If your job does not run after it has been successfully submitted, it might be due to one of the following reasons:

- The queue has reached its maximum run limit
- Your job is waiting for resources
- Your mission share has run out
- The system is going into dedicated time
- Scheduling is turned off
- Your job has been placed on hold
- Your home filesystem or default /nobackup filesystem is down
- Your hard quota limit for disk space has been exceeded

Each scenario is described in the sections below, along with troubleshooting steps. The following commands provide useful information about the job that can help you resolve the issue:

tracejob *job_id*
: Displays log messages for a PBS job on the PBS server. The table below displays the relationship among the various NAS compute systems, the compute model types, the front-end systems to use, and the PBS server that serves each model type. For jobs served by the PBS server pbspl1 or pbspl4, you can log into the server and run the command yourself. For jobs served by other PBS server, such as pbs05a, the command can only be run by a NAS staff member.

qstat -as *job_id*
: Displays all information about a job on a front-end for the compute resource model type you submitted your job for. When checking Endeavour jobs on a PFE, the *job_id* must include both the job sequence and the server name, pbspl4. For example:

```
qstat -as 2468.pbspl4
```

System Name	Compute Model	Front-End	PBS Server
LDAN	ldan	pfe[20-22,25,26],lfe[5-8]	pbspl1
Electra	bro_ele	pfe[20-22,25,26]	pbspl1
Electra	sky_ele	pfe[20-22,25,26]	pbspl1
Aitken	cas_ait	pfe[20-22,25,26]	pbspl1
Aitken	rom_ait	pfe[20-22,25,26]	pbspl1
Aitken	mil_ait	afe[01,02]	pbspl1
Endeavour	cas_end	pfe[20-22,25,26]	pbspl4
Cabeus	rom_gpu	cfe[01,02]	pbspl4
Cabeus	mil_a100	cfe[01,02]	pbspl4
Cabeus	gh200	cghfe[01,02]	pbs05a
Athena	tur_ath	athfe[01-04]	pbs06a

The Queue Has Reached Its Maximum Run Limit

Some queues have a maximum run (max_run) limit—a specified maximum number of jobs that each user can run. If the number of jobs you are running has reached that limit, any jobs waiting in the queue will not begin until one of the running jobs is completed or terminated.

To check if a queue has a max_run limit set, login to an appropriate front-end and run:

```
qstat -fQ queue_name | grep max_run
```

For example, the devel queue served by pbspl1 has a max_run limit of 1 per user, the debug and ldan queues served by pbspl1 each have a max_run limit of 2 per user.

Your Job is Waiting for Resources

Your job might be waiting for resources for one of the following reasons:

- All resources are busy running jobs, and no other jobs can be started until resources become available again
- PBS is draining the system and not running any new jobs in order to accommodate a high-priority job
- Users have submitted too many jobs at once (for example, more than 100), so the PBS scheduler is busy sorting jobs and cannot start new jobs effectively
- You requested a specific node or group of nodes to run your job, which might cause the job to wait in the queue longer than if nodes were not specified

To view job status and events for jobd served by pbspl1 or pbspl4, run the tracejob utility on the appropriate PBS server. For example:

```
pfe25% ssh pbspl1
pbspl1% tracejob 234567
Job: 234567.pbspl1.nas.nasa.gov

02/15/2019 00:23:55  S    Job Modified at request of Scheduler@pbspl1.nas.nasa.gov
```

```
02/15/2019 00:23:55  L    No available resources on nodes
02/15/2019 00:38:47  S    Job Modified at request of Scheduler@pbspl1.nas.nasa.gov
02/15/2019 00:50:30  S    Job Modified at request of Scheduler@pbspl1.nas.nasa.gov
02/15/2019 01:51:21  S    Job Modified at request of Scheduler@pbspl1.nas.nasa.gov
02/15/2019 01:55:38  S    Job Modified at request of Scheduler@pbspl1.nas.nasa.gov
02/15/2019 02:16:44  S    Job Modified at request of Scheduler@pbspl1.nas.nasa.gov
02/15/2019 07:39:48  L    Considering job to run
02/15/2019 07:39:48  L    Job is requesting an exclusive node and node is in use
```

TIP: If the scheduler has not yet reviewed the job, no information will be available and the `tracejob` utility will not provide any output.

If your job requests an exclusive node and that node is in use, you can wait for the requested node, request a different node, or submit the job again without requesting a specific node.

To view the current node usage for each processor type, run the `node_stats.sh` script. For example:

```
Node summary according to PBS as of Thu Aug 28 15:05:22 PDT 2025
 Nodes installed on Electra  : 3384  cores:  121536
 Nodes installed on Pleiades : 9266  cores:  210764
 Nodes installed on Aitken   : 3660  cores:  370272
 Nodes installed on Cabeus   : 128   cores:    8192

 Nodes free on Electra       : 260   cores:    9212
 Nodes free on Pleiades      : 639   cores:   15900
 Nodes free on Aitken        : 305   cores:   30064
 Nodes free on Cabeus        : 20    cores:    1280

 Nodes reserved on Electra   : 139
 Nodes reserved on Pleiades  : 177
 Nodes reserved on Aitken    : 68
 Nodes reserved on Cabeus    : 0

Nodes Down Across NAS: 827 == 40008 cores, 28 gpus

Nodes used/free by hardware type:
      bro :   Intel Broadwell in Pleiades :  28 Cores   Nodes Total:  1980, Down:    33, Reserved:    55,
           Used:  1705, Free:   242,  Running jobs:  167, Queued jobs want  1723 nodes
  bro_ele :   Intel Broadwell in Electra  :  28 Cores   Nodes Total:  1152, Down:    60, Reserved:     0,
        Used:   993, Free:    99,  Running jobs:   89, Queued jobs want  1057 nodes
  cas_ait : Intel Cascadelake in Aitken   :  40 Cores   Nodes Total:  1116, Down:    51, Reserved:     0,
        Used:   963, Free:   102,  Running jobs:   28, Queued jobs want  3925 nodes
  cas_gpu : Intel Cascadelake in Pleiades :  48 Cores   Nodes Total:    38, Down:     2, Reserved:     0,
        Used:    26, Free:    10,  Running jobs:    4, Queued jobs want     4 nodes
      has :    Intel Haswell in Pleiades :  24 Cores   Nodes Total:  2013, Down:    29, Reserved:   122,
           Used:  1876, Free:   108,  Running jobs:   48, Queued jobs want  9074 nodes
      ivy :   Intel Ivybridge in Pleiades :  20 Cores   Nodes Total:  5216, Down:   309, Reserved:     0,
           Used:  4646, Free:   261,  Running jobs:  905, Queued jobs want  4055 nodes
 mil_a100 :        AMD Milan in Cabeus   :  64 Cores   Nodes Total:   128, Down:     3, Reserved:     0,
       Used:   105, Free:    20,  Running jobs:   11, Queued jobs want   425 nodes
  mil_ait :        AMD Milan in Aitken   : 128 Cores   Nodes Total:   512, Down:    29, Reserved:     0,
        Used:   424, Free:    59,  Running jobs:   19, Queued jobs want  1953 nodes
  rom_ait :         AMD Rome in Aitken   : 128 Cores   Nodes Total:  2032, Down:   137, Reserved:    68,
        Used:  1751, Free:   144,  Running jobs:  115, Queued jobs want  4071 nodes
  rom_gpu :         AMD Rome in Pleiades : 128 Cores   Nodes Total:     2, Down:     0, Reserved:     0,
        Used:     0, Free:     2,  Running jobs:    0, Queued jobs want     0 nodes
  sky_ele :    Intel Skylake in Electra  :  40 Cores   Nodes Total:  2232, Down:   172, Reserved:   139,
        Used:  1899, Free:   161,  Running jobs:   79, Queued jobs want  8960 nodes
  sky_gpu :    Intel Skylake in Pleiades :  36 Cores   Nodes Total:    17, Down:     2, Reserved:     0,
        Used:    -1, Free:    16,  Running jobs:    1, Queued jobs want    22 nodes

GPUs used/free by host model type:
  cas_gpu :  4 Nvidia V100 Intel Cascadelake in Pleiades  GPUs Total:   152, Down:     8, Reserved:     0,
        Used:   104, Free:    40,  Running jobs:    4, Queued jobs want    16 gpus
 mil_a100 :  4 Nvidia A100 AMD Milan         in Cabeus    GPUs Total:   512, Down:    12, Reserved:     0,
       Used:   420, Free:    80,  Running jobs:   11, Queued jobs want  1700 gpus
  rom_gpu :  8 Nvidia A100 AMD Rome          in Pleiades  GPUs Total:    16, Down:     0, Reserved:     0,
        Used:     0, Free:    16,  Running jobs:    0, Queued jobs want     0 gpus
  sky_gpu :  4 Nvidia V100 Intel Skylake     in Pleiades  GPUs Total:    68, Down:     8, Reserved:     0,
        Used:    -4, Free:    64,  Running jobs:    1, Queued jobs want    88 gpus

Nodes currently allocated to the devel queue:
     bro :   Intel Broadwell Total:   56, Used:    7, Free:   49
 cas_ait : Intel Cascadelake Total:   28, Used:   10, Free:   18
     has :    Intel Haswell Total:  315, Used:  302, Free:   13
     ivy :   Intel Ivybridge Total:  183, Used:  158, Free:   25
 sky_ele :    Intel Skylake Total:   10, Used:    3, Free:    7

SBU rate per node type: bro:1.0 bro_ele:1.0 cas_ait:1.64 cas_gpu:27.04 has:0.8 ivy:0.66

mil_a100:37.86 mil_ait:4.38 rom_ait:4.06 rom_gpu:75.72 sky_ele:1.59 sky_gpu:27.04

FY2024 SBU cost == $0.22/SBU
```

Your Mission Share Has Run Out

If all of the cores within your mission directorate's share have been assigned or if running the new job would exceed your mission share, your job will not run. If resources appear to be available, they belong to other missions.

To view all information about your job, run qstat -as *job_id*. In the following sample output, a comment (line 5) indicates that the job would exceed the mission share:

```
pfe25% qstat -as 778574
JobID          User     Queue   Jobname  CPUs  wallt  Ss  wallt  Eff  wallt
-------------  -------  ------  -------  ----  -----  --  -----  ---  -----
778574.pbspl1  zsmith   normal  my_GC     12   04:00  Q   07:06   --  04:00
Job would exceed mission CPU share
```

To view the distribution of shares among all mission directorates, run qstat -W shares=-. For example:

```
pfe25% qstat -W shares=-

Group   Share% Use%  Share Exempt    Use  Avail Borrowed Ratio Waiting
------- ------ ---- ------ ------ ------ ------ -------- ----- -------
Overall    100    1 502872      0   9620 493252        0  0.02     400
 ARMD       26   27 128245      0 135876      0     7631  1.06  579173
 HEOMD      22   22 108514   1832 111840      0     3326  1.03 1783072
 SMD        50   40 246626     32 201220  45406        0  0.82   94316
 NAS         2    0   9864    180   2416   7448        0  0.24     128
```

If running your job would exceed your mission share, you might be able to borrow nodes from other mission directorates. To borrow nodes, your job must not request a wall-clock time that is too long (more than 4 hours). See Mission Shares Policy for more information.

The System is Going into Dedicated Time

When dedicated time is scheduled for hardware and/or software work, the PBS scheduler will not start a job if its projected end-time runs past the beginning of the dedicated time.

If you can reduce the requested wall-clock time so that your job will finish running prior to dedicated time, then your job can then be considered for running. To change the wall-clock time request for your job, follow the example below:

```
pfe25% qalter -l walltime=hh:mm:ss job_id
```

TIP: For Endeavour jobs, the *job_id* must include both the job sequence number and the Endeavour server name (for example, 2468.pbspl4).

To find out whether the system is in dedicated time, run the schedule all command. For example:

```
pfe25% schedule all
No scheduled downtime for the specified period.
```

Scheduling is Turned Off

Sometimes job scheduling is turned off by a system administrator. This is usually done when system or PBS issues need to be resolved before jobs can be scheduled to run. When this happens, you should see the following message near the beginning of the qstat -au *your_userid* output.

+++Scheduling turned off.

Your Job Has Been Placed On Hold ("H" Mode)

A job can be placed on hold either by the job owner or by someone who has root privilege, such as a system administrator. If your job has been placed on hold by a system administrator, you should get an email explaining the reason for the hold.

Your Home Filesystem or Default /nobackup Filesystem is Down

When a PBS job starts, the PBS prologue checks to determine whether your home filesystem and default /nobackup filesystem are available before executing the commands in your script. If your default /nobackup filesystem is down, PBS cannot run your job and will put the job back in the queue. If your PBS job does not need any file in that filesystem, you can tell PBS that your job will not use the default /nobackup to allow your job to run.

For example, if your default is /nobackupp17, you can add the following in your PBS script:

```
#PBS -l nobackupp17=0
```

Your Hard Quota Limit for Disk Space Has Been Exceeded

NAS quotas have hard limits and soft limits. Hard limits should never be exceeded. Soft limits can be exceeded temporarily, for a

grace period of 14 days. If your data remains over the soft limit for more than 14 days, the soft limit is enforced as a hard limit.

See Quota Policy on Disk Space and Files for more information.

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
```

There are several settings on the NAS front-end systems and compute nodes that define hard and soft limits to control the amount of resources allowed per user, or per user process. You can change the default soft limits for your sessions, up to the maximum hard limits, as needed.

Keep in mind that there might be different default limits set for an SSH login session and a PBS session, and among the various front-end systems and compute nodes.

Viewing Soft and Hard Limits

To view the soft limit settings in a session, run:

- For csh: % limit
- For bash: $ ulimit -a

The output samples below show soft limit settings on an Aitken front-end (afe01 or afe02).

For csh:

```
% limit
cputime      unlimited
filesize     unlimited
datasize     unlimited
stacksize    300000 kbytes
coredumpsize unlimited
memoryuse    unlimited
vmemoryuse   unlimited
descriptors  16384
memorylocked unlimited
maxproc      2060780
maxlocks     unlimited
maxsignal    2060780
maxmessage   819200
maxnice      0
maxrtprio    0
maxrttime    unlimited
```

For bash:

```
$ ulimit -a
core file size          (blocks, -c) unlimited
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 2060780
max locked memory       (kbytes, -l) unlimited
max memory size         (kbytes, -m) unlimited
open files                      (-n) 16384
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 300000
cpu time               (seconds, -t) unlimited
max user processes              (-u) 2060780
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

To view the hard limits, run:

- For csh: % limit -h
- For bash: $ ulimit -Ha

Modifying Soft Limits

To modify the soft limit setting of a resource, type the name of the resource after `limit` (for csh) or type the option after `ulimit` (for bash). For example, to reset the `stacksize` limit to 200,000 KB, run:

- For csh: % limit stacksize 200000
- For bash: $ ulimit -Ss 200000

Common Issues with the Default Limits

The following list describes a few common issues with default limits.

- stacksize (stack size)

  Some plotting packages, such as Tecplot, require a small stack size to function properly. On systems where the stacksize is set to unlimited, you will have to decrease the stacksize in order to use them. However, if the stacksize is set too small when applications are run, segmentation faults (segfaults) commonly occur. If you run plotting packages and your own applications in the same session, you will need to alternate between a small and a large stack size accordingly.

- maxproc (maximum user processes)

  If you encounter one of the following error messages, this indicates that you have reached the maxproc soft limit:

  ```
  can't fork process
  or
  fork: Resource temporarily unavailable
  ```

  This may happen during compilation, especially if you run a parallel make operation using make -j.

  Note: Each Tecplot session can consume more than 100 user processes. If you run multiple instances of Tecplot on the same front-end system where maxproc is set to a small value, you might reach the maxproc soft limit.

- memoryuse and vmemoryuse (maximum memory size and virtual memory)

  The vmemoryuse setting affects the outcome when you try to allocate memory—for example, when you use the allocate command as follows:

  ```
  allocate (a(i),stat=ierror)
  ```

  The memoryuse setting affects the referencing of array a. Keep in mind that running your applications on nodes with different settings will result in different behaviors.

- coredumpsize (core file size)

  Core files are useful for debugging your application. However, large core files may consume too much of your disk quota. In addition, the generation of large core files in a PBS job could occupy the buffer cache of the nodes, waiting to be flushed to disk. If the buffer cache on the nodes is not flushed by the PBS epilogue of the current job and the PBS prologue of the next job, the nodes will be unusable.

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
```

When a PBS job is running, the Linux operating system uses part of the physical memory of the compute nodes to store data that is either read from disk or written to disk. The memory used in this way is called *page cache*. In some cases, the amount or distribution of the physical memory used by page cache can affect job performance.

For more information, see the Wikipedia entry on Page cache.

Checking Page Cache Usage

To find out how much physical memory is used by page cache in a node your PBS job is running on, connect to the node via SSH and then run one of the following commands:

- cat /proc/meminfo
- top -b -n1 | head

Examples

The following example shows output of the cat /proc/meminfo command on a node of a running job. The output reports that 60,023,544 KB (~60 GB) of the node's memory is used by page cache:

```
% cat /proc/meminfo
MemTotal:      132007564 kB
MemFree:         605680 kB
Buffers:              0 kB
Cached:        60023544 kB
SwapCached:           0 kB
```

The next example shows output from the top -b -n1 | head command, run on the same node as in the previous example. The output reports that the node has 128,913 MB total memory, that 128,314 MB of the total memory is used, and that 59,220 MB (~59 GB) of the used memory is occupied by page cache:

```
% top -b -n1 | head
top - 15:04:00 up 3 days, 16:22,  2 users,  load average: 1.29, 1.23, 1.38
Tasks: 546 total,   2 running, 543 sleeping,   0 stopped,   1 zombie
Cpu(s):  0.5%us,  2.7%sy,  0.0%ni, 96.8%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   128913M total,   128314M used,      598M free,        0M buffers
Swap:        0M total,        0M used,        0M free,    59220M cached

PID    USER  PR  NI  VIRT  RES  SHR S   %CPU %MEM    TIME+ COMMAND
```

Managing Page Cache

The amount of physical memory used as page cache can be significant if your code reads a large amount of data from disk or writes a large amount of data to disk. For this reason, you may want to manage page cache using one of the methods described in this section.

Assigning Rank 0 to a Single Node

In the case of an MPI application where I/O is done solely by the rank 0 process, it is possible for the page cache to fill up the physical memory of a node, leaving no memory for the other MPI ranks. To avoid this situation, you can assign the rank 0 process to a node by itself. For example, the following PBS resource request allows for 49 MPI ranks with rank 0 to run on one node, and assigns 28 MPI ranks to each of the other two nodes:

```
#PBS -l select=1:mpiprocs=1:model=bro_ele+2:mpiprocs=28:model=bro_ele
```

Limiting Page Cache Usage

If there is enough physical memory on the head node to accommodate both the page cache and the memory used by other ranks, keep in mind that the total amount of memory in a node is split between two sockets. For example, the 128 GB of memory in an Electra Broadwell node is evenly divided between the node's two sockets, as shown in the Broadwell configuration diagram. Therefore, if other MPI processes are running on the same socket as the rank 0 process, you should check whether the amount of memory on one socket is enough for both the rank 0 process and the other ranks. If there is not enough memory, the other ranks may be forced to allocate and use memory on the other socket, causing a non-local memory access that may result in degraded performance.

In cases like this, you can set a limit for page cache usage to ensure there will be enough memory to accommodate both the I/O from the rank 0 process and the amount of memory used by other ranks on the same socket. You can accomplish this by running pcachem, a page cache management tool. For example, to set the limit at 8 GB (2^{33} bytes = 8589934592 bytes) run:

```
module load pagecache-management/0.5
setenv PAGECACHE_MAX_BYTES   8589934592
```

```
mpiexec -n XX pcachem -- a.out
```

where *XX* represents the total number of MPI processes. The exact value that you specify for PAGECACHE_MAX_BYTES is not important, however, be sure you specify a value that leaves enough physical memory on the socket (after subtracting the amount used by page cache) to accommodate the other MPI ranks on the socket.

Note: You must also load an MPI module for running MPI applications.

If you are using OVERFLOW, consider the option shown in the following example, which demonstrates the use of both page cache management (with pcachem) and process binding (with mbind.x) for better performance:

```
module load pagecache-management/0.5
setenv PAGECACHE_MAX_BYTES   8589934592
setenv MBIND /u/scicon/tools/bin/mbind.x

$HOME/bin/overrunmpi -v -n XX -aux "pcachem -- $MBIND -v -gm -cs" YY
```

where *YY* refers to the input file for OVERFLOW. The mbind.x option -gm prints memory usage for each process, and the -cs option distributes the MPI processes among the hardware cores.

For more information about mbind.x, see Using the mbind Tool for Pinning.

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).

The PBS exit value of a job may fall in one of four ranges:

- X = 0 (= JOB_EXEC_OK)

  This is a PBS special return value indicating that the job executed successfully

- X < 0

  This is a PBS special return value indicating that the job could not be executed. These negative values are:

  - -1 = JOB_EXEC_FAIL1 : Job exec failed, before files, no retry
  - -2 = JOB_EXEC_FAIL2 : Job exec failed, after files, no retry
  - -3 = JOB_EXEC_RETRY : Job exec failed, do retry
  - -4 = JOB_EXEC_INITABT : Job aborted on MOM initialization
  - -5 = JOB_EXEC_INITRST : Job aborted on MOM initialization, checkpoint, no migrate
  - -6 = JOB_EXEC_INITRMG : Job aborted on MOM initialization, checkpoint, ok migrate
  - -7 = JOB_EXEC_BADRESRT : Job restart failed
  - -8 = JOB_EXEC_GLOBUS_INIT_RETRY : Initialization of Globus job failed. Do retry.
  - -9 = JOB_EXEC_GLOBUS_INIT_FAIL : Initialization of Globus job failed. Do not retry.
  - -10 = JOB_EXEC_FAILUID : Invalid UID/GID for job
  - -11 = JOB_EXEC_RERUN : Job was rerun
  - -12 = JOB_EXEC_CHKP : Job was checkpointed and killed
  - -13 = JOB_EXEC_FAIL_PASSWORD : Job failed due to a bad password
  - -14 = JOB_EXEC_RERUN_ ON_SIS_FAIL : Job was requeued (if rerunnable) or deleted (if not) due to a communication failure between Mother Superior and a Sister

- 0 <= X < 128 (or 256 depending on the system)

  This is the exit value of the top process in the job, typically the shell. This may be the exit value of the last command executed in the shell or the `.logout` script if the user has such a script (csh).

- X >=128 (or 256 depending on the system)

  This means the job was killed with a signal. The signal is given by X modulo 128 (or 256). For example an exit value of 137 means the job's top process was killed with signal 9 (137 % 128 = 9).

Occasionally, PBS jobs may fail due to mpiexec timeout problems at startup. If this happens, you will typically see several error messages similar to the following:

- MPT: Launcher network accept (MPI_LAUNCH_TIMEOUT) timed out
- MPT: Intermediate launcher on *some_node* failed to receive connection(s) from: *some_other_nodes*

When mpiexec starts, several things must happen in a timely manner before the job runs; for example, every node must access the executable and load it into memory. In most cases, this process is successful and the job starts normally. However, if something slows down this process, mpiexec may timeout before the job can start.

This can happen when the job is running an executable located on a filesystem that is not mounted during PBS prologue activities (for example, on a colleague's filesystem); mpiexec may timeout before the filesystem is mounted on each node. (Jobs running on a large number of nodes may be more likely to experience this problem than smaller jobs.)

Workaround Steps

If you encounter an mpiexec startup failure, complete all of these steps to try and resolve the problem.

1. Ensure all the filesystems are mounted before mpiexec runs. For example, if your home filesystem is /home3, your nobackup filesystem is /nobackupp17, and the executable is under /vast_swbuild, you would add the following line to your job script:

   ```
   #PBS -l site=needed=/home3+/nobackupp17+/vast_swbuild
   ```

2. Even if all the filesystems are mounted, the filesystem servers might be slow to respond to requests to load the executable. To address this, increase the mpiexec timeout period from 20 seconds (default) to 40 seconds by setting value of the MPI_LAUNCH_TIMEOUT environment variable to 40:

   For csh/tsch:
   ```
   setenv MPI_LAUNCH_TIMEOUT 40
   ```
   For bash/sh/ksh:
   ```
   export MPI_LAUNCH_TIMEOUT=40
   ```

3. Finally, it might take several attempts to ensure that the executable is loaded into memory. You can accomplish this by adding the several_tries script to the beginning of your mpiexec command line:

   ```
   /u/scicon/tools/bin/several_tries mpiexec -np 2000 /path/to/your_executable
   ```

   The script will then attempt the mpiexec command several times until command succeeds (or runs longer than the maximum time set in the environment variables), or a threshold number of attempts is exceeded (see several_tries settings, below.

Alternative to Step 3: If it is difficult to add the several_tries tool to your job script—for example, if you have a complicated set of nested scripts and it's not clear where to insert the tool—try using the pdsh command instead, as follows:

```
pdsh -w `uniq $PBS_NODEFILE | xargs | tr " " "," `
  "md5sum /path/to/your_executable" | dshbak -c
```

Note: The pdsh command line above is shown on two lines for formatting reasons. It should be entered as one line.

This does a parallel SSH into each node and loads the executable into each node's memory so that it's available when mpiexec tries to launch it.

If the executable is already in your path, you do not need to include the entire path in the command line.

Sample PBS Script with Workaround

The following sample script incorporates all three steps described in the previous section.

```
#PBS -l select=200:ncpus=40:model=sky_ele
#PBS -l walltime=HH:MM:SS
#PBS -l site=needed=/home3+/nobackupp17+/vast_swbuild
#PBS -j oe

cd $PBS_O_WORKDIR

setenv MPI_LAUNCH_TIMEOUT 40
set path = ($path /u/scicon/tools/bin)

several_tries mpiexec -np 8000 ./my_a.out
```

several_tries Script Settings

The following environment variables are associated with the several_tries script:

SEVERAL_TRIES_MAXTIME
: Maximum time the command can run each time, and still be retried.
Default: 60 seconds

SEVERAL_TRIES_NTRIES
: Threshold number of attempts. Default: 6

SEVERAL_TRIES_SLEEPTIME
: Sleep time between attempts. Default: 30 seconds

Listed below are several common reasons why a job submission might not be successful.

Compute Model not Recognized

If you get the following message after submitting a PBS job:

```
qsub: Unrecognized model xxx
```

It is possible that you are submitting from a front-end that does not serve the model type.

As shown in the table below, NAS has various front-ends and PBS servers for different compute resources. Make sure you are on the correct front-end for the compute resource you want to use.

System Name	Compute Model	Front-End	PBS Server
LDAN	ldan	pfe[20-22,25,26], lfe[5-8]	pbspl1
Electra	bro_ele	pfe[20-22,25,26]	pbspl1
Electra	sky_ele	pfe[20-22,25,26]	pbspl1
Aitken	cas_ait	pfe[20-22,25,26]	pbspl1
Aitken	rom_ait	pfe[20-22,25,26]	pbspl1
Aitken	mil_ait	afe[01,02]	pbspl1
Endeavour	cas_end	pfe[20-22,25,26]	pbspl4
Cabeus	mil_a100	cfe[01,02]	pbspl4
Cabeus	gh200	cghfe[01,02]	pbs05a
Athena	tur_ath	athfe[01-04]	pbs06a

Queue is Unknown

Be sure to use the correct queue. The common queue names served by each PBS server are:

- pbspl1: devel, debug, normal, long, and vlong for Electra bro_ele and sky_ele, Aitken cas_ait and rom_ait
- pbspl4:
  - e_normal, e_long, e_vlong, and e_debug for Endeavour resources
  - gpu_devel, gpu_normal for Cabeus mil_a100
- pbs05a: gpu_devel, gpu_normal, gpu_long, gpu_wide for Cabeus gh200
- pbs06a: devel, normal, long, and wide for Athena tur_ath

Be aware that pbspl4 server only recognizes the queue names for Endeavour and some Cabeus jobs. If you submit a job to pbspl4 and specify a pbspl1 queue, you will get an error, as follows:

```
pfe25% qsub -q normal@pbspl4 job_script
qsub: unknown queue
```

AUID or GID not Authorized to Use a Specific Queue

If you get the following message after submitting a PBS job:

```
qsub: Unauthorized Request
```

It is possible that you tried submitting the job to a queue that is accessible only to certain groups or users. To find out, check the output of qstat -fQ *@server_name* and look for a users list called acl_groups or acl_users. If your group or username is not in the lists, you must submit your job to a different queue.

You will also get this error if your project group ID (GID) has no allocation left. SeeNot Enough or No Allocation Left below for more information on troubleshooting this issue.

AUID Not Authorized to Use a Specific GID

If you get the following message after submitting a PBS job:

```
qsub: Bad GID for job execution
```

It is possible that your Agency User ID (AUID) has not been added to use allocations under a specific GID. Please ask the principal investigator (PI) of that GID to submit a request to support@nas.nasa.gov to add your AUID under that GID.

Queue is Disabled

If you get the following message after submitting a PBS job, submit the job to a different queue that is enabled:

qsub: Queue is not enabled

Queue Has a max_queued Limit

If you get the following message after submitting a PBS job:

qsub: would exceed queue's generic per-user limit

It is possible that you tried submitting the job to a queue that has a max_queued limit—a specified maximum number of jobs that each user can submit to a queue (running or waiting)—and you have already reached that limit.

Currently, the devel queue served by pbspl1 has a max_queued limit of 1. The gpu_devel queue served by pbspl4 and pbs05a have a max_queued limit of 1 and 2, respectively.

Resource Request Exceeds Resource Limits

If you get the following message after submitting a PBS job:

qsub: Job exceeds queue resource limits

Reduce your resource request to below the limit or use a different queue.

Not Enough or No Allocation Left

An automated script checks GID usage each day. If usage exceeds a GID's allocation, the GID is removed from the PBS access control list and you will no longer be able to submit jobs under that GID.

You can check the amount of allocation remaining using the acct_ytd command; for more information, see Job Accounting Utilities. In addition, if your PBS output file includes a message stating that your GID allocation usage is near its limit or is already over its limit, ask your PI to request an increase in allocation.

Once the request for increased allocation is approved and added to your account, an hourly script will automatically add your GID back to the PBS access control list.

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
```

If your job terminates because the nodes it is running on do not have enough available memory, try using one or more of the methods described below.

Note: The memory listed for each type of node in this article refers to the total physical memory in the node. 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 can use 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.

Use A Processor With More Memory

The following list shows the amount of total physical memory per node for the various NAS x86_64 processor types:

Broadwell
    128 GB
Skylake
    192 GB
Cascade Lake
    192 GB
Rome
    512 GB
Milan
    512 GB

If your job runs out of memory on a Broadwell node, you can try using one of the other node types instead. For example, change:

#PBS -lselect=1:ncpus=**28**:model=**bro_ele**

to one of the following:

#PBS -lselect=1:ncpus=**40**:model=**sky_ele**

#PBS -lselect=1:ncpus=**40**:model=**cas_ait**

#PBS -lselect=1:ncpus=**128**:model=**rom_ait**

#PBS -lselect=1:ncpus=**128**:model=**mil_ait**

Run Fewer Processes per Node

If all processes use approximately the same amount of memory, and you cannot fit 28 processes per node (Broadwell), 40 processes per node (Skylake/Cascade Lake), or 128 processors per node (Rome/Milan), then reduce the number of processes per node and request more nodes for your job.

For example, change:

#PBS -l**select=1:ncpus=128:mpiprocs=128**:model=mil_ait

to the following:

#PBS -l**select=2:ncpus=64:mpiprocs=64**:model=mil_ait

Recommendation: Add the command mbind.x -gm -cs before your executable in order to (1) monitor the memory usage of your processes and (2) balance the workload by spreading the processes between the two sockets of each node. For example:

mpiexec -np 60 /u/scicon/tools/bin/mbind.x -gm -cs ./a.out/

If you are running a typical MPI job, where the rank 0 process performs the I/O and uses a large amount of buffer cache, assign rank 0 to one node by itself. For example, if rank 0 requires up to 80 GB of memory by itself, change:

#PBS -l**select=1:ncpus=28:mpiprocs=28**:model=bro_ele

to the following:

#PBS -l**select=1:ncpus=1:mpiprocs=1**:model=bro_ele**+1:ncpus=27:mpiprocs=27**:model=bro_ele

Use the Endeavour System

If any process or thread in a multi-process or multi-thread job needs more than about 500 GB, the job won't run on Electra or Aitken nodes. Instead, run it on Endeavour, which is a shared-memory system.

Report Bad Nodes

If you suspect that certain nodes have less total physical memory than normal, report them to the NAS Control Room staff at (800) 331-8737, (650) 604-4444, or by email at support@nas.nasa.gov. The nodes can be taken offline while we resolve any issues,

preventing them from being used before they are fixed.

When your PBS job is running, its error and output files are kept in the /PBS/spool directory of the first node of your job. However, the space under /PBS/spool is limited, and when it fills up, any job that tries to write to /PBS/spool may die. This makes the node unusable by jobs until the spool directory is cleaned up manually.

To avoid this situation, PBS enforces a limit of 200 MB on the combined sizes of error and output files produced by a job. If a job exceeds that limit, PBS terminates the job.

To prevent this from happening to your job, do *not* write large amounts of content in the PBS output/error files. If your job needs to produce a lot of output, you can use the qsub -kod option to write the output directly to your final output file, bypassing the spool directory (see Commonly Used qsub Options). Until then, you can redirect standard out or standard error within your PBS script. Here are a few options to consider:

1. Redirect standard out and standard error to a single file:

   ```
   (for csh)
   mpiexec a.out >& output
   (for bash)
   mpiexec a.out > output 2>&1
   ```

2. Redirect standard out and standard error to separate files:

   ```
   (for csh)
   (mpiexec a.out > output) >& error
   (for bash)
   mpiexec a.out > output 2> error
   ```

3. Redirect only standard out to a file:

   ```
   (for both csh and bash)
   mpiexec a.out > output
   ```

The files "output" and "error" are created under your own directory and you can view the contents of these files while your job is still running.

If you are concerned that these two files could get clobbered in a second run of the script, you can create unique filenames for each run. For example, you can add the PBS JOBID to "output" using the following:

```
(for csh)
mpiexec a.out >& output.$PBS_JOBID
(for bash)
mpiexec a.out > output.$PBS_JOBID 2>&1
```

where $PBS_JOBID contains a number (jobid) and the name of the PBS server, such as 12345.pbspl1.nas.nasa.gov.

If you just want to include the numeric part of the PBS JOBID, do the following:

```
(for csh)
set jobid=`echo $PBS_JOBID | awk -F . '{print $1}'`
mpiexec a.out >& output.$jobid

(for bash)
export jobid=`echo $PBS_JOBID | awk -F . '{print $1}'`
mpiexec a.out > output.$jobid 2>&1
```

If you used the qsub -kod option mentioned above, you will be able to see the contents of the output or error files directly from a front end as they grow. If you neither used -k nor redirected the output, you can still see the contents of your PBS output/error files before your job completes by following these steps:

1. Find out the first node of your PBS job using qstat -W o=+rank0. In the following example, the output shows that the first node is r162i0n14:

   ```
   %qstat -u your_username -W o=+rank0
   JobID          User   Queue  Jobname   TSK Nds    wallt S    wallt  Eff Rank0
   -------------- ------ ------ -------- ---- --- -------- - -------- ---- ---------
   868819.pbspl1 zsmith long   ABC       512  64 5d+00:00 R 3d+08:39 100% r162i0n14
   ```

2. Log into the first node and *cd* to /PBS/spool to find your PBS stderr/out file(s). You can view the contents of these files using *vi* or *view*.

   ```
   %ssh r162i0n14
   %cd /PBS/spool
   %ls -lrt
   -rw------- 1 zsmith a0800 49224236 Aug  2 19:33 868819.pbspl1.nas.nasa.gov.OU
   -rw------- 1 zsmith a0800  1234236 Aug  2 19:33 868819.pbspl1.nas.nasa.gov.ER
   ```