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.

You can use a script called pdsh_gdb to debug a PBS job *while the job is running*. The script is available iafter loading the module /scicon/cli_tools on NAS compute systems.

Launching this script from a front-end node allows you to connect to each compute node of a PBS job and create a stack trace of each process. The aggregated stack trace from each process will be written to a user-specified directory (by default, it is written to ~/tmp).

Here is an example of how to use the pdsh_gdb script.

```
% module load scicon/cli_tools
% mkdir tmp
% pdsh_gdb -j job_id -d tmp -s -u username -n group_id
```

Note: The -n *group_id* is needed if you have specified a non-default project group ID (GID) via #PBS -W group_list=group_id for the PBS job that you want to debug with pdsh_gdb.

For Endeavour jobs, be sure to include both the job sequence number and the Endeavour server name, pbspl4, in the *job_id* (for example, 2468.pbspl4).

To view more detailed usage information, launch pdsh_gdb without any option:

```
% pdsh_gdb
```

The pdsh_gdb script was developed by NAS staff member Steve Heistand.

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.

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).

In order to run a graphics application in a PBS batch job, you must provide a value for the **DISPLAY** environment variable—even if you do not have or need an actual display for the job. If the batch job cannot find this value, it will quit and return an error such as "Can't open display" or "Display not set."

To provide a value for the **DISPLAY** variable, you can set up a display connection using either X11 forwarding or a VNC session, or you can run the job on the NAS GPU nodes, which include graphics cards. Each of these methods is described below.

Method 1: Use X11 Forwarding to Set Up a Display

Complete these steps to use your local system as an remote X11 server, which acts as the display:

1. Set up X11 Forwarding (if it's not set up already).
2. Log in to a PFE and submit a PBS batch job with the -V option on the qsub command line or inside the PBS script. For example:

   ```
   your_local_system% ssh pfe25
   pfe25% echo $DISPLAY
   10.150.27.21:30.0
   pfe% qsub -V job_script
   ```

   In the above example, using the -V option will export the **DISPLAY** environment variable (with its value set to 10.150.27.21:30.0) from your PFE login session to the PBS batch job.

Drawbacks to using this method:

- Because the connection from the PBS job to your local system must be maintained, you cannot exit from either your local system or from the PFE login window until the job completes.
- Application performance on remote X11 servers deteriorates with latency, so unless your local system is physically close to the NAS facility, job performance may not be optimal.

Method 2: Use a VNC Session to Set Up a Display

Complete these steps on a PFE to set up a VNC viewer, which acts as the display:

1. Complete steps 1-4 of the instructions in VNC: A Faster Alternative to X11 to establish a VNC session.

   Once the VNC viewer is set up and running, you will see the following messages (or something similar) in the PFE login window:

   ```
   pfe25% vncserver

   New 'X' desktop is pfe25:9

   Starting applications specified in /u/username/.vnc/xstartup
   Log file is /u/username/.vnc/pfe25:9.log
   ```

   This information indicates that the **DISPLAY** value pfe25:9.0 is set, providing a valid display that your PBS batch jobs can access.

2. In your PBS batch job script, add the following line to instruct the job to use this display:
   - For csh: setenv DISPLAY pfe25:9.0
   - For bash: export DISPLAY=pfe25:9.0
3. Submit your PBS batch job from the PFE connection window where the VNC session is running.

Benefits of using this method:

- Job performance is not impacted.
- The VNC viewer will be maintained as long as the PFE you started it on (pfe25 in this example) is online. The session will stay open even if you exit from your local system or from the PFE connection window where the viewer is running. To reconnect to the session, simply log in to the PFE and repeat steps 3 and 4 in the VNC instructions with the same port number (5909 in this example).

Note: The number of VNC ports on each PFE is limited. You must stop the vncserver script after your PBS job completes, or the port will remain open and owned by you, which prevents other users from using it. Remember to complete step 5 in the VNC instructions to properly shut down your VNC session.

Method 3: Run Your Job on the NAS GPU Nodes

When you run a job on a GPU node, a display is enabled automatically.

Complete these steps to run your job on a GPU node:

1. Modify your PBS script to run startx and set **DISPLAY** to :0.0 by adding these lines:

   ```
   /bin/startx /usr/bin/Xvnc :1 -s 0 -ac -br -listen tcp passwordFile=$HOME/.vnc/passwd &
   ```

```
setenv DISPLAY :1.0       (for csh)
or
export DISPLAY=:1.0       (for bash)
sleep 15  # wait 15 seconds to allow the startx to fully start up before using the display
```

2. Submit a batch job to request a GPU node using the correct queue:

   To request a GPU node:

   ```
   cfe01% qsub -q a100 -lselect=1:model=mil_a100 pbs_script
   ```

   Note: The NAS GPUs with x86-64 CPUs are accessible from cfe01 or cfe02, not the PFEs.

Drawbacks to using this method:

- There are fewer GPU nodes available than non-GPU nodes.
- You cannot use this method if your graphics application needs more than 64 GB of memory.

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.

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
   ```

Packaging Multiple Runs

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
```

Using GNU Parallel to Package Multiple Jobs in a Single PBS Job

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
```

Managing Memory

Checking and Managing Page Cache Usage

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.

Using gm.x to Check Memory Usage

You can use the gm.x tool to check the memory usage of jobs running on NAS x86_64 or aarch64 systems. This tool reports the memory usage from each process at the end of a run. Optionally, gm.x can also report memory usage at a fixed interval during the run.

Note: The NAS mbind tool provides the options -gm, -gmc, -gmr for reporting memory usage information similar to gm.x.

The gm.x and mbind.x tools are available in the /u/scicon/tools/bin directory fo the NAS x86_64 systems and in the /u/.scicon/tools/arm/bin directory for the NAS aarch64 system.

On either the NAS x86_64 or aarch systems, loading the module scicon/cli_tools provides acess to gm.x, mbind.x, and a few other tools for the appropriate architecture.

Use the -h option to find out what types of memory usage can be reported:

```
% gm.x -h
Usage: gm.x [-options] [-h|-he] a.out [args]

Options -
  -hwm     ; high water mark (VmHWM)
  -rss     ; resident memory size (VmRSS)
  -wrss    ; weighted memory size (WRSS)
  -c<s:n>  ; print memory usage every <s> seconds for <n> times
  -r<list> ; print memory usage for selected ranks in the list
  -g<n:m>  ; monitor GPU usage (mem, power, energy)
  -time    ; print timing stats (wtime, utime, stime)
  -memv    ; use variable memory unit
  -v       ; verbose flag

Default: type is set by environment variable GM_TYPE (def=VmHWM)
       memory usage is printed at the end of the run by all ranks
```

Additional information about each option can be found with gm.x -he. Here are brief descriptions for some of the options:

-hwm (default)
: Reports the peak resident set size (high water mark) that the kernel captures for the process. Specifically, the status is taken from the value shown in the VmHWM field of the /proc//status file.

-wrss
: Calls the system function get_weighted_memory_size. (See **man get_weighted_memory_size** for more information about this function.)

-c
: Reports the memory usage during the run; useful in the event that a job fails to complete.

-memv
: Provided for backward compatibility; allows memory usage to be reported in a variable unit. Without this option, memory usage is reported in MB.

The gm.x tool can be used for either OpenMP or MPI applications (linked with HPE's MPT library). You do not need to recompile your application to use it.

To use gm.x for an MPI code, add gm.x just before the executable in the command line. For example:

```
mpiexec -np 4 gm.x ./a.out
Memory usage for (r1i1n0,pid=9767): 1.458 MB (rank=0)
Memory usage for (r1i1n0,pid=9768): 1.413 MB (rank=1)
Memory usage for (r1i1n0,pid=9770): 1.413 MB (rank=3)
Memory usage for (r1i1n0,pid=9769): 1.417 MB (rank=2)
```

You can feed the standard output into the gm_post.x script to compute the total memory used and the average memory used per process. For example:

```
mpiexec -np 4 gm.x ./a.out | gm_post.x -v
Memory usage for (r1i1n0,pid=9767): 1.458 MB (rank=0)
Memory usage for (r1i1n0,pid=9768): 1.413 MB (rank=1)
Memory usage for (r1i1n0,pid=9770): 1.413 MB (rank=3)
Memory usage for (r1i1n0,pid=9769): 1.417 MB (rank=2)

Number of nodes     = 1
Number of processes = 4
Processes per node  = 4
Total memory        = 5.701 MB

Memory per node     = 5.701 MB
Minimum node memory = 5.701 MB
Maximum node memory = 5.701 MB

Memory per process  = 1.425 MB
Minimum proc memory = 1.413 MB
```

```
Maximum proc memory = 1.458 MB
```

If you use dplace to pin the process, add gm.x after dplace in the command line:

```
mpiexec -np NN dplace -s1 gm.x ./a.out
```

The gm.x tool was developed by NAS staff member Henry Jin.

Using qsh.pl to Check Memory Usage

You can use the qsh.pl tool to check the memory usage of jobs running on Aitken or Electra. This tool is a Perl script that uses Secure Shell (SSH) protocol to access all of the nodes used by a PBS job in order to run a command that you supply.

For Aitken and Electra jobs, you can run qsh.pl on the PFEs.

The qsh.pl tool is available in the /u/scicon/tools/bin directory.

TIP: You can either run `module load scicon/cli/tools` or add the directory to your shell startup script so that you can invoke qsh.pl without typing the full path each time. For example:

```
set path = ( $path /u/scicon/tools/bin )
```

Syntax:

pfe21% qsh.pl *job_id command*

You can use this script to check the amount of free memory in the nodes of your PBS job.

Example:

```
pfe21% qsh.pl 30329 "cat /proc/meminfo"

running "cat /proc/meminfo" on:  r56i2n14 r56i2n15
r56i2n14 :
  MemTotal:      8079728 kB
  MemFree:        857936 kB
  Buffers:             0 kB
  Cached:        3775472 kB
...
r56i2n15 :
  MemTotal:      8079728 kB
  MemFree:       5840920 kB
  Buffers:             0 kB
  Cached:         784280 kB
...
```

The qsh.pl script was provided by NAS staff member Bob Hood.

Using qtop.pl to Check Memory Usage

You can use the qtop.pl tool to check the memory usage of jobs running on Aitken or Electra. This tool is a Perl script that uses Secure Shell (SSH) to access the nodes of a PBS job in order to perform the top command. The output provides memory usage information for the whole node and for each process.

For Aitken and Electra jobs, you can run qtop.pl on the PFEs.

The qtop.pl tool is available after loading the module scicon/cli_tools or by adding to your path in the /u/scicon/tools/bin directory.

For example:
```
set path = ( $path /u/scicon/tools/bin )
```

Syntax:

```
pfe21% qtop.pl [-b] [-p n] [-P s] [-h n] [-H s] [-t s] [-N s] job_id
```

-b
    (for running in background or batch) don't run 'resize' command
-p n
    show at most n processes per host
-P s
    show only procs in s, a comma-separated list of ranges
    e.g. -P 1,8-9
-h
    don't show the column header line
-H s
    show only header lines in s, comma-separated ranges
    e.g. -H 1-2,7
    e.g. -H 0 (don't show any lines)
-t s
    pass string s (must be one argument) to top command
-n s
    show output only from nodes in s, comma-separated ranges
    e.g. -n 0,2-3 (relative node numbers)
-N s
    show output only from nodes in s, a comma-separated list
    e.g. -N r1i1n14,r1i1n15 (absolute node numbers)

Example: To skip the header and list 8 procs per host:

```
pfe21% qtop.pl -H 0 -p 8 996093
all nodes in job 996093:  r184i2n12
r184i2n12   PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
        20027 zsmith    25   0 23.8g 148m 5320 R  101  0.6   5172:37 a.out
        20028 zsmith    25   0 23.8g 140m 5140 R  101  0.6   5173:35 a.out
        20029 zsmith    25   0 23.9g 286m 6640 R  101  1.2   5172:23 a.out
        20030 zsmith    25   0 23.9g 245m 5040 R  101  1.0   5171:18 a.out
        20031 zsmith    25   0 23.9g 265m 6040 R  101  1.1   5171:46 a.out
        20032 zsmith    25   0 23.9g 246m 5300 R  101  1.0   5171:00 a.out
        20033 zsmith    25   0 23.8g 158m 5476 R  101  0.7   5172:41 a.out
        20034 zsmith    25   0 23.8g 148m 5280 R  101  0.6   5173:02 a.out
```

The qtop.pl script was provided by NAS staff member Bob Hood.

Using qps to Check Memory Usage

You can use the qps tool to check the memory usage of jobs running on Aitken or Electra. This tool is a Perl script that uses Secure Shell (SSH) protocol to access each node of a running job in order to get process status (ps) information on each node.

For Aitken and Electra jobs, you can run qps on the PFEs.

The qps tool is available in the /u/scicon/tools/bin directory.

TIP: You can run the command module load scicon/cli_tools or add the directory to your shell startup script so that you can invoke qps without typing the full path each time. For example:
set path = ($path /u/scicon/tools/bin)

Syntax:

pfe21% qps *job_id*

Example:

pfe21% qps 26130

*** Job 26130, User abc, Procs 1
NODE TIME %MEM %CPU STAT TASK
r1i0n14 10:17:13 2.8 99.9 RL ./a.out
r1i0n14 10:17:12 2.9 99.9 RL ./a.out
r1i0n14 10:17:18 2.9 99.9 RL ./a.out
r1i0n14 10:16:34 2.9 99.8 RL ./a.out
r1i0n14 10:17:11 2.9 99.9 RL ./a.out
r1i0n14 10:17:13 2.9 99.9 RL ./a.out
r1i0n14 10:17:12 2.9 99.9 RL ./a.out
r1i0n14 10:17:15 2.9 99.9 RL ./a.out

Note: The percentage of memory usage by a process reported by this script is a percentage of the memory in the entire node.

Several factors can affect the amount of memory available for your job and the use of memory by your applications. We provide a number of tools you can use to monitor and manage memory usage.

Physical Memory Available on a Node When a Job Starts

The total physical memory of a NAS compute node varies from 128 GB (compute model bro_ele) to 192 GB (compute models sky_ele, cas_ait) for Intel Xeon processors, or 512 GB (compute models rom_ait, mil_ait) for the AMD Rome and Milan processors. However, the amount of memory on a node that is available for a PBS job is less than the total physical memory, because the system kernel can use up to four GB of memory in each node.

When a job starts running, the PBS prologue tries to clean up the memory on the nodes used by the previous job that ran on the nodes, by flushing the job's data from memory to disk. If there is a delay in this cleanup attempt—for example, due to Lustre issues—less memory will be available for the job. After the prologue has the opportunity to flush the memory, if the available memory is less than 85% of the total physical memory of the node, the prologue will terminate the job and return it to the queue. The node will be taken offline.

WARNING: For nodes with a large amount of total physical memory, the use of the 85% rule in the PBS prologue to start a job can lead to a significant imbalance of available memory for nodes assigned to your job. For example, in the case of Aitken's Rome and Milan nodes, the PBS prologue will start your job if each of the assigned nodes has more than 85% of its 512 GB physical memory available. This could lead to some nodes assigned to your job having only 435 GB while other nodes assigned to your job having more than 500 GB—and the possibility of running out of memory on nodes that had lower available memory when the job started.

Memory Used By and For Your Application

Segments and Processes

When you run an application, each of the following segments takes up space in the virtual memory:

text
: Executable instructions.

data
: Pre-defined data—global and static variables initialized by the programmer.

bss
: Uninitialized data—global and static variables that do not have explicit initialization in source code and/or are initialized to zero by the kernel.

stack
: Used for storing local variables and function parameters of a stack frame, which is created when a function is called and removed when the function returns.

heap
: Used for dynamic memory allocation during runtime.

shared libraries
: For example, math and MPI libraries.

You can run the Linux `size` command to list the bytes required (but not necessarily used) by the text, data, and bss memory segments.

Note: Some user applications have built-in reports of the memory requirement. These reports might not take into account the memory used by the shared libraries.

During runtime, the memory needed by each process in the application is paged (in 4 KB units) into the physical memory. The total amount of physical memory used by each process is called resident set size (RSS), which is reported in the RSS column of the `ps` command output or in the RES column of the `top` command output. The total physical memory used by application processes in a node can be estimated by the sum of the processes' RSS; keep in mind that the sum may be an overestimate, as the usage by the shared libraries might be counted more than once among processes.

Memory resource limits set by the system administrator for the Linux shells (such as `stacksize`, `memoryuse`, `vmemoryuse`, and so on) can affect your application. To see the settings, run the `limit` command (for csh) or `ulimit -a` (for bash) on a compute node. You might need to adjust one or more of the settings for your application to run properly.

Page Cache for Application I/O

Page cache, also known as buffer cache, is used to speed up the disk I/O of your application. Page cache usage is reported in the `Cache Mem` entry in the system usage section of the `top` command output.

If your job performs a large amount of I/O, there will be less memory available for your running processes. For some compute-intensive jobs, page cache management (limiting the amount of memory used by page cache) may improve performance. For more information, see Checking and Managing Page Cache Usage.

The /tmp Filesystem

The /tmp filesystem on a compute node is a fast, memory-based temporary local filesystem. You can access it directly via the /tmp path or by the $TMPDIR environment variable created by PBS for your job under /tmp/pbs.jobid. pbs_servername.nas.nasa.gov. Usage by /tmp is included in the Cache Mem entry of the top command output.

Note: /tmp usage is limited to 50% of the total physical memory on the node. Using more than 50% results in the error No space left on device.

Tools for Checking Memory Usage

To help you assess the memory usage of your job, we provide the following NAS-developed tools:

qtop.pl
: Invokes the top command on the compute nodes of a job, and provides a snapshot of the amount of used and free memory of the whole node and the amount used by each running process.

qps
: Invokes the ps command on the compute nodes of a job, and provides a snapshot of the %mem used by its running processes.

qsh.pl
: Can be used to invoke the cat /proc/meminfo command on the compute nodes to provide a snapshot of the total and free memory in each node.

gm.x
: Provides the memory high-water mark for each process of your job when the job finishes.

Notes:

- The qtop.pl, qpe, and qps, tools can be used only for jobs running on Aitken and Electra. The gm.x tool can be used for jobs running on Aitken, Electra, and Endeavour.
- For Aitken or Electra jobs, you can run the tools on the PFEs.
- If your Aitken or Electra job uses more than one node, be aware that the memory usage reported in the PBS output file is not the total memory usage for your job; rather it is *the memory used in the first node* of your job.

If Your Job Runs Out of Memory

If your Aitken or Electra job runs out of memory it may be killed by the kernel. If the job needs more memory, see How to Get More Memory for Your Job for possible approaches.

WARNING: The out-of-memory event is not always recorded in the system log files and when this happens, no email is sent to the owner of the job. If you suspect that your job may run out of memory, we recommend getting into the habit of monitoring memory usage using the tools listed above. If you need help, contact us at support@nas.nasa.gov.

For jobs running on Endeavour, the Linux kernel cgroup is used to enforce the resources allocated to your job, such as number of cores and amount of memory. When your job reaches the memory enforced by cgroup, your job will be terminated. If this happens, increase the memory request in your PBS script, and resubmit the job.

For additional information, see the recording and slides for the "Memory Usage on NAS Xeon-Based Systems" webinar, which are available in the HECC webinars archive.

How to Get More Memory for your PBS Job

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.

Maximizing Use of Resources

Checkpointing and Restart

NAS systems do not have an automatic checkpoint capability made available by the operating system. For jobs that need a lot of resources, a long wall-clock time, or both, be sure to implement a checkpoint/restart capability in your source code or job script.

PBS automatically restarts unfinished jobs after system crashes. If you do not want PBS to restart your job, add the following line in your PBS script:

```
#PBS -r n
```

Process/Thread Pinning

Pinning, the binding of a process or thread to a specific core, can improve the performance of your code.

You can insert the MPI function mpi_get_processor_name and the Linux C function sched_getcpu into your source code to check process and/or thread placement. The MPI function mpi_get_processor_name returns the hostname an MPI process is running on (to be used for MPI and/or MPI+OpenMP codes only). The Linux C function sched_getcpu returns the processor number the process/thread is running on.

If your source code is written in Fortran, you can use the C codemycpu.c, which allows your Fortran code to call sched_getcpu. The next section describes how to use the mycpu.c code.

C Program mycpu.c

```
#include <utmpx.h>
int sched_getcpu();

int findmycpu_ ()
{
   int cpu;
   cpu = sched_getcpu();
   return cpu;
}
```

Compile mycpu.c as follows to produce the object file mycpu.o:

```
pfe21% module load comp-intel/2020.4.304
pfe21% icc -c mycpu.c
```

The following example demonstrates how to instrument an MPI+OpenMP source code with the above functions. The added lines are highlighted.

```
      program your_program
      use omp_lib
...
      integer :: resultlen, tn, cpu
      integer, external :: findmycpu
      character (len=8) :: name

      call mpi_init( ierr )
      call mpi_comm_rank( mpi_comm_world, rank, ierr )
      call mpi_comm_size( mpi_comm_world, numprocs, ierr )
      call mpi_get_processor_name(name, resultlen, ierr)
!$omp parallel

      tn = omp_get_thread_num()
      cpu = findmycpu()
      write (6,*) 'rank ', rank, ' thread ', tn,
     &  ' hostname ', name, ' cpu ', cpu
.....
!$omp end parallel
      call mpi_finalize(ierr)
      end
```

Compile your instrumented code as follows:

```
pfe21% module load comp-intel/2020.4.304
pfe21% module load mpi-hpe/mpt
pfe21% ifort -o a.out -qopenmp mycpu.o your_program.f -lmpi
```

Sample PBS script

The following PBS script provides an example of running the hybrid MPI+OPenMP code across two nodes, with 2 MPI processes per node and 4 OpenMP threads per process, and using the mbind tool to pin the processes and threads.

```
#PBS -lselect=2:ncpus=28:mpiprocs=2:model=bro
#PBS -lwalltime=0:10:00

cd $PBS_O_WORKDIR

module load comp-intel/2020.4.304
module load mpi-hpe/mpt

mpiexec -np 4 mbind.x -cs -t4 -v ./a.out
```

Here is a sample output:

These 4 lines are generated by mbind only if you have included the -v option:
host: r627i4n1, ncpus: 56, rank: 0 (r0), nthreads: 4, bound to cpus: {0-9:3}
host: r627i4n1, ncpus: 56, rank: 1 (r1), nthreads: 4, bound to cpus: {14-23:3}
host: r627i4n8, ncpus: 56, rank: 2 (r0), nthreads: 4, bound to cpus: {0-9:3}
host: r627i4n8, ncpus: 56, rank: 3 (r1), nthreads: 4, bound to cpus: {14-23:3}

These lines are generated by your instrumented code:
rank 0 thread 3 hostname r627i4n1 cpu 0
rank 0 thread 3 hostname r627i4n1 cpu 3
rank 0 thread 3 hostname r627i4n1 cpu 6
rank 0 thread 3 hostname r627i4n1 cpu 9
rank 1 thread 3 hostname r627i4n1 cpu 14
rank 1 thread 3 hostname r627i4n1 cpu 17
rank 1 thread 3 hostname r627i4n1 cpu 20
rank 1 thread 3 hostname r627i4n1 cpu 23
rank 2 thread 3 hostname r627i4n8 cpu 0
rank 2 thread 3 hostname r627i4n8 cpu 3
rank 2 thread 3 hostname r627i4n8 cpu 6
rank 2 thread 3 hostname r627i4n8 cpu 9
rank 3 thread 3 hostname r627i4n8 cpu 14
rank 3 thread 3 hostname r627i4n8 cpu 17
rank 3 thread 3 hostname r627i4n8 cpu 20
rank 3 thread 3 hostname r627i4n8 cpu 23

Note: In your output, these lines may be listed in a different order.

For MPI codes built with HPE's MPT libraries, one way to control pinning is to set certain MPT memory placement environment variables. For an introduction to pinning at NAS, see Process/Thread Pinning Overview.

MPT Environment Variables

Here are the MPT memory placement environment variables:

MPI_DSM_VERBOSE

Directs MPI to display a synopsis of the NUMA and host placement options being used at run time to the standard error file.

Default: Not enabled

The setting of this environment variable is ignored if MPI_DSM_OFF is also set.

MPI_DSM_DISTRIBUTE

Activates NUMA job placement mode. This mode ensures that each MPI process gets a unique CPU and physical memory on the node with which that CPU is associated. Currently, the CPUs are chosen by simply starting at relative CPU 0 and incrementing until all MPI processes have been forked.

Default: Enabled

WARNING: If the nodes used by your job are not fully populated with MPI processes, use MPI_DSM_CPULIST, dplace, or omplace for pinning instead of MPI_DSM_DISTRIBUTE.

The MPI_DSM_DISTRIBUTE setting is ignored if MPI_DSM_CPULIST is also set, or if dplace or omplace are used.

MPI_DSM_CPULIST

Specifies a list of CPUs on which to run an MPI application, excluding the shepherd process(es) and mpirun. The number of CPUs specified should equal the number of MPI processes (excluding the shepherd process) that will be used.

Syntax and examples for the list:

- Use a comma and/or hyphen to provide a delineated list:

```
# place MPI processes ranks 0-2 on CPUs 2-4
# and ranks 3-5 on CPUs 6-8
setenv MPI_DSM_CPULIST "2-4,6-8"
```

- Use a "/" and a stride length to specify CPU striding:

```
# Place the MPI ranks 0 through 3 stridden
# on CPUs 8, 10, 12, and 14
setenv MPI_DSM_CPULIST 8-15/2
```

- Use a colon to separate CPU lists of multiple hosts:

```
# Place the MPI processes 0 through 7 on the first host
# on CPUs 8 through 15. Place MPI processes 8 through 15
# on CPUs 16 to 23 on the second host.
setenv MPI_DSM_CPULIST 8-15:16-23
```

- Use a colon followed by allhosts to indicate that the prior list pattern applies to all subsequent hosts/executables:

```
# Place the MPI processes onto CPUs 0, 2, 4, 6 on all hosts
setenv MPI_DSM_CPULIST 0-7/2:allhosts
```

Examples

An MPI job requesting 2 nodes and running 4 MPI processes per node will get the following placements, depending on the environment variables set:

Note: The example below uses a hypothetical model xxx with 8 cores per node for demonstration. The current NAS compute nodes all have a much larger number of cores per node. Adjust your PBS resource request and the number of MPI processes according to your usage.

```
#PBS -lselect=2:ncpus=8:mpiprocs=4
module load <mpt_module>
setenv ....
cd $PBS_O_WORKDIR
```

mpiexec -np 8 ./a.out

- setenv MPI_DSM_VERBOSE
  setenv MPI_DSM_DISTRIBUTE

  MPI: DSM information
  MPI: MPI_DSM_DISTRIBUTE enabled

grank	lrank	pinning	node name	cpuid
0	0	yes	r86i3n5	0
1	1	yes	r86i3n5	1
2	2	yes	r86i3n5	2
3	3	yes	r86i3n5	3
4	0	yes	r86i3n6	0
5	1	yes	r86i3n6	1
6	2	yes	r86i3n6	2
7	3	yes	r86i3n6	3

- setenv MPI_DSM_VERBOSE
  setenv MPI_DSM_CPULIST 0,2,4,6

  MPI: WARNING MPI_DSM_CPULIST CPU placement spec list is too short.
  MPI: MPI processes on host #1 and later will not be pinned.
  MPI: DSM information

grank	lrank	pinning	node name	cpuid
0	0	yes	r22i1n7	0
1	1	yes	r22i1n7	2
2	2	yes	r22i1n7	4
3	3	yes	r22i1n7	6
4	0	no	r22i1n8	0
5	1	no	r22i1n8	0
6	2	no	r22i1n8	0
7	3	no	r22i1n8	0

- setenv MPI_DSM_VERBOSE
  setenv MPI_DSM_CPULIST 0,2,4,6:0,2,4,6

  MPI: DSM information

grank	lrank	pinning	node name	cpuid
0	0	yes	r13i2n12	0
1	1	yes	r13i2n12	2
2	2	yes	r13i2n12	4
3	3	yes	r13i2n12	6
4	0	yes	r13i3n7	0
5	1	yes	r13i3n7	2
6	2	yes	r13i3n7	4
7	3	yes	r13i3n7	6

- setenv MPI_DSM_VERBOSE
  setenv MPI_DSM_CPULIST 0,2,4,6:allhosts

  MPI: DSM information

grank	lrank	pinning	node name	cpuid
0	0	yes	r13i2n12	0
1	1	yes	r13i2n12	2
2	2	yes	r13i2n12	4
3	3	yes	r13i2n12	6
4	0	yes	r13i3n7	0
5	1	yes	r13i3n7	2
6	2	yes	r13i3n7	4
7	3	yes	r13i3n7	6

Using the omplace Tool for Pinning

HPE's omplace is a wrapper script for dplace. It pins processes and threads for better performance and provides an easier syntax than dplace for pinning processes and threads.

The omplace wrapper works with HPE MPT as well as with Intel MPI. In addition to pinning pure MPI or pure OpenMP applications, omplace can also be used for pinning hybrid MPI/OpenMP applications.

A few issues with omplace to keep in mind:

- dplace and omplace do not work with Intel compiler versions 10.1.015 and 10.1.017. Use the Intel compiler version 11.1 or later, instead
- To avoid interference between dplace/omplace and Intel's thread affinity interface, set the environment variable KMP_AFFINITY to disabled or set OMPLACE_AFFINITY_COMPAT to ON
- The omplace script is part of HPE's MPT, and is located under the /nasa/hpe/mpt/*mpt_version_number*/bin directory

Syntax

For OpenMP:
setenv OMP_NUM_THREADS *nthreads*
omplace [OPTIONS] *program args...*
or
omplace -nt *nthreads* [OPTIONS] *program args...*

For MPI:
mpiexec -np *nranks* omplace [OPTIONS] *program args...*

For MPI/OpenMP hybrid:
setenv OMP_NUM_THREADS *nthreads*
mpiexec -np *nranks* omplace [OPTIONS] *program args...*
or
mpiexec -np *nranks* omplace -nt *nthreads* [OPTIONS] *program args...*

Some useful omplace options are listed below:

WARNING: For omplace, a blank space is required between -c and cpulist. Without the space, the job will fail. This is different from dplace.

-b *basecpu*
: Specifies the starting CPU number for the effective CPU list.

-c *cpulist*
: Specifies the effective CPU list. This is a comma-separated list of CPUs or CPU ranges.

-nt *nthreads*
: Specifies the number of threads per MPI process. If this option is unspecified, it defaults to the value set for the OMP_NUM_THREADS environment variable. If OMP_NUM_THREADS is not set, then *nthreads* defaults to 1.

-v
: Verbose option. Portions of the automatically generated placement file will be displayed.

-vv
: Very verbose option. The automatically generated placement file will be displayed in its entirety.

For information about additional options, see **man omplace**.

Examples

Note: For simplicity of illustration, the examples shown here use an older Intel Westmere processor with a total of 12 physical cores with 6 cores per socket. The Westmere processors are no longer in service at NAS. Replace the Westmere model with the appropriate one you are using, and adjust the CPU list after the omplace -c option accordingly. See the configuation of each processor for details.

- Broadwell processors
- Skylake processors
- Cascade Lake processors
- Rome processors
- Milan processors

For Pure OpenMP Codes Using the Intel OpenMP Library

Sample PBS script:

```
#PBS -l select=1:ncpus=12:model=wes

module load comp-intel/
setenv KMP_AFFINITY disabled
```

```
omplace -c 0,3,6,9 -vv ./a.out
```

Sample placement information for this script is given in the application's stout file:

```
omplace: placement file /tmp/omplace.file.21891
   firsttask cpu=0
   thread oncpu=0 cpu=3-9:3 noplace=1  exact
```

The above placement output may not be easy to understand. A better way to check the placement is to run the ps command on the running host while the job is still running:

```
ps -C a.out -L -opsr,comm,time,pid,ppid,lwp > placement.out
```

Sample output of placement.out

```
PSR COMMAND          TIME   PID  PPID   LWP
  0 openmp1       00:00:02 31918 31855 31918
 19 openmp1       00:00:00 31918 31855 31919
  3 openmp1       00:00:02 31918 31855 31920
  6 openmp1       00:00:02 31918 31855 31921
  9 openmp1       00:00:02 31918 31855 31922
```

Note that Intel OpenMP jobs use an extra thread that is unknown to the user, and does not need to be placed. In the above example, this extra thread is running on logical core number 19.

For Pure MPI Codes Using HPE MPT

Sample PBS script:

```
#PBS -l select=2:ncpus=12:mpiprocs=4:model=wes

module load comp-intel/
module load mpi-hpe/mpt

#Setting MPI_DSM_VERBOSE allows the placement information
#to be printed to the PBS stderr file

setenv MPI_DSM_VERBOSE

mpiexec -np 8 omplace -c 0,3,6,9 ./a.out
```

Sample placement information for this script is shown in the PBS stderr file:

```
MPI: DSM information
MPI: using dplace
grank  lrank  pinning  node name      cpuid
   0      0   yes      r144i3n12          0
   1      1   yes      r144i3n12          3
   2      2   yes      r144i3n12          6
   3      3   yes      r144i3n12          9
   4      0   yes      r145i2n3           0
   5      1   yes      r145i2n3           3
   6      2   yes      r145i2n3           6
   7      3   yes      r145i2n3           9
```

In this example, the four processes on each node are evenly distributed to the two sockets (CPUs 0 and 3 are on the first socket while CPUs 6 and 9 on the second socket) to minimize contention. If omplace had not been used, then placement would follow the rules of the environment variable OMP_DSM_DISTRIBUTE, and all four processes would have been placed on the first socket—likely leading to more contention.

For MPI/OpenMP Hybrid Codes Using HPE MPT and Intel OpenMP

Proper placement is more critical for MPI/OpenMP hybrid codes than for pure MPI or pure OpenMP codes. The following example demonstrates the situation when no placement instruction is provided and the OpenMP threads for each MPI process are stepping on one another which likely would lead to very bad performance.

Sample PBS script without pinning:

```
#PBS -l select=2:ncpus=12:mpiprocs=4:model=wes

module load comp-intel/
module load mpi-hpe/mpt
setenv OMP_NUM_THREADS 2

mpiexec -np 8 ./a.out
```

There are two problems with the resulting placement shown in the example above. First, you can see that the first four MPI processes on each node are placed on four cores (0,1,2,3) of the same socket, which will likely lead to more contention compared to when they are distributed between the two sockets.

```
MPI: MPI_DSM_DISTRIBUTE enabled
grank  lrank  pinning  node name      cpuid
   0      0   yes      r212i0n10          0
   1      1   yes      r212i0n10          1
   2      2   yes      r212i0n10          2
   3      3   yes      r212i0n10          3
   4      0   yes      r212i0n11          0
   5      1   yes      r212i0n11          1
   6      2   yes      r212i0n11          2
   7      3   yes      r212i0n11          3
```

The second problem is that, as demonstrated with the ps command below, the OpenMP threads are also placed on the same core where the associated MPI process is running:

```
ps -C a.out -L -opsr,comm,time,pid,ppid,lwp

PSR COMMAND         TIME   PID  PPID   LWP
  0 a.out       00:00:02  4098  4092  4098
  0 a.out       00:00:02  4098  4092  4108
  0 a.out       00:00:02  4098  4092  4110
  1 a.out       00:00:03  4099  4092  4099
  1 a.out       00:00:03  4099  4092  4106
  2 a.out       00:00:03  4100  4092  4100
  2 a.out       00:00:03  4100  4092  4109
  3 a.out       00:00:03  4101  4092  4101
  3 a.out       00:00:03  4101  4092  4107
```

Sample PBS script demonstrating proper placement:

```
#PBS -l select=2:ncpus=12:mpiprocs=4:model=wes

module load mpi-hpe/mpt
module load comp-intel/

setenv MPI_DSM_VERBOSE
setenv OMP_NUM_THREADS 2
setenv KMP_AFFINITY disabled

cd $PBS_O_WORKDIR

#the following two lines will result in identical placement

mpiexec -np 8 omplace -nt 2 -c 0,1,3,4,6,7,9,10 -vv ./a.out
#mpiexec -np 8 omplace -nt 2 -c 0-10:bs=2+st=3 -vv  ./a.out
```

Shown in the PBS stderr file, the 4 MPI processes on each node are properly distributed on the two sockets with processes 0 and 1 on CPUs 0 and 3 (first socket) and processes 2 and 3 on CPUs 6 and 9 (second socket).

```
MPI: DSM information
MPI: using dplace
grank  lrank  pinning  node name      cpuid
   0      0   yes      r212i0n10          0
   1      1   yes      r212i0n10          3
   2      2   yes      r212i0n10          6
   3      3   yes      r212i0n10          9
   4      0   yes      r212i0n11          0
   5      1   yes      r212i0n11          3
   6      2   yes      r212i0n11          6
   7      3   yes      r212i0n11          9
```

In the PBS stout file, it shows the placement of the two OpenMP threads for each MPI process:

```
omplace: This is an HPE MPI program.
omplace: placement file /tmp/omplace.file.6454
    fork skip=0  exact cpu=0-10:3
    thread oncpu=0 cpu=1 noplace=1  exact
    thread oncpu=3 cpu=4 noplace=1  exact
    thread oncpu=6 cpu=7 noplace=1  exact
    thread oncpu=9 cpu=10 noplace=1  exact
omplace: This is an HPE MPI program.
omplace: placement file /tmp/omplace.file.22771
    fork skip=0  exact cpu=0-10:3
    thread oncpu=0 cpu=1 noplace=1  exact
    thread oncpu=3 cpu=4 noplace=1  exact
    thread oncpu=6 cpu=7 noplace=1  exact
    thread oncpu=9 cpu=10 noplace=1  exact
```

To get a better picture of how the OpenMP threads are placed, using the following ps command:

```
ps -C a.out -L -opsr,comm,time,pid,ppid,lwp

PSR COMMAND         TIME   PID  PPID   LWP
  0 a.out       00:00:06  4436  4435  4436
```

```
 1 a.out      00:00:03  4436  4435  4447
 1 a.out      00:00:03  4436  4435  4448
 3 a.out      00:00:06  4437  4435  4437
 4 a.out      00:00:05  4437  4435  4446
 6 a.out      00:00:06  4438  4435  4438
 7 a.out      00:00:05  4438  4435  4444
 9 a.out      00:00:06  4439  4435  4439
10 a.out      00:00:05  4439  4435  4445
```

The Intel compiler's OpenMP runtime library has the ability to bind OpenMP threads to physical processing units. Depending on the system topology, application, and operating system, thread affinity can have a dramatic effect on code performance. We recommend two approaches for using the Intel OpenMP thread affinity capability.

Using the KMP_AFFINITY Environment Variable

The thread affinity interface is controlled using the KMP_AFFINITY environment variable.

Syntax

For csh and tcsh:

setenv KMP_AFFINITY [<*modifier*>,...]<*type*>[,<*permute*>][,<*offset*>]

For sh, bash, and ksh:

export KMP_AFFINITY=[<*modifier*>,...]<*type*>[,<*permute*>][,<*offset*>]

Using the Compiler Flag -par-affinity Compiler Option

Note: This approach does not apply when using the Intel OneAPI ifx compiler which does not support the -par-affinity flag.

Starting with the Intel compiler version 11.1, thread affinity can be specified through the compiler option -par-affinity. The use of -openmp or -parallel is required in order for this option to take effect. This option overrides the environment variable when both are specified. See **man ifort** for more information.

Note: Starting with comp-intel/2015.0.090, -openmp is deprecated and has been replaced with -qopenmp.

Syntax

-par-affinity=[<*modifier*>,...]<*type*>[,<*permute*>][,<*offset*>]

Possible Values for type

For both of the recommended approaches, the only required argument is type, which indicates the type of thread affinity to use. Descriptions of all of the possible arguments (type, modifier, permute, and offset) can be found in man ifort.

Recommendation: Use Intel compiler versions 11.1 and later, as some of the affinity methods described below are not supported in earlier versions.

Possible values for type are:

type = none (default)
: Does not bind OpenMP threads to particular thread contexts; however, if the operating system supports affinity, the compiler still uses the OpenMP thread affinity interface to determine machine topology. Specify KMP_AFFINITY=verbose,none to list a machine topology map.

type = disabled
: Specifying disabled completely disables the thread affinity interfaces. This forces the OpenMP runtime library to behave as if the affinity interface was not supported by the operating system. This includes implementations of the low-level API interfaces such as kmp_set_affinity and kmp_get_affinity that have no effect and will return a nonzero error code.

Additional information from Intel:

"The thread affinity type of KMP_AFFINITY environment variable defaults to none (KMP_AFFINITY=none). The behavior for KMP_AFFINITY=none was changed in 10.1.015 or later, and in all 11.*x* compilers, such that the initialization thread creates a "full mask" of all the threads on the machine, and every thread binds to this mask at startup time. It was subsequently found that this change may interfere with other platform affinity mechanism, for example, dplace() on Altix machines. To resolve this issue, a new affinity type disabled was introduced in compiler 10.1.018, and in all 11.*x* compilers (KMP_AFFINITY=disabled). Setting KMP_AFFINITY=disabled will prevent the runtime library from making any affinity-related system calls."

type = compact
: Specifying compact causes the threads to be placed as close together as possible. For example, in a topology map, the nearer a core is to the root, the more significance the core has when sorting the threads.

Usage example:

```
# for sh, ksh, bash
export KMP_AFFINITY=compact,verbose

# for csh, tcsh
```

```
setenv KMP_AFFINITY compact,verbose
```

type = scatter
: Specifying scatter distributes the threads as evenly as possible across the entire system. Scatter is the opposite of compact.

Note: For most OpenMP codes, type=scatter should provide the best performance, as it minimizes cache and memory bandwidth contention for all processor models.

Usage example:

```
# for sh, ksh, bash
export KMP_AFFINITY=scatter,verbose

# for csh, tcsh
setenv KMP_AFFINITY scatter,verbose
```

type = explicit
: Specifying explicit assigns OpenMP threads to a list of OS proc IDs that have been explicitly specified by using the proclist=*modifier*, which is required for this affinity type.

Usage example:

```
# for sh, ksh, bash
export KMP_AFFINITY="explicit,proclist=[0,1,4,5],verbose"

# for csh, tcsh
setenv KMP_AFFINITY "explicit,proclist=[0,1,4,5],verbose"
```

For nodes that support hyperthreading, you can use the granularity modifier to specify whether to pin OpenMP threads to physical cores using granularity=core (the default) or pin to logical cores using granularity=fine or granularity=thread for the compact and scatter types.

Examples

The following example illustrates the thread placement of an OpenMP job with four threads on an Intel Broadwell node with two sockets with different thread affinity methods. The variable OMP_NUM_THREADS is set to 4:

```
# for sh, ksh, bash
export OMP_NUM_THREADS=4

# for csh, tcsh
setenv OMP_NUM_THREADS 4
```

The use of the verbose modifier is recommended, as it provides an output with the placement.

Broadwell

As seen in the configuration diagram of a Broadwell node, each set of 14 physical cores in a socket share the same L3 cache.

Four threads running on 1 node (28 physical cores and 56 logical cores due to hyperthreading) of Broadwell will get the following thread placement:

Setting of KMP_AFFINITY	processor id	0,28	1,29	2,30	3,31	4,32	5,33	Click Here to Expand Table
granularity=core,compact,verbose	thread id	0,1	2,3					
granularity=core,scatter,verbose	thread id	0	2					
“explicit_proclist=[0,7,14,21],verbose”	thread id	0						

Process/Thread Pinning Overview

Pinning, the binding of a process or thread to a specific core, can improve the performance of your code by increasing the percentage of local memory accesses.

Once your code runs and produces correct results on a system, the next step is performance improvement. For a code that uses multiple cores, the placement of processes and/or threads can play a significant role in performance.

Given a set of processor cores in a PBS job, the Linux kernel usually does a reasonably good job of mapping processes/threads to physical cores, although the kernel may also migrate processes/threads. Some OpenMP runtime libraries and MPI libraries may also perform certain placements by default. In cases where the placements by the kernel or the MPI or OpenMP libraries are not optimal, you can try several methods to control the placement in order to improve performance of your code. Using the same placement from run to run also has the added benefit of reducing runtime variability.

Pay attention to maximizing data locality while minimizing latency and resource contention, and have a clear understanding of the characteristics of your own code and the machine that the code is running on.

Characteristics of NAS Systems

NAS provides two distinctly different types of systems: Aitken and Electra are cluster systems, and Endeavour is a global shared-memory system. Each type is described in this section.

Aitken and Electra

On Aitken and Electra, memory on each node is accessible and shared only by the processes and threads running on that node. Electra is a cluster system that consists of Broadwell and Skylake nodes, and Aitken is a cluster system that consists of Cascade Lake, AMD Rome, and Milan nodes.

Each node contains two sockets, with a symmetric memory system inside each socket. These nodes are considered non-uniform memory access (NUMA) systems, and memory is accessed across the two sockets through the inter-socket interconnect. So, for optimal performance, data locality should not be overlooked on these processor types.

However, compared to a global shared-memory NUMA system such as Endeavour, data locality is less of a concern on the cluster systems. Rather, minimizing latency and resource contention will be the main focus when pinning processes/threads on these systems.

For more information on Aitken and Electra, see the following articles:

- Aitken Configuration Details
- Electra Configuration Details

Endeavour

Endeavour comprises two hosts. Each host is a NUMA system that contains 32 sockets with a total of 896 cores. A process/thread can access the local memory on its socket, remote memory across sockets within the same chassis through the Ultra Path Interconnnect, and remote memory across chassis through the HPE Superdome Flex ASICs, with varying latencies. So, data locality is critical for achieving good performance on Endeavour.

Note: When developing an application, we recommend that you initialize data in parallel so that each processor core initializes the data it is likely to access later for calculation.

For more information, see Endeavour Configuration Details.

Methods for Process/Thread Pinning

Several pinning approaches for OpenMP, MPI and MPI+OpenMP hybrid applications are listed below. We recommend using the Intel compiler (and its runtime library) and the HPE MPT software on NAS systems, so most of the approaches pertain specifically to them. You can also use the mbind tool for multiple OpenMP libraries and MPI environments.

OpenMP codes

- Using Intel OpenMP Thread Affinity for Pinning
- Using the dplace Tool for Pinning
- Using the omplace Tool for Pinning
- Using the mbind Tool for Pinning

MPI codes

- Setting HPE MPT Environment Variables
- Using the omplace Tool for Pinniing

- Using the mbind Tool for Pinning

MPI+OpenMP hybrid codes

- Using the omplace Tool for Pinning
- Using the mbind Tool for Pinning

Checking Process/Thread Placement

Each of the approaches listed above provides some verbose capability to print out the tool's placement results. In addition, you can check the placement using the following approaches.

Use the ps Command

ps -C *executable_name* -L -opsr,comm,time,pid,ppid,lwp

In the generated output, use the core ID under the PSR column, the process ID under the PID column, and the thread ID under the LWP column to find where the processes and/or threads are placed on the cores.

Note: The ps command provides a snapshot of the placement at that specific time. You may need to monitor the placement from time to time to make sure that the processes/threads do not migrate.

Instrument your code to get placement information

- Call the mpi_get_processor_name function to get the name of the processor an MPI process is running on
- Call the Linux C function sched_getcpu() to get the processor number that the process or thread is running on

For more information, see Instrumenting your Fortran Code to Check Process/Thread Placement.

Hyperthreading

Hyperthreading is available and enabled on the Aitken and Electra compute nodes. With hyperthreading, each physical core can function as two logical processors. This means that the operating system can assign two threads per core by assigning one thread to each logical processor.

The following table shows the number of physical cores and potential logical processors available for each processor type:

Processor Model	Physical Cores (N)	Logical Processors (2N)
Broadwell	28	56
Skylake	40	80
Cascade Lake	40	80
Rome	128	256
Milan	128	256

If you use hyperthreading, you can run an MPI code using 2*N* processes per node instead of *N* process per node—so you can use half the number of nodes for your job. Each process will be assigned to run on one logical processor; in reality, two processes are running on the same physical core.

Running two processes per core can take less than twice the wall-clock time compared to running only one process per core—if one process does not keep the functional units in the core busy, and can share the resources in the core with another process.

Benefits and Drawbacks

Using hyperthreading can improve the overall throughput of your jobs, potentially saving standard billing unit (SBU) charges. Also, requesting half the usual number of nodes may allow your job to start running sooner—an added benefit when the systems are loaded with many jobs. However, using hyperthreading may not always result in better performance.

WARNING: Hyperthreading does not benefit all applications. Also, some applications may show improvement with some process counts but not with others, and there may be other unforeseen issues. Therefore, before using this technology in your production run, you should test your applications with and without hyperthreading. If your application runs more than two times slower with hyperthreading than without, do not use it.

Using Hyperthreading

Hyperthreading can improve the overall throughput, as demonstrated in the following example.

Example

Consider the following scenario with a job that uses 80 MPI ranks on Skylake. Without hyperthreading, we would specify:

#PBS -lselect=2:ncpus=40:**mpiprocs=40**:model=sky_ele

and the job will use 2 nodes with 40 processes per node. Suppose that the job takes 1000 seconds when run this way. If we run the job with hyperthreading, e.g.:

#PBS -l**select=1**:ncpus=40:**mpiprocs=80**:model=sky_ele

then the job will use 1 node with all 80 processes running on that node. Suppose this job takes 1800 seconds to complete.

Without hyperthreading, we used 2 nodes for 1000 seconds (a total of 2000 node-seconds); with hyperthreading, we used 1 node for 1800 seconds (1800 node-seconds). Thus, under these circumstances, if you were interested in getting the best wall-clock time performance for a single job, you would use two nodes without hyperthreading. However, if you were interested in minimizing resource usage, especially with multiple jobs running simultaneously, using hyperthreading would save you 10% in SBU charges.

Mapping of Physical Core IDs and Logical Processor IDs

Mapping between the physical core IDs and the logical processor IDs is summarized in the following table. The value of*N* is 28, 40, and 128 for Broadwell, Skylake/Cascade Lake, and Rome/Milan processor types, respectively.

Physical ID	Physical Core ID	Logical Processor ID
0	0	0 ; *N*
0	1	1 ; *N*+1
...	...	.....
0	*N*/2 - 1	*N*/2 - 1; *N* + *N*/2 - 1
1	*N*/2	*N*/2 ; *N* + *N*/2
1	*N*/2 + 1	*N*/2 + 1; *N* + *N*/2 + 1
1	...	...
1	*N* - 1	*N* - 1; 2*N* - 1

Note: For additional mapping details, see the configuration diagrams at the for each processor type, or run the `cat /proc/cpuinfo` command on the specific node type.

The dplace tool binds processes/threads to specific processor cores to improve your code performance. For an introduction to pinning at NAS, see Process/Thread Pinning Overview.

Once pinned, the processes/threads do not migrate. This can improve the performance of your code by increasing the percentage of local memory accesses.

dplace invokes a kernel module to create a job placement container consisting of all (or a subset of) the CPUs of the cpuset. In the current dplace version 2, an LD_PRELOAD library (libdplace.so) is used to intercept calls to the functions fork(), exec(), and pthread_create() to place tasks that are being created. Note that tasks created internal to glib are not intercepted by the preload library. These tasks will *not* be placed. If no placement file is being used, then the dplace process is placed in the job placement container and (by default) is bound to the first CPU of the cpuset associated with the container.

Note: The dplace tool is avalable on the NAS x86_64 compute nodes accessed through PBS.

Syntax

```
dplace [-e] [-v 1|2] [-o <file>] [-c <cpu_list>] [-n <process_name>] [-r <opts>] \
    [-s <skip_count>] [-x <no_place>] command [args...]
dplace [-v 1|2] [-rn|rs|-rt] [-o <file>] -p <placement_file> command [args...]
dplace [-q|-qq|-qqq]
```

If no placement file is being used, then the dplace process is placed in this container and (by default) is bound to the first cpu of the cpuset associated with the container. Then dplace "execs" the <command>. The command is executing within this placement container and continues to be bound to the first cpu of the container. As the command forks child processes, they inherit the container and are bound to the next available cpu of the container.

If a placement file is being used, then the dplace process is not placed at the time the job placement container is created. Instead, placement occurs as processes are forked and executed.

Options for dplace

Explanations for some of the options are provided below. For additional information, see **man dplace**.

-e and -c *cpu_numbers*

-e determines exact placement. As processes are created, they are bound to CPUs in the exact order specified in the CPU list. CPU numbers may appear multiple times in the list.

A CPU value of "x" indicates that binding should *not* be done for that process. If the end of the list is reached, binding starts over again at the beginning of the list.

-c *cpu_numbers* specifies a list of CPUs, optionally strided CPU ranges, or a striding pattern. For example:

- -c 1
- -c 2-4 (equivalent to -c 2,3,4)
- -c 12-8 (equivalent to -c 12,11,10,9,8)
- -c 1,4-8,3
- -c 2-8:3 (equivalent to -c 2,5,8)
- -c CS
- -c BT

Note: CPU numbers are *not* physical CPU numbers. They are logical CPU numbers that are relative to the CPUs that are in the allowed set, as specified by the current cpuset.

A CPU value of "x" (or *), in the argument list for the -c option, indicates that binding should not be done for that process. The value "x" should be used only if the -e option is also used.

Note that CPU numbers start at 0.

For striding patterns, any subset of the characters (B)lade, (S)ocket, (C)ore, (T)hread may be used; their ordering specifies the nesting of the iteration. For example, SC means to iterate all the cores in a socket before moving to the next CPU socket, whileCB means to pin to the first core of each blade, then the second core of every blade, and so on.

For best results, use the -e option when using stride patterns. If the -c option is not specified, all CPUs of the current cpuset are available. The command itself (which is "execed" by dplace) is the first process to be placed by the -c *cpu_numbers*.

Without the -e option, the order of numbers for the -c option is not important.

-x *skip_mask*
: Provides the ability to skip placement of processes. The *skip_mask* argument is a bitmask. If bit *N* of *skip_mask* is set, then the *N*+1th process that is forked is not placed. For example, setting the mask to 6 prevents the second and third processes from

being placed. The first process (the process named by the command) will be assigned to the first CPU. The second and third processes are not placed. The fourth process is assigned to the second CPU, and so on. This option is useful for certain classes of threaded applications that spawn a few helper processes that typically do not use much CPU time.

-s *skip_count*

Skips the first *skip_count* processes before starting to place processes onto CPUs. This option is useful if the first *skip_count* processes are "shepherd" processes used only for launching the application. If *skip_count* is not specified, a default value of 0 is used.

-q

Lists the global count of the number of active processes that have been placed (by dplace) on each CPU in the current cpuset. Note that CPU numbers are logical CPU numbers within the cpuset, not physical CPU numbers. If specified twice, lists the current dplace jobs that are running. If specified three times, lists the current dplace jobs and the tasks that are in each job.

-o *log_file*

Writes a trace file to *log_file* that describes the placement actions that were made for each fork, exec, etc. Each line contains a time-stamp, process id:thread number, CPU that task was executing on, taskname and placement action. Works with version 2 only.

Examples of dplace Usage

For OpenMP Codes

```
#PBS -lselect=1:ncpus=8

#With Intel compiler versions 10.1.015 and later,
#you need to set KMP_AFFINITY to disabled
#to avoid the interference between dplace and
#Intel's thread affinity interface.

setenv KMP_AFFINITY disabled

#The -x2 option provides a skip map of 010 (binary 2) to
#specify that the 2nd thread should not be bound. This is
#because under the new kernels, the master thread (first thread)
#will fork off one monitor thread (2nd thread) which does
#not need to be pinned.

#Choose how you want to place the threads carefully.
#Below is just an example, which may not be the best
#for the processor types.

dplace -x2 -c 2,1,4,5 ./a.out
```

To check the thread placement, you can add the -o option to create a log:

```
dplace -x2 -c 2,1,4,5 -o log_file ./a.out
```

Or use the following command on the running host while the job is still running:

```
ps -C a.out -L -opsr,comm,time,pid,ppid,lwp > placement.out
```

Sample Output of log_file

```
timestamp       process:thread cpu taskname| placement action
15:32:42.196786 31044          1 dplace    | exec ./openmp1, ncpu 1
15:32:42.210628 31044:0        1 a.out     | load, cpu 1
15:32:42.211785 31044:0        1 a.out     | pthread_create thread_number 1, ncpu -1
15:32:42.211850 31044:1        - a.out     | new_thread
15:32:42.212223 31044:0        1 a.out     | pthread_create thread_number 2, ncpu 2
15:32:42.212298 31044:2        2 a.out     | new_thread
15:32:42.212630 31044:0        1 a.out     | pthread_create thread_number 3, ncpu 4
15:32:42.212717 31044:3        4 a.out     | new_thread
15:32:42.213082 31044:0        1 a.out     | pthread_create thread_number 4, ncpu 5
15:32:42.213167 31044:4        5 a.out     | new_thread
15:32:54.709509 31044:0        1 a.out     | exit
```

Sample Output of placement.out

```
PSR COMMAND          TIME   PID  PPID   LWP
  1 a.out        00:00:02 31044 31039 31044
  0 a.out        00:00:00 31044 31039 31046
  2 a.out        00:00:02 31044 31039 31047
  4 a.out        00:00:01 31044 31039 31048
  5 a.out        00:00:01 31044 31039 31049
```

Note: Intel OpenMP jobs use an extra thread that is unknown to the user and it does not need to be placed. In the above example,

this extra thread (31046) is running on core number 0.

For MPI Codes Built with HPE's MPT Library

With HPE's MPT, only 1 shepherd process is created for the entire pool of MPI processes, and the proper way of pinning using dplace is to skip the shepherd process.

Here is an example for Endeavour:

```
#PBS -l ncpus=8
....
 mpiexec -np 8 dplace -s1 -c 0-7 ./a.out
```

If the number of processes in each node is less than the number of cores in that node, choose how you want to place the processes carefully. Below is just an example:

```
#PBS -l select=2:ncpus=8:mpiprocs=4 ...
mpiexec -np 8 dplace -s1 -c 2,4,1,5 ./a.out
```

To check the placement, you can set MPI_DSM_VERBOSE, which prints the placement in the PBS stderr file:

```
#PBS -l select=2:ncpus=8:mpiprocs=4
...
setenv MPI_DSM_VERBOSE
mpiexec -np 8 dplace -s1 -c 2,4,1,5 ./a.out
```

Output in PBS stderr File

```
MPI: DSM information
grank   lrank   pinning  node name      cpuid
   0       0   yes      r75i2n13          1
   1       1   yes      r75i2n13          2
   2       2   yes      r75i2n13          4
   3       3   yes      r75i2n13          5
   4       0   yes      r87i2n6           1
   5       1   yes      r87i2n6           2
   6       2   yes      r87i2n6           4
   7       3   yes      r87i2n6           5
```

If you use the -o *log_file* flag of dplace, the CPUs where the processes/threads are placed will be printed, but the node names are not printed.

```
#PBS -l select=2:ncpus=8:mpiprocs=4
....
mpiexec -np 8 dplace -s1 -c 2,4,1,5 -o log_file ./a.out
```

Output in log_file

```
timestamp       process:thread cpu taskname | placement action
15:16:35.848646 19807          - dplace     | exec ./new_pi, ncpu -1
15:16:35.877584 19807:0        - a.out      | load, cpu -1
15:16:35.878256 19807:0        - a.out      | fork -> pid 19810, ncpu 1
15:16:35.879496 19807:0        - a.out      | fork -> pid 19811, ncpu 2
15:16:35.880053 22665:0        - a.out      | fork -> pid 22672, ncpu 2
15:16:35.880628 19807:0        - a.out      | fork -> pid 19812, ncpu 4
15:16:35.881283 22665:0        - a.out      | fork -> pid 22673, ncpu 4
15:16:35.882536 22665:0        - a.out      | fork -> pid 22674, ncpu 5
15:16:35.881960 19807:0        - a.out      | fork -> pid 19813, ncpu 5
15:16:57.258113 19810:0        1 a.out      | exit
15:16:57.258116 19813:0        5 a.out      | exit
15:16:57.258215 19811:0        2 a.out      | exit
15:16:57.258272 19812:0        4 a.out      | exit
15:16:57.260458 22672:0        2 a.out      | exit
15:16:57.260601 22673:0        4 a.out      | exit
15:16:57.260680 22674:0        5 a.out      | exit
15:16:57.260675 22671:0        1 a.out      | exit
```

For MPI Codes Built with MVAPICH2 Library

With MVAPICH2, 1 shepherd process is created for each MPI process. You can use ps -L -u *your_userid* on the running node to see these processes. To properly pin MPI processes using dplace, you cannot skip the shepherd processes and must use the following:

```
mpiexec -np 4 dplace -c2,4,1,5 ./a.out
```

The mbind utility is a "one-stop" tool for binding processes and threads to CPUs. It can also be used to track memory usage. The utility, developed at NAS, works for for MPI, OpenMP, and hybrid applications, and is available after loading the module scicon/cli_tools on NAS x86_64 or aarch64 systems.

One of the benefits of mbind is that it relieves you from having to learn the complexity of each individual pinning approach for associated MPI or OpenMP libraries. It provides a uniform usage model that works for multiple MPI and OpenMP environments.

Currently supported MPI and OpenMP libraries are listed below.

MPI:

- HPE-MPT
- MVAPICH2
- INTEL-MPI
- OPEN-MPI (including NVIDIA HPC-X MPI)
- MPICH

Note: When using mbind with HPE-MPT, it is highly recommended that you use MPT 2.17r13, 2.21 or a later version in order to take full advantage of mbind capabilities.

OpenMP:

- Intel OpenMP runtime library
- GNU OpenMP library
- PGI runtime library
- Oracle Developer Studio thread library

Starting with version 1.7, the use of mbind is no longer limited to cases where the same set of CPU lists is used for all processor nodes. However, as in previous versions, the same number of threads must be used for all processes.

WARNING: The mbind tool might not work properly when used together with other performance tools.

Syntax

#For OpenMP:
mbind.x [*-options*] program [*args*]

#For MPI or MPI+OpenMP hybrid which supports mpiexec:
mpiexec -np nranks mbind.x [*-options*] program [*args*]

To find information about all available options, run the command mbind.x -help.

Here are a few recommended mbind options:

-cs, -cp, -cc;
or -c*cpulist*
: -cs for spread (default), -cp for compact, -cc for cyclic; -c*cpulist* for process ranks (for example, -c*0,3,6,9*). CPU numbers in the *cpulist* are relative within a cpuset, if present.
Note that the -cs option will distribute the processes and threads among the physical cores to minimize various resource contentions, and is usually the best choice for placement.

-t[*n*]
: Number of threads per process. The default value is given by the OMP_NUM_THREADS environment variable; this option overrides the value specified by OMP_NUM_THREADS.

-gm[*n*]
: Print memory usage information. This option is for printing memory usage of each process at the end of a run. Optional value [*n*] can be used to select one of the memory usage types: 0=default, 1=VmHWM, 2=VmRSS, 3=WRSS. Recognized symbolic values for [*n*]: "hwm", "rss", or "wrss". For default, environment variable GM_TYPE may be used to select the memory usage type:

  - VmHWM - high water mark
  - VmRSS - resident memory size
  - WRSS - weighted memory usage (if available; else, same as VmRSS)

-gmc[*s:n*]
: Print memory usage every [*s*] seconds for [*n*] times. The -gmc option indicates continuous printing of memory usage at a default interval of 5 seconds. Use additional option [*s:n*] to control the interval length [*s*] and the number of printing times [*n*]. Environment variable GM_TIMER may also be used to set the [*s:n*] value.

-gmr[*list*]
: Print memory usage for selected ranks in the list. This option controls the subset of ranks for memory usage to print. [*list*] is a comma-separated group of numbers with possible range.

-g[*n:m*]
: Work with -gm to monitor GPU usage (mem, power, energy). With [-gm], this option prints a few basic stats about GPU usage, including memory used, power in watts, and energy in joules. Default is to print one GPU per rank if running under MPI or all available GPUs otherwise. Use <n> to select a specific GPU id. Use <n>='a' to print all available GPUs; or <n>='n' to turn off

GPU usage printout. The additional option <:m> sets an internal monitoring frequency in seconds (def=5).

-l

Print node information. This option prints a quick summary of node information by calling the clist.x utility.

-v[*n*]

Verbose flag; Option -v or -v1 prints the process/thread-CPU binding information. With [*n*] greater than 1, the option prints additional debugging information. [*n*] controls the level of details. Default is n=0 (OFF).

Examples

Print a Processor Node Summary to Help Determine Proper Process or Thread Pinning

In your PBS script, add the following to print the summary:

```
#PBS -lselect=...:model=bro_ele

module load scicon/cli_tools
mbind.x -l

...
```

In the sample output below for a Broadwell node, look for the listing under column CPUs(SMTs). CPUs listed in the same row are located in the same socket and share the same last level cache, as shown in this configuration diagram.

```
Host Name           : r15i3n16
Processor Model     : Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz
Processor Speed     : 3300 MHz
Level 1 Cache (D)   : 32 KB
Level 1 Cache (I)   : 32 KB
Level 2 Cache (U)   : 256 KB
Level 3 Cache (U)   : 35840 KB (shared by 14 cores)
SMP Node Memory     : 125.4 GB (121.6 GB free, 2 numa nodes)
Memory Page Size    : 4096 Bytes

Number of Sockets   : 2
Number of L3 Caches : 2
Number of Cores     : 28    (14 cores/socket)
Number of CPUs      : 56    (2 SMTs/core)

Socket Cache Cores    CPUs(SMTs)
0 0 0-6,8-14 (0,28)(1,29)(2,30)(3,31)(4,32)(5,33)(6,34)(7,35)(8,36)(9,37)
   (10,38)(11,39)(12,40)(13,41)
1 0 0-6,8-14 (14,42)(15,43)(16,44)(17,45)(18,46)(19,47)(20,48)(21,49)
   (22,50)(23,51)(24,52)(25,53)(26,54)(27,55)
```

For Pure OpenMP Codes Using Intel OpenMP Library

Sample PBS script:

```
#PBS -l select=1:ncpus=28:model=bro_ele
#PBS -l walltime=0:5:0

module load scicon/cli_tools
module load comp-intel

setenv OMP_NUM_THREADS 4
cd $PBS_O_WORKDIR

mbind.x -cs -t4 -v ./a.out
#or simply:
#mbind.x -v ./a.out
```

The four OpenMP threads are spread (with the -cs option) among four physical cores in a node (two on each socket), as shown in the application's stdout:

```
host: r15i3n16, ncpus: 56, nthreads: 4, bound to cpus: {0,1,14,15}
OMP: Warning #181: GOMP_CPU_AFFINITY: ignored because KMP_AFFINITY has been defined
```

The proper placement is further demonstrated in the output of the ps command below:

```
r15i3n164% ps -C a.out -L -opsr,comm,time,pid,ppid,lwp
PSR COMMAND             TIME   PID  PPID   LWP
  0 a.out           00:02:06 37243 36771 37243
  1 a.out           00:02:34 37243 36771 37244
 14 a.out           00:01:47 37243 36771 37245
 15 a.out           00:01:23 37243 36771 37246
```

Note: If you use older versions of Intel OpenMP via older versions of Intel compiler modules (comp-intel/2016.181 or earlier) during

runtime, the ps output will show an extra thread that does not do any work, and therefore does not accumulate any time. Since this extra thread will not interfere with the other threads, it does not need to be placed.

For Pure MPI Codes Using HPE MPT

WARNING: mbind.x disables MPI_DSM_DISTRIBUTE and overwrites the placement initially performed by MPT's mpiexec. The placement output from MPI_DSM_VERBOSE (if set) most likely is incorrect and should be ignored.
TIP: When using HPE MPT to launch multiple processes of a non-MPI application (for example, Python), set the MPI_SHEPHERD environment variable to true to inform MPT that the executable is not an MPI application.

Sample PBS script where the same number of MPI ranks are used in different nodes:

```
#PBS -l select=2:ncpus=28:mpiprocs=4:model=bro_ele

module load scicon/cli_tools
module load comp-intel
module load mpi-hpe

#setenv MPI_DSM_VERBOSE

cd $PBS_O_WORKDIR

mpiexec -np 8 mbind.x -cs -v ./a.out
#or simply:
#mpiexec mbind.x -v ./a.out
```

On each of the two nodes, four MPI processes are spread among four physical cores (CPUs 0,1,14,15); two on each socket, as shown in the application's stdout:

```
host: r11i5n6, ncpus: 56, process-rank: 0 (r0), bound to cpu: 0
host: r11i516, ncpus: 56, process-rank: 1 (r1), bound to cpu: 1
host: r11i5n6, ncpus: 56, process-rank: 2 (r2), bound to cpu: 14
host: r11i5n6, ncpus: 56, process-rank: 3 (r3), bound to cpu: 15
host: r15i3n16, ncpus: 56, process-rank: 4 (r0), bound to cpu: 0
host: r15i3n16, ncpus: 56, process-rank: 5 (r1), bound to cpu: 1
host: r15i3n16, ncpus: 56, process-rank: 6 (r2), bound to cpu: 14
host: r15i3n16, ncpus: 56, process-rank: 7 (r3), bound to cpu: 15
```

Note: For readability in this article, the printout of the binding information from mbind.x is sorted by the process-rank. An actual printout will not be sorted.

Sample PBS script where different numbers of MPI ranks are used on different nodes:

```
#PBS -l select=1:ncpus=28:mpiprocs=1:model=bro_ele+2:ncpus=28:mpiprocs=4:model=bro_ele

module load scicon/cli_tools
module load comp-intel
module load mpi-hpe

#setenv MPI_DSM_VERBOSE

cd $PBS_O_WORKDIR

mpiexec -np 9 mbind.x -cs -v ./a.out
#Or simply:
#mpiexec mbind.x -v ./a.out
```

As shown in the application's stdout, only one MPI process is used on the first node and it is pinned to CPU 0 on that node. For each of the other two nodes, four MPI processes are spread among four physical cores (CPUs 0,1,14,15):

```
host: r15i3n16, ncpus: 56, process-rank: 0 (r0), bound to cpu: 0
host: r15i5n4, ncpus: 56, process-rank: 1 (r0), bound to cpu: 0
host: r15i5n4, ncpus: 56, process-rank: 2 (r1), bound to cpu: 1
host: r15i5n4, ncpus: 56, process-rank: 3 (r2), bound to cpu: 14
host: r15i5n4, ncpus: 56, process-rank: 4 (r3), bound to cpu: 15
host: r15i5n8, ncpus: 56, process-rank: 5 (r0), bound to cpu: 0
host: r15i5n8, ncpus: 56, process-rank: 6 (r1), bound to cpu: 1
host: r15i5n8 ncpus: 56, process-rank: 7 (r2), bound to cpu: 14
host: r15i5n8, ncpus: 56, process-rank: 8 (r3), bound to cpu: 15
```

For MPI+OpenMP Hybrid Codes Using HPE MPT and Intel OpenMP

Sample PBS script:

```
#PBS -l select=2:ncpus=28:mpiprocs=4:model=bro_ele

module load scicon/cli_tools
module load comp-intel
module load mpi-hpe
```

```
setenv OMP_NUM_THREADS 2
#setenv MPI_DSM_VERBOSE

cd $PBS_O_WORKDIR

mpiexec -np 8 mbind.x -cs -t2 -v ./a.out
#or simply:
#mpiexec mbind.x -v ./a.out
```

On each of the two nodes, the four MPI processes are spread among the physical cores. The two OpenMP threads of each MPI process run on adjacent physical cores, as shown in the application's stdout:

```
host: r11i5n2, ncpus: 56, process-rank: 0 (r0), nthreads: 2, bound to cpus: {0,1}
host: r11i5n2, ncpus: 56, process-rank: 1 (r1), nthreads: 2, bound to cpus: {2,3}
host: r11i5n2, ncpus: 56, process-rank: 2 (r2), nthreads: 2, bound to cpus: {14,15}
host: r11i5n2, ncpus: 56, process-rank: 3 (r3), nthreads: 2, bound to cpus: {16,17}
host: r11i8n5, ncpus: 56, process-rank: 4 (r0), nthreads: 2, bound to cpus: {0,1}
host: r11i8n5, ncpus: 56, process-rank: 5 (r1), nthreads: 2, bound to cpus: {2,3}
host: r11i8n5, ncpus: 56, process-rank: 6 (r2), nthreads: 2, bound to cpus: {14,15}
host: r11i8n5, ncpus: 56, process-rank: 7 (r3), nthreads: 2, bound to cpus: {16,17}
```

You can confirm this by running the following ps command line on the running nodes. Note that the HPE MPT library creates a shepherd process (shown running on PSR=18 in the output below), which does not do any work.

```
r11i5n2% ps -C a.out -L -opsr,comm,time,pid,ppid,lwp
PSR COMMAND    TIME   PID  PPID   LWP
 18 a.out   00:00:00 41087 41079 41087
  0 a.out   00:00:12 41092 41087 41092
  1 a.out   00:00:12 41092 41087 41099
  2 a.out   00:00:12 41093 41087 41093
  3 a.out   00:00:12 41093 41087 41098
 14 a.out   00:00:12 41094 41087 41094
 15 a.out   00:00:12 41094 41087 41097
 16 a.out   00:00:12 41095 41087 41095
 17 a.out   00:00:12 41095 41087 41096
```

For Pure MPI or MPI+OpenMP Hybrid Codes Using other MPI Libraries and Intel OpenMP

Usage of mbind with other MPI libraries should be the same as with HPE MPT. The main difference is that you must load the proper mpi modulefile.

Note that the Intel MPI library automatically pins processes to CPUs to prevent unwanted process migration. If you find that the placement done by the Intel MPI library is not optimal, you can use mbind to do the pinning instead. If you use version 4.0.2.003 or earlier, you might need to set the environment variable I_MPI_PIN to 0 in order for mbind.x to work properly.

The mbind utility was created by NAS staff member Henry Jin.