub

Submit a job:

```
bsub <command>
```

Submit a job to a particular queue:

```
bsub -q <queue_name> <command>
```

Submit a job which writes output and errors to file:

```
bsub -o <output_file> -e <error_file> <command>
```

Submit a job reserving 1GB (1000MB) memory:

```
bsub -M 1000 -R 'select[mem>1000] rusage[mem=1000]' <command>
```

Submit a job requiring 8 threads:

```
bsub -n 8 <command>
```

8.7 bjobs

List jobs owned by all users:

```
bjobs -u all
```

List jobs owned by a particular user:

```
bjobs -u <username>
```

List jobs for all users in a particular queue:

```
bjobs -u all -q <queue_name>
```

List your pending jobs:

```
bjobs -p
```

List your running jobs:

```
bjobs -r
```

List your finished jobs:

```
bjobs -d
```

List your suspended jobs:

```
bjobs -s
```

Get more information about a particular job:

```
bjobs -l <JOBID>
```

8.8 bstop

Suspend a job:

```
bstop <JOBID>
```

8.9 bresume

Resume a job:

```
bresume <JOBID>
```

8.10 bkill

Cancel a job:

```
bkill <JOBID>
```

8.11 bswitch

Move job to a different queue:

```
bswitch <destination_queue> <JOBID>
```

8.12 bmod

Modify the parameters of a pending job:

```
bmod <options> <JOBID>
```

Update the job name:

```
bmod -J <new_job_name> <JOBID>
```

Update the output and error file paths:

```
bmod -o <output_file> -e <error_file> <JOBID>
```

Update the job resources:

```
bmod -M <memory_in_MB> -R 'select[mem><memory_in_MB>] rusage[mem=<memory_in_MB>]' <JOBID>
```

8.13 btop

Move job to the top of your job list:

```
btop <JOBID>
```

8.14 bbot

Move job to the bottom of your job list:

```
bbot <JOBID>
```