Examples of PBS scripts

- MPI script
- OpenMP script
- Hybrid MPI/OpenMP script
- Running Multiple Parallel Jobs Sequentially
- Scripts involving more than one executable
- Batch script for multiple MPI jobs on Beagle
- Batch script for chaining jobs on Beagle
- Batch script for packing jobs on Beagle on multiple nodes
- Batch script for packing jobs on Beagle on one node
- Benchmarking

In general we tried make the following scripts as simple as possible to clarify how to use PBS in specific situations -- better approaches are certainly possible. We would be more than happy to provide you with further guidance in writing your own scripts (write to beagle-support@ci.uchicago.edu). More details about PBS and its directives can be found at Getting started: performing basic operations on Beagle2 and in its references.

Here are some basic rules that it is important to follow:

- Do all the I/O from /lustre/beagle2, this is both important for speed and stability (files can be copied from other locations in the PBS script, but the aprun arguments should contain only references to /lustre/beagle2)
- The module command might not work under all shells, if you need to load a module, please contact us at beagle-support@ci.uchicago.edu
- In general we prefer to use simply `#PBS -l mppwidth = total number of cores`, instead of more complex approaches (using mppdepth or mppnpn) because the latter seem to create more problems than they solve.
- Your application's executable line must begin with one of the application launch commands (aprun for ESM jobs), or else your application will launch on a login/service node instead of a compute node. Launching an application on a login node can cause a service disruption for all users on the system.

**MPI script**

The following is for a single program, running on 320 cores (10 nodes) using MPI on all the cores for one hour of walltime. PBS output and error will be merged. (For the meaning of specific directives see Getting started: performing basic operations on Beagle2)

```bash
#!/bin/bash
#PBS -N MyMPITest
#PBS -l walltime=1:00:00
#PBS -l mppwidth=320
#PBS -j oe
source /opt/modules/default/init/bash
#Move to the directory where the script was submitted -- by the qsub command
cd $PBS_O_WORKDIR
# Define and create a directory on /lustre/beagle2 where to run the job
LUSTREDIR=/lustre/beagle2/`whoami`/MyMPITest/${PBS_JOBID}
echo "$LUSTREDIR"
mkdir -p "$LUSTREDIR"
# Copy the input file and executable to /lustre/beagle2, these have to be user and project specific
cp /home/lpesce/tests/openMPTest/src/hello_smp hello.in $LUSTREDIR
# Move to /lustre/beagle2
cd "$LUSTREDIR"
# Note that here I was running hello_smp on 320 cores , i.e., using 320 PEs (by using -n 320)
# each with 1 thread -- i.e., just itself (default by not using -d)
echo "hello_smp < hello.in >& hello.out" > hello.sh
chmod +x hello.sh
aprun -n 320 ./hello.sh
```
OpenMP script

The following is for a single program, running on a node using 32 threads (usually it is not recommended to use 32 threads, but there are exceptions and here we wanted to produce an example that used all the resources available) for one hour of walltime. Input and output by PBS are merged. Notice that we are using only mppwidth and not mppdepth because in the end it only matters how many cores we ask for and using only one directive is simpler. (For the meaning of specific directives see Getting started: performing basic operations on Beagle2)

#!/bin/bash
#PBS -N MyTest
#PBS -l walltime=1:00:00
#PBS -l mppwidth=32
#PBS -j oe

source /opt/modules/default/init/bash
#Move to the directory where the script was submitted -- by the qsub command
cd $PBS_O_WORKDIR
# Define and create a directory on /lustre/beagle2 where to run the job
LUSTREDIR=/lustre/beagle2/`whoami`/MyTest/${PBS_JOBID}
echo $LUSTREDIR
mkdir -p $LUSTREDIR

# Copy the input file and executable to /lustre/beagle2, these have to be user and project specific
cp /home/lpesce/tests/openMPTest/src/hello_smp hello.in $LUSTREDIR
# Move to /lustre/beagle2
cd $LUSTREDIR

# Note that here I was running one PE (by using -n 1)
# each with 32 threads (by using -d 32)
# Notice the setting of the environmental variable OMP_NUM_THREADS for openMP
# if other multi-threading approaches are used they might need to be handled differently
export OMP_NUM_THREADS=32
echo "hello_smp < hello.in >& hello.out" > hello.sh
chmod +x hello.sh
aprun -n 1 -d 32 ./hello.sh

Hybrid MPI/OpenMP script

In this example on each node we are placing 4 MPI processes, one on each die, and each process may fork 8 threads to run on the same die. This example uses ceiling(15/4) = 4 compute nodes. Note that the -S aprun flag is a new one, which defines how many processes to put on each die in order to allow an even distribution for sparsely populated jobs (i.e. jobs in which fewer than 32 processes are used per node). By default processes are placed sequentially within a node, that is die 0 is filled first (top left in Figure 1), then die 1 (bottom left) then die 2 (top right) and then die 3 (bottom right). Note, if you are using a multi-threaded version of libsci 10.4.1, set GOTO_NUM_THREADS rather than OMP_NUM_THREADS in your job script.

option, -d, sets the number of OpenMP threads per MPI task. You must also set the environment variable OMP_NUM_THREADS to this same value. Make sure that this value multiplied by the value for -N does not exceed 32. (there is 32 cores per one compute nodes)

Therefore if OMP_NUM_THREADS=8

you will set:

-d 8

Value 8 is recommended because one MPI task is run per NUMA node and memory is limited to that closest to each OpenMP thread.
#!/bin/bash
#PBS -N My_job
#PBS -l mppwidth=128
#PBS -l walltime=00:20:00
#PBS -j oe

source /opt/modules/default/init/bash

export OMP_NUM_THREADS=8

# This variable should be set to allow the ALPS and the OS scheduler to
# assign the task affinity rather than the compiler. If this is not set
# you may see a large negative effect on performance.

echo "hello_smp < hello.in >& hello.out" > hello.sh

chmod +x hello.sh

aprun -n 15 -N 4 -d 8 -S 1 ./hello.sh

Running Multiple Parallel Jobs Simultaneously

Notice the "&" at the end of each aprun command. Also the "wait" command at the end of the script is very important. It makes sure the batch job won't exit before all the simultaneous apruns are completed.

#!/bin/bash
#PBS -l mppwidth=192
#PBS -l walltime=12:00:00
#PBS -N my_job
#PBS -j oe

source /opt/modules/default/init/bash

cd $PBS_O_WORKDIR

export OMP_NUM_THREADS=8

# This variable should be set to allow the ALPS and the OS scheduler to
# assign the task affinity rather than the compiler. If this is not set
# you may see a large negative effect on performance.

echo "hello_smp < hello.in >& hello.out" > hello.sh

chmod +x hello.sh

aprun -n 64 ./a.out &
aprun -n 96 ./b.out &
aprun -n 32 ./c.out &

wait

Scripts involving more than one executable

Please notice two things: 1) the aprun command will reserve one or more nodes, i.e., it is not possible to ask for just a few cores with an aprun command; 2) multiple, independent MPI jobs do not run on the same node on a Cray XE6 -- or at least we haven't found yet how to do it.

The following is an example of a PBS script to run more than one executable on a single node (for the meaning of specific directives see Getting started: performing basic operations on Beagle2)

It runs one node job, with name MyTest and walltime one hour, output and error from PBS will be merged. The job will have four programs running on the same node each using six threads.
#!/bin/bash
#PBS -N MyTest
#PBS -l walltime=1:00:00
#PBS -l mppwidth=32
#PBS -j oe

source /opt/modules/default/init/bash

# Move to the directory where the script was submitted -- by the qsub command
cd $PBS_O_WORKDIR

# Define and create a directory on /lustre/beagle where to run the job
LUSTREDIR=/lustre/beagle/`whoami`/MyTest/${PBS_JOBID}
echo $LUSTREDIR
mkdir -p $LUSTREDIR

# Copy the input files for the multiple commands and the command files themselves to /lustre/beagle2
# these have to be user and project specific
cp /home/lpesce/tests/openMPTest/src/hello_smp hello.in1 $LUSTREDIR
cp /home/lpesce/tests/openMPTest/src/hello_smp hello.in2 $LUSTREDIR
cp /home/lpesce/tests/openMPTest/src/hello_smp hello.in3 $LUSTREDIR
cp /home/lpesce/tests/openMPTest/src/hello_smp hello.in4 $LUSTREDIR

# Move to /lustre/beagle2
cd $LUSTREDIR

# Create a script on /lustre/beagle2 which will encapsulate the multiple commands
# Of course the commands in general will be user specific and the script name is up
# to the user
SHSCRIPT=run${PBS_JOBID}.sh
printf "#!/bin/bash
" >> $SHSCRIPT
printf " export OMP_NUM_THREADS=8 ./hello_smp  <hello.in1 > hello.out1 & \n" >> $SHSCRIPT
printf " export OMP_NUM_THREADS=8 ./hello_smp  <hello.in2 > hello.out2 & \n" >> $SHSCRIPT
printf " export OMP_NUM_THREADS=8 ./hello_smp  <hello.in3 > hello.out3 & \n" >> $SHSCRIPT
printf " export OMP_NUM_THREADS=8 ./hello_smp  <hello.in4 > hello.out4 & \n" >> $SHSCRIPT
printf "wait" >> $SHSCRIPT
chmod 755 $SHSCRIPT

# Note that here I was running 4 independent calculations AKA 4 PEs (by using -n 4)
# each with 8 threads (by using -d 8)
# and asked for each PE to be on a numa node (-S 1)
aprun -n 4 -S 1 -d 8 $SHSCRIPT

---

Batch script for multiple MPI jobs on Beagle

% cat myshell.sh
#!/bin/bash

njobs=32

for (( i = 0; i < $njobs ; i++ ))
do
echo "starting $i"
mkdir job_$i
cd job_$i

sed 's/""/>""' ../qscript_template > tmp
mv tmp qscript_$i
qsub qscript_$i
cd ..
done

% cat qscript_template
#!/bin/bash

---
Batch script for chaining jobs on Beagle

% cat chaining.sh
# Each job starts only after completion of previous job

#!/bin/bash
njobs=32
for (( i = 0; i < $njobs ; i++ ))
do
    mkdir job_$i
    cd job_$i

    sed 's/^.*-N.*$/\#PBS -N myjob-'$i'/' ../qscript_test > qscript_$i

    if [ -n $last_jobID ]
    then
        jobID=`qsub -W depend=afterok:$last_jobID qscript_$i`
    else
        jobID=`qsub qscript_$i`
    fi

    # Each job starts only after completion of previous job

    #!/bin/bash
    njobs=32
    for (( i = 0; i < $njobs ; i++ ))
do
    mkdir job_$i
    cd job_$i

    sed 's/^.*-N.*$/\#PBS -N myjob-'$i'/' ../qscript_test > qscript_$i

    if [ -n $last_jobID ]
    then
        jobID=`qsub -W depend=afterok:$last_jobID qscript_$i`
    else
        jobID=`qsub qscript_$i`
    fi

    # Each job starts only after completion of previous job
Batch script for packing jobs on Beagle on multiple nodes

You can pack your multiple jobs into one larger calculation.

% cat packing.sh
#!/bin/bash

njobs=64
nprocs_per_job=32
let total_procs=($njobs * $nprocs_per_job)
sed 's/^.*width.*$/#PBS -l mppwidth='$total_procs'/' qscript_test > tmp
mv tmp qscript
echo "cd job_0 " >> qscript
mkdir job_0
cd job_0
for (( i = 1; i < $njobs ; i++ )); do
    mkdir job_${i}
    cd job_${i}
    echo "cd job_${i}" >> ../../qscript
    echo "aprun -n $nprocs_per_job python /<path to your dir>/hello.py >job_${i}.out &" >> ../../qscript
    echo "cd .." >> ../../qscript
    cd..
done
cd..
echo "wait" >> qscript
qsub qscript

% cat qscript_test
#!/bin/bash
#PBS -N test
#PBS -q batch
#PBS -l walltime=00:30:00
#PBS -l mppwidth=64
#PBS -j oe

./opt/modules/default/init/bash
cd $PBS_O_WORKDIR
Batch script for packing jobs on Beagle on one node

```bash
#!/bin/bash
#PBS -N test
#PBS -q batch
#PBS -l walltime=05:30:00
#PBS -l mppwidth=32
#PBS -j oe

. /opt/modules/default/init/bash
cd $PBS_O_WORKDIR
module swap PrgEnv-cray PrgEnv-gnu
module load python/2.7.10

aprun -n 1 sh -c 'parallel -j8 :::: s.txt'
```

```bash
% cat s.txt
python /my_dgemm.py 1000 > 01.out
python /my_dgemm.py 2000 > 02.out
...
python /my_dgemm.py 16000 > 16.out
```

Benchmarking

```bash
% cat benchmarking.pbs
#!/bin/bash
#PBS -N test
#PBS -q batch
#PBS -l walltime=00:30:00
#PBS -l mppwidth=160
#PBS -j oe

. /opt/modules/default/init/bash
cd $PBS_O_WORKDIR
module swap PrgEnv-cray PrgEnv-gnu
module load python/2.7.10
resfile="runtimes.dat"
```
size_list="32 64 96 128 160"
for size in $size_list; do
    aprun -n $size python mandel.py > $size.stdout
    runtime=$(grep -w "taken" $size.stdout)
    echo "$size
    runtime
    >> $resfile"
done

cat runtimes.dat
32
Total time taken for all communications : 13.0682859421
64
Total time taken for all communications : 7.64163589478
96
Total time taken for all communications : 5.77753686905
128
Total time taken for all communications : 4.64086604118
160
Total time taken for all communications : 4.32990908623