Using the UCI “biocluster’s” queuing system with Grid Engine (GE)

Kevin Thornton
krthornt@uci.edu
June 22, 2012

Intro
The “biocluster” is a set of networked computers plus storage devices that are linked together via a high-speed private network (connecting nodes to each other, and various nodes to storage). One logs into the *login* node, which is hpc.oit.uci.edu. Users request accounts from Joseph Farran (jfarran@uci.edu).

The main feature of the cluster is that it manages user activity via a queuing system called “Grid Engine”, or GE for short. The main job of the queue is to match up a user’s request for resources with the resources that are currently both free on the system, and available to the user. If sufficient resources are available, jobs are executed in a first-in/first-out manner.

Joseph has some initial documentation here: http://hpc.oit.uci.edu/. It is very important that users read this, especially with respect to how storage is organized.

Please note that simply reading this document won’t get you up to speed--you need to put the time in and work through real examples. The work is well-rewarded: the queuing system will work to ensure that your jobs are executed as resources become available, allowing you to chug through a ton of computation in a short amount of time.

The storage
As usual, logging in puts you into your home directory, which is /data/users/username.
You do not have much storage in this folder, and the storage that is there is slow.

Instead of reading/writing stuff to your home folder, you are strongly encouraged to use the bio storage array, which has mount point /bio.

Each user in the “bio” group has a folder on the large storage array, using the user’s NetID as the folder name. My folder is called:

```
/bio/krthornt
```

Your research group may have its own storage device--this is something that the PI would have purchased, and the PI can tell you the mount point.

Practical considerations
The cluster is **<u>not</u>** a long-term storage/backup machine. It is for research, and users are expected to move output from the cluster to somewhere else for archival and/or further processing. This means that, if you have a ton of high-throughput sequencing data, *you probably need a large RAID server for your lab*, to which you can move your files.

To be totally blunt: it would be foolish for a lab to analyze large data sets on the biocluster without investing in their own storage & backup system.

Other considerations
It is likely that the storage situation on biocluster will be in flux in the future. Long-term, biosci users will need massive amounts of storage to handle the output of high-throughput genomics pipelines. We currently aren't there yet.

The queues

Joseph has written about the available queues here: http://hpc.oit.uci.edu/

Using the queues

A workflow consists of the following:
1. A user writes a *script* which specifies the commands to be executed, and what resources are required
2. That script is submitted to a queue. By default, jobs go to the free queue, but a user may name a specific queue at submission time
3. If all goes well, the jobs run to completion, and you are done.

In this section, we break down the first two parts of the workflow.

The scripts
The script is a list of commands that will be executed by the system. The computing environment is Linux (a Unix variant), and the default language that the system understand is called "bash". All "bash" is is a set of conventions that the command-line interface understands--this includes basic commands like "cd" to change directories, "ls" to list the contents of directories, etc. It also contains a set of variables specifying things about the user environment, etc., which affect the behavior of the system. Finally, it is possible to write simple *programs* using bash, which control the flow of the execution of commands. Essentially, the script which defines your job for the queue is a program written in the bash language, with some extensions to tell the queueing system about a few things that it needs to know.

I now have to state a hard truth: if you are not familiar with the Linux environment, you will quickly get lost. It is basic knowledge of the bioinformatics 101 sort, and is beyond the scope of this document (or me personally) to help you out with. Here are two good online resources:
1.http://tldp.org/HOWTO/Bash-Prompt-HOWTO/ covers the use of bash as a command-line interface to a Linux system.
2.http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html covers the basics of using bash as a programming language.

In the rest of this document, I will casually refer to many terms that are specific to working in a Linux environment. If you get lost, google. There will be a learning curve for some of you.

A very simple script
In this section, we will write a simple script that will do the following:
1. It will list the contents of our home directory
2. It will write the list of contents of our home directory to a file called “mystuff.txt”. That output file will be placed in the directory from which we submit the script to the queue.

Here is the script (in courier font so that you can see all the spacing, which matters a great deal!!!):

```
#!/bin/bash

#$ -N MYSTUFF
#$ -q krt

ls ~/ > mystuff.txt
```

If one wrote (or copy/pasted) the above text into a file called “job.sh”, then that is all you need to do to create your first script. [Note: if you don’t know how to create/edit files on a Unix system, then you need to learn about text editors, such as emacs, vi, etc. That is homework for you.]

Let us break down the script, line-by-line:

Line	Function
`#!/bin/bash`	Tells the *command interpreter* that the script is in the bash language. NOTE: this is the first line in 100% of scripts that you will write. No exceptions, unless you really know what you are doing.
`#$ -N MYSTUFF`	This tells the GE system that the name of the job is “MYSTUFF”
`#$ -q krt`	Send it to the Thornton lab queue.
`ls ~/ > mystuff.txt`	This does the work: get the contents of the home folder, and redirect output to a file called “mystuff.txt”. The output file will be written to the directory $SGE_O_WORKDIR, *which defaults to the directory from which the job is submitted.*

So now we’ve learned a couple of things:
1. The simplest script is just an embellished list of commands
2. These `#$` lines affect how your script relates with/is affected by the queuing system.
3. There are environment variables that the GE system sets automatically, such as `SGE_O_WORKDIR`, which is a variable whose value is the directory from which the script

was submitted. A full list of environment variables can be found in the man page for the command `qsub`.

Submitting a job to the queue
You now have a file called "job.sh", and you want to submit it to the free queue. Simply type

```
qsub job.sh
```

That's it--done.

To submit it to the "bio" queue:

```
qsub -q bio job.sh
```

The command "qsub" simply takes a script/job, and sends it to the queue.

NOTE: If you specify a queue in the script, and again when executing a `qsub` command, the job will be sent the queue specified when executing `qsub`.

Checking the queue status
The command "qstat" gives the current status of the queue (this is **NOT** the same as "top").

As of right now, qstat tells us:

```
[krthornt@hpc ~]$ qstat
job-ID  prior   name       user         state submit/start at     queue                          slots ja-task-
ID
-----------------------------------------------------------------------------------------------------------------
--
   1330 0.55500 e0.1.k5.r1 krthornt     r     06/18/2012 23:54:16 krt@compute-2-1.local              1 93
   1334 0.55500 e0.5.k5.r1 krthornt     r     06/20/2012 02:12:05 krt@compute-2-1.local              1 75
   1338 0.55500 e1.k5.r100 krthornt     r     06/20/2012 19:54:42 krt@compute-2-1.local              1 45
   1338 0.55500 e1.k5.r100 krthornt     r     06/21/2012 06:57:54 krt@compute-2-1.local              1 89
```

My user name is "krthornt", and I have a few simulations running on my own queue. The "r" in the "state" column means that they are running.

The first column is an ID number, which is unique for each job. My job numbers are 1330, 1334, and 1338. You will see that there are two 1338s, which is because they are two different "tasks" in an "array job", which is a topic we'll cover later.

The job ID is useful, especially if you've made a mistake. If you want to delete a job from the queue, type

```
qdel jobid
```

which works for both array and non-array jobs.

Using multiple priority queues at once

There are two basic kinds of nodes on the biocluster: those that are owned by a specific lab group, and those contributed to the cluster using funds from the School of Biological Sciences. The former type of node belongs to queues whose names were decided upon by the various labs (my queue is called "krt"). All nodes on the cluster are part of a queue called "bio".

Without going into too much detail, if you are running and using all of the "bio" nodes, I (or someone in my lab) can come along and submit to the "krt" queue, which will pause your jobs there, run mine, and your jobs wait until I'm done.

Even better, I am able to do this in my scripts:

```
#$ -q krt,bio
```

which submits to both the queues at once. This means that I'm always able to have quick access to my own nodes, while being able to scavenge available resources on the general bio queue at the same time. [This is why it is good for PIs to contribute nodes to the cluster--everyone wins.]

There is also a "free" queue, which consists of all HPC 64-core machines. The details are here: http://hpc.oit.uci.edu/free-queue. The short of it is that, if a machine is not being used, it is part of a queue called "`free64`", and includes machines owned by biosci, physical sciences, and engineering. To use the `free64` queue:

```
#$ -q free64
```

Why would you want to use the free queue? If the bio queue is currently full, it may be a way to get your jobs started more quickly. The `free64` queue is also a way to get as many cores as possible for a big array job. For example, I can do this:

```
#$ -q krt,bio,free64
```

The effect of that queue request is the following:

1. The scheduler will first try to put jobs on my queue (`krt`), which will suspend and jobs submitted to any other queue that landed on my nodes.
2. The GE scheduler will then try to put jobs on the `bio` nodes, suspending the jobs of any users with lower priority on that queue.
3. The scheduler will then survey the nodes participating in the `free64` queue, and attempt to grab some cores from there. Any jobs that end up running on a "free" node risk being suspended if/when a higher-priority user comes along.

In practice, submitting to all three queues like this has given me access to over 500 cores, so the benefit is clear, especially for single-core jobs.

Email notification

You can have the GE system email you when something happens with your job:

```
#$ -m bea
```

will send an email to your UCI netID at the (b)eginning or (e)nd of a job, or if a job (a)borts for some reason.

Warning: this is a good way to fill up your inbox! This is likely best used in combination with email filters and folders that grab up messages from the system.

Gory details

Modules

A lot of us use custom C/C++ code in our research. Running such code requires that the system can find the relevant software libraries, etc. One library that the Thornton lab uses is called “boost” (http://www.boost.org), which is a C++ library. If you have a program that requires a *run-time* boost library (as opposed to simply the compile-time headers) to run, add the following line to your script:

```
module load boost/1.49.0
```

at some point after all the `#$` business and before the line running the program depending on boost.

To see a complete list of modules, type the following on the login node:

```
module list
```

For some reason, the GNU Scientific library (GSL), is not a module, but is in the standard /usr/include and /usr/lib locations for the headers and run-time libraries, respectively.

Custom use of run-time libraries

Say you’ve installed some other library in your home folder, and that library is needed by programs that you will run on the queue. Assuming that you’ve installed the library in your “lib” directory (*e.g.,* /data/username/lib), add the following line to your script:

```
LD_LIBRARY_PATH=$HOME/lib:$LD_LIBRARY_PATH
```

at some point after all the `#$` business and before the line running the program which depends on libraries in your home folder.

I will note here that compiling custom code on the login node can be a painful task: the combination of the module system, plus not having privileges to install custom libraries outside of your home folder, means that you've got to learn the details of how to manipulate the compiler's environment variables. That will be the subject of another document.

Using array jobs to automate serial tasks

From a cluster-computing standpoint, there are two basic types of jobs that a computer can be asked to do:
1. A job can be done that uses multiple processors at once. This is usually done via a fancy program that a credible programmer has written, taking advantage of things like threads, MPI, etc. One example is the alignment of short reads to a reference genome using programs like bwa, Mosaik, etc. These aligners can use multiple CPU at once.
2. A job that consists of many repetitive tasks, each of which can be done on a single CPU. This is a task which can be "serially" parallelized. A good example is computer simulation. Why use 1 CPU to do 1000 replicates of a parameter set when you could use 1000 CPU to each to 1 replicate.

In this section, I'm going to tackle #2, the case of serially-parallelized jobs. Let's imagine that we want to take Dick Hudson's coalescent simulation "ms", and generation 1,000 replicates of a sample of size 10 from a Wright-Fisher population with mutation rate θ = 10, recombination rate ρ = 10, and the length of the gene is 1,000 nucleotides. I'm also going to assume that the binary for "ms" is in my "bin" folder (*e.g.,* ~/bin).

The usual command would be:

```
~/bin/ms 10 1000 -t 10 -r 10 1000 > output
```

and it would run in a few seconds. How can we use all available nodes in a particular queue to get it done even faster? The answer is an "array job", and it is best to learn by seeing an example:

```
#!/bin/bash

#$ -N RUNMS
#$ -t 1-1000

~/bin/ms 10 1 -t 10 -r 10 1000 > $SGE_O_WORKDIR/output.
$SGE_TASK_ID
```

The above script creates a job called "RUNMS", and the "ms" command itself has been changed to run just a single replicate (that's the second integer passed to the program), and the output file has now been changed to $SGE_O_WORKDIR/`output.`
`$SGE_TASK_ID`.

So, how does this work? The answer is in this line:

```
#$ -t 1-1000
```

which tells the GE system that the job is an “array” of job consisting of 1,000 tasks. A *task* is simply an instance of the script in the queue--this script will be executed in the queue 1,000 times. Further, each instance will be assigned a unique ID number, which is stored in the variable `SGE_TASK_ID` There is no magic to the values taken on by this variable--they will be 1 through 1,000, in order.

The above script uses the `SGE_TASK_ID` variable to ensure that the output of each replicate is written to a unique file. If you’re following along, you should realized that the full path specification of each output file will be:

```
$SGE_O_WORKDIR/output.$SGE_TASK_ID
```

Array jobs are awesome. You can automate a tremendous amount of work while keeping the script short. The following sections will cover some special tricks regarding their use. These sections will cover some points which are entirely practical, and are related to how one can set up array jobs to get a lot of work done.

Limiting your use of resources

Let us imagine that you have a large number of jobs in an array job, and each task could take several days to complete. It would be a bit rude to take up the entire bio queue with such a job array.

Adding the following line to your script:

```
#$ -tc X
```

will limit concurrent execution of tasks within an array job to a maximum of X tasks.

Let us say that you are a user in a group which has access to a queue other than bio. Let’s use the krt queue as an example, and assume that queue consists of a single, 64-core node. Saying the following:

```
#$ -q krt,bio
#$ -tc 128
```

will have the effect of filling up the krt queue with 64 jobs, and trying to fit another 64 jobs on a node on the bio queue. Basically, I’m trying to limit myself to using 2 nodes total for my array job, one of which is in the pool of bio nodes.

Resource requests

Requesting multiple execution cores

There are two types of programs that a user is likely to run on a cluster:

1. A "single-core" program, which executes only on one processor
2. A "multi-core" program, which is able to execute using multiple processors simultaneously. An example of this is the `bwa` aligner, whose "`bwa aln`" command is capable of using multiple cores/CPU at once.

[For the sake of simplicity, I will assume here that all "multi-core" are implemented using *threads*, which means that multiple CPU/cores on the same node are used. If your program uses something like MPI to communicate across multiple processors, you'll need to look elsewhere (e.g., talk to Joseph and/or Harry).]

By default, a GE job assumes that only a single-core is needed. When this is true, a user doesn't have to add anything to a script. However, let's say that you want to use 64 cores to run "`bwa aln`". In order to do this, you need to use something called a *parallel environment* to request 64 cores on the same node. One does this by adding the following line to a job script:

```
#$ -pe openmp 64
```

which tells the GE system to load the "parallel environment" called "openmp", and request 64 cores. The resource request is done such that all 64 cores are *on the same node.*

When you request a certain number of cores, the number of core is stored in two different shell variables. These are `CORES` and `OMP_NUM_THREADS`. The latter is used for programs using the MPI method of parallelizing code. The former is handy for making your scripts a little more generic, *e.g.,*

```
#!/bin/bash
#$ -pe openmp 32
module load bwa/0.6.2
bwa aln -t $CORES [ rest of command line goes here ]
```

The above will use 32 cores to align data using the "bwa" aligner.

[If you care, other parallel environments supported by GE (such as MPI) do not require/guarantee that all cores requested are on the same node. This is because those other parallel environments use the network to allow the cores to work in cahoots with one another, which is great because you could theoretically use hundreds of cores to get work done.]

Requesting a range of cores

You may have a program that could use up to 64 cores, but there is no guarantee that a node is currently free, or will be free in the foreseeable future, with 64 cores free. Let's assume that you are willing to live with a minimum of 4 cores for a bwa job, but would like 64 if they are available:

```
#!/bin/bash
#$ -q bio
#$ -pe openmp 8-64
module load bwa/0.6.2
bwa aln -t $CORES [ rest of command line goes here ]
```

Now, your alignments will run with as many cores as they can grab on a free node. Using the `openmp` environment in this way is "greedy", and will request as many cores as are free on a node. However, because of the way the scheduler works (which is complicated…), if a node has, say, 24 cores available now, the above script will get 24 cores on that node. If you monitor the queue, and see that that would happen, you can perform an on-the-fly modification to the script when you submit it:

```
qsub -pe openmp 64 scriptname.sh
```

which will force the job to launch on a node with 64 free cores. [Hint: you just learned that parameters passed to `qsub` will over-ride GE parameters set in your script. That is a good thing to know…]

Reserving cores for multi-core jobs.

[NOT SURE WHERE WE ARE HERE…]

A common problem on systems like the biocluster is the following:

1. User A has a large number of array jobs, each sub-task of which uses a single core. Currently, user A is running a task on all available nodes on the bio queue
2. User B as an array job consisting of multi-core tasks. For the sake of argument, assume each job will request 8 tasks.

The problem is that user A is "flooding" the queue (although in reality not doing anything wrong *per se*). User B is affected because it is rather unlikely that 8 cores will become available all at once on a single node. So, even though the scheduler has moved user B to the top of the priority queue, user B's jobs will not run. It is our current belief that adding a "reservation" request to your script modifies the scheduling behavior to allow your job to run:

```
#$ -R y
```

The testing that we've done suggests that reserving works.

Other resource requests

Other resource requests are handled through the `-l` option to `qsub`. They follow the general form:

```
-l resource_name=value
```

These resource requests may not be used that often, but are good to know about. Each node on the bio cluster has 512BG RAM and 64 cores. It is useful to think of that as there being 512/64 = 8GB RAM/core. That means that if you manage to get 64 jobs each taking 9GB RAM all running on the same node, the odds are good that you've just crashed that node. To help avoid this problem, request that a node have at least a certain amount of available RAM:

```
#$ -l mem_free=10GB
```

Why request 10GB if I think I need 9GB per process? No reason except that it is best to stay on the safe side.

Important caveat

As a user, one has to be very careful about resource requests for jobs that will use a lot of RAM. Let us revisit the example above. Further imagine that you have an array job of 100 tasks that will each take 9GB RAM, but it will take several hours before they hit that RAM usage. What will happen:

1. You submit your array to the queue
2. The scheduler finds you an open node, and starts loading your jobs onto it. Since Joseph has optimized the scheduler to "pack" nodes (*i.e.*, shove as many jobs onto a single node before moving on to another node), it will certainly be the case that 10GB are free at the time the scheduler checks the resource request
3. Several hours later, the 64 jobs on one node hit 9GB RAM, which is 576GB total, which is greater than what's on the system. Boom--the node goes down. That sucks, but the GE did exactly what you asked it to.

There is a workaround. In addition to the RAM request, use the parallel environment to make sure that the number of jobs running can never use up all the system RAM. For our hypothetical case, 9GB RAM per process /8GB RAM per core = 1.125 "core RAM equivalents" are needed to run our jobs. So, round that up to two:

```
#$ -pe openmp 2
#$ -l mem_free=10GB
```

Now, the perfect storm can't happen--at most 32 of these jobs can be run on a single node at a time. This may be considered "wasteful" in that it doesn't allow 100% resource utilization, but one always has to make compromises when there is the possibility of crashing the compute nodes.

Resource requests and array jobs

For an array job, the resource request applies *per-task*, not for the entire array. In other words, saying

```
#$ -t 1-1000
#$ -pe openmp 2
```

tells the GE that you want to run 1000 tasks which each will need two cores.

Setting a random number seed

Some programs require that a (usually integer) value be passed to the program in order to initialize a random number generator. In the previous examples concerning "ms", I ignored the issue of seeding, which is not ideal, as you can end up with replicates generated using identical seeds (because programs will often default to using the system time for a seed, and it is quite likely that two instances of an array job will start on the exact same second of the day, and therefore get the same seed).

Here's a trick that I add to scripts:

```
SEED=`echo "$SGE_TASK_ID*$RANDOM" | bc -l`
```

This trick uses the program "bc" to multiply the current task's `SGE_TASK_ID` by a random value from the system, and save the results in a variable called `SEED`. This works because, as in perl and many other scripting languages, encapsulating a shell command in back-ticks can be used to take the return-value of a command and save it to a variable. Now, you can pass `$SEED` as the command-line argument to a program.

Automating jobs with inconvenient file names

In the above examples, an array job was used to write to files named "`output.$SGE_TASK_ID.`" Similarly, one could read input from "`infile.$SGE_TASK_ID`". Unfortunately, many of our analyses will use files with unfortunate file names. For instance, take the following FASTQ files in one of my folders:

```
[krthornt@hpc line00_reference]$ ls line*.gz
line0_lane0_1.fastq.gz  line0_lane1_1.fastq.gz  line0_lane2_1.fastq.gz
line0_lane0_2.fastq.gz  line0_lane1_2.fastq.gz  line0_lane2_2.fastq.gz
```

Further, I have 21 directories with similarly-named files. For each directory, I want to align each FASTQ file against a common reference genome. In total, there are 134

FASTQ files. Do I need to write 134 *separate* job scripts? I could do that, and even automate writing such scripts, using a language like perl or python. You can do it that way, but I won't--there is a cleaner and better-organized method based on array jobs.

Here's how I'm going to solve the problem:
1. I will use a perl script to write each of the 134 command lines to a file. Using perl to do this is a trivial matter: simply troll through each of the 21 folders, find the files that match the correct pattern, and write the command line to an output file, which I will call BWAALNCLI.txt.
2. I will write an array job that does 134 tasks. Each task will use Linux commands to pull the i^th^ line from BWAALNCLI.txt, and execute the command contained on that line.

To make things concrete, here is the first line of BWAALNCLI.txt:

```
bwa aln -t 32 -l 13  -m 50000000 -I -R 5000 /bio/krthornt/
new_yakuba_data/dyak-all-chromosome-r1.3-newnames.fasta /bio/
krthornt/new_yakuba_data/line00_reference/line0_lane0_1.fastq.gz
> /bio/krthornt/new_yakuba_data/line00_reference/
line0_lane0_1.sai
```

The above can be broken down into:

```
bwa aln [params] reference_file fastq_file > output.sai
```

In the above, the reference is

```
/bio/krthornt/new_yakuba_data/dyak-all-chromosome-r1.3-newnames.fasta
```

The fastq_file is

```
/bio/krthornt/new_yakuba_data/line00_reference/line0_lane0_1.fastq.gz
```

and the final output file will be

```
/bio/krthornt/new_yakuba_data/line00_reference/line0_lane0_1.sai
```

Note the following:
1. bwa is installed on the cluster, and all nodes, which is why I don't have to specify a path to the binary
2. I have my reference, FASTQ files, etc., on the /bio storage device, which is the fast one with lots of room.
3. bwa's aligner prints to STDOUT, therefore I use ">" to redirect output to the final ".sai" file.

So, now I have 134 lines in BWAALNCLI.txt, and I write the following script, called bwa_aln.sh, and I've numbered each line:

```
1.#!/bin/bash
2.#$ -N BWA_ALN
3.#$ -pe openmp 32
4.#$ -t 1-134
5.COMMAND=`head -n $SGE_TASK_ID /data/users/krthornt/
alignment_jobs/BWAALNCLI.txt | tail -n -1 | cut -d">" -f1`
6.OF1=`head -n $SGE_TASK_ID /data/users/krthornt/alignment_jobs/
BWAALNCLI.txt | tail -n -1 | cut -d">" -f 2`
7.echo `date`
8.$COMMAND > $OF1
9.echo `date`
```

Let's break down the script:

Line	Effect
1	Establish "bash" as the command interpreter
2	Set the name of the job
3	Request 32 cores on 1 node for each task in the array
4	Declare that there are 134 tasks in the array
5	Parse the i^{th} line from the file of command lines, and save the part up until, but not including, the '>' character, into a variable called `COMMAND`. This variable represents the actual command to execute.
6	Parse the ith line from the file of command lines, and save the part after the '>' character into a variable called `OF1`. This variable represents the name of the output file
7	Write out the current date and time
8	Executes the command, and redirects output to the output file
9	Write out the current date and time.

One final thing--where do the date and time get written? The answer is:

```
$SGE_O_WORKDIR/BWA_ALN.oJOBID.$SGE_TASK_ID
```

In other words, the queuing system will make a file based on your job name, the job id (the one reported by qstat), and the position of the job in the array, and write anything printed to STDOUT in what I call the "o" files. In this example, printing out the date twice tells me how long the alignments took, which is useful to know.

There are also files with the format

```
$SGE_O_WORKDIR/BWA_ALN.eJOBID-$SGE_TASK_ID
```

containing anything which may have been written to STDERR. When things go wrong, look at these files first.

Submitting the above script to the “sf” queue results in aligning all 134 fastq files to the reference sequence of *Drosophila yakuba* in approximately 12 hours. If that doesn’t sell you on the usefulness of the system, nothing will.

Redirecting STDOUT and STDERR

As mentioned above, anything printed to the STDOUT (standard output) stream will end up in a file that has the suffix `.oJOBID` (for a non-array job), and `.oJOBID.$SGE_TASK_ID` (for an individual task in a larger array job).

There are two ways to change this behavior:

1. Manually redirect STDOUT and STDERR to file names of your choosing using > and 2>, respectively, as you would for any command-line program.
2. Add the following lines to your script to handle these two streams:
```
    #$ -e stderr_filename
    #$ -o stdout_filename
```

You can eliminate all output to either stream by using the `-e`/`-o` options to redirect to the file `/dev/null`. This will prevent any of the `*.sh.o*` and `*.sh.e*` files from being written. Whether or not you want to do this is another matter. For example, suppressing the `*.sh.e*` files means that you won’t see any error messages that may be printed.

NOTE: The biocluster scheduler is set up to automatically delete the STDOUT and STDERR files if they are empty when the job leaves the queue. This is a good thing--it prevents the accumulation of 1000s of tiny files. Further, if there are errors (and the program writes them to STDERR), you will see which files are still there, and they are the ones that had problems.

A technical detail

Why wasn’t the previous script slightly simpler? Why not just get the ith line and execute it directly, *e.g.*:

```
#!/bin/bash
#$ -N BWA_ALN
#$ -pe openmp 32
#$ -t 1-134
COMMAND=`head -n $SGE_TASK_ID /data/users/krthornt/
alignment_jobs/BWAALNCLI.txt | tail -n -1`
echo `date`
```

```
$COMMAND
echo `date`
```

In the above version, we just get the desired line from the file, store it in COMMAND, then take the value of COMMAND to execute.

The problem is this: it won't work. When the line contains any of the "magic" Linux/Unix objects, such as redirects (>, 2>, etc.), or pipes (|), simply saying $COMMAND will only execute everything up until the first magic character, and the rest is ignored. Thus, one must parse the file using the Unix "cut" command, and save the parts separately.

"One off" commands using the queue

What if you want to run a single command, but don't want to bother writing a job script? It is not great etiquette to run heavy-duty programs on the login node, but you can send single commands to a node using the "`qrsh`" command:

```
qrsh -q quename command
```

The above will execute the command on the desired queue, provided that resources are available.

Many commands that one may wish to execute this way will require a "module load" to make sure the binary is available in the user's execution path. To execute multiple commands, surround the commands in double quotes and separate them with semicolons:

```
[krthornt@hpc ~]$ qrsh -q krt "module load bwa/0.6.2;bwa"

Program: bwa (alignment via Burrows-Wheeler transformation)
Version: 0.6.2-r126
Contact: Heng Li <lh3@sanger.ac.uk>

Usage:   bwa <command> [options]

Command: index         index sequences in the FASTA format
         aln           gapped/ungapped alignment
         samse         generate alignment (single ended)
         sampe         generate alignment (paired ended)
         bwasw         BWA-SW for long queries
         fastmap       identify super-maximal exact matches

         fa2pac        convert FASTA to PAC format
         pac2bwt       generate BWT from PAC
         pac2bwtgen    alternative algorithm for generating BWT
         bwtupdate     update .bwt to the new format
         bwt2sa        generate SA from BWT and Occ
         pac2cspac     convert PAC to color-space PAC
         stdsw         standard SW/NW alignment
```

Here is an example executing an R script:

```
[krthornt@hpc ~]$ qrsh -q krt "module load R/2.15.0 ; R --no-
save --slave < test.R"
 [1]  0.8414710  0.9092974  0.1411200 -0.7568025 -0.9589243
-0.2794155
 [7]  0.6569866  0.9893582  0.4121185 -0.5440211
```

The script “test.R” contains just one command, `print(sin(1:10))`.

You may actually execute a whole series of commands that way.

Advanced job scripting issues

R integration

R is great. If you don’t use it, you should. One can execute interactive R code in shell scripts using (recent versions of) R’s ability to have command-line options passed to scripts:

```
#!/bin/bash
#$ -N RMADNESS
#$ -t 1-1000

module load R/2.15.0

INFILE=infile.$SGE_TASK_ID
OUTFILE=outfile.$SGE_TASK_ID

R --no-save --args $INFILE $OUTFILE < Rcode.R
```

The file `Rcode.R` may look something like this:

```
#define a function that does the work of the script
dosomething=function(infilename,outfilename)
{
     #something useful gets done here
}

#parse the command line options
a=commandArgs(trailing=T)
```

```
infile=a[1]
outfile=a[2]
dosomething(infile,outfile)
```

Implementing pipelines
One common organizational problem is that different stages of an analysis may be multi-processor/multi-core, and others may be serially-parallelized. A good example is an assembly-by-reference of paired-end Illumina data using the “bwa” aligner. A typical pipeline consists of the following stages:

1. Align each FASTQ file separately to the reference. This stage makes use of “bwa aln”, which is able to run multi-threaded.
2. At the end of step 1, the forward and reverse sequences of each lane have been aligned separately. The next step is to run “bwa sampe”, and pipe the results to samtools for sorting and compression into “bam” files. This step is single-core.

Naively, one could do all of this in one job script. But, this isn’t the best use of resources: once the aligner is done, the “sampe/samtools” step would use just 1 processor, but the job has requested 32 processors per task (in the specific example above), which means that the second steps (“sampe/samtools”) are hogging CPU cycles by preventing other jobs from running.

A more efficient use of resources would be to generate two scripts:
1. bwa_aln.sh, which runs the alignment step, requesting some large number of processors per node to get the job done.
2. make_bamfiles.sh, which executes the “sampe/samtools” step, requesting only 1 processor per node to do the work.

One can submit jobs in such a way that they wait for other jobs to complete. For example, if we have the following (incomplete) scripts:

In `script1.sh`:

```
#! /bin/bash
#$ -N JOB1
```

In `script2.sh`:

```
#! /bin/bash
#$ -N JOB2
#$ -hold_jid JOB1
```

The user now submits `job1.sh` followed by `job2.sh` to the queue, using `qsub` with no extra options being required. JOB2 will remain “held” by the queue until JOB1 is completed. If JOB1 is an array job, then JOB2 will remain held until all tasks of JOB1 are completed.

A clever user will take advantage of the fact that the Unix shell environment defaults to lexical sorting, and will therefore name the scripts in their pipeline something like this:

```
00_first_step.sh
01_second_step.sh
02_third_step.sh
etc...
```

and will submit them using a “for” loop from the shell:

```
for i in *.sh
do
qsub $i
done
```

and let the command interpreter do all of the work for them.

Scripting smarter: let bash help you

The reality of life is that things don’t always work. Bugs in code, bugs in scripts, hardware issues, etc., all conspire to make jobs fail. On top of that, a lot of the software that we use is not robust (often because we wrote it), and fails to check that input files exist, or that output files exist already and will be over-written, etc. Fortunately, you can use the programming features of bash to add some safety to your scripts.

Here is an example script which ensures that input files exist, and that output files are not larger than 0 bytes. The latter test assumes that if a file exists and were output, something would have been written to it, making it larger than 0 bytes.

```
#!/bin/bash

#$ -N MONKEYINGAROUND
#$ -t 1-1000

INFILE=$SGE_O_WORKDIR/input.$SGE_TASK_ID
OUTFILE=$SGE_O_WORKDIR/input.$SGE_TASK_ID
#first, make sure $INFILE exists:
if [ -e $INFILE ]
then
     #make sure $OUTFILE isn’t there and already written to:
     if [ ! -s $OUTFILE ]
     then
     #it is safe to execute the program!
     program $INFILE $OUTFILE
     fi
```

```
else
     echo "$INFILE doesn't exist, not executing"
fi
```

The above script will now only run if the input file does indeed exist, and the output file does not exist. This guards against mis-typed input file names, and allows you to re-run an entire job if a subset of them failed, without over-writing the output of the parts of the task which succeeded.

An annoyance

One very important note is that the whitespace in the "if" statements is important, and must be written exactly as above. If I replace each single space in the if statements with an X, we get:

```
#!/bin/bash

#$ -N MONKEYINGAROUND
#$ -t 1-1000

INFILE=$SGE_O_WORKDIR/input.$SGE_TASK_ID
OUTFILE=$SGE_O_WORKDIR/input.$SGE_TASK_ID
#first, make sure $INFILE exists:
ifX[X-eX$INFILEX]
then
     #make sure $OUTFILE isn't there and already written to:
     ifX[X!X-sX$OUTFILEX]
     then
     #it is safe to execute the program!
     program $INFILE $OUTFILE
     fi
else
     echo "$INFILE doesn't exist, not executing"
fi
```

Please note where those X are!!!!! This is an annoyance of bash--its flow-control functions are whitespace sensitive because the parser is not as smart/flexible as the interpreters and compilers that exist for languages like perl, C, C++, fortran, etc.

The checks being done inside the bracket are actually an implicit call to the shell command "test", and there are a large number of things for which you may test. See the man page for test for a list.