You can run multiple serial jobs within a single batch job on Aitken or Electra by using the PBS scripts shown in the examples below. The maximum number of processes you can run concurrently on a single node is limited by the maximum number of processes that will fit in a given node's memory. However, for performance reasons, you might want to limit the number of serial processes per node to the number of cores on the node.

System	Processor Type	Cores/Node	Total Physical Memory/Node
Electra	Broadwell	28	128 GB
Electra	Skylake	40	192 GB
Aitken	Cascade Lake	40	192 GB
Aitken	Rome	128	512 GB
Aitken	Milan	128	512 GB

Note: The amount of memory that is actually available to a PBS job is slightly less than the total physical memory of the nodes it runs on, because the operating system uses up to 4 GB of memory in each node. At the beginning of a job, the PBS prologue checks the nodes for free memory and guarantees that 85% of the total physical memory is available for the job.

PBS Script Examples

The scripts shown in these examples allow you to spawn serial jobs across nodes using the mpiexec command. The mpiexec command keeps track of $PBS_NODEFILE and properly places each serial job onto the processors listed in $PBS_NODEFILE. For this method to work for serial codes and scripts, you must set the MPI_SHEPHERD environment variable to true.

In addition, to launch multiple copies of the serial job at the same time, you must use the $MPT_MPI_RANK environment variable in order to distinguish different input/output files for each serial job. In the examples below, this is accomplished through the use of a wrapper script, called wrapper.csh, in which the input/output identifier ($rank) is calculated from the sum of $MPT_MPI_RANK and an argument provided as input by the user.

Note: These methods may not work if, in addition to mpi-hpe/mpt, you load other modules that also come with mpiexec (such as mpi-hpcx or Intel python).

Example 1

The following PBS script runs 64 copies of a serial job, assuming that 16 copies will fit in the available memory on one node and 4 nodes are used. The wrapper script is shown below the PBS script.

serial1.pbs

```
#PBS -S /bin/csh
#PBS -j oe
#PBS -l select=4:ncpus=16:model=sky_ele
#PBS -l walltime=4:00:00

module load mpi-hpe/mpt
setenv MPI_SHEPHERD true

cd $PBS_O_WORKDIR

mpiexec -np 64 wrapper.csh 0
```

wrapper.csh

```
#!/bin/csh -f
@ rank = $1 + $MPT_MPI_RANK
./a.out < input_${rank}.dat > output_${rank}.out
```

This example assumes that input files are named input_0.dat, input_1.dat, ... and that they are all located in the directory the PBS script is submitted from ($PBS_O_WORKDIR). If the input files are located in different directories, then you can modify wrapper.csh to change (cd) into different directories, as long as the directory names are differentiated by a single number that can be obtained from $rank (=0, 1, 2, 3, ...).

Be sure to include the current directory in your path, or use ./wrapper.csh in the PBS script. Also, run the chmod command as follows to ensure that you have execute permission for wrapper.csh:

```
chmod u+x wrapper.csh
```

Example 2

The following PBS script provides flexibility in situations where the total number of serial jobs may not be the same as the total

number of processors requested in a PBS job. The serial jobs are divided into a few batches, and the batches are processed sequentially. Again, the wrapper script is used where multiple versions of the a.out program in a batch are run in parallel.

serial2.pbs

```
#PBS -S /bin/csh
#PBS -j oe
#PBS -l select=3:ncpus=10:model=sky_ele
#PBS -l walltime=4:00:00

module load mpi-hpe/mpt
setenv MPI_SHEPHERD true

# Set MPI_DSM_DISTRIBUTED=0 to spread out the 10 serial processes
# across the total number of physical cores on a node
setenv MPI_DSM_DISTRIBUTE 0

cd $PBS_O_WORKDIR

# This will start up 30 serial jobs 10 per node at a time.
# There are 64 jobs to be run total, only 30 at a time.

# The number to run in total defaults here to 64 or the value
# of PROCESS_COUNT that is passed in via the qsub line like:
# qsub -v PROCESS_COUNT=48 serial2.pbs

# The total number to run at once is automatically determined
# at runtime by the number of CPUs available.
# qsub -v PROCESS_COUNT=64 -l select=4:ncpus=3:model=sky_ele serial2.pbs
# would make this 12 per pass not 30. No changes to script needed.

if ( $?PROCESS_COUNT ) then
 set total_runs=$PROCESS_COUNT
else
 set total_runs=64
endif

set batch_count=`wc -l < $PBS_NODEFILE`

set count=0

while ($count < $total_runs)
  @ rank_base = $count
  @ count += $batch_count
  @ remain = $total_runs - $count
  if ($remain < 0) then
    @ run_count = $total_runs % $batch_count
  else
    @ run_count = $batch_count
  endif
  mpiexec -np $run_count wrapper.csh $rank_base
end
```

GNU is a free software operating system that is upward-compatible with Unix. *GNU parallel* is a shell tool for executing jobs in parallel. It uses the lines of its standard input to modify shell commands, which are then run in parallel.

A copy of GNU parallel is available in the /usr/bin directory of some NAS compute node types bro_ele, sky_ele and rom_ait. For detailed information about this tool, see the GNU operating system website and the GNU parallel man page.

The following examples demonstrate how you can use GNU parallel to run multiple tasks in a single PBS batch job.

Example 1

This example script runs 64 copies of a serial executable file, and assumes that 4 copies will fit in the available memory of one node and that 16 nodes are used.

gnu_serial1.pbs

```
#PBS -lselect=16:ncpus=4:model=sky_ele
#PBS -lwalltime=4:00:00

cd $PBS_O_WORKDIR

seq 64 | parallel -j 4 -u --sshloginfile $PBS_NODEFILE \
 "cd $PWD;./myscript.csh {}"
```

In the above PBS script, the last command uses the parallel command to simultaneously run 64 copies of myscript.csh located under $PBS_O_WORKDIR. Here is the specific breakdown:

seq 64
: Generates a set of integers *1, 2, 3, ..., 64* that will be passed to the parallel command.

-j 4
: GNU parallel will determine the number of processor cores on the remote computers and run the number of tasks as specified by -j. In this case, -j 4 tells the parallel command to run 4 tasks in parallel on one compute node.

-u
: Tells the parallel command to print output as soon as possible. This may cause output from different commands to be mixed. GNU parallel runs faster with -u. This can be reversed with --group.

--sshloginfile $PBS_NODEFILE
: Distributes tasks to the compute nodes listed in $PBS_NODEFILE. **This option is not needed if the PBS script uses only one node.**

"cd $PWD; ./myscript.csh {}"
: Changes directory to the current working directory and runs myscript.csh located under $PWD. At this point, $PWD is the same as $PBS_O_WORKDIR. The {} is an input to myscript.csh (see below) and will be replaced by the sequence number generated from seq 64.

myscript.csh

```
#!/bin/csh -fe
# add the following to enable the module command:
source /usr/share/modules/init/csh
date
mkdir -p run_$1
cd run_$1

echo "Executing run $1 on" `hostname` "in $PWD"

$HOME/bin/a.out < ../input_$1 > output_$1
```

In this above sample script executed by the parallel command:

- $1 refers to the sequence numbers (*1, 2, 3, ..., 64*) from the seq command that was piped into the parallel command
- For each serial run, a subdirectory named run_$1 (*run_1, run_2*, ...) is created
- The echo line prints information back to the PBS stdout file
- The serial a.out is located under $HOME/bin
- The input for each run, input_$1 (*input_1, input_2*, ...) is located under $PBS_O_WORKDIR, which is the directory above run_$1
- The output for each run (*output_1, output_2*, ...) is created under run_$1

Potential Modifications to Example 1

There are multiple ways to pass arguments to parallel. For example, instead of using the seq command to pass a sequence of integers, you can also pass in a list of directory names or filenames using ls -1 or cat mylist, where the file *mylist* contains a list of entries.

Example 2

This script is similar to Example 1, except that 6 nodes are used instead of 16 nodes. This means that 24 serial a.outs can be run simultaneously, since a total of 24 cores (6 nodes x 4 cores) are requested. As each a.out completes its work on a core, another a.out is launched by the parallel command to run on the same core.

myscript.csh is the same as that shown in the previous example.

gnu_serial2.pbs

```
#PBS -lselect=6:ncpus=4:model=sky_ele
#PBS -lwalltime=4:00:00

cd $PBS_O_WORKDIR

seq 64 | parallel -j 4 -u --sshloginfile $PBS_NODEFILE \
 "cd $PWD; ./myscript.csh {}"
```

Example 3

In this example, an OpenMP executable is run with 16 OpenMP threads on one Electra Skylake node. To run 64 copies of this executable with 10 copies running simultaneously on 10 nodes:

gnu_openmp.pbs

```
#PBS -lselect=10:ncpus=16:mpiprocs=1:ompthreads=16:model=sky_ele
#PBS -lwalltime=4:00:00

cd $PBS_O_WORKDIR

seq 64 | parallel -j 1 -u --sshloginfile $PBS_NODEFILE \
 "cd $PWD; ./myopenmpscript.csh {}"
```

myopenmpscript.csh

```
#!/bin/csh -fe
date
mkdir -p run_$1
cd run_$1

setenv OMP_NUM_THREADS 16

echo "Executing run $1 on" `hostname` "in $PWD"

$HOME/bin/a.out < ../input_$1 > output_$1
```

A job array is a collection of jobs that differ from each other by only a single index parameter. Creating a job array provides an easy way to group related jobs together. For example, if you have a parameter study that requires you to run your application five times, each with a different input parameter, you can use a job array instead of creating five separate PBS scripts and submitting them separately.

Creating a Job Array

To create a job array, use a single PBS script and use the -J option to specify a range for the index parameter, either on the qsub command line or within your PBS script. For example, if you submit the following script, a job array with five sub-jobs will be created:

```
#PBS -lselect=1:ncpus=128:model=mil_ait
#PBS -lwalltime=8:00:00
#PBS -J 1-5

# For each sub-job, a value provided by the -J
# option is used for PBS_ARRAY_INDEX

mkdir dir.$PBS_ARRAY_INDEX
cd    dir.$PBS_ARRAY_INDEX

mpiexec ../a.out < ../input.$PBS_ARRAY_INDEX
```

Submitting the script to PBS will return a PBS_ARRAY_ID. For example:

```
% qsub job_array_script
28846[].pbspl1.nas.nasa.gov
```

Each sub-job in this job array will have a PBS_JOBID that includes both the PBS_ARRAY_ID and a unique PBS_ARRAY_INDEX value within the brackets. For example:

```
28846[1].pbspl1.nas.nasa.gov
28846[2].pbspl1.nas.nasa.gov
28846[3].pbspl1.nas.nasa.gov
28846[4].pbspl1.nas.nasa.gov
28846[5].pbspl1.nas.nasa.gov
```

When you develop your script, note the following:

- Interactive submission of job arrays is not allowed.
- Job arrays are required to be re-runnable. Submitting jobs with the -r n option is not allowed.
- The maximum array size is limited to 5 for array jobs submitted to the devel queue. Only one sub-job of the array job can be running in the devel queue.
- For all other queues, the maximum array size is limited to 501.

Checking Status

The status of the sub-jobs is not displayed by default. For example, the following qstat options shows the job array as a single job:

```
% qstat -a
or
% qstat -nu your_username
or
% qstat -J

JobID          User  Queue  Jobname  TSK Nds wallt S
-------------- ----- -----  -------- --- --- ----- -
28846[].pbspl1 user  normal myscript  20   4 00:02 B
```

In the example above, the "B" status applies only to job arrays. This status indicates that at least one sub-job has left the "Q" (queued) state and is running or has run, but not all sub-jobs have run.

To check the status of the sub-jobs, use either the -Jt option or the -t option with an array specified. For example:

```
% qstat -Jt
or
% qstat -t 28846[].pbspl1.nas.nasa.gov

JobID          User  Queue  Jobname  TSK Nds wallt S
-------------- ----- -----  -------- --- --- ----- -
28846[1].pbspl1 user  normal myscript  20   4 00:02 R
28846[2].pbspl1 user  normal myscript  20   4 00:02 R
28846[3].pbspl1 user  normal myscript  20   4 00:02 Q
28846[4].pbspl1 user  normal myscript  20   4 00:02 Q
28846[5].pbspl1 user  normal myscript  20   4 00:02 Q
```

Deleting a Job Array or Sub-Job

To delete a job array or a sub-job, use the qdel command and specify the array or sub-job. For example:

```
% qdel "28846[]"
```

```
% qdel "28846[5]"
```

More Information

For more information about job arrays, see Chapter 8 of the PBS Professional User's Guide (PDF - 3MB).

If you have an MPI application with a small number of ranks and you need to run it many times, each with a different input parameter, consider using both the GNU Parallel tool and the PBS Job Arrays feature, if it meets these criteria:

- The number of MPI ranks is equal to or smaller than the number of physical cores in a node.
- There is enough memory in the node for multiple MPI jobs.

Example

Suppose you have a 5-rank MPI application called hello_mpi_processor, and you need to run it 400 times with 400 different input parameters (represented by the files in_1, in_2, ..,in_400 in the example). The following PBS script will divide these 400 runs into 10 PBS array sub-jobs. In each sub-job, one Skylake node is used to fit eight of the 5-rank runs simultaneously. The GNU Parallel tool is used to schedule 40 of these runs in each sub-job.

runscript

```
#PBS -S /bin/csh
#PBS -l select=1:ncpus=40:model=sky_ele
#PBS -l walltime=00:10:00
#PBS -J 1-400:40
#PBS -j oe

cd $PBS_O_WORKDIR

set begin=$PBS_ARRAY_INDEX
set end=`expr $PBS_ARRAY_INDEX + 39`

seq $begin $end | parallel -j 8 -u "./my_hello.scr {}"
```

my_hello.scr

The runscript, shown above, refers to the following script which contains all the tricky details:

```
#!/bin/csh -f

# need to enable module command and load modules
source ~/.cshrc
module load mpi-hpe/mpt

set case=$1
set nprocs=5

#need to create PBS_NODEFILE to run mpiexec
echo `hostname` >>! nodefile_$case
echo `hostname` >>! nodefile_$case
echo `hostname` >>! nodefile_$case
echo `hostname` >>! nodefile_$case
echo `hostname` >>! nodefile_$case
setenv PBS_NODEFILE `pwd`/nodefile_$case

# need to disable pinning to avoid having multiple processes run
# on the same set of CPUs
setenv MPI_DSM_DISTRIBUTE 0

date                                    >  out_$case
echo "running case $case on $nprocs processors"    >>  out_$case
mpiexec -np $nprocs ./hello_mpi_processor in_$case >>  out_$case
date                                    >>  out_$case

# remove nodefile at end of run
\rm nodefile_$case
```

Job Submission

```
pfe% qsub runscript
1468444[].pbspl1.nas.nasa.gov
```

Output

At the end of the run, you should have files out_[1-400] and runscript.o1468444.[1,41,81,..,361].

The following example shows sample contents of one of the out* files:

```
Wed Mar 15 16:05:12 PDT 2017
running case 42 on 5 processors
...
<output generated by hello_mpi_processor>
...
Wed Mar 15 16:05:13 PDT 2017
```