All NAS facility supercomputers use the Portable Batch System (PBS) from Altair for batch job submission, job monitoring, and job management. Note that different systems may use different versions of PBS, so the available features may vary slightly from system to system.

Batch Jobs

Batch jobs run on compute nodes, not the front-end nodes. A PBS scheduler allocates blocks of compute nodes to jobs to provide exclusive access. You will submit batch jobs to run on one or more compute nodes using the `qsub` command from an interactive session on an appropriate front-end node, such as PFEs for Electra (bro_ele, sky_ele), Aitken (cas_ait, rom_ait), AFEs for Aitken (mil_ait), CFEs for Cabeus (mil_a100) and CGHFEs for Cabeus (gh200).

Normal batch jobs are typically run by submitting a script. A "jobid" is assigned after submission. When the resources you request become available, your job will execute on the compute nodes. When the job is complete, the PBS standard output and standard error of the job will be returned in files available to you.

When porting job submission scripts from systems outside of the NAS environment or between the supercomputers, be careful to make changes to your existing scripts to make them work properly on these systems.

Interactive Batch Mode

PBS also supports an interactive batch mode, using the command `qsub -I`, where the input and output are connected to the user's terminal, but the scheduling of the job is still under control of the batch system.

Queues

The available queues on different systems vary, but all typically have constraints on maximum wall time and/or the number of CPUs allowed for a job. Some queues may also have other constraints or be restricted to serving certain users or groups. In addition, to ensure that each NASA mission directorate is granted their allocated share of resources at any given time, mission directorate limits (called "shares") are also set on NAS compute resources.

See **man pbs** for more information.

PBS was originally created by a team in the NASA Advanced Supercomputing (NAS) Division.

This article provides a simplified explanation of the PBS scheduling policy on Aitken, Electra, and Endeavour. According to the current policy, jobs are sorted in the following order:

1. Current processor use (by mission directorate)
2. Job priority
3. Queue priority
4. Job size (wide jobs first)

However, job sorting is adjusted frequently in response to varying demands and workloads. In each scheduling cycle, PBS examines the jobs in sorted order, starting a job if it can. If the job cannot be started immediately, it is either rescheduled or simply bypassed for this cycle.

There are numerous reasons that the scheduler may not start a job, including:

- 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

For more information about each item in this list, see Common Reasons Why Jobs Won't Start.

Note that a high-priority job might be blocked by one of the limits in this list, while a lower priority job from a different user or requesting fewer resources might not be blocked.

If your job is waiting in the queue, you can run the qstat command as follows to obtain information that can indicate why it has not started running.

```
pfe21% qstat -s job_id
or
pfe21% qstat -f job_id | grep -i comment
```

For jobs submitted to run on Electra or Aitken, output from the qs command shows the amount of resources used and borrowed by each mission directorate, and the resources each mission is waiting for, by processor type:

```
pfe21% /u/scicon/tools/bin/qs
```

The following command provides the order of jobs that PBS schedules to start at the current scheduling cycle. It also provides information regarding processor type(s), mission, and job priority:

```
pfe21% qstat -W o=+model,mission,pri -i
```

Note: To prevent jobs from languishing in the queues for an indefinite time, PBS reserves resources for the top *N* jobs (between 6 and 12), and doesn't allow lower-priority jobs to start if they would delay the start time of a higher-priority job. This is known as backfilling.

PBS Sorting Order

Mission Shares

Each NASA mission directorate is allocated a certain percentage of the processors on Aitken and Electra. (See Mission Shares Policy.) A job cannot start if that action would cause the mission to exceed its share, unless another mission is using less than its share and has no jobs waiting. In this case, the high-use mission can "borrow" processors from the lower-use mission for up to a specified time (currently, max_borrow is 4 hours).

So , if the job itself needs less than max_borrow hours to run, or if a sufficient number of other jobs from the high-use mission will finish within max_borrow hours to get back under its mission share, then the job can borrow processors.

When jobs are sorted, jobs from missions using less of their share are picked before jobs from missions using more of their share.

Job Priority

Job priority has three components. First is the native priority (the -p parameter to qsub or qalter). Added to that is the queue priority. If the native priority is 0, then a further adjustment is made based on how long the job has been waiting for resources. Waiting jobs get a "boost" of up to 20 priority points, depending on how long they have been waiting and which queue they are in.

Queue priority

Some queues are given higher or lower priorities than the default (run qstat -Q to get current values). Note that because the mission share is the most significant sort criterion, job and queue priorities have little effect mission-to-mission.

Job Size

Jobs asking for more nodes are favored over jobs asking for fewer. The reasoning is that, while it is easier for narrow jobs to fill in gaps in the schedule, wide jobs need help collecting enough CPUs to start.

Backfilling

The policy described above could result in a large, high-priority job being blocked indefinitely by a steady stream of smaller, low-priority jobs. To prevent jobs from languishing in the queues for an indefinite time, PBS reserves resources for the top *N* jobs (*N* is between 6 and 12), and doesn't allow lower priority jobs start if they would delay the start time of one of the top job ("backfilling"). Additional details are given below.

As mentioned above, when PBS cannot start a job immediately, if it is one of the first *N* such jobs, PBS sets aside resources for the job before examining other jobs. That is, PBS looks at the currently running jobs to see when they will finish (using the wall-time estimates). From those finish times, PBS decides when enough resources (such as CPUs, memory, mission share, and job limits) will become available to run the top job.

PBS then creates a virtual reservation for those resources at that time. Now, when PBS looks at other jobs to see if they can start immediately, it also checks whether starting the job would collide with one of these reservations. Only if there are no collisions will PBS start the lower priority jobs.

The four most commonly used PBS commands, qsub, qstat, qdel, and qhold, are briefly described below. See **man pbs** for a list of all PBS commands.

qsub

To submit a batch job to the specified queue using a script:

%qsub -q *queue_name job_script*

Queue names on Aitken and Electra include normal, debug, long, devel, and low. When *queue_name* is omitted, the job is routed to the default queue, which is the normal queue.

To submit an interactive PBS job:

%qsub -I -q *queue_name* -l *resource_list*

No *job_script* should be included when submitting an interactive PBS job.

The *resource_list* typically specifies the number of nodes, CPUs, amount of memory and wall time needed for this job. The following example shows a request for a single Aitken Rome node and a wall time of 2 hours.

%qsub -I -lselect=1:ncpus=128:model=rom_ait,walltime=2:00:00

See **man pbs_resources** for more information on what resources can be specified.

Note: If -l *resource_list* is omitted, the default resources for the specified queue is used. When *queue_name* is omitted, the job is routed to the default queue, which is the *normal* queue.

qstat

To display queue information:

%qstat -Q *queue_name*
%qstat -q *queue_name*
%qstat -fQ *queue_name*

Each option uses a different format to display all of the queues available, and their constraints and status. The *queue_name* is optional.

To display job status:

-a
: Display all jobs in any status (running, queued, held)

-r
: Display all running or suspended jobs

-n
: Display the execution hosts of the running jobs

-i
: Display all queued, held or waiting jobs

-u *username*
: Display jobs that belong to the specified user

-s
: Display any comment added by the administrator or scheduler. This option is typically used to find clues of why a job has not started running.

-f *job_id*
: Display detailed information about a specific job

-xf *job_id*,
-xu *user_id*
: Display status information for finished jobs (within the past 7 days).

TIP: Some of these flags can be combined when you are checking the job status.

Note: The versions of qstat on newer NAS systems (including Athena and Cabeus GH200 nodes) and older NAS systems (including Electra, Aitken, and Endeavour, and Cabeus A100 nodes) are different. To get the same look and feel of qstat on the newer systems as the older systems, it is recommended to use a NAS provided script called nas_qstat, available under /usr/local/bin on athfe[01-04] and cghfe[01-02]. In addition, the script showq under /u/hsp/bin can be used to visually see all jobs that are running or pending and their status, along with node availability at the bottom of the output.

qdel

To delete (cancel) a job:

%qdel *job_id*

qhold

To hold a job:

%qhold *job_id*

Only the job owner or a system administrator with "su" or "root" privilege can place a hold on a job. The hold can be released using the qrls command.

For more detailed information on each command, see their corresponding **man** pages.

Commonly Used qsub Command Options

The qsub options can be read from the PBS directives of a PBS job script or from the qsub command line. For a complete list of available options, see **man qsub**. Commonly used options include:

-S *shell_name*
: Specifies the shell that interprets the job script.

-V
: Declares that all environment variables in the qsub command's environment are to be exported to the batch job.

-v *variable_list*
: Lists environment variables to be exported to the job. For example, to pass the variable Year and its value 2025 to the job script, use qsub -v Year="2025" job_script. Within the job_script, reference the variable with ${Year}.

-q *queue_name*
: Defines the destination of the job. For a list of available queues, run the command qstat -Q on an appropriate front-end node. For example, for jobs submitted from the PFEs to run on Electra or Aitken, queue names include normal, debug, devel, long, and low. Endeavour queue names include e-normal, e-long, and e-debug. See PBS Job Queue Structure.

-l *resource_list*
: Specifies the resources that are required by the job and establishes a limit to the amount of resources that can be consumed. Commonly specified resources include select, ncpus, walltime, and mem (memory). You can also specify filesystems. See **man pbs_resources** for a complete list.

-e *path*
: Directs the standard error output produced by the request to the stated file path.

-o *path*
: Directs the standard output produced by the request to the stated file path.

-j *join*
: Declares that the standard output and error streams of the job should be merged (joined). Possible values for *join* are:
oe- Standard output and error streams are merged in the standard output file.
eo- Standard error and output streams are merged in the standard error file.

-k *keep*
: Declares that the standard output and error streams of the job should be retained on the (first) execution host. This is not typically useful on NAS systems; however, when the d (direct) option is included, it changes the way the standard output and error files from a job are handled. Usually, these files are collected until the job ends and only then copied back to their destination files. With -kod, the job writes directly to its output file as it is running. Similarly, with -ked, the error file is written directly. You can then view those files from a front-end host to monitor how the job is running.

  There are a few caveats to this feature. First, the output or error files must be available to the nodes where the job runs (the files should be on /homeX or /nobackup filesystems, not on a local filesystem such as /tmp). Second, if the files already exist, they will be appended to, rather than overwritten, as would be the case without -k. If multiple jobs specify the same output file, the outputs from the jobs can be intermingled in the final file. Third, there can be a delay of a few seconds between the time the job writes to the files and the time the change is visible on the front-end host.

-m *mail_options*
: Defines the set of conditions under which the execution server will send mail message about the job. See **man qsub** for a list of mail options.

-N *name*
: Declares a name for the job.

-W *addl_attributes*
: Allows for the specification of additional job attributes. The most common additional attributes are:
-W group_list=*g_list* - Specifies the group the job runs under.
-W depend=afterany:*job_id.server_name.nas.nasa.gov* - Submits a job that is to be executed after *job_id* has finished with any exit status (for example, 12345.pbspl1.nas.nasa.gov).
-W depend=afterok:*job_id.server_name.nas.nasa.gov* - Submits a job that is to be executed after *job_id* has finished with no errors.

-r *y|n*
: Declares whether the job is re-runnable. See Checkpointing and Restart.

PBS on Aitken

To help you prepare for running jobs on Aitken's Milan nodes, this short user guide includes information on the general configuration of Milan nodes, compiling your code, and running PBS jobs.

Aitken Milan Front-End Systems

Two new Aitken front ends, afe01 and afe02, serve the Aitken Milan nodes. Each AFE includes one or two AMD EPYC 7763 processors with a total of 128 physical cores (sum of two sockets), and 512 gigabytes (GB) of memory.

Note: The Aitken Milan compute nodes are not accessible from the PFEs.

You can access the AFEs using one of the following methods.

Initiating SSH from Your Local Desktop (Recommended)

- Two-step login:

  ```
  local_desktop% ssh sfe6.nas.nasa.gov  (or replace sfe6 with sfe7 or sfe8)
  sfe6% ssh afe01    (or ssh afe02)
  ```

- One-step login (using SSH passthrough):

  Before you begin: Follow the instructions in Setting Up SSH Passthrough, and make sure that the configuration file in the .ssh directory of your local desktop contains the following lines:

  ```
  Host afe01
     HostKeyAlias          afe01.nas.nasa.gov
     ProxyCommand          ssh -ax -oCompression=no sfe ssh-balance %h

  Host afe02
     HostKeyAlias          afe02.nas.nasa.gov
     ProxyCommand          ssh -ax -oCompression=no sfe ssh-balance %h
  ```

  Once you have SSH Passthrough set up, you can log into the AFE in one step:

  ```
  local_desktop% ssh afe01   (or ssh afe02)
  ```

Initiating SSH from a PFE

Note: This method is not recommended but is made available for convenience.

```
pfe% ssh afe01 (or ssh afe02)
```

Aitken Milan Compute Nodes

Aitken includes 512 Milan compute nodes, which are partitioned into four physical racks. Each node contains two 64-core AMD EPYC 7763 Milan sockets (2.45 GHz) and 512 GB of memory.

These nodes are inter-connected with four-lane High Data Rate (4X HDR) devices and four-lane High Data Rate (4X HDR) switches. They are currently housed in the NAS main building and have access to the NAS compute resources home and Lustre filesystems.

Note: For the long term, the Milan nodes will be relocated to be near the existing Aitken resources when a new compute module becomes available. At that time, these nodes will be connected to the Rome nodes with additional HDR cables on the 8th dimension of the Aitken cluster hybercube.

Operating System and Software Modules

The operating system on all Milan compute nodes is the Red Hat Enterprise Linux-based Tri-Lab Operating System Stack Version 4 (TOSS 4). Batch jobs use :aoe=toss4 by default, so you do not need to add :aoe=toss4 in your PBS resource request. The default $MODULEPATH includes:

- /usr/share/Modules/modulefiles
- /nasa/modulefiles/toss4
- /nasa/modulefiles/spack/
- /nasa/modulefiles/pkgsrc/toss4/

Compilers For Milan Nodes

There are several compilers you can use to build your application:

Intel Compilers

Some Intel compiler modules are available in the /nasa/modulefiles/toss4 directory, however, several are in the /nasa/modulefiles/testing directory. To see what versions are available in these two directories, run:

```
module use -a /nasa/modulefiles/testing
module avail comp-intel
```

To build your application using the Intel compilers, use `-march=core-avx2`.

Existing executables that are built for the Aitken Rome nodes should run on the Milan nodes.

Note: Although AMD recommends using -axCORE-AVX2, we do not recommend using it, as the resulting executable will run but will not actually take advantage of AVX2 instructions. Executables built with -xCORE-AVX2, -xSSE4.2, and so on will not run on Milan processors.

GNU Compilers

The default version of GNU Compiler Collection (GCC) under TOSS 4 (/usr/bin/gcc) is GCC 8.5.0. However, support for the Rome architecture is only available with GCC 11.xx and later. GCC version 13.2 has been installed and is accessible via:

```
module use -a /nasa/modulefiles/testing
module load gcc/13.2
```

The recommended compiler flag for Milan with gcc is: `-march=znver3`

AMD Optimizing C/C++ Compilers (AOCC)

To see available versions before loading an AOCC compiler, use the `module avail comp-aocc` command. For example:

```
module use -a /nasa/modulefiles/testing
module avail comp-aocc
---------- /nasa/modulefiles/testing ----------
comp-aocc/4.0.0  comp-aocc/4.1.0  cmp-aocc/4.2.0

---------- /nasa/modulefiles/toss4 ------------
comp-aocc/3.2.0
```

The AOCC C, C++ and Fortran compiler drivers are clang, clang++ and flang, respectively.

The recommended compiler flag for Milan with AOCC is: `-march=znver3`

Note: The default optimization level is `-O1` in AOCC 3.2, and `-O2` in AOCC 4.1 and newer versions.

For more compiler options, see AMD's article Compiler Options Quick Reference Guide for AMD EPYC 7xx3 Series Processor.

MPI Libraries For Milan Nodes

HPE MPT

Use the modulefile `mpi-hpe/mpt`, which is available in the /nasa/modulefiles/toss4 directory:

```
module load mpi-hpe/mpt
```

OpenMPI

There are a few OpenMPI modulefiles available in the /nasa/modulefiles/testing directory.

```
module use -a /nasa/modulefiles/testing
module avail openmpi
---- /nasa/modulefiles/testing ----
openmpi/4.1.5-toss4-intel_2022.1  openmpi/4.1.6-toss4-gnu  openmpi/4.1.6-toss4-aocc
```

Note: There are several newer versions that are not listed in the output above.

The compiler used to build each of these modulefiles is indicated in their filenames. For example, `openmpi/4.1.6-toss4-gnu` was built with gcc 8.5.0, and `openmpi/4.1.6-toss4-aocc` was built with aocc 4.2.0.

After loading one of these modulefiles, run the command `ompi_info` to learn more about each installation.

Running OpenMP or MPI/OpenMPI Hybrid Codes

Each Milan node has 128 physical cores. When simultaneous multi-threading (SMT) is enabled in BIOS, there are 256 logical cores for use by OpenMP or MPI/OpenMPI hybrid codes.

To check whether SMT is enabled, run the following command on a Milan compute node:

```
cat /proc/cpuinfo | grep processor | wc -l
```

If the output shows 256, then SMT is enabled.

Note: During Milan's early production, some nodes were configured with the SMT on and some with the SMT off. Currently, all nodes have been set with SMT on.

We recommend using the mbind.x tool for process and thread binding, as shown in the following example with four MPI processes and 32 OpenMP threads per process:

```
mpiexec -n 4 /u/scicon/tools/bin/mbind.x -t 32 a.out
```

If you use HPE MPT, you can also use the omplace tool for pinning MPI/OpenMP hybrid code.

Running PBS Jobs on Aitken Milan Nodes

Aitken's Milan nodes are accessible only through PBS jobs submitted from afe01 and afe02 and served by the PBS server pbspl1.

To request Aitken Milan nodes, use :model=mil_ait in your PBS script, as shown in the example below.

Sample PBS Script For Aitken Milan Nodes

```
#PBS -l select=2:ncpus=128:mpiprocs=128:model=mil_ait
#PBS -l walltime=8:00:00
#PBS -q normal
module use -a /nasa/modulefiles/testing
module load mpi-hpe/mpt
module load comp-intel
cd $PBS_O_WORKDIR
mpiexec -np 256 ./a.out
```

For more information, see AMD Milan Processors.

To help you prepare for running jobs on Aitken's Rome nodes, this short user guide includes information on the general configuration of Rome nodes, compiling your code, and running PBS jobs.

Overview of Aitken Rome Nodes

Aitken includes 2,008 Rome nodes, which are partitioned into sixteen physical racks. Each node contains two 64-core AMD EPYC 7742 Rome sockets (2.25 GHz) and 512 GB of memory. The nodes are connected to the Aitken InfiniBand network (ib0 and ib1) via four-lane High Data Rate (4X HDR) devices and four-lane High Data Rate (4X HDR) switches for internode communication. The ib1 fabric is used mainly for I/O and is connected to the Lustre filesystems. In addition, Aitken Rome nodes share the same home filesystems, PFEs, and PBS server as Electra, and the Aitken Cascade Lake and Milan nodes. You can access Aitken Rome nodes only through PBS jobs submitted from the PFEs.

Note: Aitken Milan nodes use a different set of front ends. See Preparing to Run on Aitken Milan Nodes for more information.

Operating System and Software Modules

The operating system on all Rome compute nodes is the Red Hat Enterprise Linux-based Tri-Lab Operating System Stack Version 4 (TOSS 4). Batch jobs use `:aoe=toss4` by default, so you do not need to add `:aoe=toss4` in your PBS resource request. The default $MODULEPATH includes:

- /usr/share/Modules/modulefiles
- /nasa/modulefiles/toss4
- /nasa/modulefiles/spack/
- /nasa/modulefiles/pkgsrc/toss4/

Compilers for Rome Nodes

Note: Some executables built under the TOSS 3 environment will work under TOSS 4; however, some may not. If you need to rebuild your executable under TOSS 4, and you experience issues such as missing certain software packages or libraries, send an email to support@nas.nasa.gov

There are several compilers you can use to build your application:

- Intel Compilers:

  Some Intel compiler modules are available in the `/nasa/modulefiles/toss4` directory, however, several are in the `/nasa/modulefiles/testing` directory. To see what versions are available in these two directories, run:

  ```
  module use -a /nasa/modulefiles/testing
  module avail comp-intel
  ```

  Existing executables that are built with support for AVX512 (for running on Skylake and Cascade Lake processors) will not run on the Rome nodes. Others that are built with support for AVX2 may run on the Rome nodes; however, you should always check the result of your runs using existing executables. To rebuild your application using Intel compilers, use `-march=core-avx2`.

  Note: Although AMD recommends using -axCORE-AVX2, we do *not* recommend using it, as the resulting executable will run but will not actually take advantage of AVX2 instructions. Executables built with -xCORE-AVX2, -xSSE4.2, etc., will not run on Rome processors.
- GNU Compilers:

  The default version of GNU Compiler Collection (GCC) under TOSS 4 (/usr/bin/gcc) is GCC 8.5.0. However, support for the Rome architecture is only available with GCC 9.x and later. Among the several GCC versions available in the /nasa/pkgsrc/toss4 directory, load the 9.3 or 10.3 version for use with the Rome nodes. For example:

  ```
  module avail gcc
  -- /nasa/modulefiles/pkgsrc/toss4/ --
  gcc/6.5  gcc/7.5  gcc/8.4  gcc/9.3  gcc/10.3  gcc/13.2
  module load gcc/10.3
  ```

  The recommended compiler flag to use with GCC is `-march=znver2`.
- AMD Optimizing C/C++ and Fortran Compilers (AOCC)

  To see available versions before loading an AOCC compiler, use the `module avail comp-aocc` command. For example:

  ```
  module avail comp-aocc
  -- /nasa/modulefiles/toss4 --
  comp-aocc/3.2.0

  -- /nasa/modulefiles/testing --
  comp-aocc/4.0.0  comp-aocc/4.1.0  comp-aocc/4.2.0  comp-aocc/5.0.0

  module load comp-aocc/3.2.0
  ```

The recommended compiler flag to use with AOCC is -march=znver2.

- PGI Compilers:

  Some PGI compiler modules are available in the /nasa/modulefiles/testing directory. There is no support specifically for Rome processors. If you want to compile your code using a PGI compiler, the recommended compiler flag is -tp=zen

For more compiler options, see Compiler Options Quick Reference Guide for AMD EPYC 7xx2 Series Processors.

MPI Library for Rome Nodes

HPE MPT is the only recommended MPI library for use on the AMD Rome nodes under TOSS 4. Use thempi-hpe/mpt.2.30 modulefile in the /nasa/modulefiles/toss4 directory. Earlier versions such as mpi-hpe/mpt.2.25 do not work properly for the Rome nodes.

```
module load mpi-hpe/mpt.2.30
```

Note: Although there is more than one version of MPT library installed on NAS systems, mpi-hpe/mpt.2.30 is currently the only reliable version for running on the Rome nodes. You can access this version by loading mpi-hpe/mpt, which points to the recommended version (currently mpi-hpe/mpt.2.30) and environmental settings:

```
% module load mpi-hpe/mpt
```

Running OpenMP or MPI/OpenMPI Hybrid Codes

Each Rome node has 128 physical cores. When Simultaneous Multi-Threading (SMT) is enabled in BIOS, there are 256 logical cores for use by OpenMP or MPI/OpenMPI hybrid codes.

To check whether SMT is enabled, run the following command on a Rome compute node:

```
cat /proc/cpuinfo | grep processor | wc -l
```

If it shows 256, then SMT is enabled.

We recommend using the mbind.x tool for process and thread binding, as shown in the following example with four MPI processes and 32 OpenMP threads per process:

```
mpiexec -n 4 /u/scicon/tools/bin/mbind.x -t 32 a.out
```

If you use HPE MPT, you can also use the omplace tool for pinning MPI/OpenMP hybrid code.

Running PBS Jobs on Aitken Rome Nodes

To request Aitken Rome nodes, use :model=rom_ait in your PBS script, as shown in the example below.

The devel, normal, long, debug, and low queues are available for use with Rome nodes.

Note: The Standard Billing Unit (SBU) rate for Rome processors is 4.06.

Sample PBS Script For Aitken Rome Nodes

```
#PBS -l select=2:ncpus=128:mpiprocs=128:model=rom_ait
#PBS -l walltime=8:00:00
module load mpi-hpe/mpt
module load comp-intel/2020.4.304

cd $PBS_O_WORKDIR

mpiexec -np 256 ./a.out
```

For more information, see AMD Rome Processors.

To help you prepare for running jobs on Aitken's Cascade Lake compute nodes, this short user guide includes information on the general configuration of Cascade Lake nodes, compiling your code, running PBS jobs, and checking allocation usage.

Overview of Aitken Cascade Lake Nodes

Aitken includes 1,152 Cascade Lake nodes, which are partitioned into eight physical racks. Each node contains two 20-core Xeon Gold 6248 sockets (2.5 GHz) and 192 GB of memory. The nodes are connected to the Aitken InfiniBand network (ib0 and ib1) via four-lane Enhanced Data Rate (4X EDR) devices and four-lane High Data Rate (4X HDR) switches for internode communication. The ib1 fabric is used mainly for I/O and is connected to the NAS Lustre filesystems. In addition, Aitken Cascade Lake nodes use the NAS compute resources home filesystems, front-end systems (PFEs), and PBS server (pbspl1) as Electra and the Aitken Rome nodes. You can access Aitken Cascade Lake nodes only through PBS jobs submitted from the PFEs.

Note: Aitken Milan nodes use a different set of front ends. See Preparing to Run on Aitken Milan Nodes for more information.

Compiling Your Code For Cascade Lake Nodes

The Cascade Lake processors include the Advanced Vector Extensions 512 (AVX-512). Intel AVX-512 optimizations are included in Intel compiler version 16.0 and later versions. We recommend that you test the latest Intel compiler, through the command module load comp-intel/, which may include better optimizations for AVX-512.

For Cascade Lake-specific optimizations, use the compiler option -xCORE-AVX512.

If you want a single executable that will run on any of the Aitken and Electra processor types, with suitable optimization to be determined at runtime, you can compile your application using the option:
-O3 -axCORE-AVX512,CORE-AVX2 -xAVX.

Note: The use of either of these options, -xCORE-AVX512 or -axCORE-AVX512, could either improve or degrade performance of your code. Be sure to check performance with and without these flags before using them for production runs.

Running PBS Jobs on Aitken Cascade Lake Nodes

To request Aitken Cascade Lake nodes, use :model=cas_ait in your PBS script, as shown in the example below.

Note: Because MPT 2.15 and earlier versions do not support the ConnectX-5 host channel adapters (HCAs), the environment variables MPI_IB_XRC and MPI_XPMEM_ENABLED have been disabled for jobs running on Cascade Lake. If your MPI applications perform significant MPI collective operations and rely on having these two variables enabled to get good performance, use MPT 2.17 or newer versions. You can use the NAS-recommended MPT version with the command:
module load mpi-hpe/mpt

Sample PBS Script For Aitken Cascade Lake Nodes

```
#PBS -l select=10:ncpus=40:mpiprocs=40:model=cas_ait
#PBS -l walltime=8:00:00
#PBS -q normal

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

cd $PBS_O_WORKDIR

mpiexec -np 400 ./a.out
```

For more information, see Cascade Lake Processors.

When you request PBS resources for your job, you must specify one or more Aitken or Electra node model (processor) types, so it is helpful to keep this information in mind:

- Usage charges on each HECC node type are based on a common Standard Billing Unit (SBU).
- The total physical memory of each node type varies from 128 GB to 512 GB.

The following table provides the SBU rate and memory for each Aitken and Electra node type:

Processor Type	SBU Rate/Node	Memory/Node
Milan	4.38 (128 cores/node)	512 GB
Rome	4.06 (128 cores/node)	512 GB
Cascade Lake	1.64 (40 cores/node)	192 GB
Skylake	1.59 (40 cores/node)	192 GB
Broadwell	1.00 (28 cores/node)	128 GB

Note: The amount of memory available to a PBS job is less than the node's total physical memory, because the system kernel can use up to 4 GB of memory in each node.

Checking Available Nodes Before Requesting Resources

If your application can run on any of the Aitken or Electra processor types, you might be able to reduce wait time by requesting a type that has more free nodes (nodes currently unoccupied by other running jobs).

Before you submit your job, you can find out the current numbers of used and free nodes for each processor type by running the node_stats.sh script, which is accessible by loading the scicon/cli_tools modulefile. For example:

```
% module load scicon/cli_tools
% node_stats.sh
```

You can also identify which processor models are in use for each job currently running on HECC systems. Run the following command, and check the Model field in the output:

```
% qstat -a -W o=+model
```

Resource Request Examples

As shown in these examples, use the model=[bro_ele,sky_ele,cas_ait,rom_ait, mil_ait] attribute to request the node model type(s) for your job.

The default operating system for all model types is TOSS (Tri-Lab Operating System Stack). For all model types, the default is aoe=toss4:aoe=toss4 to your PBS resource request.)

Example 1: Basic Resource Request

Each of the following sample command lines requests a single node model type for a 128-process job:

```
#PBS -l select=5:ncpus=28:model=bro_ele
#to run all 28 cores on each of the 5 Broadwell nodes
#(12 cores in the 5th node will go unused)

#PBS -l select=4:ncpus=40:model=sky_ele
#to run all 40 cores on each of the 4 Skylake nodes
#(32 cores in the 4th node will go unused)

#PBS -l select=4:ncpus=40:model=cas_ait
#to run all 40 cores on each of the 4 Cascade Lake nodes
#(32 cores in the 4th node will go unused)

#PBS -l select=1:ncpus=128:model=rom_ait
#to run all 128 cores on 1 Rome node

#PBS -l select=1:ncpus=128:model=mil_ait
#to run all 128 cores on 1 Milan node
```

You can specify both the queue type -q normal, debug, long, low, or devel and the processor type. For example:

```
#PBS -q normal
#PBS -l select=128:ncpus=128:model=mil_ait
```

Example 2: Requesting Different ncpus in a Multi-Node Job

For a multi-node PBS job, the number of CPUs ncpus used in each node can be different. This feature is useful for jobs that need more memory for some processes, but less for other processes. You can request resources in "chunks" for a job with varying ncpus per node.

This example shows a request of two resource chunks. In the first chunk, one CPU in one Broadwell node is requested, and in the second chunk, 28 CPUs in each of three other Broadwell nodes are requested:

```
#PBS -l select=1:ncpus=1:model=bro_ele+3:ncpus=8:model=bro_ele
```

Example 3: Requesting Multiple Node Model Types

A PBS job can run across more than one node model type if the nodes are all from Electra or all from Aitken, but not across the systems. This can be useful in two scenarios:

- When you cannot find enough free nodes within one model for your job
- When some of your processes need more memory while others need much less

You can accomplish this by specifying the resources in chunks, with one chunk requesting one node model type and another chunk requesting a different type.

This example shows a request for one Skylake node (which provides ~185 GB of memory per node) and three Broadwell nodes (which provide ~120 GB of memory per node):

```
#PBS -lplace=scatter:excl:group=model

#PBS -lselect=1:ncpus=24:mpiprocs=24:model=sky_ele+3:ncpus=28:mpiprocs=28:model=bro_ele
```

Running Jobs on Athena

To run a PBS job on the Athena compute nodes, submit the job from one of the Athena front-end systems (ATHFEs) to one of the queues managed by the PBS server pbs06a: devel, normal, long, and wide.

Note: You cannot submit jobs to run on Athena from a PFE.

Running Jobs on Aitken and Electra

To run a PBS job on the Aitken or Electra compute nodes, submit the job from one of the PFEs to one of the queues managed by the PBS server pbspl1.

Aitken Milan Nodes: You can submit batch jobs to run on Aitken's Milan nodes (mil_ait) either from a PFE or from one of the two Aitken Milan front-end systems (afe01, afe02). Interactive PBS jobs for Aitken Milan nodes must be submitted from the AFEs because currently there is no direct network connection to the PFEs.

Running Jobs on Endeavour

To run a PBS job on the shared memory systems Endeavour3/4, submit the job from a PFE to one of the queues managed by the PBS server pbspl4.

Running Jobs on Cabeus

To run a PBS job on:

- Milan A100 (mil_a100) submit the job from a Cabeus front end (cfe01 or cfe02) to one of the queues managed by the PBS server pbspl4.
- Grace Hopper nodes (gh200), submit your job from a Cabeus Grace Hopper front end (cghfe01 or cghfe02) to one of the queues served by the PBS server pbs05a.

PBS Queues

To view a complete list of all queues, including their defaults or limits on the number of cores, wall time, and priority, run the qstat command from an appropriate front-end system as follows:

```
qstat -Q
```

To view more detailed information about each queue, such as access control, defaults, limits on various resources, and status:

```
qstat -fQ queue_name
```

Brief summaries of the various queues are provided in the following sections.

Queues for Athena Users

You can run jobs on Athena using any of the following queues:

```
Queue   NCPUs/      Time/
name      max/def    max/def    pr
======= =====/=== ======/====== ===
devel     256/ 1    02:00/--      100
normal     --/ 1    08:00/--        0
long       --/ 1  120:00/--        0
wide      128/64   24:00/--       20
```

Queues for Aitken and Electra Users

You can run jobs on Aitken and Electra using any of the following queues:

```
Queue   NCPUs/      Time/
name      max/def    max/def    pr
======= =====/=== ======/====== ===
low        --/  8  04:00/ 00:30 -10
normal      --/  8  08:00/ 01:00   0
long       --/  8 120:00/ 01:00   0
debug       --/  8  02:00/ 00:30  15
devel      --/  1  02:00/    -- 101
```

The normal, long and low queues are for production work. The debug and devel queues have higher priority and are for debugging and

development work.

For the debug queue, each user is allowed a maximum of 2 running jobs using a total of up to 128 nodes. For thedevel queue, each user is allowed only 1 job (either queued or running).

Queues for Cabeus GPU Users

For information about queues for the Cabeus GPU nodes, seeRequesting GPU Resources.

See also:

- Preparing to Run on Cabeus Grace Hopper Nodes
- Preparing to Run on Cabeus A100 GPU Nodes

Queues for Endeavour Users

Jobs submitted from PFEs to the e_normal, e_long, e_debug, and e_low queues are routed to run on the Endeavour shared-memory system, which has its own PBS server.

See Endeavour PBS Server and Queue Structure for more information.

Queue for Lou Users

The ldan queue provides access to one of the Lou data analysis nodes (LDANs) for post-processing archive data on Lou.

The ldan queue is managed by the same PBS server as all of the Electra and Aitken queues and is accessible by submitting jobs from an LFE or PFE.

Each job on the ldan queue is limited to 1 node and 72 hours. Each user can have a maximum of 2 running jobs on this queue.

See Lou Data Analysis Nodes for more information.

Queues with Restricted Access

The vlong Queue

Access to the vlong queue requires authorization from the NAS User Services manager.

A maximum of 1024 nodes and 16 days are allowed on this queue.

To avoid losing all useful work in the event of a system or application failure, ensure that your application includes checkpoint and restart capability; this is especially important when using this queue.

The wide Queue

The wide queue is a high priority queue used for jobs that meet either one of the following conditions:

- The number of requested NCPUs (number of nodes * number of cores per node) exceeds 25% of the user's mission shares
- The number of nodes requested of a processor model exceeds 25% of the total number of nodes of that model. See the Electra and Aitken configuration pages for the total number of nodes for each model.

Users do not have direct access to thewide queue. Instead, a job that meets the above criteria is moved into thewide queue by NAS staff. Only 1 job in the wide queue is allowed among Electra and Aitken. When more than one job meets the criteria, preference is given to the job that has been waiting for longer time or whose owner does not have other jobs running.

Athena: You can submit jobs to the Athena wide queue (with minimum of 64 nodes and maximum of 128 nodes for up to 24 hours) without requesting approval from NAS staff.

Various Special Queues

Special queues are higher priority queues for time critical projects to meet important deadlines. Some examples of special queues are armd_spl, astro_ops, heomd_spl, kepler, smd1, smd2, smd_ops , and so on. Request for creation and access of a special queue is carefully reviewed by the mission directorate or by the program POC who authorizes the project's SBU allocation. The request is rarely granted unless the project's principal investigator demonstrates that access to a special queue is absolutely necessary.

Advanced Reservation Queues

An advanced reservation queue allows you to reserve resources for a certain period ahead of time. One circumstance in which the NAS User Services manager may authorize the use of a reservation queue is when a user plans to run an extra-large job, such as one exceeding 15,000 cores, which would never start running without a reservation.

A reservation queue is created by NAS staff for the user and is normally named with the letter R followed by a number.

To find the status of existing PBS advanced reservations, issue the pbs_rstat command on a PFE.

Note: Queues such as alphatst, ded_time, diags, dpr, idle, resize_test, and testing are used for staff testing purposes, and are not accessible by general users.

Electra and Aitken devel Queue

NAS provides a special queue on Electra and Aitken, called the devel queue, that is designated for development work. Using this queue for development work provides faster turnaround than using other, non-designated queues.

Note: The devel queue should not be used for production work. Misuse of the devel queue may result in loss of access to the queue. See devel Queue Policies for more information.

Flexible Node Types

From the devel queue, you can access nodes of all Aitken and Electra processor types.

Requesting Resources

Because the devel queue does not have a default processor type, you must specify the processor type by including the :model=*xxx* attribute in your PBS resource request (where *xxx* is bro_ele, sky_ele, cas_ait or rom_ait. Note that mil_ait is not allowed to be used in devel queue.) For example, to request a Rome node:

```
#PBS -q devel
#PBS -l select=1:ncpus=128:model=rom_ait
```

Submitting Jobs

To submit your job to the devel queue, run the qsub command:

```
pfe21% qsub -q devel job_script
1234.pbspl1.nas.nasa.gov
```

To check the status of a job submitted to the devel queue, run the qstat command:

```
pfe21% qstat -u your_username devel
```

Determining Available Resources

In order to obtain the fastest possible start time, run the node_stats.sh script to view the number and types of nodes currently available in the devel queue before you submit your job. For example:

```
pfe21% node_stats.sh
```

devel Queue Policy

When using the devel queue, be aware of the following policy:

- Each user is allowed only one job on the devel queue, either running or queued. You cannot stack up a list of queued jobs or create self-submitting jobs.
- The maximum wall-clock time of a job is always 2 hours.
- A job may request no more than 512 nodes, either of a single processor/node type or a mixture of different processor/node types.
- The PBS scheduler predominately considers which users have used the most devel queue resources to decide which jobs to run next.
- devel queue usage is applied to your mission share.

The devel queue is intended for developing and testing code and cases to prepare them for production runs in other queues, such as the normal and long queues. It should not be used for performing parameter studies or for breaking a long production run into multiple consecutive runs that each fit within the 2-hour limit.

Disregarding these devel queue policies and guidelines may result in loss of access to the queue.

PBS on Athena

Preparing to Run on Athena Turin Nodes

To help you prepare for running jobs on Athena's Turin nodes, this short user guide includes information on the general configuration of Turin nodes, compiling your code, and running PBS jobs.

Athena Front-End Systems

Four new front-ends (athfe[*01-04*]) serve the Athena Turin compute nodes. Each Athena front-end node includestwo 128-core AMD EPYC 9745 processors with a total of 256 physical cores and 768 GB memory, two 200 Gb/s Slingshot interconnects, and two 25 Gb/s dedicated Ethernet interfaces for external connections.

From your local desktop, access the ATHFEs via a two-step login or a one-step login:

- Two-step login:

  local_desktop% ssh -l *your_NAS_username* sfe6.nas.nasa.gov
  (or replace sfe6 with sfe7 or sfe8)
  sfe6% ssh athfe01 (or ssh athfe02, athfe03, athfe04)

  You can omit -l *your_NAS_username* if your usernames on NAS systems and your local system are identical. If you are not using the NAS config file template, you can add the following lines in your local config file to include all of the ATHFEs:

  ```
  Host athfe01
    HostKeyAlias        athfe01.nas.nasa.gov
    ProxyCommand        ssh -ax -oCompression=no sfe ssh-balance %h

  Host athfe02
    HostKeyAlias        athfe02.nas.nasa.gov
    ProxyCommand        ssh -ax -oCompression=no sfe ssh-balance %h

  Host athfe03
    HostKeyAlias        athfe03.nas.nasa.gov
    ProxyCommand        ssh -ax -oCompression=no sfe ssh-balance %h

  Host athfe04
    HostKeyAlias        athfe04.nas.nasa.gov
    ProxyCommand        ssh -ax -oCompression=no sfe ssh-balance %h
  ```

- Then, run the command:

  local_desktop% ssh athfe01 (or ssh athfe02, athfe03, athfe04)

Athena Turin Compute Nodes

Athena currently includes 1,024 Turin compute nodes partitioned into four physical racks. These racks are housed in Compute Module 2 of the Modular Computing Facility (MSF), located less than one mile from the NAS main building.

Note: We plan to add more nodes to Athena in 2026.

Each node contains two 128-core AMD EPYC 9745 Turin sockets (2.4 GHz) for a total of 256 cores. Each node has 768 GB of memory. With Simultaneous Multithreading (SMT) enabled, each node has 256 physical cores and 512 logical cores.

TIP: It is possible that during Turin's early production, some nodes are configured with SMT on and some with SMT off. To check whether SMT is enabled on a particular Turin compute node, run lscpu | grep "Thread(s) per core" on the node. If it shows 2, then SMT is enabled on that node.

The Athena compute nodes are inter-connected with Cray Slingshot 11 Cassini network interface cards (NICs) and Rosetta Ethernet switches. All nodes have access to the NAS compute resources home filesystems (/u/*username*) and Lustre filesystems (/nobackup/*username*).

The Athena compute nodes are only accessible by submitting PBS jobs from the ATHFEs, not from the PFEs. The PBS server for these compute nodes is pbs06a. Users cannot login to pbs06a.

Requesting CPU Allocations

An SBU allocation for the NAS CPU systems, named hecc in the acct_ytd command output, is required to use the Athena compute nodes through PBS. If you are new to the NAS systems, you can apply for allocations on this web site.

The SBU rates (for using 1 node for 1 hour) for all NAS Systems can be found underJob Accounting.

Operating System and Software Modules

The operating system on all Turin compute nodes is the Red Hat Enterprise (9.6) Linux-based Tri-Lab Operating System Stack Version 5 (TOSS 5). The HPE Performance Cluster Manager (HPCM 1.14) is used to manage the cluster.

Software packages are managed with the Tool Command Language (Tcl) environment modules system. On Athena, you should use the module command from sourcing init scripts under /opt/cray/pe/modules/3.2.11.7, not the one under the /usr/local directory. In your login sessions or job scripts, check whether you are using the correct version of the module command by using:

```
% module -V
```

A module -V output that begins with VERSION=3.2.11.7 shows you are using the right version of the module command:

If your output shows Modules Release 5.3.0 you should modify your scripts to use the following lines for Athena:

```
source /opt/cray/pe/modules/3.2.11.7/init/csh (for csh)
or
source /opt/cray/pe/modules/3.2.11.7/init/bash (for bash)
```

One way to accomplish this is as follows:

For .cshrc:

```
if ( -e /opt/cray/pe/modules/default/init/csh ) then
   source /opt/cray/pe/modules/default/init/csh
else
   source /usr/local/lib/init/global.cshrc
endif
```

For .bashrc:

```
if [ -f /opt/cray/pe/modules/default/init/bash ]; then
  source /opt/cray/pe/modules/default/init/bash
else
  source /usr/local/lib/global.profile
fi
or
source /opt/cray/pe/modules/default/init/bash >& /dev/null || \
   source /usr/local/lib/global.profile
```

Note: The last command line in the code box above is too long to be formatted as one line, so it is broken with a backslash (\).

The default $MODULEPATH upon login includes:

- /opt/modulefiles
- /opt/cray/modulefiles
- /opt/cray/pe/modulefiles
- /opt/cray/pals/modulefiles
- /opt/cray/pe/perftools/25.09.0/modulefiles
- /opt/cray/pe/craype-targets/default/modulefiles
- /nasa/modulefiles/toss5_crayex
- /usr/share/Modules/modulefiles
- /usr/share/modulefiles
- /etc/modulefiles
- /etc/scl/modulefiles

To see what software modules are available, run module avail at the command prompt.

Porting Your Application

The Athena cluster is based on the x86_64 architecture and Cray's proprietary Slingshot interconnects. In order to take advantage of the high-performance Slingshot interconnects, an MPI or an MPI/OpenMP hybrid application needs to be built/linked with the Cray MPICH library. An existing executable built on other NAS CPU systems with the HPE/SGI MPT library or OpenMPI library will not run on Athena. One must rebuild the MPI application on Athena with a cray-mpich module, which is integrated in the Cray Programming Environment (CPE).

The Cray Programming Environment offers choices of various compilers. The default is to use a Cray compiler integrated in the PrgEnv-cray module. If you prefer to use the Intel, AMD AOCC or GNU compiler, one is available and integrated in the corresponding PrgEnv-xxx module.

Note: PrgEnv-xxx is case-sensitive.

Cray Programming Environment

The HPE Cray Programming Environment offers a set of tools tightly integrated for developing, porting, debugging, and tuning applications. It is the main software stack recommended for use on the Athena Turin nodes.

Various PrgEnv-xxx modules are available for choosing an environment with a specific compiler flavor. After loading a PrgEnv-xxx module, invoking the ftn, cc, CC drivers will correctly use the corresponding Fortran, C and C++ compiler integrated within that environment with appropriate options to build and link applications with relevant dynamic libraries.

Choosing the Cray Compiler Integrated in PrgEnv-cray

The CPE version 25.09 is installed on Athena. By default, several modules of CPE 25.09 are loaded automatically when you login to an Athena front-end or compute node.

Note: When a new CPE is added and loaded by default as configured by the NAS system adminstrators in the future, some of the following modules may be updated with a new version.

```
% module list
Currently Loaded Modulefiles:
  1) cray-libpals/1.7.1       7) cce/20.0.0
  2) cray-pals/1.7.1          8) craype/2.7.35
  3) craype-x86-turin         9) cray-dsmml/0.3.1
  4) libfabric/12.0          10) cray-mpich/8.1.33
  5) craype-network-ofi      11) cray-libsci/25.09.0
  6) perftools-base/25.09.0  12) PrgEnv-cray/8.6.0
```

where, in the order of relevance to porting your application,

- 12) PrgEnv-cray
  is a meta-module that sets up a complete programming environment based on Cray's compilers and tools
- 8) craype (Cray Programming Environment Drivers)
  provides compiler/link driver wrappers (ftn, cc, CC) that automatically include the correct compiler options and link relevant libraries as required by the currently loaded modules. The --help option of ftn, cc, CC will print the help page of the corresponding compiler in that environment.
- 7) cce (Cray Compiling Environment)
  includes a suite of Cray compilers, libraries, and utilities that enable users to compile, analyze code to automatically generate highly optimized executables for Cray systems. This modulefile defines the system paths and environment variables needed to run the Cray Compiling Environment. With PrgEnv-cray, ftn, cc, CC invokes the Cray compiler drivers crayftn, craycc, crayCC. See crayftn, craycc, crayCC man pages or use the --help option for available Cray compiler flags.
- 10) cray-mpich
  is a Cray-developed high-performance, optimized implementation of open-source MPICH

  Note: Loading a cray-mpich module provides access to scripts such as mpifort, mpif77, mpif90, mpicc, mpicxx, mpic++ as well as various libmpi* for building MPI applications. Read the intro_mpi man page for more information.

  During runtime, a cray-pals module is needed to access the mpiexec executable in Cray's parallel application launch service (PALS).

- 2) cray-pals
  is a part of PALS that handles the startup and coordination of parallel applications across compute nodes; loading this module automatically loads the cray-libpals module
- 1) cray-libpals
  is a library of PALS
- 5) craype-network-ofi
  enables MPI applications to use the OpenFabrics Interface (OFI) or Libfabric network transport; works in conjunction with cray-mpich to implement MPI communication over the OFI framework, offering performance characteristics that can be compared with the alternative transports like UCX.
- 4) libfabric
  provides a standardized, high-performance communication library for applications running on Cray systems, abstracting the underlying Slingshot network
- 11) cray-libsci
  is a collection of highly optimized numerical routines, such as BLAS, LAPACK, and ScaLAPACK, designed to enhance the performance of scientific and HPC applications on Cray's systems
- 9) cray-dsmml (Distributed Symmetric Memory Management Library)
  is a proprietary library that manages distributed memory heaps for PGAS (Partitioned Global Address Space) languages like Coarray Fortran, UPC, and OpenSHMEM
- 3) craype-x86-turin
  is a module that adds reference to libraries for the purpose of building AMD Turin target apps; For Fortran, it turns on the Cray Fortran compiler flag -h cpu=x86-turin; for C/C++, it sets -target-cpu to znver5.
- 6) pertools-base
  is a foundational component for accessing the suite of Cray performance analysis tools (CrayPat, Apprentice2/3) without altering a program's behavior

If you purge your modules using the module purge command, reloading the PrgEnv-cray/8.6.0 module afterwards will load all of the above except the cray-pals, cray-libpals, craype-x86-turin and the perftools-base modules.

WARNING: If your system startup script (such as .cshrc or .profile) explicitly sources /usr/share/Modules/init/{csh,tcsh,bash,ksh} or /usr/local/lib/init/global.{cshrc,profile} or equivalent, you will encounter errors when you try module load PrgEnv-xxx .

For example:

```
% module purge
% module load PrgEnv-cray
Loading craype/2.7.35
  ERROR: Conflicting PrgEnv-cray/8.6.0 is loading

Loading PrgEnv-cray/8.6.0
  ERROR: Load of requirement craype failed
```

The solution is to (1) comment out sourcing those files, and (2) source /opt/cray/pe/modules/3.2.11.7/init/csh (for csh) or

/opt/cray/pe/modules/3.2.11.7/init/bash (for bash) in your system startup scripts (for Athena) and job scripts.

Choosing the Intel compiler integrated in PrgEnv-intel

If you prefer to use the Intel compiler, loading the PrgEnv-intel/8.6.0 module after the module purge command will load the following modules:

1) intel/2023.2.1 4) libfabric/12.0 7) cray-libsci/25.09.0
2) craype/2.7.35 5) craype-network-ofi 8) PrgEnv-intel/8.6.0
3) cray-dsmml/0.3.1 6) cray-mpich/8.1.33

You can switch module from intel/2023.2.1 to intel-classic/2023.2.1 or intel-oneapi/2023.2.1. The Intel compilers invoked by ftn, cc, and CC when each of these intel modules is loaded are:

module loaded	intel/2023.2.1	intel-classic/2023.2.1	intel-oneapi/2023.2.1
ftn	ifort	ifort	ifx
cc	icx	icc	icx
CC	icpx	icpc	icpx

Note: When you choose to use PrgEnv-intel, the wrappers ftn, cc, CC will use the Intel compilers and link to Cray's libraries, not Intel's libraries. To link to Intel's library, for example, MKL library, add -qmkl={sequential,parallel,cluster} to the ftn, cc, CC command. During runtime, you may have to set the LD_LIBRARY_PATH to include the path of the MKL library (/nasa/intel/oneapi/mkl/2023.2.0/lib/intel64).

Choosing the AMD AOCC compiler integrated in PrgEnv-aocc

If you want to use the AMD AOCC compiler, loading PrgEnv-aocc/8.6.0 will load the aocc/5.0.0 module together with the craype, cray-libsci, cray-mpich, cray-network-ofi, libfabric, cray-dsmml modules.

Choosing the GNU compiler integrated in PrgEnv-gnu

Loading the PrgEnv-gnu/8.6.0 module will load gcc-native/14.2. In addition to the compiler module, the craype, cray-libsci, cray-mpich, cray-network-ofi, libfabric, cray-dsmml modules are also loaded.

Building and Running MPI Applications

Your existing executables built with HPE/SGI MPT or OpenMPI library will not run on the Athena Turin nodes. Rebuild your application using the recommended Cray compiler wrappers (ftn, cc, CC) and MPI library.

Building with PrgEnv-cray

athfe01% module load PrgEnv-cray

This should have also loaded a cray-mpich module. If not, load it explictly.

athfe01% module load cray-mpich

Method 1: Use ftn, cc, CC wrappers

When the modules PrgEnv-cray and cray-mpich are loaded, the Cray compiler wrappers ftn, cc, CC will automatically link the Cray MPI library. This is the preferred way to build an MPI application as the Cray compiler wrappers are designed to work specifically with the Cray MPICH implementation and other Cray system libraries, ensuring compatibility and optimized performance.

athfe01% ftn <FFLAGS> source.f90
athfe01% cc <CFLAGS> source.c
athfe01% CC <CXXFLAGS> source.cpp

TIP: Use the ftn, cc, CC flag -craype-verbose to print the full include and link information used by the compiler wrapper. For even more detailed information, use crayftn flag -v (accepted by ftn) to see what the compiler, lister (which generates a detailed, annotated listing of source code) and linker is doing and what it is calling. The Cray C and C++ compiler option -### acts as a dryrun (i.e., it prints what the compiler, linker, etc. will be doing for your code but does not actually run the command).

Method 2: Use mpifort, mpicc, mpicxx wrappers

Instead of using ftn, cc, CC, you can also use the various mpixxx scripts to build your MPI applications. Furthermore, this is the way to build your MPI application if cray-mpich is loaded but not PrgEnv-cray.

athfe01% mpifort <FFLAGS> source.f90
athfe01% mpicc <CFLAGS> source.c
athfe01% mpicxx <CXXFLAGS> source.cpp

Note: mpif77, mpif90 are soft links to mpifort. mpic++ is a soft link to mpicxx.

Building with PrgEnv-intel

athfe01% module switch PrgEnv-cray PrgEnv-intel

Method 1: Use ftn, cc, CC wrappers

With PrgEnv-intel, the default compiler module loaded is intel/2023.2.1. You can switch from intel/2023.2.1 to intel-classic/2023.2.1 or intel-oneapi/2023.2.1 depending on which combination of Fortran (ifort, ifx), C (icc, icx), and C++ compilers (icpc, icpx) you want. The ftn, cc, CC wrappers will correctly use the compilers associated with each of these three Intel compiler modules and link with the Cray MPICH library.

TIP: You can pass in the Intel compiler flag -dryrun to the ftn, cc, CC wrappers to show what the Intel driver tool commands to be used.

Method 2: Use mpifort, mpicc, mpicxx wrappers

If you prefer to use the mpifort, mpicc, mpicxx from the cray-mpich module to build your MPI applications, be aware that mpifort, mpicc, mpicxx default to using ifx, icx, icpx regardless of which of intel, intel-classic, intel-oneapi module you have loaded. Therefore, it is recommended to set the following environment variables to use the compiler you want before building with mpifort, mpicc, mpicxx from Cray MPICH. For example, to make sure ifort, icc, icpc are used by the mpifort, mpicc, mpicxx wrappers, do the following before invoking mpifort, mpicc, mpicxx:

With csh:

```
setenv MPICH_FC ifort
setenv MPICH_CC icc
setenv MPICH_CXX icpc
```

With bash:

```
export MPICH_FC=ifort
export MPICH_CC=icc
export MPICH_CXX=icpc
```

Method 3: Explictly invoke compilers and link to MPI libraries

If you prefer not to use any wrappers, you will need to explictly provide the paths to the appropriate include files and libraries. For example, if you are building with ifort and linking with the Cray MPICH 8.1.33 library, do:

```
ifort [options] program.f90 \
-I/opt/cray/pe/mpich/8.1.33/ofi/intel/2022.1/include \
-L/opt/cray/pe/mpich/8.1.33/ofi/intel/2022.1/lib \
-lmpi -lmpifort
```

Running MPI applications

To run your MPI executable, submit a PBS job as described in the Running PBS Jobs section. In addition, you will need to load the cray-pals module in order to access the mpiexec command.

```
% module use /opt/cray/pals/modulefiles
% module load cray-pals
Loading cray-pals/1.7.1
Loading requirement: cray-libpals/1.7.1

% which mpiexec
/opt/cray/pals/1.7/bin/mpiexec
```

Read man mpiexec for all options and environment variables.

Note: On Slingshot 11 systems, Cray MPICH sets the variable MPICH_OFI_USE_PROVIDER to cxi (the Cassini interfaces) by default. This variable controls the selection of a provider to use from a list of already-loaded providers. The variable FI_PROVIDER controls what providers to load. Setting FI_PROVIDER to just cxi (to be selected for use by MPICH_OFI_USER_PROVIDER) willl speed up the MPI startup.

Note: When running more than 1 MPI rank per node, both Cassini interfaces on a node will be used by default. There are environment variables (MPI_OFI_NIC_VERBOSE, MPI_OFI_NUM_NICS, MPICH_OFI_NIC_POLICY, MPICH_OFI_NIC_MAPPING, etc.) to display NIC selection, choose the number of NICs to use per node, select the rank-to-NIC assignment policy, specify the precise rank-to-NIC mapping on each node.

Building and Running OpenMP or MPI/OpenMP Hybrid Codes

With the Cray compiler, use the compiler flag -fopenmp or -homp to enable support for OpenMP directives. Without these options, the OpenMP directives are ignored.

Be aware that on a compute node, the default value of the variable OMP_NUM_THREADS is set to the value of ncpus in your PBS resource request. Remember to reset OMP_NUM_THREADS to a value of your choice before running.

TIP: If your OpenMP application is built with Cray OpenMP, set the variable CRAY_OMP_CHECK_AFFINITY to true to see how the OpenMP threads are placed on the cores.

For MPI/OpenMP hybrid code, it is critical to apply process and thread binding to avoid multiple threads running on the same core. The Cray PALS mpiexec includes some options to help with binding. For example, when running on 4 nodes with a total of 256 MPI processes where each node runs 64 MPI processes and 4 OpenMP threads, the following command provides one way for process/thread binding:

```
mpiexec -n 256 --ppn 64 --depth 4 --cpu-bind depth a.out
```

Another way to do process/threading binding is to use NAS's inhouse tool mbind.x, which is available at /u/scicon/tools/bin. You can also access mbind.x by loading the module scicon/cli_tools. To eliminate interference between Cray MPI's own binding scheme with mbind.x, one must add --cpu-bind none to mpiexec when using mbind.x. In addition, mbind.x takes the thread number either from the value of its -t option (higher precedence) or $OMP_NUM_THREADS. When the -t option is not specified and therefore $OMP_NUM_THREADS is used by mbind.x, be aware that the default value of $OMP_NUM_THREADS in a PBS job may be set to the value of NCPUS, which likely is not what you want. The default value of $OMP_NUM_THREADS can also cause surprising binding behavior when using mbind.x to bind MPI processes of pure MPI code. Thus, it is recommended to set $OMP_NUM_THREADS to 1 explicitly for PBS jobs running pure MPI applications.

With mbind.x, to run on 4 nodes with a total of 256 MPI processes where each node runs 64 MPI processes and 4 OpenMP threads, use:

```
mpiexec -n 256 --ppn 64 --cpu-bind none /u/scicon/tools/bin/mbind.x \
 -cs -t4 -v a.out

or

export OMP_NUM_THREADS=4 (bash)
#setenv OMP_NUM_THRREADS 4 (csh)
mpiexec -n 256 --ppn 64 --cpu-bind none /u/scicon/tools/bin/mbind.x \
 -cs -v a.out
```

Running PBS Jobs on Athena Turin Compute Nodes

The Athena Turin compute nodes are accessible only through PBS jobs submitted from athfe[01-04] served by the PBS server pbs06a.

To request Athena Turin compute nodes, you must use :model=tur_ath in your PBS script or the qsub command line, as shown in the example below.

TIP: The use of -lplace=scatter:excl is recommended if you plan to run across nodes with less than 512 processes on each node when SMT is enabled. This placement should be set as default. However, it is a good practice to double-check the setting.

Sample PBS Script For Athena Turin Nodes

```
#PBS -l select=2:ncpus=256:mpiprocs=256:model=tur_ath
# If mpiprocs is not set, it defaults to 1
#PBS -l walltime=8:00:00
#PBS -l place=scatter:excl
#PBS -q normal

# If using ProEnv-intel instead of PrgEnv-cray, uncomment the line below
# module swap PrgEnv-cray PrgEnv-intel
# If the module cray-pals is not loaded by default, load it manually
module use /opt/cray/pals/modulefiles
module load cray-pals

#set FI_PROVIDER may reduce MPI startup time
#with bash
FI_PROVIDER=cxi
#with csh
#setenv FI_PROVIDER cxi

cd $PBS_O_WORKDIR

# Run a total of 512 MPI processes with 256 processes per node
mpiexec -np 512 --ppn 256 ./a.out

# If wanting to use mbind.x for process/thread binding,
# reset OMP_NUM_THREADS from the default value to 1 for pure MPI code
# and you must disable Cray MPI's binding with --cpu-bind none

#with bash
OMP_NUM_THREADS=1
#with csh
#setenv OMP_NUM_THREADS 1
mpiexec -np 512 --ppn 256 --cpu-bind none /u/scicon/tools/bin/mbind.x \
 -cs -v ./a.out
```

Note: The versions of qstat on newer NAS systems (including Athena and Cabeus GH200 nodes) and older NAS systems (including Electra, Aitken, and Endeavour, and Cabeus V100 and A100 nodes) are different. To get the same look and feel of qstat on the newer systems as the older systems, it is recommended to use a NAS provided script called nas_qstat, available under /usr/local/bin on athfe[01-04] and cghfe[01-02]. In addition, the script showq under /u/hsp/bin can be used to visually see all jobs that are running or pending and their status, along with node availability at the bottom of the output.

References

- Cray Programming Environment 25.09 Software for HPCM
- Cray Compiling Environment (25.09)
- Cray Compiler Guide: C and C++ Reference Manual (24.07)
- Cray Compiler Guide: Fortran Reference Manual (24.07)

Running Jobs on Athena

To run a PBS job on the Athena compute nodes, submit the job from one of the Athena front-end systems (ATHFEs) to one of the queues managed by the PBS server pbs06a: devel, normal, long, and wide.

Note: You cannot submit jobs to run on Athena from a PFE.

Running Jobs on Aitken and Electra

To run a PBS job on the Aitken or Electra compute nodes, submit the job from one of the PFEs to one of the queues managed by the PBS server pbspl1.

Aitken Milan Nodes: You can submit batch jobs to run on Aitken's Milan nodes (mil_ait) either from a PFE or from one of the two Aitken Milan front-end systems (afe01, afe02). Interactive PBS jobs for Aitken Milan nodes must be submitted from the AFEs because currently there is no direct network connection to the PFEs.

Running Jobs on Endeavour

To run a PBS job on the shared memory systems Endeavour3/4, submit the job from a PFE to one of the queues managed by the PBS server pbspl4.

Running Jobs on Cabeus

To run a PBS job on:

- Milan A100 (mil_a100) submit the job from a Cabeus front end (cfe01 or cfe02) to one of the queues managed by the PBS server pbspl4.
- Grace Hopper nodes (gh200), submit your job from a Cabeus Grace Hopper front end (cghfe01 or cghfe02) to one of the queues served by the PBS server pbs05a.

PBS Queues

To view a complete list of all queues, including their defaults or limits on the number of cores, wall time, and priority, run theqstat command from an appropriate front-end system as follows:

```
qstat -Q
```

To view more detailed information about each queue, such as access control, defaults, limits on various resources, and status:

```
qstat -fQ queue_name
```

Brief summaries of the various queues are provided in the following sections.

Queues for Athena Users

You can run jobs on Athena using any of the following queues:

```
Queue   NCPUs/      Time/
name      max/def    max/def    pr
======= =====/=== ======/====== ===
devel     256/ 1    02:00/--      100
normal     --/ 1    08:00/--        0
long       --/ 1  120:00/--        0
wide      128/64   24:00/--       20
```

Queues for Aitken and Electra Users

You can run jobs on Aitken and Electra using any of the following queues:

```
Queue   NCPUs/      Time/
name      max/def    max/def    pr
======= =====/=== ======/====== ===
low        --/  8  04:00/ 00:30 -10
normal      --/  8  08:00/ 01:00   0
long       --/  8 120:00/ 01:00   0
debug       --/  8  02:00/ 00:30  15
devel      --/  1  02:00/    -- 101
```

The normal, long and low queues are for production work. The debug and devel queues have higher priority and are for debugging and

development work.

For the debug queue, each user is allowed a maximum of 2 running jobs using a total of up to 128 nodes. For thedevel queue, each user is allowed only 1 job (either queued or running).

Queues for Cabeus GPU Users

For information about queues for the Cabeus GPU nodes, seeRequesting GPU Resources.

See also:

- Preparing to Run on Cabeus Grace Hopper Nodes
- Preparing to Run on Cabeus A100 GPU Nodes

Queues for Endeavour Users

Jobs submitted from PFEs to the e_normal, e_long, e_debug, and e_low queues are routed to run on the Endeavour shared-memory system, which has its own PBS server.

See Endeavour PBS Server and Queue Structure for more information.

Queue for Lou Users

The ldan queue provides access to one of the Lou data analysis nodes (LDANs) for post-processing archive data on Lou.

The ldan queue is managed by the same PBS server as all of the Electra and Aitken queues and is accessible by submitting jobs from an LFE or PFE.

Each job on the ldan queue is limited to 1 node and 72 hours. Each user can have a maximum of 2 running jobs on this queue.

See Lou Data Analysis Nodes for more information.

Queues with Restricted Access

The vlong Queue

Access to the vlong queue requires authorization from the NAS User Services manager.

A maximum of 1024 nodes and 16 days are allowed on this queue.

To avoid losing all useful work in the event of a system or application failure, ensure that your application includes checkpoint and restart capability; this is especially important when using this queue.

The wide Queue

The wide queue is a high priority queue used for jobs that meet either one of the following conditions:

- The number of requested NCPUs (number of nodes * number of cores per node) exceeds 25% of the user's mission shares
- The number of nodes requested of a processor model exceeds 25% of the total number of nodes of that model. See the Electra and Aitken configuration pages for the total number of nodes for each model.

Users do not have direct access to thewide queue. Instead, a job that meets the above criteria is moved into thewide queue by NAS staff. Only 1 job in the wide queue is allowed among Electra and Aitken. When more than one job meets the criteria, preference is given to the job that has been waiting for longer time or whose owner does not have other jobs running.

Athena: You can submit jobs to the Athena wide queue (with minimum of 64 nodes and maximum of 128 nodes for up to 24 hours) without requesting approval from NAS staff.

Various Special Queues

Special queues are higher priority queues for time critical projects to meet important deadlines. Some examples of special queues are armd_spl, astro_ops, heomd_spl, kepler, smd1, smd2, smd_ops , and so on. Request for creation and access of a special queue is carefully reviewed by the mission directorate or by the program POC who authorizes the project's SBU allocation. The request is rarely granted unless the project's principal investigator demonstrates that access to a special queue is absolutely necessary.

Advanced Reservation Queues

An advanced reservation queue allows you to reserve resources for a certain period ahead of time. One circumstance in which the NAS User Services manager may authorize the use of a reservation queue is when a user plans to run an extra-large job, such as one exceeding 15,000 cores, which would never start running without a reservation.

A reservation queue is created by NAS staff for the user and is normally named with the letter R followed by a number.

To find the status of existing PBS advanced reservations, issue the pbs_rstat command on a PFE.

Note: Queues such as alphatst, ded_time, diags, dpr, idle, resize_test, and testing are used for staff testing purposes, and are not accessible by general users.

PBS on Cabeus

Cabeus includes 538 GPU nodes based on the NVIDIA GH200 Grace Hopper Superchip.

To help you prepare for running jobs on the Grace Hopper GPU nodes, this short user guide includes information on the general configuration of the nodes, compiling your code, and running PBS jobs.

Overview of Cabeus Nodes with NVIDIA Grace Hopper Superchip

The nodes are accessed via two front ends:

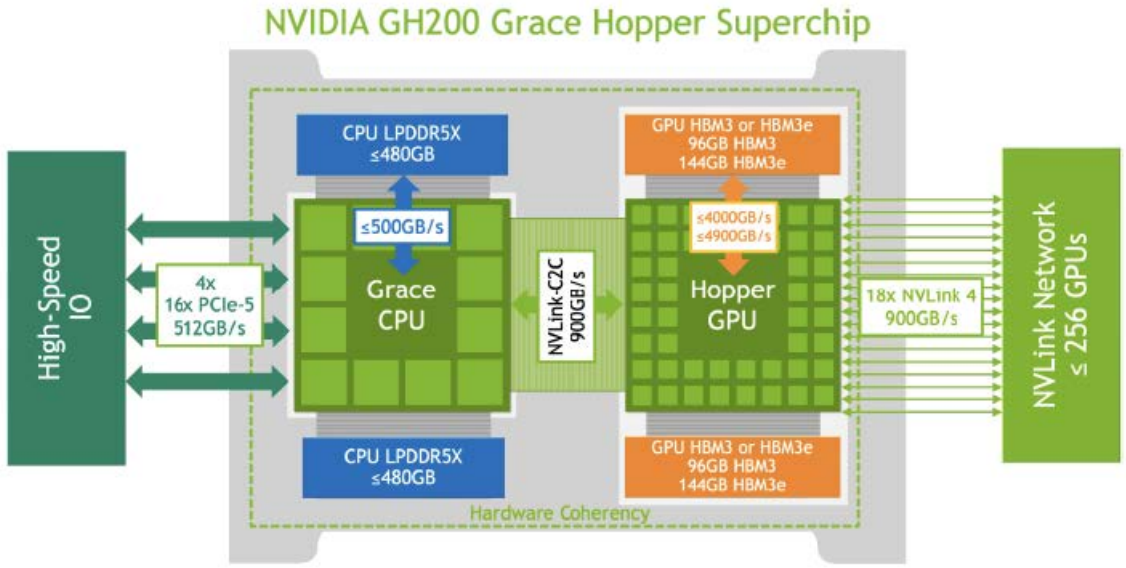
- cghfe01
- cghfe02

The hostnames for the GH200 compute nodes are as follows:

- cgh[1-3,6-15,17-19,21-23]n[01-26]
- cgh[04,20]n[01-12]
- cgh16n[01-20]

The PBS resource model name is `gh200`.

Each of the front ends and compute nodes contains a single GH200 superchip, which is shown in this diagram published by NVIDIA:



As shown in the diagram, the GH200 superchip includes:

- One Grace CPU containing 72 Arm Neoverse V2 physical cores (maximum 3.465 GHz on the front ends; slight variation on compute nodes but should be all above 3.4 GHz; use the `lscpu` command to check) and up to 480 GB Low-Power Double Data Rate 5X (LPDDR5X) host memory with a memory bandwidth of up to 500 GB/s.

  Note: The memory bandwidth varies with memory capacity: 384 GB/s with a capacity of 480 GB as in all NAS GH200 nodes, and 512 GB/s with a capacity of 120 GB or 240 GB.

  The Arm Neoverse V2 CPU uses the Armv9.0-A architecture which comes with four 128-bit Single Instruction Multiple Data (SIMD) units that support Scalable Vector Extension 2 (SVE2) instructions.
- One H100 (Hopper) GPU containing 132 Streaming Multiprocessors (SMs) with a clock rate of 1.980 GHz and 96 GB High Bandwidth Memory 3 (HBM3) device memory with a memory bandwidth of up to 4000 GB/s. Each SM contains 128 single precision CUDA cores (total is 16,896 per H100 GPU), 64 double precision CUDA cores (total is 8,448 per H100 GPU), and 4 Tensor cores (total is 528 per H100 GPU).
- A high-bandwidth, memory-coherent NVIDIA NVLink-C2C (chip-2-chip) interconnect that connects the CPU and GPU with a bidirectional bandwidth of 900 GB/s.

  NVLink-C2C enables applications to oversubscribe the Hopper GPU's memory and directly utilize the Grace CPU's memory at high bandwidth. The Grace CPU can also directly access the GPU memory. The memory coherency enables programming with a unified programming model.
- 4x (4 links) PCIe 5.0 x16 (16 lanes) with a total bi-directional bandwidth of 512 GB/s.

  The GH200 nodes at NAS use the NVIDIA MGX platform design where the nodes are interconnected with PCIe and host channel adapter (HCA) on each node and NVIDIA's internode InfiniBand networking solution. Specifically, InfiniBand Next-Generation Data Rate (NDR) HCAs, switches, and cables are utilized in a fat tree topology.

  Note: The 18x NVLink 4 network shown on the right side of the diagram above does not apply to the NVIDIA MGX platform. It is

used in the NVIDIA DGX platform for connecting up to 256 GPUs together.

For more information about the Grace Hopper superchip, see NVIDIA's GH200 architecture whitepaper.

Requesting GPU Allocations

A Standard Billing Unit (SBU) allocation specifically for the GPUs is required to use all GPU resources, including the GH200 compute nodes. The SBU rate for the GH200 nodes is 18.34.

Accessing the Cabeus Grace Hopper Front Ends and Compute Nodes

The GH200 front end nodes (cghfe01 and cghfe02) are for building, testing your software dependencies, and running short tests of your executables.

Note: Termination of long-running CPU and GPU processes may be enforced in the future.

You can access the GH200 front ends from your local desktop using one of the following methods:

One-Step Connection With SSH passthrough

If you normally connect to NAS front-ends using SSH passthrough, you can modify your existing .ssh/config file on your local host to add cghfe01 and cghfe02. If you have been using a configuration file based on the NAS config template, you can simply replace this line:

```
Host *.nas.nasa.gov lou lou? ?fe ?fe? ?fe?? *-last pbspl?
```

with the following line:

```
Host *.nas.nasa.gov lou lou? *fe* *-last pbs*
```

where *fe* matches any NAS front-end system. Then, run:

```
your_localhost% ssh cghfe01 (or ssh cghfe02)
```

Two-Step Connection Without SSH Passthrough

To access the front ends without SSH passthrough, run:

```
your_localhost% ssh sfe6 (or ssh sfe7, ssh sfe8)
sfe% ssh cghfe01  (or ssh cghfe02)
Password: xxxxxxxxxxx
cghfe01%
```

The gh200 compute nodes are for running long testing or production jobs. They are accessible only through PBS jobs submitted from the cghfe01 or cghfe02 front ends. The PBS server for the GH200 nodes is pbs05a (users are not allowed to SSH into pbs05a).

You can access your filesystems from the GH200 front-ends and compute nodes, including: $HOME, Lustre /nobackup, /nobackupnfs1, /nasa, /nasa2, and /vast_swbuild.

Operating Systems

The operating system on all GH200-based front ends and compute nodes is TOSS 5 with Red Hat Enterprise Linux 9.8. The Linux kernel is Release 5.x for the Arm architecture 64-bit execution state (aarch64).

```
% cat /etc/os-release
NAME="Red Hat Enterprise Linux"
VERSION="9.8 (Plow)"

% uname -r
5.14.0-687.39.1.1toss.t5.aarch64
```

Software Modules

For AI/ML Applications

If you run AI/ML applications: the miniconda3/gh2 module in the /swbuild/analytix/tools directory is the only one that will work with the GH200 nodes:

```
cghfe01% module use -a /swbuild/analytix/tools/modulefiles
cghfe01% module load miniconda3/gh2
```

For more information, including the conda environments that can be used with the Grace Hopper nodes, see Using Conda Environments on NAS Systems.

The procedure to access JupyterLab on the GH200 system is the same as those outlined in Secure Setup for Using Jupyter Notebook on NAS Systems and Using Jupyter Notebook for Machine Learning Development on NAS Systems, except on the local system the

user needs to proxyjump through a Grace Hopper front end. For example:

```
your_local_system% ssh -l username -o "StrictHostKeyChecking ask" \
-L 8080:/tmp/user.sock -o ProxyJump=sfe,cghfe01
```

For HPC Applications

You can find HPC software modules for the Arm-based GH200 nodes in the /nasa/arm directory. The current default $MODULEPATH includes /nasa/arm/modulefiles/toss5, where you can find many of the modules mentioned in this article.

```
cghfe01% echo $MODULEPATH
/etc/scl/modulefiles:/usr/share/Modules/modulefiles:/etc/modulefiles:/usr/share/ \
modulefiles:/nasa/arm/modulefiles/toss5
```

Note: The command line above is too long to be formatted as one line, so it is broken with a backslash (\).

Some of the available software packages are mentioned in the Programming and Compiling for Grace Hopper Nodes and Running a GPU Application sections. Be aware that many nvhpc or nvhpc-xxx modules (packaged and pre-built by NVIDIA) contain multiple components (such as CUDA, compiler, math, MPI, .., and so on.). If you want to use only one of the components, you can try other modules, for example, those with module names such as cuda, hpcx, openmpi, and so on.

WARNING: Software modules and executables built for the x86_64 platform (such as NAS PFEs, AFEs, and CFEs; bro_ele, sky_ele, cas_ait, rom_ait, mil_ait, and mil_a100 GPU nodes) cannot be used on the aarch64 platform (i.e., gh200 GPU nodes) because x86_64 and aarch64 architectures have different instruction sets. You must use modules under the /nasa/arm directory to recompile and run your applications for the Arm-based GH200 systems.

If your system startup script loads modules meant for use on x86_64 based systems (for example, the .bashrc, .profile, .cshrc, or .login scripts, which are accessible from all NAS x86_64 and Arm systems), you should unload those modules when you are on the GH200 front-ends and compute nodes.

Programming and Compiling for Grace Hopper Nodes

In addition to the information in this section, read Programming for NVIDIA GPUs and the references listed there for information on the recommended programming models for developing applications suitable for running on the NVIDIA GPUs. Some of the most commonly used software packages needed with these programming languages are CUDA toolkits, compilers, and math and MPI libraries.

CUDA Toolkit

You can find the CUDA version and its associated driver installed on the GH200 front ends and compute nodes by running the nvidia-smi command. Currently, it is CUDA version 13.3 and driver is 610.57.04.

Since the CUDA driver is backward compatible, an application compiled with an older CUDA Toolkit version can run with a newer driver release. For example, you can build your GPU application with CUDA Toolkit 12.6, and run it on a system with CUDA 13.3. See CUDA Toolkit and CUDA Driver for more information.

The following CUDA Toolkits are available for use on the GH200 nodes:

- **CUDA 12.6**

  ```
  module load nvhpc/25.1 (or module load nvhpc-nompi/25.1)
  nvcc --version
  nvcc: NVIDIA (R) Cuda compiler driver
  Copyright (c) 2005-2024 NVIDIA Corporation
  Built on Wed_Oct_30_00:08:18_PDT_2024
  Cuda compilation tools, release 12.6, V12.6.85
  Build cuda_12.6.r12.6/compiler.35059454_0
  ```

  OR

  ```
  module load nvhpc/24.11 (or module load nvhpc-nompi/24.11)
  nvcc --version
  nvcc: NVIDIA (R) Cuda compiler driver
  Copyright (c) 2005-2024 NVIDIA Corporation
  Built on Thu_Sep_12_02:27:38_PDT_2024
  Cuda compilation tools, release 12.6, V12.6.77
  Build cuda_12.6.r12.6/compiler.34841621_0
  ```

- **CUDA 12.3**

  ```
  module load nvhpc/24.3 (or nvhpc-nompi/24.3 or nvhpc-openmpi3/24.3)
  nvcc --version
  nvcc: NVIDIA (R) Cuda compiler driver
  Copyright (c) 2005-2023 NVIDIA Corporation
  Built on Fri_Nov__3_17:22:02_PDT_2023
  Cuda compilation tools, release 12.3, V12.3.103
  Build cuda_12.3.r12.3/compiler.33492891_0
  ```

Compilers

There are several compiler options for building your applications on the GH200 front ends. With any of these compilers, make sure your application builds, runs successfully, and validates correctly.

NVIDIA Compilers: nvc, nvc++, nvfortran, nvcc

To use an NVIDIA compiler, load one of the nvhpc modulefiles on either the cghfe01 or cghfe02 front end or on the GH200 compute nodes. Below are examples of a few possible choices:

- For the 25.1 version of the NVIDIA compilers, use:

  ```
  % module load nvhpc/25.1
  or
  % module load nvhpc-nompi/25.1
  ```

- For the 24.11 version of the NVIDIA compilers, use:

  ```
  % module load nvhpc/24.11
  or
  % module load nvhpc-nompi/24.11
  ```

- For the 24.3 version of the NVIDIA compilers, use:

  ```
  % module load nvhpc/24.3
  or
  % module load nvhpc-nompi/24.3
  ```

The NVIDIA compilers include the HPC compilers and the CUDA C/C++ compiler:

- NVIDIA HPC Compilers: nvc, nvc++, nvfortran

  These compilers can be used to compile pure CPU applications or GPU applications written in C/C++ and Fortran language standards (through the -stdpar[=gpu,multicore] flag); OpenACC (-acc[=gpu,multicore] flag); OpenMP (-mp[=gpu,multicore] flag), and CUDA Fortran. The gpu and multicore options let you choose whether to compile the application to run on the GPUs or on the multicore CPUs.

  Use the -tp=native flag or the -mcpu=native flag when compilation and execution occur on the same CPU processor (in this case, the Grace CPU) and use the -gpu=cc90 flag (i.e., compute capability 9.0) to target the Hopper GPU.

- NVIDIA CUDA C/C++ Compiler: nvcc

  The nvcc compiler is used to build CUDA C/C++ applications to run on the GPUs. This compiler splits the source code into *device code* and *host code*. The device code is compiled by nvcc. nvcc sends the host code to a C/C++ compiler on the host for compilation—for example, the gcc and g++ GNU compilers in the /usr/bin directory. You can choose a different host compiler—for example, a different version of GNU compiler—through the nvcc flag -ccbin=/*path_to_compiler* or --compiler-bindir=/*path_to_compiler*.

For more information, see Compiling GPU Applications for Different Generations of NVIDIA GPUs at NAS.

GNU Compiler Collection: gcc, g++, gfortran

- GNU compilers at the system location:

  ```
  cghfe01% which gcc
  /usr/bin/gcc
  cghfe01% gcc --version
  gcc (GCC) 11.5.0 20240719 (Red Hat 11.5.0-14)

  cghfe01% which gfortran
  /usr/bin/gfortran
  cghfe01% gfortran --version
  GNU Fortran (GCC) 11.5.0 20240719 (Red Hat 11.5.0-14)
  ```

  These compilers can only be used for building CPU-only applications, as they were not built with offloading support.

For all of the above compilers, NVIDIA recommends:

- Use the -mcpu=native flag instead of the -march=xxx or -mtune=xxx flags to optimize for the native CPU and use its features.
- Depending on the char sign type you assume for your application. you may need to use one of these C language flags: -fsigned-char, -funsigned-char, or -fnosigned-char.
  Note: For NVIDIA HPC compilers (nvc, nvc++, nvfortran), the default char sign type is -fsigned-char on x86_64 systems and -funsigned-char on aarch64 systems. For the Arm Compiler for Linux, the default is -fno-signed-char.

For the GNU compiler and/or Arm compiler:

- If using -Ofast results in unacceptable fast math optimizations, use -O3 -ffp-contract=fast. For even more accuracy, use -ffp-contract=off to disable floating point operation contraction used in fused multiply-add.
- Use -flto to enable link-time optimization, which can be significant.

For the GNU compiler, you can use `-fsanitize=address -fsanitize=undefined` to help debugging memory issues that might appear on aarch64 systems (but are hidden on x86 systems due to differences on how compiler generate codes and layout data).

Math Libraries

There are two sets of math libraries. One targets the GPUs and the other targets the Grace CPUs.

- **CUDA libraries targeting the Hopper GPUs:**
  This set of libraries includes: cuBLAS, cuFFT, cuRAND, cuSOLVER, cuSPARSE, and cuTENSOR. They are located in the following directories:
  - If using module nvhpc/25.1:
    /nasa/arm/software/nvhpc/25.1/Linux_aarch64/25.1/math_libs/lib64
  - If using module nvhpc/24.11: /nasa/arm/software/nvhpc/24.11/Linux_aarch64/24.11/math_libs/lib64
  - If using module nvhpc/24.3: /nasa/arm/software/nvhpc/24.3/Linux_aarch64/24.3/math_libs/lib64

  When using the NVIDIA HPC compilers (nvc, nvc++, and nvfortran), these libraries can be linked with the flag `-cudalib[=cublas|cufft|...]`.

- **NVIDIA Performance Libraries (NVPL) targeting the Grace CPUs:**
  NVPL is a set of optimized math libraries (BLAS, FFT, LAPACK, RAND, SCALAPACK, SPARSE, and TENSOR) that are used for the Grace CPUs. The NVPL set is available with NVIDIA HPC SDK version 24.3 (and later) in the /nasa/arm/.../math_libs/nvpl/libs directory.
  When using the NVIDIA HPC compilers (nvc, nvc++, and nvfortran), these libraries (with the exception of SCALAPCK) can be linked with the flag `-Mnvpl[=blas,fft,...]`. To link with NVPL SCALAPACK, use both `-Mscalapack` and `-Mnvpl`.
  If using the GNU compiler for linking with NVPL, use the `-l`, `-L`, and `-lnvpl_...` options explicitly.

MPI Libraries

The GH200 nodes are not supported by HPE, and therefore HPE MPT library is not available on these nodes.

There are many modules containing OpenMPI. If you are not sure which one to use, start with one of these OpenMPI modules. Read Using OpenMPI on NAS Systems to learn how to use OpenMPI.

OpenMPI Version 4.1.7

- nvhpc/25.1: provided by NVIDIA, and built with gcc, g++, nvfortran, HPCX 2.21, and CUDA 12.6.
- nvhpc/24.11: provided by NVIDIA, and built with gcc, g++, nvfortran, HPCX 2.20, and CUDA 12.5.
- nvhpc/24.3: provided by NVIDIA, and built with gcc, g++, nvfortran, HPCX 2.17.1, and CUDA 12.2.
- hpcx/2.21: open source version of precompiled HPCX. The OpenMPI provided in this distribution was built with gcc, g++, gfortran, HPCX 2.21, and CUDA 12.6.
- openmpi/4.1.7-cuda12.6-gnu: built by NAS staff with GNU compiler, UCX 1.17, and CUDA 12.6.
  Loading `openmpi/4.1.7-cuda12.6-gnu` automatically loads `nvhpc-nompi/24.11`, which provides access to CUDA 12.6.
- openmpi/4.1.7-cuda12.6-nv: built by NAS staff with nvc, nvc++, nvfortran, UCX 1.17, and CUDA 12.6.
  Loading `openmpi/4.1.7-cuda12.6-nv` automatically loads `nvhpc-nompi/24.11` which provides access to NVIDIA HPC compilers and CUDA 12.6.

OpenMPI Version 3.1.5

- nvhpc-openmpi3/24.3: provided by NVIDIA, and built with nvc, nvc++, nvfortran, CUDA 11.0, and without UCX.

TIP: When using newer versions of OpenMPI with UCX, for example, from nvhpc/24.11 or hpcx/2.21, setting the UCX_PROTO_ENABLE=n and UCX_GVA_ENABLE=n environment variables may help to reduce run-to-run variations or degraded performance in MPI communication. Set them as follows:

```
mpiexec -np .. -x UCX_PROTO_ENABLE=n -x UCX_GVA_ENABLE=n ...
```

For applications that perform UCX atomic operations on the Grace CPU, you may also need to add:

```
UCX_IB_MLX5_DEVX=no
```

See NVIDIA HPCX Known Issues (3606445, 3884209, 4026461 and 4050321) for more information.

Running a GPU Application

PBS jobs running on the GH200 nodes are served by a dedicated PBS server, pbs05a, running with PBS version 2024.1.4.

A GH200 compute node (72 CPU cores; 1 GPU; all host and device memory) cannot be divided or shared between PBS jobs or between different chunks of a PBS job (for example, `-lselect=`*N* means *N* chunks). The GH200 node model name is gh200. In your PBS resource request, you must use `:model=gh200`. You do not need to use `-lplace=scatter:excl`.

To run on the GH200 nodes, submit your PBS interactive or batch job from one of the two Grace Hopper front ends, cghfe01 or cghfe02. Use one of the following queues:

- gpu_normal: default priority (0), maximum walltime 24 hours.

- gpu_devel: priority 10, maximum walltime 2 hours, number of nodes 2.
- gpu_wide: priority 20, maximum walltime 12 hours (wide is considered 85+ nodes requested for the job).
- gpu_long: default priority (0); maximum walltime 120 hours; number of nodes 8; number of running jobs 2.

Note: The available queues will change from time to time. Use the command qstat -Q and qstat -fQ to find what queues have been created and enabled to be used by your group.

See the following sample PBS script.

Sample PBS Script for Running Jobs on Grace Hopper Nodes

```
#!/bin/bash
#PBS -lselect=2:ncpus=72:ngpus=1:model=gh200
#PBS -q name_of_queue

module load nvhpc/24.11

# for running 144 MPI ranks on the CPU cores across two nodes
# with 72 MPI ranks per node
# if necessary, also add -x LD_LIBRARY_PATH
mpiexec -np 144 -N 72 --machinefile $PBS_NODEFILE ./a_CPU_executable

# for running 2 MPI ranks on two GPUs across two nodes
# with 1 MPI rank per node
# if necessary, also add -x LD_LIBRARY_PATH
mpiexec -np 2 -N 1 --machinefile $PBS_NODEFILE ./a_GPU_executable
```

After job submission, you should see a JOBID similar to *xxx*.pbs0a5.gh.nas.nasa.gov.

Note: The versions of qstat on newer NAS systems (including Athena and Cabeus GH200 nodes) and older NAS systems (including Electra, Aitken, and Endeavour, and Cabeus V100 and A100 nodes) are different. To get the same look and feel of qstat on the newer systems as the older systems, it is recommended to use a NAS provided script called nas_qstat, available under /usr/local/bin on athfe[01-04] and cghfe[01-02]. In addition, the script showq under /u/hsp/bin can be used to visually see all jobs that are running or pending and their status, along with node availability at the bottom of the output.

References

- NVIDIA Grace Hopper Superchip Architecture White Paper
- NVIDIA Grace Performance Tuning Guide

The Cabeus cluster comprises multiple NVIDIA GPU nodes, including:

- 128 nodes with Milan CPU and NVIDIA A100 GPUs (model=mil_a100)
- 350 nodes with NVIDIA GH200 Superchip (model=gh200)

Among them, the gh200 nodes are served by their own front ends (cghfe01 and cghfe02) and PBS server (pbs05a), while the other GPU nodes are served by the front ends cfe01, cfe02 and PBS server pbspl4.

In this article, the term *Cabeus A100 GPU nodes* refers to the 128 nodes with Milan CPU and NVIDIA A100 GPUs.

To help you prepare for running jobs on the Cabeus A100 GPU nodes, this short user guide includes information on the general configuration of the nodes, compiling your code, and running PBS jobs.

Overview of Cabeus Nodes with AMD EPYC CPU and NVIDIA A100 GPUs

The Cabeus cluster includes two Cabeus front-end nodes (cfe01 and cfe02) for the GPU nodes with x86_64-based CPUs (mil_a100). Each of the CFEs includes one 16-core AMD EPYC 7313p processor and 256 GB memory.

Note: The Cabeus compute nodes are not accessible from the PFEs.

Each of the 128 A100 compute nodes, referred to as the mil_a100 nodes, contains:

- One 64-physical-core AMD EPYC 7763 Milan CPU socket (2.45 GHz) with 512 GB DDR4 host memory.
- Four NVIDIA A100 accelerator cards each with 3,456 double precision CUDA cores, 432 Tensor cores (boost clock 1410 MHz), and 80 GB HBM2e device memory
- Third-generation NVLink with 12 links yielding 600 GB/sec total bandwidth for GPU-to-GPU communication
- PCIe 4.0 bus connecting the CPU socket and the GPU accelerator cards with a bandwidth of 31.5 GB/s per direction

These 128 compute nodes are partitioned into twelve physical racks. They are connected in a fat-tree topology to the Cabeus InfiniBand network (ib0 and ib1) via four-lane High Data Rate (4X HDR) devices and 4X HDR switches. These HDR devices and switches provide a maximum bandwidth of 200 Gbits/s (i.e., 25 GB/s) for internode communication.

All existing HECC filesystems such as the home, VAST, and Lustre filesystems are mounted on both CFEs and on the Cabeus compute nodes.

PBS jobs for running on Cabeus A100 nodes are managed by the PBS server pbspl4.

Requesting GPU Allocations

An SBU allocation specifically for the GPUs is required to use the GPU resources. For more information about requesting GPU allocations, see this article.

The SBU rates for using one GPU node for one hour, for each type of node, are:

- mil_a100 with four A100s: 37.86
- gh200 with one GH200 Superchip: 18.34

Depending on how you request the GPU resources, your job may be charged the full rate or a fraction of the SBU rate. See the next section for some examples.

Accessing the CFEs and the mil_a100 nodes

You can access the Cabeus front ends (CFEs) either from your local desktop or from the PFEs. If you have set upSSH passthrough for the CFEs, then do:

```
your_localhost% ssh cfe01   (or ssh cfe02)
```

If you are connected to a PFE, then do:

```
pfe% ssh cfe01 (or ssh cfe02)
```

You can access the Cabeus Milan A100 compute nodes only through PBS jobs submitted from the CFEs or from pbspl4. To request Milan A100 (mil_a100) resources, specify one of the queue names shown below and the model type mil_a100.

The following PBS attributes are required when requesting the A100 resources:

- **model=mil_a100:** The A100 nodes with the AMD EPYC 7763 CPU are designated with a model name mil_a100.
- **-q**, followed by one of the queues shown below, to access the mil_a100 nodes.

The PBS queues for the Milan A100 nodes are as follows:

- **gpu_normal**
  - Maximum wall time: 24 hrs
  - Maximum node count: 32

- **gpu_devel**
  - Maximum wall time: 2 hrs
  - Maximum node count: 2
  - Maximum user jobs (queued or running): 1
  - Priority: 10

- **np_gpu_wide (non-primetime)**
  - Maximum wall time: 12 hrs
  - Maximum node count: 64
  - Minimum node count: 24
  - Maximum user jobs (queued or running): 2
  - Priority: 20

Important: The **np_gpu_wide** queue will run jobs between 7:00 a.m. Thursdays and 7:00 a.m. Saturdays (Pacific Time). You can submit jobs to the **np_gpu_wide** queue at any time during the week, but the jobs will run only during this period.

As shown in the examples below, adding @pbspl4 to the queue name is optional, whether you submit the job from a CFE or from pbspl4.

```
cfe% qsub -I -q gpu_devel@pbspl4
   -lselect=1:ncpus=10:ngpus=1:mem=xxGB:model=mil_a100
```

or

```
pbspl4% qsub -I -q gpu_devel
   -lselect=1:ncpus=10:ngpus=1:mem=xxGB:model=mil_a100
```

Note: The qsub commands are shown on two lines due to a KB formatting issue. They should be on one line.

When not specified, the default value of ncpus, ngpus, and host mem for all GPU node types (including mil_a100) is 1, 1, and 32 GB, respectively. The minimum value of ncpus, ngpus, and host mem is 1, 1, and 256 MB, respectively.

From the perspective of PBS, one physical node can be made up of multiple virtual nodes (*vnodes*). In the case of mil_a100, one physical node with 4 GPUs is partitioned as 4 vnodes, where each vnode includes 1 GPU, 16 physical cores, and 128 GB of memory.

PBS allocates resources in the vnodes differently depending on the actual selection specified in -lselect=... and the choice of the PBS -lplace statement, which includes an arrangement attribute (free, pack, scatter, vscatter) and a sharing attribute (shared, excl, and exclhost). Read **man qsub** for the definition of each value of the arrangement and the sharing attribute.

If you plan to use fewer than 4 GPUs per physical mil_a100 node, you can use the shared sharing attribute which allows other jobs to use the remaining GPUs in the node. It also allows your job to be charged for only the number of GPUs assigned to your job. For example, the following request will be charged for the use of only 1 GPU in a mil_a100 node:

```
-lselect=1:ncpus=16:ngpus=1:mem=60GB:model=mil_a100 -lplace=scatter:shared
```

If you do not want other jobs to share resources in the mil_a100 nodes assigned to your job, it is easier if you simply request the use of all 4 GPUs in each mil_a100 node. For example:

```
-lselect=1:ncpus=16:ngpus=4:mem=60GB:model=mil_a100 -lplace=scatter:excl
```

This request will result in being charged for all 4 GPUs in a mil_a100 node.

Operating System and Software Modules

The operating system on all A100 compute nodes is TOSS 4 (Red Hat Enterprise Linux-based Tri-Lab Operating System Stack Version 4). Batch jobs use :aoe=toss4 by default, so you do not need to add :aoe=toss4 in your PBS resource request. The default $MODULEPATH includes:

- /usr/share/Modules/modulefiles
- /nasa/modulefiles/toss4
- /nasa/modulefiles/spack
- /nasa/modulefiles/pkgsrc/toss4

Programming and Compiling For Cabeus A100 Nodes

Read Programming for NVIDIA GPUs and the references listed there for information on the recommended programming models for developing applications suitable for running on the NVIDIA GPUs.

Depending on the programming model and language chosen, use an appropriate compiler available in the NVIDIA HPC SDK to compile your application:

- NVIDIA HPC Compilers
  - nvfortran
  - nvc
  - nvc++
- NVIDIA CUDA C/C++ Compiler - nvcc

Run the command module avail nvhpc to find available versions of NVIDIA HPC SDK on the system. The examples shown in this article use nvhpc/23.7. However, you can choose a different version.

```
% module load nvhpc/23.7
```

or

```
% module load nvhpc-nompi/23.7
```

For building NVIDIA GPU applications with MPI for inter-GPU or inter-node communication, you can use either OpenMPI or HPE MPT:

OpenMPI from nvhpc

Example 1:

```
module load nvhpc/23.7
mpif90 -o your_application.exe -acc=gpu -gpu=cc80
    -Minfo=accel your_application.f
```

Note: The mpif90 line is shown on two lines due to KB formatting issues. It should be a single line.

Example 2:

```
module load nvhpc/23.7
mpicc -o main.o -c main.c
nvcc  -o cuda_code.o -c cuda_code.cu
mpicc -o your_program.exe main.o cuda_code.o -lcudart
```

Note: The default OpenMPI version in nvhpc/23.7 is 3.1.5.

HPE MPT library

This is recommended for building/running applications when using NAS GPU nodes supported by HPE.

Example 1:

```
module load nvhpc-nompi/23.7
module load mpi-hpe/mpt
nvfortran -o your_application.exe \
-acc=gpu -gpu=cc80 -Minfo=accel your_application.f -lmpi
```

Example 2:

```
module load nvhpc-nompi/23.7
module load mpi-hpe/mpt
nvc  -o main.o -c main.c -lmpi
nvcc -o cuda_code.o -c cuda_code.cu
nvc  -o your_program.exe main.o cuda_code.o -lcudart -lmpi
```

Instead of explicitly using nvfortran/nvc/nvc++ ... -lmpi in the command line, you can use the mpif90, mpicc, and mpicxx wrappers from HPE MPT. However, it is better to set the following variables before invoking mpif90, mpicc, and mpicxx to ensure nvfortran/nvc/nvc++ are used instead of gfortran/gcc/g++:

```
setenv MPIF90_F90 nvfotran
mpif90 [options] program.f

setenv MPICC_CC   nvc
mpicc  [options] program.c

setenv MPICXX_CXX nvc++
mpicxx [options] program.cc
```

TIP for Advanced Developers: Instead of using MPI, you can develop applications using the NVIDIA Collective Communications Library (NCCL) for multi-GPU and multi-node communication. Read this NVIDIA page to learn more.

Running a CUDA Application

The following environment variables may be useful during runtime:

- MPI_USE_CUDA: If using HPE MPT library for running a CUDA application across multiple GPUs or across multiple nodes, setting this variable will enable CUDA support so that the buffers on the GPUs can be used with sends, receives, collectives, and MPI RMA calls.
- CUDA_VISIBLE_DEVICES: Use this variable to control which GPUs your application uses. This is useful when you want to target a specific GPU or a set of GPUs for debugging or testing.

Sample PBS Script

```
#PBS -l select=1:ncpus=16:ngpus=4:mem=240GB:model=mil_a100
#PBS -l walltime=8:00:00
## If you request more than 1 chunk (i.e., select=2 or more),
## with resources which are scattered (i.e., not using the same physical nodes)
## and you want exclusive access to the vnodes assigned,
## uncomment the following place statement
##PBS -l place=scatter:excl
#PBS -q gpu_devel

module load nvhpc-nompi/23.7
module load mpi-hpe/mpt

cd $PBS_O_WORKDIR

setenv MPI_USE_CUDA 1

## if you want to run on two specific GPUs,
## for example, GPU0 and GPU2

setenv CUDA_VISIBLE_DEVICES 0,2
mpiexec -np 2 ./your_application.exe

## if you want to run on all four GPUs in the node

setenv CUDA_VISIBLE_DEVICES 0,1,2,3
mpiexec -np 4 ./your_application.exe
```

References

- Cabeus Configuration Details
- PBS Job Sharing on GPU Nodes
- Requesting GPU Resources
- Programming for NVIDIA GPUs

Cabeus GPU Nodes

Cabeus, named after a lunar crater in the south polar region of the Moon, is a cluster system that supports HPC and machine learning/artificial intelligence (ML/AI) applications that can benefit from the GPU technology.

GPU Hardware Summary

The node statistics are shown in two groups: nodes based on the x86_64 operating system (OS), and those based on the aarch64 OS.

Cabeus x86_64 GPU Nodes

- 12 racks
- 128 nodes
- 8,192 CPU cores + 1,769,472 double-precision GPU cores
- Total memory: 106 terabytes (65 TB from CPU host memory + 41 TB from GPU memory)
- Theoretical-double precision peak performance: 5.31 Petaflop/s (0.32 Pflop/s from CPUs + 4.99 Pflop/s from GPUs)

Cabeus aarch64 GPU Nodes

- 23 racks
- 538 nodes
- 38,736 CPU cores + 4,545,024 double-precision GPU cores
- Total memory: 258,240 GB (CPU) + 51,648 GB (GPU)
- Theoretical-double precision peak performance: 2,147 Teraflops (CPU) + 17,998 Tflops (GPU)

GPU Nodes by Model

Each GPU compute node includes one CPU socket as the host and one or four GPU cards as the coprocessors. These compute nodes are grouped by the CPU processor models as listed below:

- **model=mil_a100**

  128 nodes total. Each node contains one AMD Milan CPU socket and four NVIDIA Amper A100 GPU cards, where the CPU and GPUs are connected via PCI Express 4.0 bus.

  The Milan CPU socket used in the `mil_a100` nodes are AMD EPYC 7763.

- **model=gh200**

  538 nodes total. Each node contains one NVIDIA GH200 Grace Hopper Superchip, which consists of one Grace CPU (Neoverse V2 core with ARM v9.0-A architecture) and one H100 GPU, connected by a high-bandwidth, memory-coherent NVLink-C2C interconnect.

Cabeus Processor, Memory, and Network Subsystems Statistics

	Node Details	
	mil_a100	**gh200**
Architecture	Apollo 6500 Gen10 Plus	Supermicro SuperServer ARS-111GL-NHR
Total # of Nodes	128	538
Host name	cb[01-05,08-09]n[01-12], cb[06-07]n[01-10], cb[10-12]n[01-08]	cgh[1-3,6-15,17-19,21-23]n[01-26], cgh[4,20]n[01-12], cgh16n[01-20]
	CPU Host	
Processors Model	64-core EPYC 7763	72-core Grace (Neoverse V2)
# of Physical CPU Cores/Node	64	72
CPU-Clock	2.45 GHz	3.465 GHz
Maximum Double Precision Floating Point Operations per cycle per core	16	16
Total CPU Double Precision Flops/Node	2,508 GFlops	3.99 TFlops
Host Memory/Node	512 GB (8 x 64 GB DDR4)	480 GB (LPDDR5X)
	GPU	
GPU Architecture	Ampere	Hopper
Compute Capability**	cc80	cc90

Device Name	NVIDIA A100-SXM4-80GB	NVIDIA GH200 480GB
Clock Rate (Base/Boost)	1,275 MHz/1,410 MHz	(n/a)/1,980 MHz
# of coprocessors/Node	4	1
# of Streaming Multiprocessors (SMs)/coprocessor	108	132
# of Single Precision CUDA Cores/coprocessor	6,912	16,896
# of Double Precision CUDA Cores/coprocessor	3,456	8,448
# of Tensor Cores/coprocessor	432	528
Maximum double precision floating point operations per cycle per CUDA core	2	2
Total GPU Double Precision Flops/Coprocessor (Base/Boost) using Double Precision CUDA Cores	8.8 TFlops /9.7 TFlops	33.4 TFlops
Total GPU Double Precision Flops/Node (Base/Boost) using Double Precision CUDA Cores	35.2 TFlops/38.8 TFlops	33.4 TFlops
L2 Cache	40 MB	60 MB
Global Memory/coprocessor	80 GB (HBM2e)	96 GB (HBM3)
Memory Clock Rate	1,593 MHz	2,619 MHz
Memory Bus Width	5,120 bits	6,144 bits
Memory Bandwidth	2,039 GB/s (if using SXM form factor; 1,935 GB/s if using PCI form factor)	4,000 GB/s
GPU-to-GPU Communication	with NVLink 3.0: 600 GB/s	N/A
CPU-to-GPU Communication	PCIe 4.0 with 31.5 GB/s bandwidth for each direction	NVLink-C2C with 900 GB/s bi-directional
	Inter-node Network	
IB Device on Node	1 HDR 200 Gb 2-port card + 1 GbE 4-port card	NDR 200 Gb 1-port
	Local Disk	
SSD/node***	raw (2 x 1.92 TB SSD drives + 2 x 3.84 TB NVMe SSD) or (6 x 2.93 TB NVMe SSD)	960 GB Samsung NVMe M.2

** The compute capability identifies the features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the present GPU. Do not confuse the compute capability number with the CUDA version.

*** Not yet available for public usage, pending configuration decisions.

For more hardware details, access a GPU node through a PBS session and run one of the following commands:

For CPU info:

```
cat /proc/cpuinfo
```

For host memory info:

```
cat /proc/meminfo
```

Note: For the gh200 nodes with the unified memory feature,
the total memory shown in the output of /proc/meminfo is the sum
of the CPU memory and the GPU memory.

For GPU info:

```
/usr/bin/nvidia-smi -q
```

or load a nvhpc module and run the command pgaccelinfo or nvaccelinfo.

For example, for systems with x86_64 architecture:

```
module avail nvhpc
---------- /nasa/modulefiles/toss4 ----------

nvhpc-nompi/22.3 nvhpc-nompi/23.11 nvhpc/23.5 nvhpc/24.1
nvhpc-nompi/23.5 nvhpc-nompi/24.1 nvhpc/23.7 nvhpc/openmpi-3.1.5
nvhpc-nompi/23.7 nvhpc/22.3 nvhpc/23.11 nvhpc/openmpi-4.0.5
```

```
module load nvhpc/23.11
```

For example, for systems with aarch64 architecture:

```
module avail nvhpc
module load nvhpc/24.11

pgaccelinfo
or
nvaccelinfo
```

Running Jobs on Athena

To run a PBS job on the Athena compute nodes, submit the job from one of the Athena front-end systems (ATHFEs) to one of the queues managed by the PBS server pbs06a: devel, normal, long, and wide.

Note: You cannot submit jobs to run on Athena from a PFE.

Running Jobs on Aitken and Electra

To run a PBS job on the Aitken or Electra compute nodes, submit the job from one of the PFEs to one of the queues managed by the PBS server pbspl1.

Aitken Milan Nodes: You can submit batch jobs to run on Aitken's Milan nodes (mil_ait) either from a PFE or from one of the two Aitken Milan front-end systems (afe01, afe02). Interactive PBS jobs for Aitken Milan nodes must be submitted from the AFEs because currently there is no direct network connection to the PFEs.

Running Jobs on Endeavour

To run a PBS job on the shared memory systems Endeavour3/4, submit the job from a PFE to one of the queues managed by the PBS server pbspl4.

Running Jobs on Cabeus

To run a PBS job on:

- Milan A100 (mil_a100) submit the job from a Cabeus front end (cfe01 or cfe02) to one of the queues managed by the PBS server pbspl4.
- Grace Hopper nodes (gh200), submit your job from a Cabeus Grace Hopper front end (cghfe01 or cghfe02) to one of the queues served by the PBS server pbs05a.

PBS Queues

To view a complete list of all queues, including their defaults or limits on the number of cores, wall time, and priority, run theqstat command from an appropriate front-end system as follows:

```
qstat -Q
```

To view more detailed information about each queue, such as access control, defaults, limits on various resources, and status:

```
qstat -fQ queue_name
```

Brief summaries of the various queues are provided in the following sections.

Queues for Athena Users

You can run jobs on Athena using any of the following queues:

```
Queue   NCPUs/      Time/
name      max/def    max/def    pr
======= =====/=== ======/====== ===
devel     256/ 1    02:00/--      100
normal     --/ 1    08:00/--        0
long       --/ 1  120:00/--        0
wide      128/64   24:00/--       20
```

Queues for Aitken and Electra Users

You can run jobs on Aitken and Electra using any of the following queues:

```
Queue   NCPUs/      Time/
name      max/def    max/def    pr
======= =====/=== ======/====== ===
low        --/  8  04:00/ 00:30 -10
normal      --/  8  08:00/ 01:00   0
long       --/  8 120:00/ 01:00   0
debug       --/  8  02:00/ 00:30  15
devel      --/  1  02:00/    -- 101
```

The normal, long and low queues are for production work. The debug and devel queues have higher priority and are for debugging and

development work.

For the debug queue, each user is allowed a maximum of 2 running jobs using a total of up to 128 nodes. For thedevel queue, each user is allowed only 1 job (either queued or running).

Queues for Cabeus GPU Users

For information about queues for the Cabeus GPU nodes, seeRequesting GPU Resources.

See also:

- Preparing to Run on Cabeus Grace Hopper Nodes
- Preparing to Run on Cabeus A100 GPU Nodes

Queues for Endeavour Users

Jobs submitted from PFEs to the e_normal, e_long, e_debug, and e_low queues are routed to run on the Endeavour shared-memory system, which has its own PBS server.

See Endeavour PBS Server and Queue Structure for more information.

Queue for Lou Users

The ldan queue provides access to one of the Lou data analysis nodes (LDANs) for post-processing archive data on Lou.

The ldan queue is managed by the same PBS server as all of the Electra and Aitken queues and is accessible by submitting jobs from an LFE or PFE.

Each job on the ldan queue is limited to 1 node and 72 hours. Each user can have a maximum of 2 running jobs on this queue.

See Lou Data Analysis Nodes for more information.

Queues with Restricted Access

The vlong Queue

Access to the vlong queue requires authorization from the NAS User Services manager.

A maximum of 1024 nodes and 16 days are allowed on this queue.

To avoid losing all useful work in the event of a system or application failure, ensure that your application includes checkpoint and restart capability; this is especially important when using this queue.

The wide Queue

The wide queue is a high priority queue used for jobs that meet either one of the following conditions:

- The number of requested NCPUs (number of nodes * number of cores per node) exceeds 25% of the user's mission shares
- The number of nodes requested of a processor model exceeds 25% of the total number of nodes of that model. See the Electra and Aitken configuration pages for the total number of nodes for each model.

Users do not have direct access to thewide queue. Instead, a job that meets the above criteria is moved into thewide queue by NAS staff. Only 1 job in the wide queue is allowed among Electra and Aitken. When more than one job meets the criteria, preference is given to the job that has been waiting for longer time or whose owner does not have other jobs running.

Athena: You can submit jobs to the Athena wide queue (with minimum of 64 nodes and maximum of 128 nodes for up to 24 hours) without requesting approval from NAS staff.

Various Special Queues

Special queues are higher priority queues for time critical projects to meet important deadlines. Some examples of special queues are armd_spl, astro_ops, heomd_spl, kepler, smd1, smd2, smd_ops , and so on. Request for creation and access of a special queue is carefully reviewed by the mission directorate or by the program POC who authorizes the project's SBU allocation. The request is rarely granted unless the project's principal investigator demonstrates that access to a special queue is absolutely necessary.

Advanced Reservation Queues

An advanced reservation queue allows you to reserve resources for a certain period ahead of time. One circumstance in which the NAS User Services manager may authorize the use of a reservation queue is when a user plans to run an extra-large job, such as one exceeding 15,000 cores, which would never start running without a reservation.

A reservation queue is created by NAS staff for the user and is normally named with the letter R followed by a number.

To find the status of existing PBS advanced reservations, issue the pbs_rstat command on a PFE.

Note: Queues such as alphatst, ded_time, diags, dpr, idle, resize_test, and testing are used for staff testing purposes, and are not accessible by general users.

The Cabeus cluster comprises two types of NVIDIA GPU nodes, including nodes with x86_64-based CPUs and nodes with aarch64-based CPUs.

Note: You must have a Standard Billing Unit (SBU) allocation specifically for GPUs in order to use the GPU nodes.

Nodes with x86_64-Based CPUs

- 128 nodes each with one Milan CPU and four NVIDIA A100 GPUs (model=mil_a100)

These GPU nodes are served by PBS server pbspl4 and are accessible from the Cabeus front ends, cfe01 and cfe02.

The nodes are sharable by different jobs. Instead of each node being treated as one unit for exclusive access by a single job, each node is logically split into four *vnodes*.

If you want exclusive access to a node, request the use of all the resources in a node and specify the placement -lplace=scatter:excl. for example:

```
#PBS -lselect=1:ncpus=128:ngpus=4:mem=480GB:model=mil_a100
#PBS -lplace=scatter:excl
#PBS -q queue_name@pbspl4
```

where the use of @pbspl4 in the line #PBS -q is optional.

Note: There are 64 physical cores per mil_a100 node. However, PBS sees each mil_a100 node as having 128 logical cores due to hyperthreading enabled. If you specify :ncpus=64, PBS will assign 64 logical cores all from the first 32 physical cores. To get all 64 physical cores assigned to your job, use :ncpus=128.

Nodes with aarch64-based CPU

Cabeus contains 538 nodes with NVIDIA GH200 Superchips (model=gh200).

These nodes are accessible from the Cabeus Grace Hopper front ends, cghfe01 and cghfe02. PBS jobs are served by pbs05a, where user login is not allowed.

The gh200 nodes have only one GPU per node. A node is configured via PBS for exclusive use by a single job, and it is not necessary to explicitly request the amount of resources. For example, the following two PBS resource requests provide you with the same exclusive access to all resources of a whole node:

```
#PBS -lselect=1:model=gh200
or
#PBS -lselect=1:ncpus=72:ngpus=1:mem=460GB:model=gh200
#PBS -q queue_name@pbs05a
```

where the use of @pbs05a in the line #PBS -q is optional.

Requesting AMD Milan A100 GPU Nodes (mil_a100)

Each mil_a100 node contains only one socket; each socket is further split into four vnodes where eachmil_a100 vnode contains 1 GPU, 16 CPU physical cores, and 128 GB of memory.

The new queue structure utilizes the PBS primetime feature. A "nonprime-time" queue, np_gpu_wide, will run jobs only from 7:00 a.m. Thursday to 7:00 a.m. Saturday (Pacific Time). The gpu_normal and gpu_devel queues will run during both primetime and non-primetime (i.e., any time). You can submit jobs to any of the queues at any time during the week, but the non-prime time jobs will run only during their designated period.

The following PBS queues are enabled for use:

- **gpu_normal**
  - Maximum wall time: 24 hrs
  - Maximum node count: 32
- **gpu_devel**
  - Maximum wall time: 2 hrs
  - Maximum node count: 2
  - Maximum user jobs (queued or running): 1
  - Priority: 10
- **np_gpu_wide (non-primetime)**
  - Maximum wall time: 12 hrs
  - Maximum node count: 64
  - Minimum node count: 24
  - Maximum user jobs (queued or running): 2

- Priority: 20

Notes:

- Long running jobs need to implement checkpoint/restart if they cannot fit into the wall-time limit of these queues.
- The maximum amount of resources or certain settings for these queues may change from time to time. To get the current setting, run the command qstat -fQ *queue_name*. Note that the resources_max.nodect value refers to the number of physical nodes allowed per job.

In your job request, specify the number of CPU cores, the number of GPUs, and the amount of host memory you want for your job. The following example shows how to request exclusive usage of a whole node:

```
#PBS -lselect=1:ncpus=128:ngpus=4:mem=480GB:model=mil_a100
#PBS -lplace=scatter:excl
```

If ncpus, ngpus, or mem is not specified, the default values (ncpus=1, ngpus=1, mem=32GB) will be used.

Requesting NVIDIA GH200 GPU Nodes (gh200)

To run on the GH200 nodes, submit your PBS interactive or batch job from one of the two Grace Hopper front ends, cghfe01 or cghfe02. Use one of the following queues:

- **gpu_normal**
  - Maximum wall time: 24 hrs
  - Priority: Default (0)

- **gpu_devel**
  - Maximum wall time: 2 hrs
  - Maximum node count: 2
  - Maximum user jobs (queued or running): 1
  - Priority: 10

- **gpu_wide**
  - Maximum wall time: 12 hrs
  - Priority: 20
  - Wide is considered 85 or more nodes requested for a job.

- **gpu_long**
  - Maximum wall time: 120 hrs
  - Maximum node count: 8
  - Maximum user jobs (queued or running): 2
  - Priority: Default (0)

Note: Since each gh200 node is set up to not allow job sharing, the use of PBS placement such as -lplace=scatter:excl is not required.

TIP: The available queues will change from time to time. Use the command qstat -Q and qstat -fQ to find what queues have been created and enabled to be used by your group.

Charging for Use of the GPU Nodes

A separate SBU allocation exclusively for the GPU nodes is required. For information on requesting SBU allocation for the GPU nodes, see this article.

The SBU rates for using 1 GPU node for 1 hour are:

- mil_a100 with four A100s: 37.86
- gh200 with one H100: 18.34

Note: For the GPU nodes with aarch64-based CPU (model=gh200), your job will be charged for the SBU cost of a 1 whole node regardless of how the resources are requested, since these nodes are not set up for job sharing (each job is allocated with all of the resources in a whole node).

For the GPU nodes with x86_64 based CPUs, since a node is sharable by multiple jobs, your job may be charged for the SBU cost of 1 whole node or a fraction of the node depending on the resource request and the place statement in your PBS job. In addition, charging is based on the number of GPUs allocated to or excluded from use by other jobs, with no consideration of CPUs and memory. Below are three examples.

Example 1

This resource request will be charged for one-quarter of the mil_a100 SBU rate of 37.86, because only 1 GPU will be allocated to the job. In addition, it allows the unused CPU, GPU, and memory in the node to be used by other jobs:

```
#PBS -lselect=1:ncpus=1:ngpus=1:mem=60GB:model=mil_a100
#PBS -lplace=scatter:shared
```

Example 2

This job will be charged for 2 times the SBU rate of 37.86, because all four GPUs in each of the two mil_a100 nodes will be allocated and no other jobs will be able to share the resources:

```
#PBS -lselect=2:ncpus=1:ngpus=4:mem=60GB:model=mil_a100
#PBS -lplace=scatter:excl
```

Example 3

This job will be charged for one half (2 x 0.25) the SBU rate of 37.86, because only 1 GPU each from 2 different nodes will be allocated and the remaining 3 GPUs in each node can be used by other jobs:

```
#PBS -lselect=2:ncpus=1:ngpus=1:mem=60GB:model=mil_a100
#PBS -lplace=scatter:excl
```

Checking Job Status

To check the status for all of your jobs submitted to a particular queue, run the qstat command. The following examples use the queue gpu_normal served by the pbspl4 PBS server. For jobs submitted to queues served by the pbs05a PBS server, change the front end from cfe to cghfe01 or cghfe02, the PBS server from pbspl4 to pbs05a, and use an appropriate queue name.

```
cfe% qstat gpu_normal@pbspl4 -u your_username
```

You can also check the status of a single job. For example, to check the status of job 140101 on pbspbl4:

```
cfe% qstat 140101.pbspl4.nas.nasa.gov
```

For single-node I/O that uses memory-based temporary filesystem, we recommend using a job-based temporary directory, $TMPDIR, instead of the node-wide /tmp directory. This recommendation is particularly important for jobs running on the NAS GPU nodes with x86_64-based CPUs (mil_a100) where node sharing is enabled.

Background

All of the NAS compute nodes—including the CPU nodes in Electra, Aitken, and the GPU nodes in Cabeus—are configured to allow up to 50% of the node's main memory for use as a temporary filesystem (/tmp) to achieve faster single-node I/O. However, using the /tmp directory is not feasible for multi-node I/O, as /tmp is a filesystem local to a single node.

Many users use the /tmp directory on the CPU nodes to speed up their jobs' I/O. When these jobs end, data under /tmp are not always removed by the application or user. In order to provide a clean /tmp directory for a new job that uses the same node, the PBS prologue will first unmount /tmp to clean up leftover data from the previous job, then will remount it for the new job.

On the GPU nodes where node sharing is enabled, the resources on each node (including /tmp) can be shared by multiple jobs. Therefore, the process of cleaning up /tmp depends on whether a new job is the only job on a node or whether there are other jobs running on the node:

- If a new job starting is the only job on a node, the PBS prologue will use the same process as on CPU nodes by unmounting and remounting /tmp.
- If there are other jobs currently running on the same node, the prologue will not unmount /tmp, because doing so would remove the data needed by the other running jobs. Instead, the prologue will look for /tmp/pbs.*jobID* directories that do *not* have a running job, and delete those directories if they exist. The $TMPDIR path is /var/tmp/pbs.*jobid*.

Therefore, to ease the removal of user data from old jobs, you should use the $TMPDIR directory instead of writing directly to /tmp.

Using the $TMPDIR Directory

Note: For some NAS systems, /var/tmp is a symlink of /tmp. For the Milan A100 (mil_a100) nodes, /var/tmp and /tmp are two different directories, but they both use the same 50% main memory pool of the nodes.

How much $TMPDIR space can a job use?

On the GPU nodes, $TMPDIR cannot use more than the amount of CPU host memory requested by a job. For example, in the following PBS job request, the 16 GB of CPU host memory requested by the job will be shared by the memory usage of the application; buffer cache usage of the application (if buffered I/O is enabled); and the usage for storing data under /tmp or $TMPDIR:

```
#PBS -lselect=1:ncpus=10:ngpus=1:mem=16GB:model=mil_a100
```

Therefore, the maximum space available for $TMPDIR is 16 GB in this example.

In addition, the combined $TMPDIR usage by all jobs running on the same node cannot exceed 50% of the node's main memory. If another job running on the same node as your job already has data stored in the job's $TMPDIR with an amount equal to the size of 50% of the node's main memory, you will encounter a "No space left on device" error when you attempt to write data into your job's $TMPDIR.

Sample Script for Using $TMPDIR

A job's resources (CPU, GPU, and memory) are accessible only during the lifetime of the job. To use $TMPDIR for I/O, you must copy data into $TMPDIR before the data are needed by your application. Any generated data needed for post-processing or restarting the next job must be copied to the $HOME or Lustre /nobackup directory before the current job is terminated. You can accomplish this by using the following sample PBS script:

```
#PBS -lselect=1:ncpus=16:ngpus=1:mem=16GB:model=mil_a100
#PBS -q gpu_devel

module load ....

cd $TMPDIR

cp /nobackup/username/project/input .

mpiexec -np ....

cp output /nobackup/username/project
#End of script
```

PBS on Electra

To help you prepare for running jobs on Electra's Skylake compute nodes, this short user guide includes information on the general configuration of Skylake nodes, compiling your code, running PBS jobs, and checking allocation usage.

Overview of Electra Skylake Nodes

Electra includes 2,196 Skylake nodes, which are partitioned into sixteen physical racks. Each node contains two 20-core Xeon Gold 6148 sockets (2.4 GHz) and 192 GB of memory. The nodes are connected to the Electra InfiniBand network (ib0 and ib1) via four-lane Enhanced Data Rate (4X EDR) devices and switches for internode communication. The ib1 fabric is used mainly for I/O and is connected to the NAS Lustre filesystems. In addition, Electra has access to the NAS compute resources home filesystems, front-end systems (PFEs), and PBS server (pbspl1). You can access Electra only through PBS jobs submitted from the PFEs.

Compiling Your Code For Skylake Nodes

The Skylake processors include the Advanced Vector Extensions 512 (AVX-512). Intel AVX-512 optimizations are included in Intel compiler version 16.0 and later versions. We recommend that you test the latest Intel compiler, which may include better optimizations for AVX-512.

For Skylake-specific optimizations, use the compiler option -xCORE-AVX512.

If you want a single executable that will run on any of the Electra processor types, with suitable optimization to be determined at runtime, you can compile your application using the option:
-O3 -axCORE-AVX512,CORE-AVX2 -xAVX.

Note: The use of either of these options, -xCORE-AVX512 or -axCORE-AVX512, could either improve or degrade performance of your code. Be sure to check performance with and without these flags before using them for production runs.

Running PBS Jobs on Electra Skylake Nodes

To request Electra Skylake nodes, use :model=sky_ele in your PBS script, as shown in the example below.

Important: Because MPT 2.15 and earlier versions do not support the ConnectX-5 host channel adapters (HCAs), the environment variables MPI_IB_XRC and MPI_XPMEM_ENABLED have been disabled for jobs running on Skylake. If your MPI applications perform significant MPI collective operations and rely on having these two variables enabled to get good performance, use MPT 2.16 or newer versions. To use the NAS-recommended MPT version, use the command:
module load mpi-hpe/mpt

Sample PBS Script For Electra Skylake Nodes

```
#PBS -l select=10:ncpus=40:mpiprocs=40:model=sky_ele
#PBS -l walltime=8:00:00
#PBS -q normal

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

cd $PBS_O_WORKDIR

mpiexec -np 400 ./a.out
```

Checking Allocation Usage For Electra Skylake Jobs

To track allocation usage for jobs running on Electra Skylake nodes, run:

```
acct_query -c electra_S
```

Examples:

- To track the total SBUs usage for one of your GIDs (for example, s0001) since 9/30/17:

  ```
  % acct_query -c electra_S -ps0001 -b 9/30/17
  ```

- To track the SBUs usage for each of your jobs run today:

  ```
  % acct_query -c electra_S -olow
  ```

For more information, see Skylake Processors.

To help you prepare for running jobs on Electra's Broadwell compute nodes, this short review includes information on the general configuration, compiling your code, running PBS jobs, and checking allocation usage.

Overview of Electra Broadwell Nodes

Electra is housed in a module outside of the primary NAS facility. The system has 16 Broadwell racks, each containing 72 nodes. Each of the 72 nodes contains two 14-core E5-2680v4 (2.4 GHz) processor chips and 128 GB of memory.

Electra's InfiniBand fabric (ib1) is used mainly for I/O and is connected to the NAS Lustre filesystems. In addition, Electra and all other NAS compute systems share the same home filesystems. It also share the front-end systems (PFE) and PBS server (pbspl1) with some NAS compute nodes such as sky_ele, sky_el, and rom_ait. You can access Electra only through PBS jobs submitted from the PFEs.

Usage of Electra Broadwell nodes is charged to your project's HECC Standard Billing Unit (SBU) allocation.

Compiling Your Code For Broadwell Nodes

Like the Haswell processors, the Broadwell processors support the Advanced Vector Extensions 2 (AVX2) instructions, in addition to AVX (introduced with Sandy Bridge processors), SSE4.2 (introduced with Nehalem processors), and earlier generations of SSE.

AVX2 supports floating point fused multiply-add, integer vector instructions extended to 256-bit, and vectorization gather support, among other features. Your application may be able to take advantage of AVX2, however, not all applications can make effective use of this set of instructions. We recommend that you use the latest Intel compiler, by using the command module load comp-intel and experiment with the following sets of compiler options before making production runs on the Broadwell nodes:

- -O*2* -xCORE-AVX2
- -O*3* -xCORE-AVX2

These compiler options generate an executable that is optimized for running on Broadwell and Haswell, but cannot run on any of the older processor types. If you prefer to generate a single executable that will run on many Intel processor types (Sandy Bridge, Ivy Bridge, Haswell, Broadwell, Skylake, and Cascade Lake), try using one of the following sets of compiler options:

- -O*2* (or -O*3*)
- -O*2* (or -O*3*) -axCORE-AVX512,CORE-AVX2 -xAVX

The use of -axCORE-AVX512,CORE-AVX2 allows the compiler to generate multiple code paths with suitable optimization to be determined at run time.

If your application does not show performance improvements with -xCORE-AVX2 or -axCORE-AVX512,CORE-AVX2 -xAVX (as compared with just -O2 or -O3) when running on Broadwell nodes, it is more advantageous to use the executable built with just -o2 or -o3 for production runs.

TIP: You can add the compiler options -ip or -ipo to instruct the compiler to look for ways to better optimize and/or vectorize your code. Also, to generate a report on how well your code is vectorized, add the compiler flag -vec-report2.

Notes:

- If you have an MPI code, we strongly recommend that you load the mpi-hpe/mpt module, which always points to the NAS recommended MPT version.
- Ensure your jobs run correctly on Broadwell nodes before you start production work.

Running PBS Jobs on Broadwell Nodes

To request Broadwell nodes, use model=bro_ele in your PBS script.

```
#PBS -l select=10:ncpus=28:mpiprocs=28:model=bro_ele
#PBS -l site=static_broadwell
```

Sample PBS Script For Broadwell Nodes

```
#PBS -l select=10:ncpus=28:mpiprocs=28:model=bro_ele
#PBS -l walltime=8:00:00
#PBS -q normal
module load comp-intel/2020.4.304 mpi-hpe/mpt
cd $PBS_O_WORKDIR
mpiexec -np 280 ./a.out
```

For more information about Broadwell nodes, see:

- Electra Configuration Details
- Broadwell Processors

Running Jobs Before Dedicated Time

The PBS batch scheduler supports a feature called shrink-to-fit (STF). This feature allows you to specify a range of acceptable wall times for a job, so that PBS can run the job sooner than it might otherwise. STF is particularly helpful when scheduling jobs before an upcoming dedicated time.

For example, suppose your typical job requires 5 days of wall time. If there are fewer than 5 days before the start of dedicated time, the job won't run until after dedicated time. However, if you know that your job can do enough useful work running for 3 days or longer, you can submit it in the following way:

qsub -l min_walltime=72:00:00,max_walltime=120:00:00 *job_script*

When PBS attempts to run your job, it will initially look for a time slot of 5 days. When no such time slot is found between now and the dedicated time, it will look for shorter and shorter time slots, down to the minimum wall time of 3 days.

If you have an existing job that is still queued, you can use the qalter command to add these min_walltime and max_walltime attributes:

qalter -l min_walltime=*hh:mm:ss*,max_walltime=*hh:mm:ss job_id*

You can also use the qalter command to change the wall time:

qalter -l walltime=*hh:mm:ss job_id*

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

If you have any questions or problems, please contact the NAS Control Room at (800) 331-8737, (650) 604-4444, or by email at support@nas.nasa.gov.

Checking the Time Remaining in a PBS Job from a Fortran Code

During job execution, it can be useful to find out the amount of time remaining for your PBS job. This allows you to decide if you want to gracefully dump restart files and exit before PBS kills the job.

If you have an MPI code, you can call MPI_WTIME and see if the elapsed wall time has exceeded some threshold in order to decide if the code should go into the shutdown phase.

For example:

```
include "mpif.h"

real (kind=8) :: begin_time, end_time

begin_time=MPI_WTIME()
do work
end_time = MPI_WTIME()

if (end_time - begin_time > XXXXX) then
   go to shutdown
endif
```

In addition, the following library has been made available on NAS compute systems for the same purpose:

```
/u/scicon/tools/lib/pbs_time_left.a
```

To use this library in your Fortran code, you need to:

1. Modify your Fortran code to define an external subroutine and an integer*8 variable

   ```
   external pbs_time_left
   integer*8 seconds_left
   ```

2. Call the subroutine in the relevant code segment where you want the check to be performed

   ```
   call pbs_time_left(seconds_left)
   print*,"Seconds remaining in PBS job:",seconds_left
   ```

   Note: The return value from pbs_time_left is only accurate to within a minute or two.

3. Compile your modified code and link with the above library using, for example:

   ```
   LDFLAGS=/u/scicon/tools/lib/pbs_time_left.a
   ```

Many jobs running across multiple nodes perform setup and pre-processing tasks on the first node, then launch computations across many nodes, and finish with post-processing and cleanup steps running only on the first node. The post-processing and cleanup steps may take minutes or even hours, during which the remaining nodes in the job are idle. You can use the pbs_release_nodes command to release these idle nodes so that you are not charged for them, and they can be made available to other jobs.

Some users have already taken steps to minimize the number of idle nodes by separating their workflow into multiple jobs of appropriate sizes. The downside to this approach is additional time waiting in the queue for these multiple jobs. Using pbs_release_nodes may allow these jobs to be combined, reducing the amount of wait time in the queue.

Node Tracking and Job Accounting

When you use the pbs_release_nodes command, the released nodes are removed from the contents of $PBS_NODEFILE. The job accounting system tracks when the nodes are released, and adjusts the amount of SBUs charged accordingly.

Running the pbs_release_nodes Command

You can either include the pbs_release_nodes command in your job script or you can run it interactively, for example, on a PFE. If you run it in your job script, you can use the $PBS_JOBID environment variable to supply the *job_id*. You can use pbs_release_nodes multiple times in a single job.

There are two ways to use the pbs_release_nodes command. In both cases, the first node is retained.

- **Keep the first node and release all other nodes**
  If you want to retain the first node only, run the command as follows to release the remaining nodes:

  pfe21% pbs_release_nodes -j *job_id* -a

- **Release one or more specific nodes**
  If you want to retain one or more nodes in addition to the first node, run the command as follows to release specific idle nodes:

  pfe21% pbs_release_nodes -j *job_id node1 node2 ... nodeN*

To find out which nodes are being used by a job before you run the pbs_release_nodes command, use one of the following methods:

- **For interactive jobs:** Run qstat -f *job_id*. In the output, look for the nodes listed under the exec_vnode job attribute.
- **For job scripts:** Run cat $PBS_NODEFILE. The output provides a list of the nodes in use.

Note: $PBS_NODEFILE may have duplicate entries for nodes, depending on how resources were requested. See **man pbs_resources** for details.

Sample Job Script with pbs_release_nodes Command

The following sample job script keeps the first node and releases all other nodes before running post-processing steps:

```
#PBS -S /bin/csh
#PBS -N cfd
#PBS -l select=4:ncpus=8:mpiprocs=8:model=sky_ele
#PBS -l walltime=4:00:00
#PBS -j oe
#PBS -W group_list=a0801
#PBS -m e
module load comp-intel/
module load mpi-hpe/mpt
cd $PBS_O_WORKDIR
mpiexec dplace -s1 -c 16-23 ./grinder < run_input > intermediate_output
# Before executing some post-processing steps that will run only on the
# first node of the job you can release all other nodes
pbs_release_nodes -j $PBS_JOBID -a
# Now run post-processing steps
./post_process < intermediate_output > final_output
# -end of script-
```

PBS Resource Request Examples

When you request PBS resources for your job, you must specify one or more Aitken or Electra node model (processor) types, so it is helpful to keep this information in mind:

- Usage charges on each HECC node type are based on a common Standard Billing Unit (SBU).
- The total physical memory of each node type varies from 128 GB to 512 GB.

The following table provides the SBU rate and memory for each Aitken and Electra node type:

Processor Type	SBU Rate/Node	Memory/Node
Milan	4.38 (128 cores/node)	512 GB
Rome	4.06 (128 cores/node)	512 GB
Cascade Lake	1.64 (40 cores/node)	192 GB
Skylake	1.59 (40 cores/node)	192 GB
Broadwell	1.00 (28 cores/node)	128 GB

Note: The amount of memory available to a PBS job is less than the node's total physical memory, because the system kernel can use up to 4 GB of memory in each node.

Checking Available Nodes Before Requesting Resources

If your application can run on any of the Aitken or Electra processor types, you might be able to reduce wait time by requesting a type that has more free nodes (nodes currently unoccupied by other running jobs).

Before you submit your job, you can find out the current numbers of used and free nodes for each processor type by running the node_stats.sh script, which is accessible by loading the scicon/cli_tools modulefile. For example:

```
% module load scicon/cli_tools
% node_stats.sh
```

You can also identify which processor models are in use for each job currently running on HECC systems. Run the following command, and check the Model field in the output:

```
% qstat -a -W o=+model
```

Resource Request Examples

As shown in these examples, use the model=[bro_ele,sky_ele,cas_ait,rom_ait, mil_ait] attribute to request the node model type(s) for your job.

The default operating system for all model types is TOSS (Tri-Lab Operating System Stack). For all model types, the default is aoe=toss4:aoe=toss4 to your PBS resource request.)

Example 1: Basic Resource Request

Each of the following sample command lines requests a single node model type for a 128-process job:

```
#PBS -l select=5:ncpus=28:model=bro_ele
#to run all 28 cores on each of the 5 Broadwell nodes
#(12 cores in the 5th node will go unused)

#PBS -l select=4:ncpus=40:model=sky_ele
#to run all 40 cores on each of the 4 Skylake nodes
#(32 cores in the 4th node will go unused)

#PBS -l select=4:ncpus=40:model=cas_ait
#to run all 40 cores on each of the 4 Cascade Lake nodes
#(32 cores in the 4th node will go unused)

#PBS -l select=1:ncpus=128:model=rom_ait
#to run all 128 cores on 1 Rome node

#PBS -l select=1:ncpus=128:model=mil_ait
#to run all 128 cores on 1 Milan node
```

You can specify both the queue type -q normal, debug, long, low, or devel and the processor type. For example:

```
#PBS -q normal
#PBS -l select=128:ncpus=128:model=mil_ait
```

Example 2: Requesting Different ncpus in a Multi-Node Job

For a multi-node PBS job, the number of CPUs `ncpus` used in each node can be different. This feature is useful for jobs that need more memory for some processes, but less for other processes. You can request resources in "chunks" for a job with varying `ncpus` per node.

This example shows a request of two resource chunks. In the first chunk, one CPU in one Broadwell node is requested, and in the second chunk, 28 CPUs in each of three other Broadwell nodes are requested:

```
#PBS -l select=1:ncpus=1:model=bro_ele+3:ncpus=8:model=bro_ele
```

Example 3: Requesting Multiple Node Model Types

A PBS job can run across more than one node model type if the nodes are all from Electra or all from Aitken, but not across the systems. This can be useful in two scenarios:

- When you cannot find enough free nodes within one model for your job
- When some of your processes need more memory while others need much less

You can accomplish this by specifying the resources in chunks, with one chunk requesting one node model type and another chunk requesting a different type.

This example shows a request for one Skylake node (which provides ~185 GB of memory per node) and three Broadwell nodes (which provide ~120 GB of memory per node):

```
#PBS -lplace=scatter:excl:group=model

#PBS -lselect=1:ncpus=24:mpiprocs=24:model=sky_ele+3:ncpus=28:mpiprocs=28:model=bro_ele
```

Running "At-Scale" Jobs in the sky_wide Queue

TEMPORARY NOTICE: The sky_wide queue on Electra's Skylake nodes has been disabled until further notice in order to process high-priority wide jobs.

The sky_wide queue is intended to help you run very wide ("at scale") jobs on Electra's Skylake nodes by ensuring that at least once a week, the jobs in the queue will have enough available nodes to run. Job requests submitted to the sky_wide queue must specify between 576 and 1,152 Skylake nodes (between 25% and 50% of the total number of Skylake nodes, which is currently 2,304). The maximum wall time is currently 5 days.

Submitting Your Job to the sky_wide Queue

You can submit jobs to the queue each week. Jobs submitted to the queue by Thursday at 5:00 p.m. (Pacific Time) are scheduled to run starting the following Monday at 10:00 a.m. PBS runs the jobs one at a time, prioritized by job size; wider jobs are run before those requesting fewer nodes.

When it is your job's turn to run, a reservation is created for the number of nodes requested, plus a few extra nodes. The job is then moved from the sky_wide queue into the reservation's queue, where it starts running.

If the job fails early, you can submit a replacement job to the reservation queue without losing your place in line. Eventually, the queue will go idle or the reservation will expire, and the next job will be selected.

Important: Jobs submitted to the sky_wide queue after Thursday at 5:00 p.m. (Pacific Time) will not be scheduled to run the following Monday; rather, they will carry over to the next week.

Sample Job Submission Script

The following sample script includes several instructions. Each line is described below the script.

```
#PBS -q sky_wide
#PBS -l select=NNN:model=sky_ele:ncpus=nn:aoe=sles12
#PBS -l max_walltime="24:00:00",min_walltime="5:30:00"
#PBS -m abe
#PBS -l site="extra_nodes=4:extra_time=20:idle_timeout=20"
```

Job Submission

The first two lines request the sky_wide queue and submit your job:

```
#PBS -q sky_wide
#PBS -l select=NNN:model=sky_ele:ncpus=nn:aoe=sles12
```

Wall Time Request

This line specifies how much wall time the job needs. Keep in mind that if the job fails, you can submit a replacement job, but its wall time must fit within the time remaining from the original job. By specifying a maximum and minimum useful wall time in the original job script, you can use the same script, unedited, for the replacement job. The sample script requests a total of 24 hours of time, and any replacement job will not run unless at least 5.5 hours of wall time remain:

```
#PBS -l max_walltime="24:00:00",min_walltime="5:30:00"
```

Note: This technique, a PBS feature known as shrink-to-fit, is described in the article Running Jobs Before Dedicated Time.

Notification

If there are several jobs in the sky_wide queue, your job might not be the first to run. You can tell PBS to send you an email notification when your job starts, ends, or aborts, so you'll know whether you need to submit a replacement job:

```
#PBS -m abe
```

Additional Options

PBS automatically adds a few extra nodes to the job reservation, beyond the number of nodes in the request, in case nodes fail during the run. That way, a replacement job will still have enough reserved nodes. The default is 12 nodes. To specify a different value, use site=extra_nodes=nn:

```
#PBS -l site=extra_nodes=4
```

Recommendation: There are four Skylake nodes per blade. Because it is possible for a failing node to take down the whole blade, it makes sense to set aside extra nodes in multiples of four.

PBS also adds extra time to each job reservation. The default is 15 minutes. To specify different value, use site=extra_time=nn:

```
#PBS -l site=extra_time=20
```

Once your per-job reservation starts, PBS monitors the reservation. If the reservation queue is idle for a set number of minutes, the reservation is terminated early. The default is 30 minutes. To specify a different value, use site=idle_timeout=nn:

```
#PBS -l site=idle_timeout=20
```

To specify multiple site options in one line, separate them by colons (":"), as shown in the sample script:

```
#PBS -l site="extra_nodes=4:extra_time=20:idle_timeout=20"
```

Note: When you consider these requests, keep in mind that your account is billed for the full number of nodes and for the actual duration of the job reservation.

Checking Status of Wide Jobs

Run the pbs_scale_jobs command, which uses qstat to display the status of jobs submitted to the sky_wide queue. For example:

```
$ pbs_scale_jobs
                                        Req'd    Elap
JobID       User     Queue    Jobname    TSK Nds wallt S wallt Eff
----------- -------- -------- --------- ----- --- ----- - ----- ---
3827.testpbs zsmith   R3853    testing3 24000 600 10:15 Q 02:26  --
3792.testpbs zsmith   sky_wide testing1 23040 576 00:15 Q 03:11  --
3808.testpbs zsmith   sky_wide testing2 23040 576 00:15 Q 03:11  --

== Jobs to consider next cycle ==
                                        Req'd    Elap
JobID       User     Queue    Jobname    TSK Nds wallt S wallt Eff
----------- -------- -------- --------- ----- --- ----- - ----- ---
4001.testpbs zsmith   sky_wide testing5 23040 576 00:15 Q 01:10  --
4002.testpbs zsmith   sky_wide testing6 23040 576 00:15 Q 01:10  --
```

In this example, the jobs named testing1-3 were submitted before the cutoff time. Jobs testing5 and testing6 were submitted too late for this week's run and will be considered next week.

The testing3 job has been moved to its run queue (R3853, in this example). Note that although testing3 was submitted after testing1 and 2, the PBS scheduler runs it first because it uses more nodes than the other jobs.

Submitting Replacement Jobs

Replacement jobs are submitted to the reservation queue associated with the original job, rather than to the sky_wide queue. This queue will have a name like R12345, which changes for each reservation.

However, there is an easier way to submit a replacement job to the reservation queue. You can use the queue's alias, which takes the form NAS_job_*username* (where *username* is your NAS user name).

Also, instead of editing your qsub script to change the queue name when you submit the replacement, you can use the original job script, but override the queue on the command line. For example:

```
qsub -q NAS_job_zsmith job_script
```

Running Jobs on Athena

To run a PBS job on the Athena compute nodes, submit the job from one of the Athena front-end systems (ATHFEs) to one of the queues managed by the PBS server pbs06a: devel, normal, long, and wide.

Note: You cannot submit jobs to run on Athena from a PFE.

Running Jobs on Aitken and Electra

To run a PBS job on the Aitken or Electra compute nodes, submit the job from one of the PFEs to one of the queues managed by the PBS server pbspl1.

Aitken Milan Nodes: You can submit batch jobs to run on Aitken's Milan nodes (mil_ait) either from a PFE or from one of the two Aitken Milan front-end systems (afe01, afe02). Interactive PBS jobs for Aitken Milan nodes must be submitted from the AFEs because currently there is no direct network connection to the PFEs.

Running Jobs on Endeavour

To run a PBS job on the shared memory systems Endeavour3/4, submit the job from a PFE to one of the queues managed by the PBS server pbspl4.

Running Jobs on Cabeus

To run a PBS job on:

- Milan A100 (mil_a100) submit the job from a Cabeus front end (cfe01 or cfe02) to one of the queues managed by the PBS server pbspl4.
- Grace Hopper nodes (gh200), submit your job from a Cabeus Grace Hopper front end (cghfe01 or cghfe02) to one of the queues served by the PBS server pbs05a.

PBS Queues

To view a complete list of all queues, including their defaults or limits on the number of cores, wall time, and priority, run the qstat command from an appropriate front-end system as follows:

```
qstat -Q
```

To view more detailed information about each queue, such as access control, defaults, limits on various resources, and status:

```
qstat -fQ queue_name
```

Brief summaries of the various queues are provided in the following sections.

Queues for Athena Users

You can run jobs on Athena using any of the following queues:

```
Queue   NCPUs/      Time/
name      max/def    max/def    pr
======= =====/=== ======/====== ===
devel     256/ 1    02:00/--      100
normal     --/ 1    08:00/--        0
long       --/ 1  120:00/--        0
wide      128/64   24:00/--       20
```

Queues for Aitken and Electra Users

You can run jobs on Aitken and Electra using any of the following queues:

```
Queue   NCPUs/      Time/
name      max/def    max/def    pr
======= =====/=== ======/====== ===
low        --/  8  04:00/ 00:30 -10
normal      --/  8  08:00/ 01:00   0
long       --/  8 120:00/ 01:00   0
debug       --/  8  02:00/ 00:30  15
devel      --/  1  02:00/    -- 101
```

The normal, long and low queues are for production work. The debug and devel queues have higher priority and are for debugging and

development work.

For the debug queue, each user is allowed a maximum of 2 running jobs using a total of up to 128 nodes. For thedevel queue, each user is allowed only 1 job (either queued or running).

Queues for Cabeus GPU Users

For information about queues for the Cabeus GPU nodes, seeRequesting GPU Resources.

See also:

- Preparing to Run on Cabeus Grace Hopper Nodes
- Preparing to Run on Cabeus A100 GPU Nodes

Queues for Endeavour Users

Jobs submitted from PFEs to the e_normal, e_long, e_debug, and e_low queues are routed to run on the Endeavour shared-memory system, which has its own PBS server.

See Endeavour PBS Server and Queue Structure for more information.

Queue for Lou Users

The ldan queue provides access to one of the Lou data analysis nodes (LDANs) for post-processing archive data on Lou.

The ldan queue is managed by the same PBS server as all of the Electra and Aitken queues and is accessible by submitting jobs from an LFE or PFE.

Each job on the ldan queue is limited to 1 node and 72 hours. Each user can have a maximum of 2 running jobs on this queue.

See Lou Data Analysis Nodes for more information.

Queues with Restricted Access

The vlong Queue

Access to the vlong queue requires authorization from the NAS User Services manager.

A maximum of 1024 nodes and 16 days are allowed on this queue.

To avoid losing all useful work in the event of a system or application failure, ensure that your application includes checkpoint and restart capability; this is especially important when using this queue.

The wide Queue

The wide queue is a high priority queue used for jobs that meet either one of the following conditions:

- The number of requested NCPUs (number of nodes * number of cores per node) exceeds 25% of the user's mission shares
- The number of nodes requested of a processor model exceeds 25% of the total number of nodes of that model. See the Electra and Aitken configuration pages for the total number of nodes for each model.

Users do not have direct access to thewide queue. Instead, a job that meets the above criteria is moved into thewide queue by NAS staff. Only 1 job in the wide queue is allowed among Electra and Aitken. When more than one job meets the criteria, preference is given to the job that has been waiting for longer time or whose owner does not have other jobs running.

Athena: You can submit jobs to the Athena wide queue (with minimum of 64 nodes and maximum of 128 nodes for up to 24 hours) without requesting approval from NAS staff.

Various Special Queues

Special queues are higher priority queues for time critical projects to meet important deadlines. Some examples of special queues are armd_spl, astro_ops, heomd_spl, kepler, smd1, smd2, smd_ops , and so on. Request for creation and access of a special queue is carefully reviewed by the mission directorate or by the program POC who authorizes the project's SBU allocation. The request is rarely granted unless the project's principal investigator demonstrates that access to a special queue is absolutely necessary.

Advanced Reservation Queues

An advanced reservation queue allows you to reserve resources for a certain period ahead of time. One circumstance in which the NAS User Services manager may authorize the use of a reservation queue is when a user plans to run an extra-large job, such as one exceeding 15,000 cores, which would never start running without a reservation.

A reservation queue is created by NAS staff for the user and is normally named with the letter R followed by a number.

To find the status of existing PBS advanced reservations, issue the pbs_rstat command on a PFE.

Note: Queues such as alphatst, ded_time, diags, dpr, idle, resize_test, and testing are used for staff testing purposes, and are not accessible by general users.

Electra and Aitken devel Queue

NAS provides a special queue on Electra and Aitken, called the devel queue, that is designated for development work. Using this queue for development work provides faster turnaround than using other, non-designated queues.

Note: The devel queue should not be used for production work. Misuse of the devel queue may result in loss of access to the queue. See devel Queue Policies for more information.

Flexible Node Types

From the devel queue, you can access nodes of all Aitken and Electra processor types.

Requesting Resources

Because the devel queue does not have a default processor type, you must specify the processor type by including the :model=*xxx* attribute in your PBS resource request (where *xxx* is bro_ele, sky_ele, cas_ait or rom_ait. Note that mil_ait is not allowed to be used in devel queue.) For example, to request a Rome node:

```
#PBS -q devel
#PBS -l select=1:ncpus=128:model=rom_ait
```

Submitting Jobs

To submit your job to the devel queue, run the qsub command:

```
pfe21% qsub -q devel job_script
1234.pbspl1.nas.nasa.gov
```

To check the status of a job submitted to the devel queue, run the qstat command:

```
pfe21% qstat -u your_username devel
```

Determining Available Resources

In order to obtain the fastest possible start time, run the node_stats.sh script to view the number and types of nodes currently available in the devel queue before you submit your job. For example:

```
pfe21% node_stats.sh
```

devel Queue Policy

When using the devel queue, be aware of the following policy:

- Each user is allowed only one job on the devel queue, either running or queued. You cannot stack up a list of queued jobs or create self-submitting jobs.
- The maximum wall-clock time of a job is always 2 hours.
- A job may request no more than 512 nodes, either of a single processor/node type or a mixture of different processor/node types.
- The PBS scheduler predominately considers which users have used the most devel queue resources to decide which jobs to run next.
- devel queue usage is applied to your mission share.

The devel queue is intended for developing and testing code and cases to prepare them for production runs in other queues, such as the normal and long queues. It should not be used for performing parameter studies or for breaking a long production run into multiple consecutive runs that each fit within the 2-hour limit.

Disregarding these devel queue policies and guidelines may result in loss of access to the queue.

PBS on Endeavour

The Endeavour supercomputer comprises two nodes, Endeavour3 and Endeavour4. Each of the two systems includes 896 cores and six terabytes (TB) of global shared memory. See Endeavour Configuration Details for more hardware information.

Endeavour 3 & 4 are accessible through the PFEs and are served by PBS server pbspl4. They have access to the NAS compute resources home filesystem and NAS Lustre nobackup filesystems.

Update to Endeavour Support Model: Due to the age of Endeavour's hardware, we are lowering the priority of those nodes in our monthly patching work. This means that Endeavour may not always be returned to service at the same time as newer systems, and the patch windows may need to be extended to account for additional time needed to bring the nodes online.

Operating System

Endeavour 3 & 4 are running the Red Hat Enterprise Linux-based Tri-Lab Operating System Stack (TOSS4).

Connecting to Endeavour

To access the systems, submit PBS jobs from the PFEs with the server name pbspl4 included in the qsub command line or PBS script. You cannot log directly into Endeavour3 and 4 from the PFEs unless you have a PBS job running there.

Accessing Your Data

Use your NAS home filesystems (/u/*username*) and your Lustre filesystem (/nobackup) for Endeavour.

For more information on using Lustre filesystems, see Lustre Basics and Lustre Best Practices.

Compiling and Running Your Code

There are no default modules for compilers, Message Passing Interface (MPI) components, or math libraries. To use these modules, you must first load them.

OpenMP Applications

Load an Intel compiler module and use the Intel compiler -qopenmp option to build the executable. Since the Cascade Lake processors used in Endeavour3 and 4 support AVX512, you can experiment with different compiler versions and flags for better performance. For example:

```
% module load comp-intel/2020.0.166
% ifort -O2 -qopenmp program.f
```

or

```
% module load comp-intel/2020.4.304
% ifort -Ofast -ipo -axCORE-AVX512,CORE-AVX2,AVX -xSSE4.2 -qopenmp program.f
```

Set a specific number of OpenMP threads in your PBS job, and use a thread-pinning method to ensure that the threads do not get in the way of each other on the same core or migrate from one core to another. For example:

```
#PBS ...

module load comp-intel/2020.4.304
setenv OMP_NUM_THREADS 28

/u/scicon/tools/bin/mbind.x -cs -t28 -v ./a.out
```

or

```
setenv KMP_AFFINITY compact # or: setenv KMP_AFFINITY scatter
./a.out > output
```

or

```
setenv KMP_AFFINITY disabled
dplace -x2 ./a.out > output
```

See Process/Thread Pinning Overview for more information about different pinning methods.

Note: The default OMP_STACKSIZE is set to 200 megabytes (MB). If your application experiences segmentation fault (segfault), experiment with increasing the $OMP_STACKSIZE value to see if it helps.

Applications that Require Scientific and Math Libraries

The Intel Math Kernel Library (MKL) is included in Intel compiler modules (versions 11.1 and later). SeeMKL to learn how to link to MKL libraries. In many cases, the following commands may be sufficient:

```
% module load comp-intel/2020.4.304
% ifort -O2 program.f -mkl
```

Note: By itself, the -mkl option implies -mkl=parallel, which will link to the threaded MKL library. If your application can benefit from using multiple OpenMP threads when the MKL routines are called, you should also set the environment variable OMP_NUM_THREADS, as shown in the OpenMP example in the previous section.

If you want to use a non-threaded MKL library, change the -mkl option to -mkl=sequential.

MPI Applications

HPE's latest Message Passing Toolkit (MPT) library is recommended. Load the library as shown in the following example:

```
% module load comp-intel/2020.4.304
% module load mpi-hpe/mpt
% ifort -O2 program.f -lmpi
```

During runtime, load the Intel compiler and the MPT modules and use mpiexec to launch the executable:

```
#PBS ...
module load comp-intel/2020.4.304
module load mpi-hpe/mpt

mpiexec -np xx ./a.out > output
```

Running PBS Jobs

Each of the Endeavour nodes has 32 sockets, and each socket includes 28 cores and 192 gigabytes (GB) of physical memory. In each socket, PBS has access to 28 cores and 185 GB of memory, which is known as a minimum allocatable unit (MAU). In addition, one of the sockets is set aside for system operations, leaving 31 sockets available for user jobs.

PBS does not allow jobs to share resources in the same socket. However, the amount of resources in a socket that a job will receive is restricted to the amount requested. This restriction is enforced by the Linux kernel cgroups feature. For example, if you request one core and 100 GB of memory, one socket will be assigned exclusively for your job, but 27 cores and 85 GB of memory of the socket will not be accessible by your job.

When job accounting is enabled, your job will be charged by the number of MAUs assigned exclusively for your job.

The maximum amount of resources a job can request is:

- **Endeavour3:** 868 cores with 5,750 GB are the maximum resources available for PBS jobs.
- **Endeavour4:** 868 cores with 5,750 GB are the maximum resources available for PBS jobs.

Endeavour PBS Queues

The PBS server pbspl4 manages jobs for Endeavour3 and 4 (:model=cas_end) and the NAS A100 GPU nodes (:model=[mil_a100]). Among the queues listed by the command qstat -Q @pbspl4, you can run jobs on the two systems using the e_normal, e_long, e_vlong, and e_debug queues. You can also find detailed settings of each queue, such as the maximum walltime and the default or minimum ncpus and memory, using the following command line (the e_normal queue is used in the example):

```
pfe% qstat -fQ e_normal@pbspl4
```

PBS Commands

Run PBS commands (such as qsub, qstat, and qdel, etc.) from the PFEs and specify the PBS server, pbspl4. For example:

```
pfe21% qsub -q queue_name@pbspl4 job_script
pfe21% qstat -nu username @pbspl4
pfe21% qstat job_id.pbspl4
pfe21% qdel job_id.pbspl4
```

If you do not normally run jobs on Aitken and Electra (which also use the PFEs to submit jobs), and your primary workload is on Endeavour3 and Endeavour4, you may want to consider setting the PBS_DEFAULT environment variable to pbspl4. This allows you to simplify the PBS commands, as follows:

```
pfe21% setenv PBS_DEFAULT pbspl4
pfe21% qsub job_script
pfe21% qstat -nu username
pfe21% qstat job_id
pfe21% qdel job_id
```

Job Submission Examples

The following examples demonstrate how PBS allocates resources for your job.

Note: If you do not specify a resource attribute (such as ncpus or mem), a default value for that attribute will be assigned to your job.

- To request using all the resources in two MAUs:

  pfe*21*% qsub -l -lselect=2:ncpus=28:mem=185G:model=cas_end -q *queue_name*@pbspl4

- To request 56 cores:

  pfe*21*% qsub -l -lselect=1:ncpus=56:model=cas_end -q *queue_name*@pbspl4

  Your job will be given access to 56 cores across in two sockets and only the default amount of memory (32G).

- To request 1,850 GB of memory:

  pfe*21*% qsub -l -lselect=1:mem=1850G:model=cas_end -q *queue_name*@pbspl4

  Your job will be given access to 1,850 GB of memory spread in ten sockets and only the default number of cores (one).

- If you do not specify which Endeavour node to run your job, PBS will assign one for you. The following example specifies Endeavour3 and requests 28 cores and 185 GB of memory:

  pfe*21*% qsub -l -lselect=host=endeavour3:ncpus=28:mem=185G:model=cas_end -q *queue_name*@pbspl4

Job Accounting

The Standard Billing Unit (SBU) rate for using one MAU per hour is 1.31.

Your group's HECC SBU allocation is charged for usage on any HECC resources, including Electra, Aitken, and Endeavour. The output from the acct_ytd command does not have separate entries for the different systems. To find Endeavour3 and Endeavour4 usage for your group, use the acct_query command instead.

For example, to view the statistics of all of your jobs that ran on Endeavour3 on March 10, 2021, type:

pfe*21*% acct_query -c endeavour3 -d 03/10/21 -u *your_username* -olow

See Job Accounting Utilities to learn more about using the acct_ytd and acct_query tools.

Endeavour jobs should be submitted from the PFEs with the specification of the PBS server pbspl4 using one of the two methods described below. Without specifying pbspl4, your Endeavour job will be rejected by the default PBS server, pbspl1, for jobs submitted from the PFEs.

Specify pbspl4 in the Command Line

You can explicitly add the server name pbspl4 to your command line. For example:

pfe*25*% qsub -q *queue_name*@pbspl4 *job_script*

2468.pbspl4.nas.nasa.gov

In the example above, a queue is specified. If you submit the job without specifying a queue, PBS will assign the job to an appropriate Endeavour queue. For example:

pfe*25*% qsub -q @pbspl4 *job_script* 2468.pbspl4.nas.nasa.gov

Additional examples of PBS command lines that specify the pbspl4 server:

pfe*25*% qstat -nu *username* @pbspl4

pfe*25*% qstat 2468.pbspl4@pbspl4

pfe*25*% qalter -lwalltime=2:00:00 2468.pbspl4@pbspl4

pfe*25*% qdel 2468.pbspl4@pbspl4

Change the Default Server to pbspl4

If your primary workload is on Endeavour, and you do not normally run jobs other NAS distributed memory cluster systems, set the environment variable PBS_DEFAULT to pbspl4. This allows you to omit pbspl4 from your command lines. For example:

pfe*25*% setenv PBS_DEFAULT pbspl4

pfe*25*% qsub -q *queue_name job_script*

2468.pbspl4.nas.nasa.gov

To submit a job without specifying a queue:

pfe*25*% qsub *job_script*

2468.pbspl4.nas.nasa.gov

Additional examples of PBS command lines you can use after you set the PBS_DEFAULT environment variable to pbspl4:

pfe*25*% qstat -nu *username*

pfe*25*% qstat 2468

pfe*25*% qalter -lwalltime=2:00:00 2468

pfe*25*% qdel 2468

Queue Structure

Each Endeavour node contains 896 cores with six terabytes (TB) of memory.

To use these resources, submit PBS jobs to one of the Endeavour queues. Endeavour queue names begin with an "e_" prefix.

View the general queues for Endeavour by using the qstat command:

pfe*25*% qstat -Q @pbspl4 | grep e_

```
Queue    Ncpus/      Time/         State counts
name       max/def    max/def   jm T/Q/H/W/R/E/B pr
======== =====/=== ======/===== == ============= ==
e_normal    --/  8  08:00/01:00 -- 0/0/0/0/0/0/0  0
e_long      --/  8  72:00/36:00 -- 0/0/0/0/3/0/0  0
e_vlong     --/  8 600:00/24:00 -- 0/0/0/0/0/0/0  0
e_debug    128/  8  02:00/00:30 -- 0/0/0/0/0/0/0 15
```

Among the Endeavour queues, the higher-priority e_debug queue is the only one that has a limit on the maximum number of cores

(128 cores) per job. If you want to use the e_debug queue, you must specify -q e_debug either in the qsub command line or inside the PBS script.

To view more information about each Endeavour queue:

pfe*25*% qstat -fQ *queue_name*@pbspl4

Running Jobs Before Dedicated Time

The PBS batch scheduler supports a feature called shrink-to-fit (STF). This feature allows you to specify a range of acceptable wall times for a job, so that PBS can run the job sooner than it might otherwise. STF is particularly helpful when scheduling jobs before an upcoming dedicated time.

For example, suppose your typical job requires 5 days of wall time. If there are fewer than 5 days before the start of dedicated time, the job won't run until after dedicated time. However, if you know that your job can do enough useful work running for 3 days or longer, you can submit it in the following way:

qsub -l min_walltime=72:00:00,max_walltime=120:00:00 *job_script*

When PBS attempts to run your job, it will initially look for a time slot of 5 days. When no such time slot is found between now and the dedicated time, it will look for shorter and shorter time slots, down to the minimum wall time of 3 days.

If you have an existing job that is still queued, you can use the qalter command to add these min_walltime and max_walltime attributes:

qalter -l min_walltime=*hh:mm:ss*,max_walltime=*hh:mm:ss job_id*

You can also use the qalter command to change the wall time:

qalter -l walltime=*hh:mm:ss job_id*

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

If you have any questions or problems, please contact the NAS Control Room at (800) 331-8737, (650) 604-4444, or by email at support@nas.nasa.gov.

Checking the Time Remaining in a PBS Job from a Fortran Code

During job execution, it can be useful to find out the amount of time remaining for your PBS job. This allows you to decide if you want to gracefully dump restart files and exit before PBS kills the job.

If you have an MPI code, you can call MPI_WTIME and see if the elapsed wall time has exceeded some threshold in order to decide if the code should go into the shutdown phase.

For example:

```
include "mpif.h"

real (kind=8) :: begin_time, end_time

begin_time=MPI_WTIME()
do work
end_time = MPI_WTIME()

if (end_time - begin_time > XXXXX) then
   go to shutdown
endif
```

In addition, the following library has been made available on NAS compute systems for the same purpose:

```
/u/scicon/tools/lib/pbs_time_left.a
```

To use this library in your Fortran code, you need to:

1. Modify your Fortran code to define an external subroutine and an integer*8 variable

   ```
   external pbs_time_left
   integer*8 seconds_left
   ```

2. Call the subroutine in the relevant code segment where you want the check to be performed

   ```
   call pbs_time_left(seconds_left)
   print*,"Seconds remaining in PBS job:",seconds_left
   ```

   Note: The return value from pbs_time_left is only accurate to within a minute or two.

3. Compile your modified code and link with the above library using, for example:

   ```
   LDFLAGS=/u/scicon/tools/lib/pbs_time_left.a
   ```

Managing Jobs

The *reservable front end* feature allows you to reserve dedicated access to an Electra compute node for up to 90 days. You can use the node the same way you would use a PFE—to submit jobs and perform other tasks such as post-processing, visualization, and so on—except that the node will be reserved specifically for you (or your group) so you can use all of the node's processor and memory resources. No other users will have access to the node.

Reserving a Dedicated Node

To request a dedicated node, run the "reservable front end" command (pbs_rfe), specifying what type of node you want and for how long. The command makes a PBS advanced reservation for a node, and tells you which node is reserved.

You can submit batch or interactive jobs, check status, and so on, from your reserved node just as you would from a PFE.

pbs_rfe Command Usage

Run pbs_rfe --help to see the command options and arguments, as shown below:

```
$ pbs_rfe --help
usage: pbs_rfe [-h] [--debug] [--duration DURATION] [--group GROUPS]
               [--model MODEL] [--name NAME] [--starttime STARTTIME]
               [--user USERS] [--version] [-W BILL_GROUP]
               [{request,delete,status,which}] [clue]

...

positional arguments:
  {request,delete,status,which}
                        action to perform
  clue                  node, reservation id, or reservation name

optional arguments:
  -h, --help            show this help message and exit
  --debug, -d           increase debugging verbosity
  --duration DURATION, -D DURATION
                        how long to reserve front-end. Format: [days+]hours
  --group GROUPS, -G GROUPS
                        groups whose members can also use front-end
  --model MODEL, -m MODEL
                        processor type
  --name NAME, -N NAME  name to give reservation
  --starttime STARTTIME, -R STARTTIME
                        when reservation should start. Format as for date(1)
                        command
  --user USERS, -U USERS
                        other userid[s] who can also use front-end
  --version             show program's version number and exit
  -W BILL_GROUP, --group_list BILL_GROUP
                        GID to charge reservation to

In addition to the default "request" action, you can "delete" a
reservation, get the "status" of a reservation, or see "which" node is
assigned to a reservation.

 pbs_rfe delete resv_id       cancel a reservation (only one at a time)
 pbs_rfe status               list your reservations and their statuses
 pbs_rfe which                display the nodes assigned to you
```

The --starttime option allows you to specify when the reservation will start (the default is to start as soon as possible).

The --user option lets you specify other users who will be permitted to log in to your reserved host with you. For example, this might be useful during a shared debugging session. You can also use the --group option to permit all users whose primary group is in the specified list.

If you forget which node is assigned, you can run pbs_rfe which to identify the correct node. For example:

```
$ pbs_rfe which
r617i0n13
```

You can also use the which option in scripts—for example, when you need to learn which node is yours.

To find out the status of your reservation, run pbs_rfe status. For example:

```
$ pbs_rfe status
Resv name    Resv ID   User      ST         Start / End           Node
-----------  --------  --------  --  ------------ / ------------  ----
```

```
FE_zsmith    R3286463  zsmith    CO  Feb 05 15:38 / Feb 15 15:38  r617i0n13
```

Note: You cannot use the node until the reservation starts. PBS will send you an email at the start of the reservation period.

When the reservation is ready, the status (ST) will change from CO to RN.

Ending the Node Reservation

When you are finished with a reserved node, you should delete the reservation using the pbs_rfe delete command. For example:

```
$ pbs_rfe status
Resv name    Resv ID   User      ST        Start / End          Node
-----------  --------  --------  --  ------------ / ------------  ----
FE_zsmith    R3286638  zsmith    CO  Feb 09 10:00 / Feb 10 10:00  r601i2n6

$ pbs_rfe delete R3286638
R3286638.pbspl1.nas.nasa.gov deleted.
```

Setting Up the Node

While you can simply SSH into the reserved node from a PFE to run commands, the node is more useful if you start a screen or VNC session there. This allows your sessions on the node to survive even if the PFE you logged into crashes.

Note: This section describes how to set up VNC on the reserved node. For information on using the screen tool, see **man screen** or **info screen**.

VNC Setup

If you want to use a graphical user interface (GUI) to run commands on your node, you'll want to use VNC to set that up. We recommend reading the article VNC: A Faster Alternative to X11 to learn more about VNC and how to establish a VNC session. However, using VNC on a reserved node is simpler than described in that article, once a few preliminary steps are accomplished. These steps are described below.

Before You Begin: Set up SSH passthrough.

Complete these steps:

1. If you do not have the **$HOME/.vnc/xstartup** file, create one on a PFE by following steps 1 and 2 in VNC: A Faster Alternative to X11.
2. Using ssh, log into your reserved node and start vncserver. Do not use the -localhost option.

   ```
   $ ssh r601i1n3 vncserver
       New 'r601i1n3:1 (zsmith)' desktop is r601i1n3:1
       Starting applications specified in /u/zsmith/.vnc/xstartup
       Log file is /u/zsmith/.vnc/r601i1n3:1.log
   ```

3. Create a network tunnel from your workstation to the reserved node. The following command logs you into a PFE with a tunnel that leads to the node:

   ```
   $ ssh -L 5901:node_name:5901 pfe
   ```

   where node_name is the name of the reserved node (r601i1n3, in our example). Note that since you are the only user on your node, the 5901 parts of the command line are constant, unlike when you run vncserver on a PFE node.
4. Start your VNC client on your workstation and connect to localhost:5901. You will be connected to your graphical environment on your reserved node.

Limitations

- The compute node reserved via pbs_rfe does not have the PBS_NODEFILE environment variable set by default. If you plan to run any application that relies on having PBS_NODEFILE set, you will have to set it manually. For example, if you try to run an MPI job on the node without presetting PBS_NODEFILE, you will get the following error message:

  ```
  name_of_compute_node> mpiexec -np X a.out
  Not invoked from a known Work Load Manager:
  o For PBS   : PBS_NODEFILE is not set.
  o For SLURM : SLURM_JOB_ID is not set.
  mpiexec.real error: Aborting.
  ```

  The solution is to create a file, for example a file named "*hfile*", and put the name of your reserved node in it as many times as you want MPI ranks. Then, set the PBS_NODEFILE environment variable to the path to *hfile*.

  ```
  (for bash)
  export PBS_NODEFILE=/path/to/hfile

  (for csh)
  setenv PBS_NODEFILE /path/to/hfile
  ```

- Compute nodes do not have all of the software packages that are installed on the PFEs. You might need to load the current

pkgsrc module to get access to some commands. See Using Software Packages in pkgsrc for more information.

- /tmp on a node is small and takes memory away from programs.
- The reserved nodes don't have direct network access to hosts outside Electra. Use your PFE login to transfer files, to outside hosts.
- The cron daemon does not run on reserved nodes. Again, use a PFE for that type of activity.
- While reservations are allowed for up to 90 days, a reserved node might need to be rebooted sooner than that (e.g., for system patches), or you might run the node out of memory. You'll still have the node reserved after the reboot, but anything you were doing will be lost. Please be sure to save your work frequently.
- Currently, each user is limited to only one node reserved at a time.

Account Charging

Reserved nodes are charged Standard Billing Units (SBUs) at the same rate as PBS jobs, whether you are actively using the node or not. Reservations are billed daily. Be sure to delete your reservation when you no longer need it.

To reduce costs, a few users can share the same reservation. (See the--user and --group options above, and Sharing with Multiple Users.

Running Jobs Before Dedicated Time

The PBS batch scheduler supports a feature called shrink-to-fit (STF). This feature allows you to specify a range of acceptable wall times for a job, so that PBS can run the job sooner than it might otherwise. STF is particularly helpful when scheduling jobs before an upcoming dedicated time.

For example, suppose your typical job requires 5 days of wall time. If there are fewer than 5 days before the start of dedicated time, the job won't run until after dedicated time. However, if you know that your job can do enough useful work running for 3 days or longer, you can submit it in the following way:

qsub -l min_walltime=72:00:00,max_walltime=120:00:00 *job_script*

When PBS attempts to run your job, it will initially look for a time slot of 5 days. When no such time slot is found between now and the dedicated time, it will look for shorter and shorter time slots, down to the minimum wall time of 3 days.

If you have an existing job that is still queued, you can use the qalter command to add these min_walltime and max_walltime attributes:

qalter -l min_walltime=*hh:mm:ss*,max_walltime=*hh:mm:ss job_id*

You can also use the qalter command to change the wall time:

qalter -l walltime=*hh:mm:ss job_id*

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

If you have any questions or problems, please contact the NAS Control Room at (800) 331-8737, (650) 604-4444, or by email at support@nas.nasa.gov.

Checking the Time Remaining in a PBS Job from a Fortran Code

During job execution, it can be useful to find out the amount of time remaining for your PBS job. This allows you to decide if you want to gracefully dump restart files and exit before PBS kills the job.

If you have an MPI code, you can call MPI_WTIME and see if the elapsed wall time has exceeded some threshold in order to decide if the code should go into the shutdown phase.

For example:

```
include "mpif.h"

real (kind=8) :: begin_time, end_time

begin_time=MPI_WTIME()
do work
end_time = MPI_WTIME()

if (end_time - begin_time > XXXXX) then
   go to shutdown
endif
```

In addition, the following library has been made available on NAS compute systems for the same purpose:

```
/u/scicon/tools/lib/pbs_time_left.a
```

To use this library in your Fortran code, you need to:

1. Modify your Fortran code to define an external subroutine and an integer*8 variable

   ```
   external pbs_time_left
   integer*8 seconds_left
   ```

2. Call the subroutine in the relevant code segment where you want the check to be performed

   ```
   call pbs_time_left(seconds_left)
   print*,"Seconds remaining in PBS job:",seconds_left
   ```

   Note: The return value from pbs_time_left is only accurate to within a minute or two.

3. Compile your modified code and link with the above library using, for example:

   ```
   LDFLAGS=/u/scicon/tools/lib/pbs_time_left.a
   ```

Many jobs running across multiple nodes perform setup and pre-processing tasks on the first node, then launch computations across many nodes, and finish with post-processing and cleanup steps running only on the first node. The post-processing and cleanup steps may take minutes or even hours, during which the remaining nodes in the job are idle. You can use the pbs_release_nodes command to release these idle nodes so that you are not charged for them, and they can be made available to other jobs.

Some users have already taken steps to minimize the number of idle nodes by separating their workflow into multiple jobs of appropriate sizes. The downside to this approach is additional time waiting in the queue for these multiple jobs. Using pbs_release_nodes may allow these jobs to be combined, reducing the amount of wait time in the queue.

Node Tracking and Job Accounting

When you use the pbs_release_nodes command, the released nodes are removed from the contents of $PBS_NODEFILE. The job accounting system tracks when the nodes are released, and adjusts the amount of SBUs charged accordingly.

Running the pbs_release_nodes Command

You can either include the pbs_release_nodes command in your job script or you can run it interactively, for example, on a PFE. If you run it in your job script, you can use the $PBS_JOBID environment variable to supply the *job_id*. You can use pbs_release_nodes multiple times in a single job.

There are two ways to use the pbs_release_nodes command. In both cases, the first node is retained.

- **Keep the first node and release all other nodes**
  If you want to retain the first node only, run the command as follows to release the remaining nodes:

  pfe21% pbs_release_nodes -j *job_id* -a

- **Release one or more specific nodes**
  If you want to retain one or more nodes in addition to the first node, run the command as follows to release specific idle nodes:

  pfe21% pbs_release_nodes -j *job_id node1 node2 ... nodeN*

To find out which nodes are being used by a job before you run the pbs_release_nodes command, use one of the following methods:

- **For interactive jobs:** Run qstat -f *job_id*. In the output, look for the nodes listed under the exec_vnode job attribute.
- **For job scripts:** Run cat $PBS_NODEFILE. The output provides a list of the nodes in use.

Note: $PBS_NODEFILE may have duplicate entries for nodes, depending on how resources were requested. See **man pbs_resources** for details.

Sample Job Script with pbs_release_nodes Command

The following sample job script keeps the first node and releases all other nodes before running post-processing steps:

```
#PBS -S /bin/csh
#PBS -N cfd
#PBS -l select=4:ncpus=8:mpiprocs=8:model=sky_ele
#PBS -l walltime=4:00:00
#PBS -j oe
#PBS -W group_list=a0801
#PBS -m e
module load comp-intel/
module load mpi-hpe/mpt
cd $PBS_O_WORKDIR
mpiexec dplace -s1 -c 16-23 ./grinder < run_input > intermediate_output
# Before executing some post-processing steps that will run only on the
# first node of the job you can release all other nodes
pbs_release_nodes -j $PBS_JOBID -a
# Now run post-processing steps
./post_process < intermediate_output > final_output
# -end of script-
```

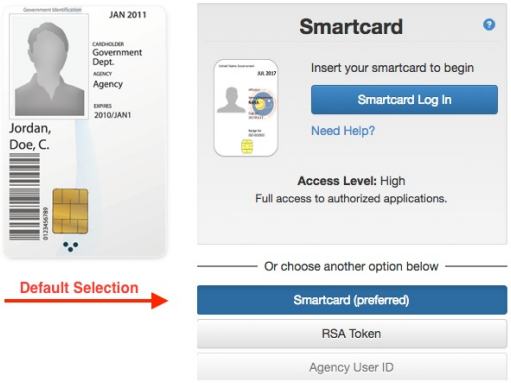
Monitoring with myNAS

Authenticating to NASA's Access Launchpad

In order to perform some basic tasks that use NAS web applications, such as changing your NAS password or logging into the myNAS portal, you must authenticate to NASA's Access Launchpad using your NASA Smartcard or RSA SecurID token.

Using Your NASA Smartcard

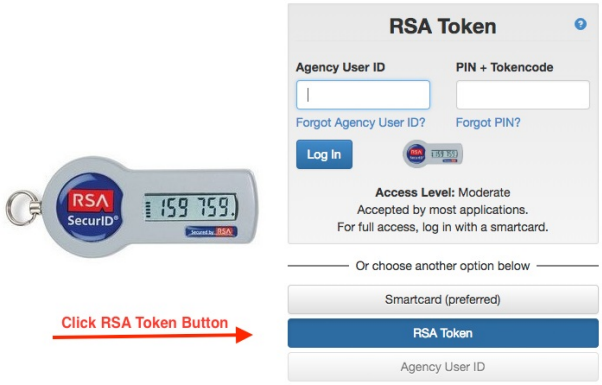
This is Launchpad's default option. Make sure **Smartcard (preferred)** is selected, insert your NASA smartcard into your workstation's card reader, and click **Smartcard Log In**, as shown below. Enter your PIN when prompted.



Using Your RSA SecurID Token

Select the **RSA Token** option, provide your Agency User ID, and enter your passcode, as shown below.

If you have an RSA SecurID fob, your passcode is your PIN + the token code displayed on the fob. If you have a soft token, enter your PIN in the RSA SecurID app to get the token code, then enter *only* the token code in the **PIN + Tokencode** field.



Once you are authenticated to Launchpad, you will be connected to the NAS password change form, the myNAS web portal, or another NAS web application to complete your task.

The myNAS web portal provides a central location where you can easily monitor all aspects of your NAS user account activity, including PBS jobs, resource allocations, and usage for the project group IDs (GIDs) you belong to. If you use the Shift tool to transfer your data, you can also check the status of your file transfers.

If you are a principal investigator (PI), you will see information about jobs and usage by all users who are assigned to your GIDs.

Follow this link to access the myNAS web portal: https://portal.nas.nasa.gov

Note: You will log into myNAS via NASA's Access Launchpad, using your NASA Smartcard or RSA SecurID token.

Home Page: Information at a Glance

Once you log in, myNAS will automatically load information for your NAS user account. On the home page, you will see:

- The number of jobs currently in each state (running, waiting, recently finished, and held)
- All of your jobs that recently changed state
- The status of your open support tickets
- An estimate of today's usage for each of your GIDs
- The latest HECC news and announcements
- Scheduled downtimes for all HECC resources

For more detailed information about all of your jobs, click **Jobs** in the navigation panel on the left side of the page. To check allocation and usage, click **Accounts**. To check storage quotas and usage, along with the status of file transfers, click **Data**.

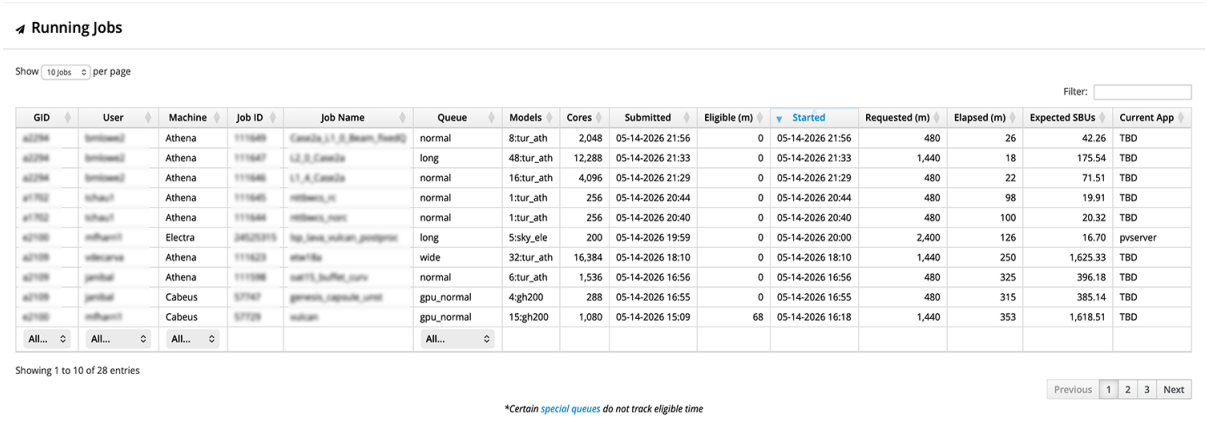
TIP: Take a quick feature tour of each myNAS page by clicking the question mark icon in the top right corner.

Jobs Page: Detailed Job Information

The **Jobs** page provides details about each of your running, waiting, recently finished, and held jobs, including:

- GID, job ID, and job name
- Requested queue, machine, processor model, and number of cores
- Date/time submitted, date/time started, and requested and elapsed time
- Number of standard billing units (SBUs) the job is expected to use
- Information about interrupted jobs, if applicable

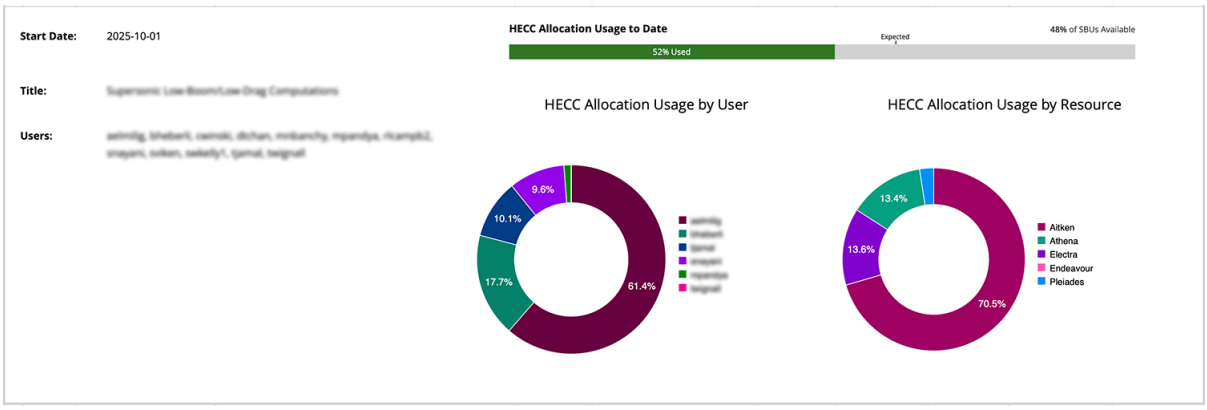
You can easily sort the tables by column and filter by any text string such as partial job name, model type, machine, and so on. For example:



Accounts Page: Detailed Allocation and Usage Information

The **Accounts** page provides information about all your GIDs, such as PI, project name, mission, and allocation, expected usage to date, and actual usage. If you have a GPU allocation, you will also see information about your GPU allocation and usage.

Click the disclosure arrow in the left column of any GID listed in the table to see interactive charts that provide allocation and usage information at a glance. The thermometers show overall allocation usage for the GID, while the first donut chart shows either your own usage (if you are an individual user) or usage for all GID members (if you are a PI). The second donut chart shows usage for the GID by resource (machine). For example:

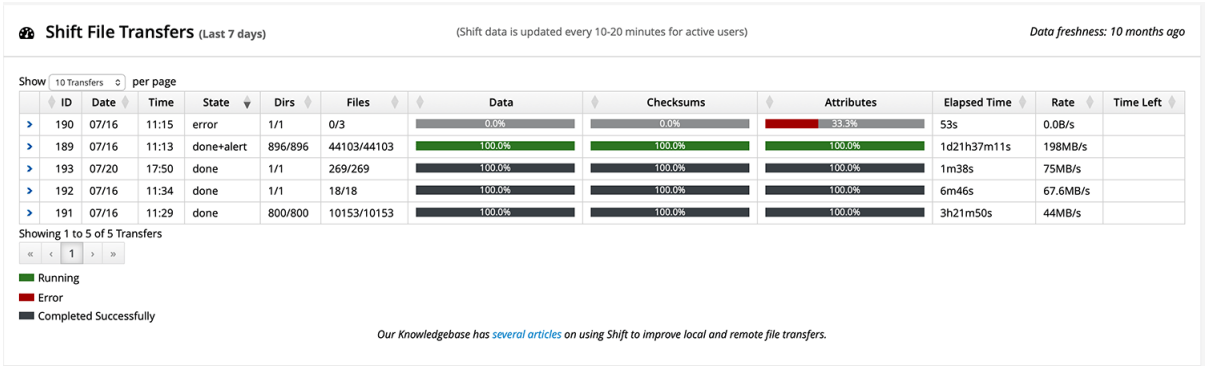


For those users in groups with GPU allocations, a second table will appear below the first one with similar information on GPU allocations and usage.

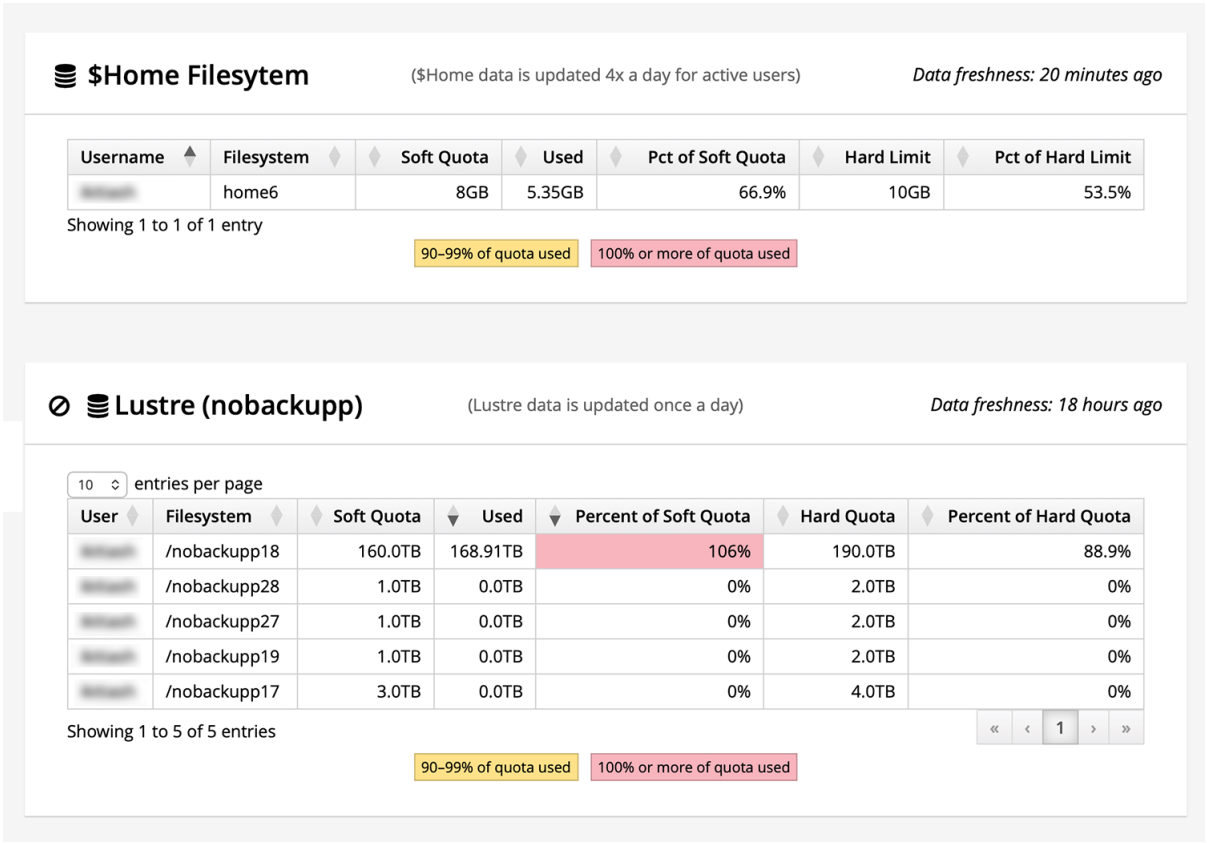
Data Page: Detailed Storage and Transfer Information

The **Data** page provides information about your Shift file transfers, $Home filesystem quota/usage, and Lustre (nobackupp) filesystem quota/usage.

The **Shift File Transfer table** shows the details and status of your Shiftc file transfers within the last seven days. Click on the disclosure arrow in the left column for a transfer to reveal more information about the source, destination, and initiating command.



The **$Home and Lustre (nobackkup) tables** show your current quotas and usage for each of the relevant filesystems.



Continuous Monitoring

We recommend bookmarking the myNAS website in your browser, and taking some time to explore all of the useful information that is available at a glance via interactive tables and charts. Once you are logged in, you can keep the browser window open to easily monitor your job and account activity.

Optimizing/Troubleshooting

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.

Using pdsh_gdb for Debugging PBS Jobs

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.

PBS exit codes

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).

How to Run a Graphics Application in a PBS Batch Job

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.

MPT Startup Failures: Workarounds

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

Common Reasons for Being Unable to Submit Jobs

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.

Avoiding Job Failure from Overfilling /PBS/spool

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
```

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
```

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.

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.

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.

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

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						

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 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 -lselect=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.

PBS Reference

PBS Environment Variables

Several environment variables are provided to PBS jobs. Some are taken from the user's environment and carried with a job, and others are created by PBS. There are also some environment variables that you can explicitly create for exclusive use by PBS jobs.

All PBS-provided environment variable names start with the characters "PBS_". Some start with "PBS_O_", which indicates that the variable is taken from the job's originating environment (that is, the user's environment).

A few useful PBS environment variables are described in the following list:

PBS_O_WORKDIR
: Contains the name of the directory from which the user submitted the PBS job

PBS_O_PATH
: Value of *PATH* from submission environment

PBS_JOBID
: Contains the PBS job identifier

PBS_JOBDIR
: Pathname of job-specific staging and execution directory

PBS_NODEFILE
: Contains a list of vnodes assigned to the job

TMPDIR
: The job-specific temporary directory for this job. Defaults to /tmp/pbs.*job_id* on the vnodes.

Default Variables Set by PBS

You can use the env command--either in a PBS script or on the command line of a PBS interactive session--to find out what environment variables are set within a PBS job. In addition to the PBS_*X* environment variables, the following variables are useful to know:

NCPUS
: Defaults to number of CPUs that you requested for the node.

OMP_NUM_THREADS
: Defaults to 1 unless you explicitly set it to a different number. If your PBS job runs an OpenMP or MPI/OpenMP application, this variable sets the number of threads in the parallel region.

FORT_BUFFERED
: Defaults to 1. Setting this variable to 1 enables records to be accumulated in the buffer and flushed to disk later.