Login

Idle SSH Connections are Broken After 10-15 Minutes

If you find that idle SSH connections tend to be dropped after 10 to 15 minutes and your are using a router or other device that implements Network Address Translation (NAT), such as a DSL router, you may want to enable the ServerAliveInterval parameter in your ~/.ssh/config.

The issue is that some devices that implement NAT will drop information from their state tables if a TCP connection has been idle for just a few minutes. In such cases, it may be necessary to ensure that the connection is never truly idle for an extended period.

The ServerAliveInterval parameter (supported in OpenSSH 3.8 and later) causes the SSH client to periodically send an encrypted query to determine if the server is still responsive. The packets will make the connection appear to be active and can prevent the NAT state table entry for the connection from timing out.

For example, adding the following to your ~/.ssh/config will cause an encrypted query to be sent every five minutes:

```
ServerAliveInterval 5m
```

TIP: You may need to experiment with the length of the interval to determine an appropriate value to prevent your connection from being broken. If you use the NAS config template, ServerAliveInterval is set to 10 minutes.

SSH Connection is Periodically Broken Even When Connection is Active

During an SSH session, the client and server will re-negotiate the keys used to encrypt the session every so often. Very old versions of OpenSSH did not implement this feature, which can cause a problem when connected to the SFEs. Your connection may appear to hang or may abort with an error message such as the following:

```
Dispatch protocol type 20
```

If you encounter this issue, you will need to upgrade to a newer version of OpenSSH. Note: Other implementations that were derived from the OpenSSH distribution, such as SUN Secure Shell, are likewise known to exhibit this problem.

Slow Performance Transferring Data

While OpenSSH performs reasonably for local data transfers, the performance will tend to be reduced due to the latency of long-haul network connections.

Depending on the severity of the impact you may have several options to improve this situation.

Upgrade to OpenSSH 4.7 or Later

If you are using a version of OpenSSH that is older than 4.7, you may see an improvement in file transfer performance by upgrading to OpenSSH 4.7 or later. This is due to the use of a larger channel buffer introduced in that version.

Read File Transfer Tips for more recommendations.

Use the information in this section to help troubleshoot any problems you encounter when logging into NAS systems.

SSH Known-Hosts Error

A successful host verification indicates that your Secure Shell (SSH) client has established a secure connection with the SSH server, and that no intermediate machines have access to that connection. The identity of the remote host is verified by checking the host public key of the remote host, which is stored in one of the following locations:

- The system-wide /etc/ssh/ssh_known_hosts file (for some systems, /etc/ssh_known_hosts)
- The ~/.ssh/known_hosts file on your local system

There are three ways SSH can respond to an unrecognized or changed SSH host key, based on the value of the StrictHostKeyChecking variable in either your ~/.ssh/config file or in the /etc/ssh/ssh_config (or /etc/ssh_config) file on your local system. The possible values are:

StrictHostKeyChecking=no
: This is the most insecure setting as it will blindly connect to the server. It will add the server's key if it's not present locally, and if the key has changed it will add the key without asking.

StrictHostKeyChecking=ask
: With this setting, if you have no host key for the server, it will show you the fingerprint and ask you to confirm. If you connect and the key does not match, it will prevent you from logging in, and will tell you where to find the conflicting key inside the known_hosts file.

StrictHostKeyChecking=yes
: This is the most secure setting. If you have no host key for this server, then it will prevent you from logging in at all.

If you receive an SSH known-hosts error message the first time you log into an SFE, and the StrictHostKeyChecking variable is set to yes, you can resolve the problem by adding -o "stricthostkeychecking=ask" to your SSH command. For example:

SSH Known-Hosts Error Example

your_local_system% ssh sfe*6*.nas.nasa.gov

WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
SHA256:lKccVJBCXNjEEJ0Gq2XWKNfWtzimfyGEMu64asz5bwM.
Please contact your system administrator.
Add correct host key in /Users/userid/.ssh/known_hosts2 to get rid
of this message.
Offending key in /etc/ssh_known_hosts2:24
RSA host key for sfe*6*.nas.nasa.gov has changed and you have
requested strict checking.
No RSA host key is known for sfe*6*.nas.nasa.gov and you have
requested strict checking.
Host key verification failed.

SSH Known-Hosts Error: Solution

Add -o "stricthostkeychecking=ask" to your SSH command. For example:

your_local_system% ssh -o "stricthostkeychecking=ask" sfe*6*.nas.nasa.gov

---
The authenticity of host 'sfe*6*.nas.nasa.gov (198.9.4.40)'
can't be established.
RSA key fingerprint is
SHA256:lKccVJBCXNjEEJ0Gq2XWKNfWtzimfyGEMu64asz5bwM.
Are you sure you want to continue connecting (yes/no)? **yes**

Warning: Permanently added 'sfe*6*.nas.nasa.gov,198.9.4.40' (RSA) to
the list of known hosts.

Plugin authentication
Enter PASSCODE: **type your passcode**

Common RSA SecurID Passcode Problems

Incorrect Passcode or Locked Token

Each RSA SecurID tokencode can be used only once. If you try to use a single tokencode multiple times, you will be prompted again to enter a new passcode. If you do not provide a correct passcode in a few consecutive attempts, your RSA SecurID token will be temporarily disabled and you will need to contact the NAS Control Room at (800) 331-8737 or (650) 604-4444 to unlock the token.

Note: If you have an RSA SecurID hard token (fob), the passcode is your PIN + the tokencode. If you have anRSA SecurID soft token (mobile app), the passcode is the tokencode only.

Incomplete PIN Setup (Hard Token Only)

If you have a hard token (fob) and you cannot log into NAS systems after you first create your PIN, it is possible that you did not successfully complete the new-PIN process. For example, you might have used a special character in your PIN, which is not supported.

In this case, try logging into an SFE again. You will be prompted to complete the new-PIN process, as shown below.

Note: Your PIN must consist of exactly 8 alphanumeric characters, with at least 1 letter and at least 1 number, and no special characters.

your_local_system% ssh sfe*6*
user@sfe*6*'s password:
Authenticated with partial success.
Plugin authentication
Enter PASSCODE:
Plugin authentication
You may create your own PIN or accept a server assigned PIN.
Would you like to create your own new PIN (yes/no)? *yes*
Plugin authentication
Enter a new PIN of 8 alphanumeric characters:
Re-enter new PIN to confirm:
Enter PASSCODE:
Plugin authentication
Enter PASSCODE:
Permission denied, please try again.
user@sfe*6*'s password:
Permission denied, please try again.
user@sfe*6*'s password:
Permission denied ().

Common NASA PIV Problems

PIN Always Prompted when Connecting to SFEs

NASA-supported laptops attempt to use PIV by default when a NASA badge is in an attached card reader. To avoid having to enter your NASA PIV PIN every time you log in, set up SSH Passthrough. Alternatively, you can disable the NASA PIV as discussed below.

Too Many Authentication Failures

The hook that allows SSH to use NASA PIV authentication may attempt to utilize all of the certificates on a NASA badge instead of just the one needed for authentication. This can cause more authentication failures than are allowed on NAS systems. In this case, you can disable PIV in one of the following ways.

- Disable PIV on the command line each time you log in:

  ssh -oPKCS11Provider=none *username*@sfeX.nas.nasa.gov

- Disable PIV by permanently modifying your SSH configuration in the ~/.ssh/config file by adding the following line (in the section of the file related to SFE configuration):

  PKCS11Provider none

Alternatively, if you want to continue using NASA PIV, sometimes a different provider is available on the systems (such as NASA-supported laptops) using:

ssh -oPKCS11Provider=/usr/lib/ssh-keychain.dylib *username*@sfeX.nas.nasa.gov

(or in the ~/.ssh/config file):

PKCS11Provider /usr/lib/ssh-keychain.dylib

Common Passthrough Problems

If you have previously set up SSH passthrough correctly, when you log into a NAS host you should be prompted only for your RSA SecurID passcode before you are "passed through" directly to the host you requested. If this does not happen, see the following common problems and solutions.

Authentication Failure

If authentication fails when you enter the passcode, it is possible that your username for your local system is not the same as your username for the NAS systems, and your .ssh/config file does not contain your NAS username. To solve this problem, add your NAS username to your .ssh/config file, as follows:

- If you use the .ssh/config file to connect to multiple computer sites, then add your NAS username to the ProxyCommand lines for corresponding NAS hosts. For example:

  Host lfe
  ProxyCommand ssh *username*@sfe*6*.nas.nasa.gov /*usr*/local/bin/ssh-proxy lfe

  In this case, you will still need to include your NAS username when you run the ssh command, as shown in the following example, in order to avoid being prompted for a password:

  %ssh *your_nas_username*@lfe

- If you use the .ssh/config file to connect only to NAS, then you can simply add the following line at the beginning of your .ssh/config file:

  User your_nas_username

  In this case, you do not need to add your NAS username to the ProxyCommand lines. Additionally, you can run the ssh command without your NAS username in the command line, and you will not be prompted for a password. For example:

  %ssh lfe

Password Prompt After Entering Passcode

If you are prompted for a password in addition to the passcode, there are several possible causes:

- Your local system username is different from your NAS system username. To solve this problem, see the previous section, **Authentication Failure**.
- Either your home directory or the .ssh/authorized_keys file under your NAS account specifies write permission for group or others. You must ensure that write permission is specified only for you.
- One or both of the following:
  - You have not run ssh-key-init on *sfeX*.
  - The .ssh/authorized_keys file of the NAS host that you want to connect to is missing.

Passphrase Prompt After Entering Passcode

If you are prompted for your passphrase in addition to the passcode, the most likely reason is that you did not run following commands to forward your private key before you issued the ssh command:

- ssh-agent
- ssh-add ~/.ssh/id_rsa

See Setting Up SSH Passthrough for more information about these commands.

Password Expiration

Your NAS password is valid for 60 days. Before it expires, you will receive an email notification reminding you to change your password. If your password expires you will still be able to log into NAS systems, but you will not be able to proceed until you follow the prompt to change it.

Account Expiration or Deactivation

Your NAS account is active as long as you have a valid project and a valid account request form on file at NAS. If your account expires, it will be removed from the NAS database and from the /etc/passwd files, and you will not be able to log in.

To maintain a valid account, you must submit a new account request form each year. You will receive an email notification when it is time to submit a new one.

If you violate NAS security rules such as those listed in the NAS Systems Environment Rules of Behavior, your account can be deactivated.

Error Connecting to a Cloud SFE

The cloud SFE updates new PIV credentials once every 24 hours. If you receive the following error when trying to connect to create a mesh key, your new PIV card credentials may not yet have been added to the cloud SFE:

WARNING: Your password has expired.
Password change required but no TTY available.

The solution is to wait 24 hours and try again.

Code Development

Debugging

Commonly Used Options for Debugging

-O*0*
: Disables optimizations. Default is -O*2*

-g
: Produces symbolic debug information in object file (implies -O*0* when another optimization option is not explicitly set)

-traceback
: Tells the compiler to generate extra information in the object file to provide source file traceback information when a severe error occurs at runtime.
Specifying -traceback will increase the size of the executable program, but has no impact on runtime execution speeds.

-check all
: Checks for all runtime failures.
Fortran only.

-check bounds
: Alternate syntax: -CB. Generates code to perform runtime checks on array subscript and character substring expressions.
Fortran only.
Once the program is debugged, omit this option to reduce executable program size and slightly improve runtime performance.

-check uninit
: Checks for uninitialized scalar variables without the *SAVE* attribute.
Fortran only.

-check-uninit
: Enables runtime checking for uninitialized variables. If a variable is read before it is written, a runtime error routine will be called. Runtime checking of undefined variables is only implemented on local, scalar variables. It is not implemented on dynamically allocated variables, extern variables or static variables. It is not implemented on structs, classes, unions or arrays.
C/C++ only.

-ftrapuv
: Traps uninitialized variables by setting any uninitialized local variables that are allocated on the stack to a value that is typically interpreted as a very large integer or an invalid address. References to these variables are then likely to cause run-time errors that can help you detect coding errors. This option sets -g.

-debug all
: Enables debug information and control output of enhanced debug information. To use this option, you must also specify the -g option.

-gen-interfaces
-warn interfaces
: Tells the compiler to generate an interface block for each routine in a source file; the interface block is then checked with -warn interfaces.

Options for Handling Floating-Point Exceptions

-fpe{*0*|*1*|*3*}
: Allows some control over floating-point exception (divide by zero, overflow, invalid operation, underflow, denormalized number, positive infinity, negative infinity or a NaN) handling for the main program at runtime.
Fortran only.
-fpe*0*: underflow gives 0.0; abort on other IEEE exceptions
-fpe*3*: produce NaN, signed infinities, and denormal results
Default is -fpe*3* with which all floating-point exceptions are disabled and floating-point underflow is gradual, unless you explicitly specify a compiler option that enables flush-to-zero. Note that use of the default -fpe*3* may slow runtime performance.

-fpe-all={*0*|*1*|*3*}
: Allows some control over floating-point exception handling for each routine in a program at runtime. Also sets -assume ieee_fpe_flags. Default is -fpe-all=*3*.
Fortran only.

-assume ieee_fpe_flags
: Tells the compiler to save floating-point exception and status flags on routine entry and restore them on routine exit. This option can slow runtime performance.
Fortran only.

-ftz
: Flushes denormal results to zero when the application is in the gradual underflow mode. This option has effect only when compiling the main program. It may improve performance if the denormal values are not critical to your application's behavior. Every optimization option O level, except -O*0*, sets -ftz.

Options for Handling Floating-Point Precision

-mp
: Enables improved floating-point consistency during calculations. This option limits floating-point optimizations and maintains declared precision. -mp*1* restricts floating-point precision to be closer to declared precision. It has some impact on speed, but less than the impact of -mp.

-fp-model precise

Tells the compiler to strictly adhere to value-safe optimizations when implementing floating-point calculations. It disables optimizations that can change the result of floating-point calculations. These semantics ensure the accuracy of floating-point computations, but they may slow performance.

-fp-model strict

Tells the compiler to strictly adhere to value-safe optimizations when implementing floating-point calculations and enables floating-point exception semantics. This is the strictest floating-point model.

-fp-speculation=*off*

Disables speculation of floating-point operations. Default is -fp-speculation=*fast*

-pc{*64*|*80*}

For Intel EM64 only. Some floating-point algorithms are sensitive to the accuracy of the significand, or fractional part of the floating-point value. For example, iterative operations like division and finding the square root can run faster if you lower the precision with the -pc[*n*] option. -pc*64* sets internal FPU precision to 53-bit significand. -pc*80* is the default and it sets internal FPU precision to 64-bit significand.

A segmentation fault (often called a *segfault*) can occur if a program you are running attempts to access an invalid memory location. When a segmentation fault occurs, the program will terminate abnormally with an error similar to the following message:

```
SIGSEGV: Segmentation fault - invalid memory reference.
forrtl: severe (174): SIGSEGV, segmentation fault occurred
```

The program may generate a core file, which can help with debugging.

If you use an Intel Classic or OneAPI compiler, and you include the `-g -traceback` options, the runtime system will usually point out the function and line number in your code where a segmentation fault occurred. However, the location of the segmentation fault might not be the root problem—a segfault is often a symptom, rather than the cause of a problem.

Common Segfault Scenarios

Common scenarios that can lead to segmentation faults include running out of stack space and issues resulting from bugs in your code.

Running Out of Stack Space

Stack space is a segment of program memory that is typically used by temporary variables in the program's subroutines and functions. Attempting to access a variable that resides beyond the stack space boundary will cause segmentation faults.

The usual remedy is to increase the stack size and re-run your program. For example, to set the stack size to unlimited, run:

For csh
: `unlimit stacksize`

For bash
: `ulimit -s unlimited`

On some of the NAS front-end nodes, the default stack size is set to 300,000 kilobytes (KB), while others are set with an unlimited stacksize. Use the command `limit` in csh and `ulimit -a` in bash to find the default stacksize set on the front-end you are on. On all NAS compute nodes, PBS sets the stack size to `unlimited`. However, for those compute nodes served by a front-end with a default stacksize set to 300,000 KB, if you use `ssh` to connect from one compute node to another (or several others) in order to run programs, then the stack size on the other node(s) is set to 300,000 KB.

Note: Setting the stack size to `unlimited` on a front-end node might cause problems with Tecplot. For more information, see Tecplot.

Bugs in Your Fortran Code

In Fortran programs, the most common bugs that cause segmentation faults are array bounds violations—attempts to write past the declared bounds of an array. Occasionally, uninitialized data can also cause segmentation faults.

Array Bounds Violations

To find array bounds violations, re-compile your code with the Intel Classic ifort or OneAPI ifx compiler using the `-check` (or `-check all`) option in combination with your other compiler options. When you use the `-check` option, the Fortran runtime library will flag occurrences of array bounds violations (and some other programming errors).

When the runtime library encounters the first array bounds violation, it will halt the program and provide an error message indicating where the problem occurred. You may need to re-run the code multiple times if there is more than one array bounds violation.

Note: Code compiled with the `-check` option may run significantly slower than code compiled with normal optimization (without the `-check` option).

Uninitialized Variables

You can use the `-init=keyword` option (available in the 2015 Intel Fortran compiler and later versions) to check uninitialized variables. The following keywords can be used with the `-init` option:

`[no]arrays`
: Determines whether the compiler initializes variables that are arrays or scalars. Specifying `arrays` initializes variables that are arrays or scalars. Specifying `noarrays` initializes only variables that are scalars. You must also specify either `init snan` or `init zero` when you specify `init [no]arrays`.

`[no]snan`
: Determines whether the compiler initializes to signaling NaN all uninitialized variables of intrinsic type REAL or COMPLEX that are saved, local, automatic, or allocated.

`[no]zero`
: Determines whether the compiler initializes to zero all uninitialized variables of intrinsic type REAL, COMPLEX, INTEGER, or

LOGICAL that are saved, local, automatic, or allocated.

Additional choices of keyword to set initial values include `infinity`, `minus_infinity` (valid only for floating point), `tiny`, `minus_tiny` (valid only for floating point), `huge`, and `minus_huge`.

Note: The `-init` compiler option does not catch all possible uninitialized variables. To find more, you can use the NAS-developed `uninit` tool. For information about using this tool, see the NAS training presentation uninit: Locating Use of Uninitialized Data in Floating Point Computation in Big Applications.

For more information about segmentation faults, see:

- Determining Root Cause of Segmentation Faults SIGSEGV or SIGBUS errors (Intel Developer Zone)
- Segmentation Fault (Wikipedia)

TotalView is a GUI-based debugging tool that provides control over processes and thread execution, as well as visibility into program state and variables, for C, C++ and Fortran applications. It also provides memory debugging to detect errors such as memory leaks, deadlocks, and race conditions. You can use TotalView to debug serial, OpenMP, or MPI codes.

Using TotalView to Debug Your Code

To find out which versions are available as modules, use the `module avail` command.

Before You Begin

Our current licenses allow using TotalView with MemoryScape, ReplayEngine, and CUDA support for up to a total of 256 processes on NAS x86_64 cluster systems and shared memory systems. Use the following command to find out whether there are unused licenses before you start TotalView:

```
pfe% /u/scicon/tools/bin/check_licenses -t
```

Ensure that X11 forwarding is set up on your system. Alternately, you can use the `ssh -X` or `-Y` options to enable X11 forwarding for your SSH session.

Note: If you are using a NAS-supported workstation or compute server, X11 forwarding should already be set up on your system. If the response of the GUI via X11 forwarding is slow, you will need to set up a VNC session.

Complete the following steps:

1. Compile your program using the `-g` option.
2. Start a PBS session.

   On on NAS x86_64 cluster systems:

   ```
   % qsub -I -X -q devel -lselect=2:ncpus=8:model=xxx,walltime=1:00:00
   ```

   where xxx is one of the following: `bro_ele`, `sky_ele`, `cas_ait`, `rom_ait`, or `mil_ait`.

   On Endeavour3/4:

   ```
   % qsub -I -X  -lselect=1:ncpus=28:mem=185GB:model=cas_end,walltime=1:00:00 \
   -q queue_name@pbspl4
   ```

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

3. Test the X11 forwarding with `xclock`:

   ```
   % xclock
   ```

Debugging with TotalView

Load the module:

```
% module load totalview/2023.4.16
```

The method you use to run TotalView depends on the application you want to debug.

For Serial Applications

Launch TotalView by running the `totalview` command and specifying the application:

```
% totalview ./a.out
```

Or, if your application requires arguments:

```
% totalview ./a.out -a arg_1 arg_2
```

For MPI Applications Built with HPE MPT

The old `-tv` option of HPE MPT's `mpiexec_mpt` is no longer supported (since mpt.2.23). Use the following method instead:

1. Load the latest MPT module:

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

2. Launch your program as follows:

```
% totalview mpiexec_mpt.real -a -np 16 ./a.out
```

If the MPI launcher fails to launch the executable and recommends setting MPI_SHEPHERD=true, then set that environment variable and try running TotalView again.

For more information, see the TotalView documentation.

The GNU Debugger (GDB) is available on HECC systems in the /usr/bin directory. GDB can be used to debug programs written in C, C++, Fortran, and Modula-a.

GDB can perform four main tasks:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what happened when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

Be sure to compile your code with -g for symbolic debugging.

GDB is typically used in the following ways:

- Start the debugger by itself:

  % gdb
  (gdb)

- Start the debugger and specify the executable:

  % gdb *your_executable*
  (gdb)

- Start the debugger, and specify the executable and core file:

  % gdb *your_executable core-file*
  (gdb)

- Attach gdb to a running process:

  %gdb *your_executable pid*
  (gdb)

At the prompt (gdb), enter commands such as break (for setting a breakpoint), run (to start running your executable, and step (for stepping into next line). Read **man gdb** to learn more on using gdb.

The Intel VTune Profiler (renamed from Amplifier starting with 2020.0 version) is an analysis and tuning tool that provides predefined analysis configurations to address various performance questions. Among them, the *hotspots* analysis type can help you to identify the most time-consuming parts of your code and provide call stack information down to the source lines.

The hotspots analysis type allows two data collection methods: (1) user-mode sampling and tracing collection, and (2) hardware event-based sampling collection. Both methods are supported on all current Aitken and Electra Intel processor types: Broadwell, Skylake, and Cascade Lake.

Note: For the AMD Rome and Milan nodes, the user-mode sampling and tracing collection is supported but the hardware event-based sampling collection is not.

The instructions below apply to using the user-mode sampling and tracing collection type, with a fixed sampling interval of 10 ms.

Setting Up to Run a Hotspots Analysis

Complete these steps to prepare for profiling your code:

1. Add the -g option to your usual set of compiler flags in order to generate a symbol table, which is used by VTune during analysis. Keep the same optimization level that you intend to run in production.
2. For MPI applications, build the code with the latest version of MPT library, as it will likely work better with Intel Vtune.
3. Start a PBS interactive session or submit a PBS batch job.
4. Load a VTune module in the interactive PBS session or PBS script, as follows:

```
module load vtune/2021.9
```

You can now run an analysis, as described in the next section.

Running a Hotspots Analysis

Run the vtune command line that is appropriate for your code, as listed below. Use the-collect (or -c) option to run the hotspots collection and the -result-dir (or -r) option to specify a directory.

Note: The vtune command replaces amplxe-cl, which was used prior to the 2020.0 version.

Running a Hotspots Analysis on Serial or OpenMP Code

To profile a serial or OpenMP application (for example):

```
vtune -collect hotspots -result-dir r000hs a.out
```

Data collected by VTune for the a.out application are stored in the r000hs directory.

Running a Hotspots Analysis on Python Code

To profile a Python application:

```
vtune -collect hotspots -result-dir r000hs /full/path/to/python3 python_script
```

Data collected by VTune while running the Python script will be stored in ther000hs directory.

Note: See Additional Resources below for more information about Python code analysis.

Running a Hotspots Analysis on MPI Code Using HPE MPT

To profile an MPI application using HPE MPT:

```
setenv MPI_SHEPHERD true
setenv MPI_USING_VTUNE true
mpiexec -np XX vtune -collect hotspots -result-dir r000hs a.out
```

Note: If your vtune run fails with MPT ERROR and a suggestion to set additional MPT environment variables, for example MPI_UNBUFFERED_STDIO, follow the suggestion and try again.

At the end of the collection, VTune generates a summary report that is written by default tostdout. The summary includes information such as the name of the compute host, operating system, CPU, elapsed time, CPU time, and average CPU utilization.

Data collected by VTune are stored and organized with one directory per host. Within each directory, data are further grouped into subdirectories by rank. As an example, for a 48-rank MPI job running on two nodes with 24 ranks per node, VTune generates two directories, r000hs.r587i0n0 and r000hs.r587i0n1, where each directory contains 24 subdirectories, (data.[*0-23*]).

To reduce the amount of data collected, consider profiling a subset of the MPI ranks, instead of all of them. For example, to profile

only rank 0 of the 48-rank MPI job:

```
mpiexec -np 1 vtune -collect hotspots -result-dir r000hs  a.out : -np 47 a.out
```

Viewing Results with the VTune GUI

Once data are collected, view the results with the VTune graphical user interface (GUI), `vtune-gui`. Since the PFEs do not have sufficient memory to run `vtune-gui`, run it on a compute node as follows:

1. On a PFE, run `echo $DISPLAY` to find its setting (for example, `pfe22:102.0`).
2. On the same PFE window, start a VNC session.
3. On your desktop, use a VNC viewer (for example, TigerVNC) to connect to the PFE.
4. On the PFE (via the VNC viewer window), request a compute node either through an interactive PBS session (`qsub -I -X`) or a reservable front end (`pbs_rfe`).
5. On the compute node, start `vtune-gui` with name of the result directory, for example, `r000hs`.

```
compute_node% module load vtune/2021.9
compute_node% vtune-gui r000hs
```

Generating a Report

You can also use the `vtune` command with the `-report` option to generate a report. There are several ways to group data in a report, as follows:

- To report time grouped by functions (in descending order) and print the results to `stdout`, or to a specific output text file, such as `output.txt`:

  ```
  vtune -report hotspots -r r000hs
  ```

  or

  ```
  vtune -report hotspots -r r000hs -report-output output
  ```

- To report time grouped by source lines, in descending order:

  ```
  vtune -report hotspots -r r000hs -group-by source-line
  ```

- To report time grouped by module:

  ```
  vtune -report hotspots -r r000hs -group-by module
  ```

You can also generate a report showing the differences between two result directories (such as from two different ranks or two different runs), or select different types of data to display in a report, as follows:

- To report the differences between two result directories:

  ```
  vtune -report hotspots -r r000hs -r r001hs
  ```

- To display CPU time for call stacks:

  ```
  vtune -report callstacks -r r000hs
  ```

- To display a call tree and provide CPU time for each function:

  ```
  vtune -report top-down -r r000hs
  ```

Additional Resources

See the following Intel VTune articles and documentation:

- hotspots Command Line Analysis
- hotspots Analysis for CPU Usage Issues
- Python Code Analysis

Gprof is a compiler-assisted performance profiler for C, Fortran, and Pascal applications running on Unix systems. You can use Gprof to help identify hotspots in your application where code optimization efforts may be most useful.

Gprof uses a hybrid of sampling and instrumentation, and provides the following information:

- The number of calls to each function and the amount of time spent there.
- Information about the caller-callee relationship.

Note: Gprof only measures the user code; it does not provide information on time spent in the kernel (such as system calls or I/O wait time).

The profiling data will be collected in a file called `gmon.out`, which will be generated at the end of a successful, uninterrupted run.

Gprof is available in the `/usr/bin` directory on all NAS compute systems. To use this tool, follow the instructions in the sections below.

Compiling and Linking to Enable Profiling with Gprof

To enable profiling with Gprof, add one of the options shown below when you compile your code:

- With Intel compilers, add the `-p` option (alternatively you can add `-pg`, which is deprecated, but still works).
- With PGI compilers, add the `-pg` option.
- With GNU compilers, add the `-pg` option. You might also have to use the `-O0` option, if you do not get meaningful profiling results when using higher levels of optimization.

Collecting Profiling Data

To collect profiling data, simply run your gprof-enabled executable the same way you would run a non-gprof-enabled executable. The data will be collected in a file called `gmon.out`.

OpenMP Applications

If you are using an OpenMP application, gprof does not generate per-thread profiling data. Only one `gmon.out` file is produced.

Note: If a file named `gmon.out` already exists in the directory, it will be overwritten.

MPI Applications

For MPI applications, if you use the Intel MPI library, no additional steps are required before you run your code. However, if you use the HPE MPT library, you also need to set the MPI_SHEPHERD variable as follows before you run the code; otherwise, the timing information will not be shown:

```
export MPI_SHEPHERD=true (bash)
setenv MPI_SHEPHERD true (csh)
```

Each MPI process will generate a profile with the same filename, `gmon.out`. These files will overwrite one another when they are written to a global filesystem. Therefore, to avoid this behavior and produce a profile with a distinct filename for each process, do:

```
export GMON_OUT_PREFIX=gmon.out (bash)
setenv GMON_OUT_PREFIX gmon.out (csh)
```

For example:

```
#PBS ...

module load comp-intel
module load mpi-hpe/mpt
export MPI_SHEPHERD=true
export GMON_OUT_PREFIX=gmon.out

mpiexec a.out
```

This operation will generate a file called `gmon.out.`*pid*, where *pid* is the process ID of an MPI process. With *N* ranks, you should get *N* such files, with filenames differing only in their *.pid* extensions. You can then analyze an individual `gmon.out.`*pid* file or several at the same time.

Generating ASCII Gprof Output

You can use the `gprof` command to convert binary data in `gmon.out` into a human-readable format.

```
gprof [options] executable_name [gmon.out] [ > analysis.output ]
```

There are two types of output: *flat profile* and *call graph.*

The flat profile shows how much time your program spent in each function, and how many times that function was called. This profile helps to identify hotspots in your application. Hotspots are shown at the top of the flat profile.

The call graph shows, for each function, which functions called it; which other functions it called; and how many times the calls occurred. The call graph also provides an estimate of how much time was spent in the subroutines of each function, which can suggest places where you might try to eliminate function calls that use a lot of time.

Commonly Used gprof Command Options

Some common options are described here. Read **man gprof** for more information.

- -p prints a flat profile. For example:

  ```
  gprof -p a.out gmon.out.1001 gmon.out.1002
  ```

- -q prints a call graph. For example:

  ```
  gprof -q a.out gmon.out.*
  ```

- -s sums up the information from multiple profiling data files and produces a file called `gmon.sum` for analysis. For example:

  ```
  gprof -s a.out gmon.out.*
  gprof a.out gmon.sum > analysis.output
  ```

- -b omits verbose texts that explain the meaning of all of the fields in the tables.

Performance Analysis

The Intel Application Performance Snapshot (APS) tool provides a quick overview of your application's performance on processor and memory usage, message passing interface (MPI), and I/O, as well as load imbalance between threads or processes. In addition to the performance metrics listed below, APS also provides suggestions for performance enhancement opportunities and additional Intel profiling tools you can use to get more in-depth analysis.

Note: Although the Intel APS tool can be used for any of the Aitken and Electra Intel Xeon processor types, it does not fully support the Sandy Bridge and Haswell processor types. APS cannot be used for the Aitken Rome and Milan processors.

Before You Begin

Note the following information:

- The Intel C/C++ or Fortran Compiler is not required, but is recommended. However, analysis of OpenMP imbalance is only available for applications that use the Intel OpenMP runtime library.
- There is no support for OpenMPI.
- Analysis of MPI imbalance is only available when you are using the Intel MPI Library version 2017 and later. (With the exception of the MPI imbalance metric, APS MPI analysis works for applications that use the NAS-recommended HPE MPT library.)

TIP: The NAS-developed MPIProf tool provides similar MPI analysis with MPI imbalance information included.

Running an APS Analysis

On NAS Intel x86_64 systems, APS is available via the vtune/2021.9 module. See aps --help for available APS options.

To run an analysis for serial and OpenMP applications:

```
module load vtune/2021.9
aps <options> ./a.out
```

To run an analysis for MPI applications:

```
module load vtune/2021.9

module load mpi-hpe

setenv MPI_SHEPHERD true
setenv MPI_USING_VTUNE true
mpiexec -np xx aps <options> ./a.out
```

TIP: The process/thread pinning tools mbind.x and omplace can be used with aps. Either of the following two methods may work:

- mpiexec -np xx aps mbind (or omplace) <mbind or omplace *options*> a.out
- mpiexec -np xx mbind (or omplace) <mbind or omplace *options*> aps <*options*> a.out

At the end of the run, APS provides a directory that contains the data collected during the analysis. The default name of the directory is aps_result_*yyyymmdd*.

WARNING: The directory will be overwritten if you run another APS analysis in the same directory on the same day.

To send the results to another directory, you can include --result-dir=*dir_name* or -r=*dir_name* option in the aps command line (where *dir_name* represents the directory name of your choice).

Viewing the Report

APS provides a text summary and an HTML file named aps_report_*yyyymmdd_hhmmss*.html (where *yyyymmdd_hhmmss* is the date and time when the HTML file is created). The HTML file contains the same information as the text summary, but also offers the following features when you open the file with a web browser:

- A description of each metric is shown when you hover your mouse over the metric name.
- Possible performance issues of your run are highlighted in red.
- Suggestions with links to Intel tools are provided to help with performance enhancement.

Note: NAS security policy prohibits using web browsers on NAS high-end computing systems. To view the HTML file, transfer it to your own workstation.

For Serial and OpenMP Applications

For serial or OpenMP applications, no action is required to generate the reports. At the end of the run, APS automatically generates the text summary (appended to your application's stdout) and the HTML file.

For MPI Applications

For MPI applications, you must generate the text summary (shown on your screen) and HTML file after the analysis completes, as follows:

aps --report=*dir_name*

where *dir_name* is the name of the directory created at the end of the analysis run.

If the summary report shows that your application is MPI-bound, run the following command to get more details about message passing, such as message sizes, data transfers between ranks or nodes, and time in collective operations:

aps-report <*options*> *dir_name*

See aps-report --help for available options.

You can also rerun the analysis after setting additional environmental variables. For example:

setenv MPS_STAT_LEVEL *n*
or
setenv APS_STAT_LEVEL *n*

where *n* is 2, 3 or 4 to get more detailed information on your application's MPI performance.

Note: If MPS_STAT_LEVEL (or APS_STAT_LEVEL) is set to 3 or 4, you must use mpi-hpe/mpt.2.18r160 or a later version when you run the analysis to avoid a segmentation fault (SEGV).

Quick Metrics Reference

Intel APS collects the metrics described in the following list.

Note: These descriptions are adapted from the Application Performance Snapshot User's Guide for Linux OS, where you can find additional details about each metric.

Elapsed Time
: Execution time of specified application in seconds.

SP GFLOPS
: Number of single precision giga-floating point operations (gigaflops) calculated per second. SP GFLOPS metrics are only available for 3rd Generation Intel Core processors, 5th Generation Intel processors, and 6th Generation Intel processors.

DP GFLOPS
: Number of double precision giga-floating point operations calculated per second. DP GFLOPS metrics are only available for 3rd Generation Intel Core processors, 5th Generation Intel processors, and 6th Generation Intel processors.

Cycles per Instruction Retired (CPI) Rate
: The amount of time each executed instruction took measured by cycles. A CPI of 1 is considered acceptable for high performance computing (HPC) applications, but different application domains will have varied expected values. The CPI value tends to be greater when there is long-latency memory, floating-point, or SIMD operations, non-retired instructions due to branch mis-predictions, or instruction starvation at the front end.

CPU Utilization
: Helps evaluate the parallel efficiency of your application. Estimates the utilization of all the logical CPU cores in the system by your application. 100% utilization means that your application keeps all the logical CPU cores busy for the entire time that it runs. Note that the metric does not distinguish between useful application work and the time that is spent in parallel runtimes.

MPI Time
: Time spent inside the MPI library. Values more than 15% might need additional exploration on MPI communication efficiency. This might be caused by high wait times inside the library, active communications, non-optimal settings of the MPI library. See MPI Imbalance metric to see if the application has load balancing problem.

MPI Imbalance
: Mean unproductive wait time per process spent in the MPI library calls when a process is waiting for data.

Serial Time
: Time spent by the application outside any OpenMP region in the master thread during collection. This directly impacts application Collection Time and scaling. High values might signal a performance problem to be solved via code parallelization or algorithm tuning.

OpenMP Imbalance
: Indicates the percentage of elapsed time that your application wastes at OpenMP synchronization barriers because of load imbalance.

Memory Stalls
: Indicates how memory subsystem issues affect the performance. Measures a fraction of slots where pipeline could be stalled due to demand load or store instructions. This metric's value can indicate that a significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. See the second level metrics to define if the application is cache- or DRAM-bound and the NUMA efficiency.

Cache Stalls
: indicates how often the machine was stalled on L1, L2, and L3 cache. While cache hits are serviced much more quickly than hits in DRAM, they can still incur a significant performance penalty. This metric also includes coherence penalties for shared data.

DRAM Stalls
: Indicates how often the CPU was stalled on the main memory (DRAM) because of demand loads or stores.

DRAM Bandwidth
: The metrics in this section indicate the extent of high DRAM bandwidth utilization by the system during elapsed time. They include:

- **Average Bandwidth:** Average memory bandwidth used by the system during elapsed time.
- **Peak:** Maximum memory bandwidth used by the system during elapsed time.
- **Bound:** The portion of elapsed time during which the utilization of memory bandwidth was above a 70% threshold value of the theoretical maximum memory bandwidth for the platform.

Some applications can execute in phases that use memory bandwidth in a non-uniform manner. For example, an application that has an initialization phase may use more memory bandwidth initially. Use these metrics to identify how the application uses memory through the duration of execution.

NUMA: % of Remote Accesses

In non-uniform memory architecture (NUMA) machines, memory requests missing last level cache may be serviced either by local or remote DRAM. Memory requests to remote DRAM incur much greater latencies than those to local DRAM. It is recommended to keep as much frequently accessed data local as possible. This metric indicates the percentage of remote accesses, the lower the better.

Vectorization

The percentage of packed (vectorized) floating point operations. The higher the value, the bigger the vectorized portion of the code. This metric does not account for the actual vector length used for executing vector instructions. As a result, if the code is fully vectorized, but uses a legacy instruction set that only utilizes a half of the vector length, the Vectorization metric is still equal to 100%.

Instruction Mix

This section contains the breakdown of micro-operations by single precision (SP FLOPs) and double precision (DP FLOPs) floating point and non-floating point (non-FP) operations. SP and DP FLOPs contain next level metrics that enable you to estimate the fractions of packed and scalar operations. Packed operations can be analyzed by the vector length (128, 256, 512-bit) used in the application.

- **FP Arith/Mem Rd Instr. Ratio:** This metric represents the ratio between arithmetic floating point instructions and memory read instructions. A value less than 0.5 might indicate unaligned data access for vector operations, which can negatively impact the performance of vector instruction execution.
- **FP Arith/Mem Wr Instr. Ratio:** This metric represents the ratio between arithmetic floating point instructions and memory write instructions. A value less than 0.5 might indicate unaligned data access for vector operations, which can negatively impact the performance of vector instruction execution. The metric value might indicate unaligned access to data for vector operations.

Additional Resources

- Getting Started with Application Performance Snapshot - Linux OS
- Application Performance Snapshot User's Guide for LinuxOS

You can use Intel Advisor to identify issues and help you with threading and vectorization optimization on the specific Intel processors that your Fortran, C, or C++ applications run on. The software, which is available as a module on HECC systems, provides recommended workflows for these two optimization areas.

Intel Advisor offers both a command-line interface (CLI) and a graphical user interface (GUI). We recommend that you use the CLI (advixe-cl) in a PBS batch job to collect data, and then use the GUI (advixe-gui) to analyze the result, as it provides a more comprehensive view than the CLI. However, please note that learning how to navigate the GUI will take some effort and time. For more information, see the Intel documentation, Intel Advisor GUI.

Analysis Types

Each Intel Advisor workflow involves a combination of some of the analysis types listed below. For detailed information about these workflows, see the Intel documentation, Getting Started with Intel Advisor, or access the CLI help page by running advixe-cl -h workflow.

Survey analysis
: Helps to identify loop hotspots and provides recommendations for how to fix vectorization issues.

Trip counts and flops analysis
: Counts the number of times loops are executed and provides data about floating-point operations (flops), memory traffic, and AVX-512 mask usage. This analysis requires more instrumentation and will perturb your application more. Thus, it takes longer time than the survey analysis.

Dependencies analysis
: Checks for real data dependencies in loops the compiler did not vectorize because of assumed dependencies.

Memory access patterns analysis
: Checks for memory access issues such as non-contiguous, or non-unit stride access.

Suitability analysis
: Predicts the maximum speed-up of your application, based on the inserted annotations and a variety of what-if modeling parameters you can experiment with. Use this information to choose the best candidates for parallelization with threads.

Roofline analysis
: Runs two analyses one by one: (1) the survey analysis, and (2) the trip counts and flops analysis. A roofline chart is then generated which plots an application's achieved floating-point performance and arithmetic intensity (AI) against the machine's maximum achievable performance.

  Note: For more information about running this analysis, see Roofline Analysis with Intel Advisor.

Compiling Your Code

At the minimum, you should include the following options when you compile your code:

```
-g -O2 (or higher)
-g -O2 (or higher) -qopenmp  (for OpenMP applications)
```

Depending on the processors your application runs on, you may consider adding an option that instructs the compiler to generate a binary that targets certain instruction sets. The options are listed in the following table.

Processor Type	Option
Cascade Lake, Skylake	-xCORE-AVX512
Broadwell	-xCORE-AVX2
Multiple auto-dispatch code paths	-axCORE-AVX512,CORE-AVX2,AVX -xSSE4.2

Collecting Data

Load the module and collect data:

```
module use -a /nasa/modulefiles/testing
module load advisor/
advixe-cl -collect=<string> [-action-option] [-global-option] [--] <target> [<target options>]
```

where:

- <string> is one of the following analysis types: survey, tripcounts, dependencies, map, or suitability
  Note: For non-MPI applications, <string> can also be roofline. For MPI applications, roofline analysis can only be done by performing survey and tripcounts in two separate steps.
- <target> is your executable.

For example:

```
advixe-cl -collect survey -project-dir my_result -- ./a.out
```

For information about action-option and global-option, run advixe-cl -h collect.

Analyzing the Result

The data collected are stored in a directory tree (for example, my_result), which you specify using an action-option called -project-dir.

Depending on the analysis type used in data collection, you might find the subdirectories hsxxx, trcxxx, dpxxx, and mpxxx in the my_result/e000 directory (for serial or pure OpenMP applications) or in the my_result/rank.n directory (for the nth-rank of an MPI application).

To view the result on a PFE, use either the Intel Advisor GUI (advixe-gui) or the CLI (advixe-cl).

Use the Intel Advisor GUI

Load the module and run advixe-gui:

```
module use -a /nasa/modulefiles/testing
module load advisor/
advixe-gui my_result
```

After the GUI starts, click **Show My Result**. In the main window, you can choose one of three panels: **Summary** (default view), **Survey & Roofline**, or **Refinement Reports**. All three are described below.

Summary
: Might show gigaflops count, AI, number of threads, loop metrics, vectorization gain/efficiency, top time-consuming loops, and platform information.

Survey & Roofline
: Might show the roofline chart, source code, assembly code, detailed info on flops, AI, efficiency, reason for no vectorization of the top-time-consuming loops.

Refinement Reports
: Might show memory access patterns report (1-stride, 0-stride, constant-stide, irregular stride) and dependencies report.

Use the Intel Advisor CLI

Load the module and run advixe-cl:

```
module use -a /nasa/modulefiles/testing
module load advisor/
advixe-cl -report=<string> [-action-option] [-global-option] [--] <target> [<target options>]
```

For information on <string>, action-option, and global-option, run: advixe-cl -h report

Note: If the job runs too quickly, there is not enough data collected to generate a report.

Additional Resources

- Getting Started with Intel Advisor
- Intel Advisor CLI
- Intel Advisor GUI
- Intel Advisor presentation by Intel engineer Jackson Marusarz
- Transforming Serial Code to Parallel Using Threading and Vectorization Part 1 and Part 2.

A roofline analysis helps you determine whether your application has achieved the best possible performance, limited by the machine capabilities. If not, you can explore the possibility of algorithm changes to improve performance.

Running the Analysis

The Intel Advisor tool is available as a module on HECC resources. To do a roofline analysis, load the module and run two analyses: survey and tripcounts, as follows:

```
module use /nasa/modulefiles/testing
module load mpi-hpe/mpt.2.18r160
module load advisor/2018
mpiexec -np x advixe-cl -collect survey -project-dir my_result -- ./a.out
mpiexec -np x advixe-cl -collect tripcounts -flop -project-dir my_result -- ./a.out
```

Note: Be sure to specify the same directory (*-project-dir*) for the survey and tripcounts analyses. The tripcounts run will take longer time than the survey run.

If these two runs are successful, you can start the Intel Advisor GUI and choose one of the ranks to analyze.

Reviewing the Results

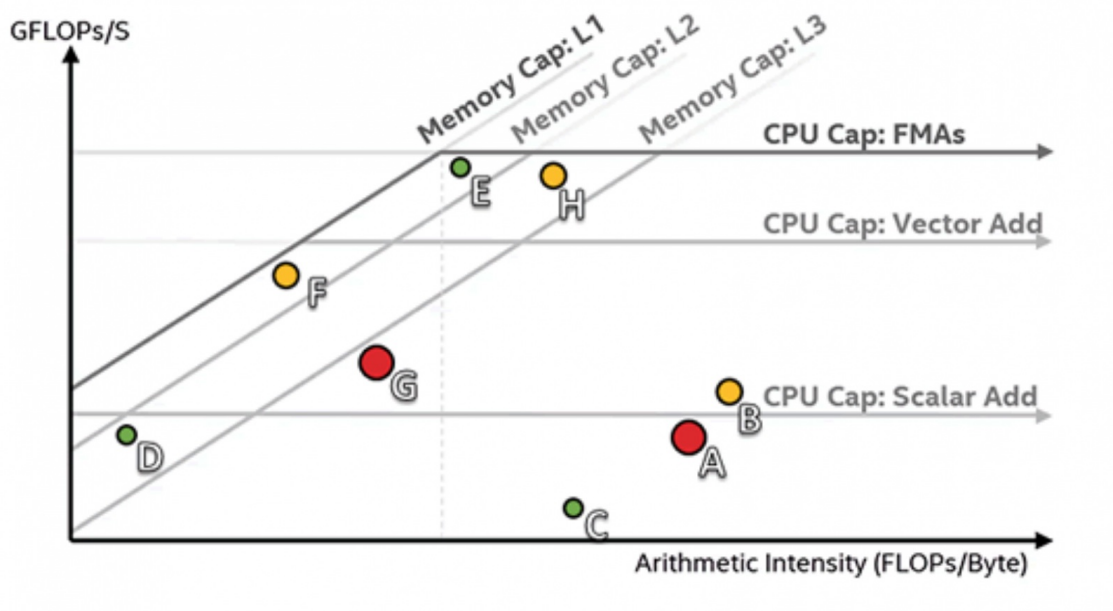
To start the GUI:

```
advixe-gui my_result
```

To see a roofline chart for the rank chosen, do one of the following:

- In the navigation panel, click the black icon (⚑) under **Run Roofline**.
- Click the **Survey & Roofline** panel in the main window, then click the vertical bar labeled **ROOFLINE**.

In the sample roofline chart shown below, the X axis is arithmetic intensity (measured in flops/byte) and the Y axis is the performance in Gflops/second, both in logarithmic scale:



Note: This sample chart is reproduced from the Intel documentation, Getting Started with Intel Advisor.

Before it collects data on your application, Intel Advisor runs benchmarks to measure the hardware limitations of your machine. It plots these on the chart as lines, called *roofs*. The horizontal lines represent the number of floating point computations of a given type your hardware can perform in a given span of time. The diagonal lines are representative of how many bytes of data a given memory subsystem can deliver per second.

In the chart, each dot is a loop or function in your application. The dot's position indicates the performance of the loop or function, which is affected by its optimization and its arithmetic intensity. The dot's size and color indicate how much of the total application time the loop or function takes. In the sample chart shown above, loops A and G (large red dots), and to a lesser extent loop B (yellow dot far below the roofs), are the best candidates for optimization. Loops C, D, and E (small green dots) and H (yellow dot) are poor candidates because they do not have much room to improve.

Important: Intel Advisor uses a cache-aware roofline model. In the classic roofline model, a kernel's arithmetic intensity would

change with problem size or cache usage optimization, because the byte count was based on DRAM traffic only. This is not the case in the cache-aware roofline model, where arithmetic intensity is a fixed value tied to the algorithm itself; it only changes when the algorithm is altered, either by the programmer or occasionally by the compiler.

Additional Resources

- Getting Started with Intel Advisor
- Intel Advisor Roofline
- Getting Started with Intel Advisor Roofline Feature
- Intel video: Introduction to Intel Advisor Roofline Feature
- Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architecture, 2009 (PDF)
- Cache-aware Roofline Model: Upgrading the Loft, 2013 (PDF)

MPIProf is a lightweight, profile-based application performance analysis tool that works with many MPI implementations, including HPE MPT. The tool gathers statistics in a counting mode via PMPI, the MPI standard profiling interface.

This tool is available for either the NAS x86_64 or aarch64 systems.

MPIProf can report profiling information about:

- Collective and/or point-to-point MPI functions called by an application (time spent, number of calls, and message size)
- MPI and POSIX I/O statistics
- Memory used by processes on each node
- Memory and power usage by applications
- Call path information of MPI calls (requires the -g compiler option and the -cpath mpiprof option)

In addition, mpiprof incorporates the mbind process binding capabilities through its various options. For more information about the options, read mpiprof -h after loading the scicon/mpiprof module.

There are two ways you can use MPIProf:

- The mpiprof profiling tool on the command line:

  mpiexec -n <N> mpiprof [-options] [-h|-help] a.out [args]

- MPIProf API routines

mpiprof Usage Example

Because the command-line method does not require changing or recompiling your application, we recommend this method for general use. To run mpiprof, load the modulefiles for your compiler, MPI library, and the mpiprof tool, then run the command line. For example:

```
module load comp-intel mpi-hpe/mpt
module load scicon/mpiprof
mpiexec -n 128 mpiprof -o mpiprof.out a.out
```

Note: If the -o option is not included, the profiling results are written to the mpiprof_stats.out file, or the <a.out>_mpiprof_stats.out file if the executable name is known.

For more details about using this tool, see the MPIProf user guide on a NAS system:

/u/scicon/tools/opt/mpiprof/default/doc/mpiprof_userguide.pdf

MPIProf was developed by NAS staff member Henry Jin.

MPInside, an MPI application performance analysis and diagnostic tool from HPE, is a lightweight program that can be used with thousands of ranks. The tool is profile-based via PMPI, the MPI standard profiling interface. An MPInside analysis can provide you with communication statistics (collective and point-to-point) as well as information that can help you diagnose communication issues such as:

- Slow MPI communications caused by non-synchronous send/receive pairs
- Load imbalance (compute time or message bandwidth) among ranks
- Reasons a code runs well with one MPI library or one platform, but not another

In addition to MPI communication profiling, MPInside provides the following optional features:

- MPI performance modeling
- I/O measurement

MPInside supports the full MPI 2.2 standard and the commonly used MPI libraries: HPE MPT, Intel MPI, and OpenMPI.

Note: Some features may be limited to HPE MPT.

Basic Usage

Most features of MPInside do not require instrumentation, recompiling, or relinking of your application. However, using the `-g` compiler flag is recommended.

The basic MPInside measurement produces a flat profile. No information is provided about the source lines that make the MPI calls.

Environment Setup

To set up your environment to use MPInside, load the MPInside module and an MPI library module. Set the `MPINSIDE_LIB` environment variable to match the MPI library module. For example:

```
module load MPInside/3.6.2

module load mpi-hpe/mpt
setenv MPINSIDE_LIB MPT (or IMPI, or OPENMPI)
```

Note: `MPINSIDE_LIB` defaults to `MPT`.

Communication Profiling

To perform an analysis using MPInside, run the following commands (where *XX* = the number of MPI processes):

```
mpiexec -np XX MPInside a.out
```

Note: The `MPInside` command is case-sensitive.

By default, MPInside measures the elapsed times of the MPI functions and collects message statistics. After the MPI application completes successfully, the measurement and the statistics data are reported in a text file called mpinside_stats. This output file contains five sections:

- The elapsed time of each function in seconds (broken down into columns for computation time and for various MPI function times, with one rank per row)
- Total megabytes (MBytes) sent by each rank
- Number of "send" calls by each rank
- Total MBytes received by each rank
- Number of "recv" calls by each rank

You can import the mpinside_stats file to an Excel spreadsheet and plot the data for visual analysis (for example, analysis of possible load imbalance or communication characteristics). You can also compute derived metrics, such as average message size, from the mpinside_stats file.

Additional Features

MPInside uses environment variables to set all of its options. These optional variables are documented in the **mpinside** man page.

Recommended Features

We recommend the following settings:

`MPINSIDE_LITE=1`
: Use when synchronization frequency is very high

MPINSIDE_CUT_OFF=0.0
Do not cut off reporting of any MPI functions
MPINSIDE_OUTPUT_PREFIX={run i.d.}
Change default report file name
MPINSIDE_PRINT_ALL_COLUMNS=1
Enable reporting of all columns (even if the values are zero) to allow easier comparisons between runs
MPINSIDE_PRINT_SIZ_IN_K=1
Print in KB units instead of MB

Advanced Features

The advanced features described below may also be useful.

MPINSIDE_SHOW_READ_WRITE

Measures the I/O time and includes several columns in the mpinside_stats output file indicating the time, number of Mbytes, and number of calls associated with the libc I/O functions. Note that the MPI function times and I/O times are subtracted from the total elapsed run times, and the result is reported as "Comput" time.

MPINSIDE_SIZE_DISTRI [T+]nb_bars[:*first-last*]

Prints a histogram of the request sizes distribution at the end of mpinside_stats for each rank specified in [:*first-last*]. If T+ is specified, size distribution time histograms are also printed. For example, setting MPINSIDE_SIZE_DISTRI to T+12:0-47 will print two histograms, as follows, for each rank (from 0 to 47) at the end of mpinside_stats:

1. Size distribution for sizes 0, 32, 64, 128, 256, 512, ..., up to 65536
2. Size distribution time histogram

MPINSIDE_CALLSTACK_DEPTH

Unwinds the stack up to the depth specified, and creates an mpinside_clstk.xxx file for each rank. The mpinside_clstk.xxx files can be post-processed with the utility MPInside_post -l to get the routine names and source line numbers. Use the-h option of MPInside_post to get more information.

MPINSIDE_EVAL_COLLECTIVE_WAIT

Checks whether a slow collective operation (such as MPI_Bcast or MPI_Allreduce) is caused by some ranks reaching the rendezvous point much later than others.

Setting this variable will put an MPI_Barrier and time it before any MPI collective operation. This assumes that the time of a collective operation is the sum of the time for all processors to synchronize plus the time of the actual operation (i.e, the physical transfer of data). In the mpinside_stats output file, the "b_*xxx*" column will give the MPI_Barrier time of the corresponding *xxx* MPI collective function, and the "*xxx*" column gives the remaining time.

MPINSIDE_EVAL_SLT

Checks whether a slow point-to-point communication is caused by late senders.

Setting this variable will measure the time the Sends arrived late compared to Recv or Wait arrivals. In the mpinside_stats output file, the column "w_*xxx*" (for example, w_wait or w_recv) represents the send-late-time (SLT) for the corresponding *xxx* function.

Note: The MPINSIDE_EVAL_COLLECTIVE_WAIT and MPINSIDE_EVAL_SLT variables are useful for identifying some problems that are intrinsic to an application, i.e., issues that cannot be remedied even with an ideal system (zero latency and infinite bandwidth).

References

- **mpinside** man page
- MPInside Reference Manual on the PFEs: /nasa/sgi/MPInside/3.6.2/doc/mpinside_3.5_ref_manual.pdf
- HPE Online Documentation: MPInside Reference Guide
- Publication: MPInside: a Performance Analysis and Diagnostic Tool for MPI Applications

The Intel VTune Profiler is an analysis and tuning tool that provides predefined analysis configurations to address performance questions.

This article provides basic information on several Intel VTune Profiler analysis types that examine various aspects of performance and identify potential benefits for your application from available hardware resources.

VTune Profiler Analysis Types

The following list provides a brief description of each analysis type that can be used with NAS resources.

performance-snapshot
: Get a quick snapshot of your application performance and identify next steps for deeper analysis. This analysis type became available starting with VTune Profiler Version 2020 Update 2. The Intel Xeon Sandy Bridge processors are not supported for this analysis.

hotspots
: Investigate call paths and find where your code is spending the most time; identify opportunities to tune your algorithms. This analysis type is in the Vtune Profiler's Algorithm analysis group.

threading
: Discover how well your application is using parallelism to take advantage of all available CPU cores. Best for visualizing thread parallelism on available cores, locating causes of low concurrency, and identifying serial bottlenecks in your code. This analysis type is in the Parallelism analysis group.

memory-consumption
: Analyze memory consumption by your application, its distinct memory objects, and their allocation stacks. This analysis type is in the Algorithm analysis group.

uarch-exploration
: Analyze CPU microarchitecture bottlenecks affecting the performance of your application. This analysis type is in the Microarchitecture analysis group.

memory-access
: Measure a set of metrics to identify issues related to memory access. Best for memory-bound applications to determine which level of the memory hierarchy is impacting your performance by reviewing CPU cache and main memory usage, including possible NUMA issues. This analysis type is in the Microarchitecture analysis group.

hpc-performance
: Analyze performance aspects of compute-intensive applications, including CPU and GPU utilization. Get information on OpenMP efficiency, memory access, and vectorization. This analysis type is in the Parallelism analysis group.

io
: Analyze utilization of IO subsystems, CPUs, and processor buses. This analysis type is in the Input and Output analysis group.

VTune Profiler Collection Types

There are two collection types:

- User-mode sampling and tracing collection.
- Hardware event-based sampling collection.

Each VTune analysis type uses one of the two collection types by default.

User-Mode Sampling and Tracing Collection

This collection method uses the operating system interrupts. No sampling drivers are needed. Default resolution is 10 millisecond (ms). The method works for the Intel Xeon nodes—PFEs and compute—and the AMD nodes.

Hardware Event-based Sampling (EBS) Collection

This collection method uses the counter overflow feature of the on-chip Performance Monitoring Unit (PMU). Default resolution is 1 ms. The is enabled for the Intel Xeon compute nodes only; it is not applicable to the PFEs (for security reasons) or the AMD processors (for hardware reasons).

There are two ways to facilitate EBS:

- Install and load the Intel sampling drivers
- Enable the Perf driverless collection, where the Linux Perf driver is used instead of the Intel sampling drivers

Each method is described below. Both require administrator privilege to configure them.

Install and Load the Intel Sampling Drivers

Different VTune Profiler versions may need different versions of the sampling drivers. A change in the kernel may result in the need to reinstall and reload the drivers. Here is one way to check whether the VTune sampling drivers have been loaded on the system for a particular version of VTune:

% <VTUNE_PROFILER_DIR>/sepdk/src insmod-sep -q

For example:

% /nasa/intel/Compiler/2021.4.0/vtune/2021.9.0/sepdk/src insmod-sep -q

Enable the Perf Driverless Collection

With this method, the Linux Perf driver is used instead of the Intel sampling drivers. The Intel VTune Profiler can use this driverless mode if the following requirements are satisfied:

- Access to core and uncore events. All hardware event-based collections in VTune Profiler use core PMU events. Some of them, such as the Memory Access and IO analysis types, also require access to uncore events that enable collecting metrics such as DRAM bandwidth, QPI/UPI bandwidth, PCI bandwidth, and others.
- Perf for Linux kernel 2.6.32 or later. PMU events are exposed by the Linux kernel through /sys/bus/event_source/devices/cpu and /sys/bus/event_source/devices/uncore_* directories.
- The value of /proc/sys/kernel/perf_event_paranid must be 0 or 1.

See Intel Processor Events Reference to learn more about core and uncore events.

Compared to the Intel sampling drivers method, there are limitations to using the Perf driverless collection mode. For example, in order to use Perf driverless collection mode with the uarch-exploration and memory-access analysis types, a system administrator must set the default limit of opened file descriptors (listed in the /etc/security/limits.conf file) to exceed 100 x *number_of_logic_CPU_cores*. More limitations are listed in the Intel documentation, Profiling Hardware Without Intel Sampling Drivers.

In all cases except for using the hotspots analysis type with stack collection, VTune Profiler uses the Intel sampling drivers if they are loaded. To make the VTune Profiler use the driverless Perf mode for sampling without stacks, create a custom analysis type and select the **Enable driverless collection** option in the GUI, or set the knob value in the command line to enable-driverless-collection=true as follows:

```
vtune -collect-with runsa -knob enable-driverless-collection=true \
-knob event-config=<event-list> <application>
```

For example,

```
-knob enable-driverless-collection=true -knob \
event-config=CPU_CLK_UNHALTED.CORE,CPU_CLK_UNHALTED.REF,INST_RETIRED.ANY
```

The enable-driverless-collection option is available starting with VTune 2019 Update 4.

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

Readiness of Running Various Analysis Types

To check whether VTune Profiler is ready for use on a system, you can use the vtune-self-checker.sh script, which can be found under <VTUNE_PROFILER_DIR>/bin64. This script runs a subset of the available analysis types against a matrix multiply application, and reports whether the analysis runs successfully. If EBS is used for the analysis, it also states whether the Intel sampling drivers or the Perf driverless mode is used.

Using the VTune Profiler Command Line

We recommend using the latest version of VTune on HECC systems, vtune/2021.9:

```
module load vtune/2021.9
```

To learn more about various analysis types, use:

```
vtune -h collect
```

or

vtune -h collect *name_of_analysis_type*

To run the VTune self checker (to confirm whether the correct drivers are installed and the system is set up properly):

```
vtune-self-checker.sh > output
```

When using the HPE MPT libraries with VTune, set the MPI_SHEPHERD and MPI_USING_VTUNE variables as follows.

Note: For some analyses, MPT may generate error message in your PBS output files stating that additional variables (such as MPI_UNBUFFERED_STDIO) need to be set.

```
(bash)
export MPI_SHEPHERD=true
export MPI_USING_VTUNE=true

(csh)
setenv MPI_SHEPHERD true
setenv MPI_USING_VTUNE true
```

For an example of running an analysis type and viewing the results, see Finding Hotspots in Your Code with the Intel VTune Command-Line Interface.

Summary Table

The information provided above was tested on NAS systems using vtune/2021.9 and is summarized in the following table.

Analysis Type	Additional Knobs Used	Analysis Group	Sampling Method	If EBS, Which Mode?	Unsupported Processor Types
performance-snapshot		N/A	EBS	Intel Driver	Sandy Bridge, AMD processors
hotspots		Algorithm	User-Mode	N/A	None
hotspots	-knob sampling-mode=hw	Algorithm	EBS	Intel Driver	AMD processors
hotspots	-knob sampling-mode=hw -knob enable-stack-collection=true	Algorithm	EBS	Perf[1]	AMD processors
threading		Parallelism	User-Mode	N/A	None
threading	-knob sampling-and-waits=hw	Parallelism	EBS	Intel Driver	AMD processors
threading	-knob sampling-and-waits=hw -knob enable-stack-collection=true	Parallelism	EBS	Intel Driver	AMD processors
memory-consumption		Algorithm	User-Mode	N/A	None
uarch-exploration		Microarchitecture	EBS	Intel Driver	Haswell[2], AMD processors
memory-access		Microarchitecture	EBS	Intel Driver	AMD processors
hpc-performance		Parallelism	EBS	Intel Driver	Sandy Bridge[3], AMD processors
io		Input and Output	EBS	Intel Driver	AMD processors
-collect-with runsa	-knob event-config=<event-list>	Custom Analysis	EBS	Intel Driver	AMD processors
-collect-with runsa	-knob event-config=<event-list> -knob enable-driverless-collection=true	Custom Analysis	EBS	Perf	AMD processors

[1] Starting with VTune 2019 Update 4, hotspots with EBS and stacks uses the Perf driverless mode by default even when the Intel sampling drivers are available.

[2] No L2, L3, DRAM bound (in % of clockticks), memory bandwidth or memory latency available for Haswell when hyperthreading is turned on.

[3] Vectorization analysis is limited for Sandy Bridge. Only metrics based on binary static analysis such as vector instruction set will be available.

Additional References

- VTune Analysis Types
- Intel VTune Profiler User Guide
- Intel VTune Profiler Performance Analysis Cookbook

The Intel VTune Profiler (renamed from Amplifier starting with 2020.0 version) is an analysis and tuning tool that provides predefined analysis configurations to address various performance questions. Among them, the *hotspots* analysis type can help you to identify the most time-consuming parts of your code and provide call stack information down to the source lines.

The hotspots analysis type allows two data collection methods: (1) user-mode sampling and tracing collection, and (2) hardware event-based sampling collection. Both methods are supported on all current Aitken and Electra Intel processor types: Broadwell, Skylake, and Cascade Lake.

Note: For the AMD Rome and Milan nodes, the user-mode sampling and tracing collection is supported but the hardware event-based sampling collection is not.

The instructions below apply to using the user-mode sampling and tracing collection type, with a fixed sampling interval of 10 ms.

Setting Up to Run a Hotspots Analysis

Complete these steps to prepare for profiling your code:

1. Add the -g option to your usual set of compiler flags in order to generate a symbol table, which is used by VTune during analysis. Keep the same optimization level that you intend to run in production.
2. For MPI applications, build the code with the latest version of MPT library, as it will likely work better with Intel Vtune.
3. Start a PBS interactive session or submit a PBS batch job.
4. Load a VTune module in the interactive PBS session or PBS script, as follows:

```
module load vtune/2021.9
```

You can now run an analysis, as described in the next section.

Running a Hotspots Analysis

Run the vtune command line that is appropriate for your code, as listed below. Use the-collect (or -c) option to run the hotspots collection and the -result-dir (or -r) option to specify a directory.

Note: The vtune command replaces amplxe-cl, which was used prior to the 2020.0 version.

Running a Hotspots Analysis on Serial or OpenMP Code

To profile a serial or OpenMP application (for example):

```
vtune -collect hotspots -result-dir r000hs a.out
```

Data collected by VTune for the a.out application are stored in the r000hs directory.

Running a Hotspots Analysis on Python Code

To profile a Python application:

```
vtune -collect hotspots -result-dir r000hs /full/path/to/python3 python_script
```

Data collected by VTune while running the Python script will be stored in ther000hs directory.

Note: See Additional Resources below for more information about Python code analysis.

Running a Hotspots Analysis on MPI Code Using HPE MPT

To profile an MPI application using HPE MPT:

```
setenv MPI_SHEPHERD true
setenv MPI_USING_VTUNE true
mpiexec -np XX vtune -collect hotspots -result-dir r000hs a.out
```

Note: If your vtune run fails with MPT ERROR and a suggestion to set additional MPT environment variables, for example MPI_UNBUFFERED_STDIO, follow the suggestion and try again.

At the end of the collection, VTune generates a summary report that is written by default tostdout. The summary includes information such as the name of the compute host, operating system, CPU, elapsed time, CPU time, and average CPU utilization.

Data collected by VTune are stored and organized with one directory per host. Within each directory, data are further grouped into subdirectories by rank. As an example, for a 48-rank MPI job running on two nodes with 24 ranks per node, VTune generates two directories, r000hs.r587i0n0 and r000hs.r587i0n1, where each directory contains 24 subdirectories, (data.[*0-23*]).

To reduce the amount of data collected, consider profiling a subset of the MPI ranks, instead of all of them. For example, to profile

only rank 0 of the 48-rank MPI job:

```
mpiexec -np 1 vtune -collect hotspots -result-dir r000hs  a.out : -np 47 a.out
```

Viewing Results with the VTune GUI

Once data are collected, view the results with the VTune graphical user interface (GUI), vtune-gui. Since the PFEs do not have sufficient memory to run vtune-gui, run it on a compute node as follows:

1. On a PFE, run echo $DISPLAY to find its setting (for example, pfe22:102.0).
2. On the same PFE window, start a VNC session.
3. On your desktop, use a VNC viewer (for example, TigerVNC) to connect to the PFE.
4. On the PFE (via the VNC viewer window), request a compute node either through an interactive PBS session (qsub -I -X) or a reservable front end (pbs_rfe).
5. On the compute node, start vtune-gui with name of the result directory, for example, r000hs.

```
compute_node% module load vtune/2021.9
compute_node% vtune-gui r000hs
```

Generating a Report

You can also use the vtune command with the -report option to generate a report. There are several ways to group data in a report, as follows:

- To report time grouped by functions (in descending order) and print the results to stdout, or to a specific output text file, such as output.txt:

  ```
  vtune -report hotspots -r r000hs
  ```

  or

  ```
  vtune -report hotspots -r r000hs -report-output output
  ```

- To report time grouped by source lines, in descending order:

  ```
  vtune -report hotspots -r r000hs -group-by source-line
  ```

- To report time grouped by module:

  ```
  vtune -report hotspots -r r000hs -group-by module
  ```

You can also generate a report showing the differences between two result directories (such as from two different ranks or two different runs), or select different types of data to display in a report, as follows:

- To report the differences between two result directories:

  ```
  vtune -report hotspots -r r000hs -r r001hs
  ```

- To display CPU time for call stacks:

  ```
  vtune -report callstacks -r r000hs
  ```

- To display a call tree and provide CPU time for each function:

  ```
  vtune -report top-down -r r000hs
  ```

Additional Resources

See the following Intel VTune articles and documentation:

- hotspots Command Line Analysis
- hotspots Analysis for CPU Usage Issues
- Python Code Analysis

Darshan is an open-source, lightweight high-performance computing I/O characterization toolkit developed at Argonne National Lab. Darshan instruments applications and captures calls to the POSIX, MPI-IO, STDIO, and HDF5 interfaces, and provides some limited information about PnetCDF. Statistics collected by Darshan include (but are not limited to) the number, size, and time spent on I/O calls, as well as the number and names of files, POSIX I/O access patterns (such as the access sizes and the numbers of total, consecutive, and sequential operations), timespan from first to last access, and estimated performance.

Darshan on NAS Systems

Two different Darshan builds are available. One was built on a PFE for MPI applications that use HPE MPT, and the other was built for non-MPI applications. Both are described below. Note that the two builds are installed in different locations.

Darshan for HPE MPI Applications

After loading an Intel compiler module and an HPE MPT module, this Darshan runtime was built with these options:

--prefix=/nasa/darshan/3.4.0_mpt
--with-log-path-by-env=DARSHAN_LOG_PATH
--enable-hdf5-mod
--with-hdf5=/nasa/hdf5/1.12.0_mpt
--with-jobid-env=PBS_JOBID CC=mpicc

Darshan for Non-MPI Applications

This Darshan runtime was built with these options:

--prefix=/nasa/darshan/3.4.0_non_mpi --with-log-path-by-env=DARSHAN_LOG_PATH
--with-jobid-env=PBS_JOBID --without-mpi CC=gcc

Instrumenting Your Application Using darshan-runtime

You can enable Darshan instrumentation either at compile/link time or at runtime. Runtime instrumentation is a quicker way to get started using Darshan, as it does not require modification to your existing executable; rather, it's done by setting the LD_PRELOAD environment variable for your PBS batch job as described below.

As the application runs, Darshan records statistics for each process. At the end of the run, Darshan collects, aggregates, compresses, and writes a log file to a location specified by the environment variable DARSHAN_LOG_PATH. The filename of the Darshan binary log will likely follow this format:

<USERNAME>_<BINARY_NAME>_<JOB_ID>_<DATE>_<UNIQUE_ID>_<TIMING>.darshan.

You can change the location and name of the Darshan log by setting the variable DARSHAN_LOGFILE=/path/*name*.darshan.

Using darshan-runtime for HPE MPI Applications

Use the following PBS script to run your HPE MPI application with Darshan.

Note: For MPI applications, the MPI_SHEPHERD environment variable must be set to true (highlighted in the script below) to avoid deadlock when preloading the Darshan shared library.

```
#!/bin/bash
#PBS -lselect=....

source /usr/local/lib/global.profile
module load mpi-hpe/mpt
#Note: Profiling HDF5 app with Darshan should NOT use hdf5 version 1.8.18_mpt
#Uncomment the following two lines if running HDF5 applications
#module use -a /nasa/modulefiles/testing
#module load hdf5/1.12.0_mpt

export DARSHAN_DIR=/nasa/darshan/3.4.0_mpt
export DARSHAN_LOG_PATH=/location_where_you_want_darshan_log_to_be_written_to
export MPI_SHEPHERD=true

mpiexec -genv LD_PRELOAD ${DARSHAN_DIR}/lib/libdarshan.so -np X mpi_executable
```

TIP: By default, Darshan aggregates statistics for files accessed by all ranks and collapses them into a single cumulative file record. If you want to retain the per-process information (which will result in a larger log file), set this variable:

```
export DARSHAN_DISABLE_SHARED_REDUCTION=1
```

Using darshan-runtime for Non-MPI Applications

Use the following PBS script to run your non-MPI application with Darshan.

Note: For non-MPI applications, the DARSHAN_ENABLE_NONMPI environment variable must be set to 1 (highlighted in the script, below).

```
#!/bin/bash
#PBS -lselect=....

source /usr/local/lib/global.profile
export DARSHAN_DIR=/nasa/darshan/3.4.0_non_mpi
export DARSHAN_LOG_PATH=/location_where_you_want_darshan_log_to_be_written_to
export DARSHAN_ENABLE_NONMPI=1
env LD_PRELOAD=${DARSHAN_DIR}/lib/libdarshan.so non-mpi_executable
```

Note: Darshan does not necessarily interfere with other profiling tools, such as the NAS-developed mpiprof tool. However, outputs generated by other profiling tools will likely also be counted in the Darshan binary log, which may confuse your analysis.

Darshan Extended Tracing

Darshan can perform eXtended Tracing (DXT) of the MPI-IO and POSIX I/O to provide details on each read or write operation issued by each rank, as well as which ranks are performing I/O and how long they are spending on I/O. A memory limit of 2 MebiByte (MiB) each for DXT_MPIIO and DXT_POSIX modules is set as default.

Note: DXT may result in longer runtime and higher memory overheads. It is also possible that the tracing log may be incomplete if a Darshan DXT module runs out of memory to store new record data.

To use DXT to trace all files, add one of the following lines in your PBS script:

```
export DARSHAN_MOD_ENABLE="DXT_POSIX,DXT_MPIIO"
or
export DXT_ENABLE_IO_TRACE=1
(This is a variable for earlier Darshan versions. It also works with 3.4.0.)
```

You can also select which items you want to include in or exclude from DXT (for example—files, ranks, or small I/O operations) by providing a trace configuration file or by setting certain DXT environment variables. For more information, read Darshan-runtime installation and usage documentation.

Analyzing Darshan Log Files with darshan-util Tools

The Darshan log file generated can be ported to and analyzed on systems wheredarshan-util is installed. You can perform the analysis on a PFE by loading either the HPE MPI version or the non-MPI version of the Darshan modulefiles, as shown below.

Note: Some of the darshan-util tools also require Perl, pdflatex, gnuplot, and epstopdf, which can be found in the pkgsrc/2023Q3 directory on NAS systems.

```
pfe% module use -a /nasa/modulefiles/testing

pfe% module load darshan/3.4.0_mpt
or
pfe% module load darshan/3.4.0_non_mpi

pfe% module load pkgsrc/2023Q3
```

After you load the Darshan and pkgsrc modulefiles, you can use one of the followingdarshan-util tools on a PFE.

darshan-job-summary.pl

The darshan-job-summary.pl tool processes the Darshan binary log (for example, *name*.darshan) and generates a multi-page PDF (*name*.darshan.pdf) containing graphs, tables, and performance estimates characterizing the I/O activity of the job:

```
pfe% darshan-job-summary.pl name.darshan
```

A sample PDF provided by the Darshan developers can be foundhere.

darshan-summary-per-file.sh

This script is similar to darshan-job-summary.pl except that it produces a separate PDF summary for every file accessed by the application:

```
pfe% darshan-summary-per-file.sh name.darshan output-dir
```

where *output-dir* is a directory to be created; it will contain the collection of PDFs (one PDF per file).

Note: Using this utility is not recommended if your application opens a large number of files.

darshan-parser

The darshan-parser tool extracts everything from a Darshan log and displays it in text format. It provides more information than darshan-job-summary.pl.

pfe% darshan-parser [options] *name*.darshan > *name*.txt

To show the available options, do: darshan-parser -h

To extract information for a specific file from a Darshan log containing statistics for many files, and produce a Darshan log for that file, complete these steps:

1. Show a list of filenames along with each file's record ID:
   darshan-parser --file-list *name*.darshan
2. Generate the log for the file you want:
   darshan-convert --file *a_record_id name*.darshan *a_file*.darshan.

darshan-dxt-parser

If you used DXT to generate a trace file (for example, *dxt_name*.darshan), use the darshan-dxt-parser tool to generate a text output:

pfe% darshan-dxt-parser [--show-incomplete] *dxt_name*.darshan > *dxt_name*.txt

The option --show-incomplete will display results even if the log is incomplete.

Known Issues

When you post-process the Darshan binary log for runs performed under a Lustre filesystem with progressive file layout, the following error will occur:

Error: failed to parse LUSTRE module record.

However, this error does not affect other I/O statistics (such as the POSIX or MPIIO stats) collected by Darshan for your application.

References

- Darshan-runtime installation and usage
- Darshan-util installation and usage

I/O characterization helps to identify opportunities for I/O related performance optimization for your application which, in turn, may also result in improved health of a shared filesystem used by many users.

The NAS in-house MPIProf tool is light-weight and easier to use compared to other I/O characterization tools. This article gets you started with I/O profiling of your application using MPIProf on NAS systems, and understanding some of the I/O statistics (such as overall, per-function, or per-rank) gathered in its text output.

After completing I/O profiling using MPIProf, you may choose to use DARSHAN to obtain finer-grain (per-file, per-call or time tracing) I/O information such as the name of the file(s), and when and where in the file the I/O occurs.

Basics on I/O Libraries

A user application performs I/O from and/or to the storage devices by calling some I/O libraries. These libraries can be divided into three layers:

- low-level layer

  The Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE to facilitate application portability. The POSIX I/O library with functions such as open, read, write, and close is at the lowest layer of all I/O libraries, and it interacts with the filesystem client and server software and hardware. All other I/O libraries above the POSIX I/O layer will eventually call the POSIX I/O functions.

- mid-level layer

  Examples of mid-level I/O libraries include those provided by the Fortran, C, C++ languages and the MPI I/O libraries.

  MPI I/O allows multiple processes to perform I/O on a file in either a collective or a non-collective way. The collective way requires all processes to participate when performing the I/O operations. The non-collective way lets each process do I/O independently.

  The names of MPI I/O functions start with MPI_File. Some examples are:

  - collective MPI I/O functions

    MPI_File_open,
    MPI_File_close,
    MPI_File_read_all, MPI_File_read_at_all, MPI_File_read_ordered
    MPI_File_write_all, MPI_File_write_at_all, MPI_File_write_ordered

  - non-collective MPI I/O functions

    MPI_File_read, MPI_File_read_at, MPI_File_read_shared
    MPI_File_write, MPI_File_write_at, MPI_File_write_shared

  Some implementations of collective MPI I/O provide performance improvement. This is done through reducing the number of times the filesystem is touched by transforming a large number of small, non-contiguous pieces of data in a file accessed by many MPI processes into fewer but larger, contiguous chunks of data handled by a smaller number of MPI processes.

- high-level layer

  There are many different high-level I/O libraries, for example, HDF5 and NETCDF. These libraries provide natural expressions of scientific data, which makes more sense to the scientists.

  The serial version of these high-level libraries calls the POSIX I/O functions directly while the parallel versions call the mid-level MPI I/O which in turn calls the POSIX I/O functions.

MPIProf does not instrument high-level I/O libraries such as HDF5 and NETCDF. It can selectively instrument the MPI communication, MPI I/O and/or POSIX I/O functions through its various options. After loading the mpiprof modulefile, you can read output from mpiprof -h for all available MPIProf command-line options. More detailed information is available through the output from mpiprof -help.

Usage

Load the appropriate compiler and MPI libraries used to build your application and the mpiprof modulefile before using MPIProf for I/O profiling, for example:

```
module load comp-intel
module load mpi-hpe/mpt
module load scicon/mpiprof
```

Choose one of the following three most commonly used methods for I/O profiling:

- Use the default:

  mpiexec -np X mpiprof a.out

This will instrument MPI communication functions, MPI I/O functions and POSIX I/O functions called by the application. However, it does not instrument the underlying POSIX I/O functions called from the MPI I/O functions.

- Use the -mio option:

  mpiexec -np X mpiprof -mio a.out

  This will instrument MPI communication functions, MPI I/O functions, and POSIX I/O functions called by the application. Furthermore, it will keep track of the underlying POSIX I/O functions called from the MPI I/O functions.

- Use the -ios option:

  mpiexec -np X mpiprof -ios a.out

  The -ios option does not instrument MPI communication functions. It instruments the MPI I/O functions and POSIX I/O functions called by the application. Furthermore, it will keep track of the underlying POSIX I/O functions called from the MPI I/O functions.

The MPIProf Statistics Output File

MPIProf generates a text output file at the end of the run. When the -o option is not used to specify an output file name, the profiling results are written to "mpiprof_stats.out" or "<a.out>_mpiprof_stats.out" if the executable name is known. The output file includes various statistics in multiple sections separated by the ==> header. A few of the sections are briefly described here:

==> Summary of this run

This section reports the execution time, total message and I/O size, communication and I/O rate, and some remarks about this run. For the execution time, since MPIProf does not profile the computation of the application, the execution time for computation is deduced by substracting communication time and/or I/O time from the total wall clock time.

MPIProf marks an application as I/O intensive when the effective I/O time is more than 10%.

Note: The effective I/O time is estimated by dividing the total I/O data size with the aggregated I/O rate from all active ranks. This is a more meaningful representation than the average I/O time for cases where I/O occurs only on a small number of ranks, such as rank 0.

Note: For runs with the mpiprof -ios option, since MPI communcation is not profiled, no communiciation time is reported. The reported computation time includes both communication time and computation time which are not separable from the gathered statistics.

TIP: Tip: Use this section to get a high-level characterization of the application.

==> Instrumented function list

See Appendix A of the MPIProf User Guide (/u/scicon/tools/opt/mpiprof/default/doc/mpiprof_userguide.pdf) for a complete list of all functions available for instrumentation.

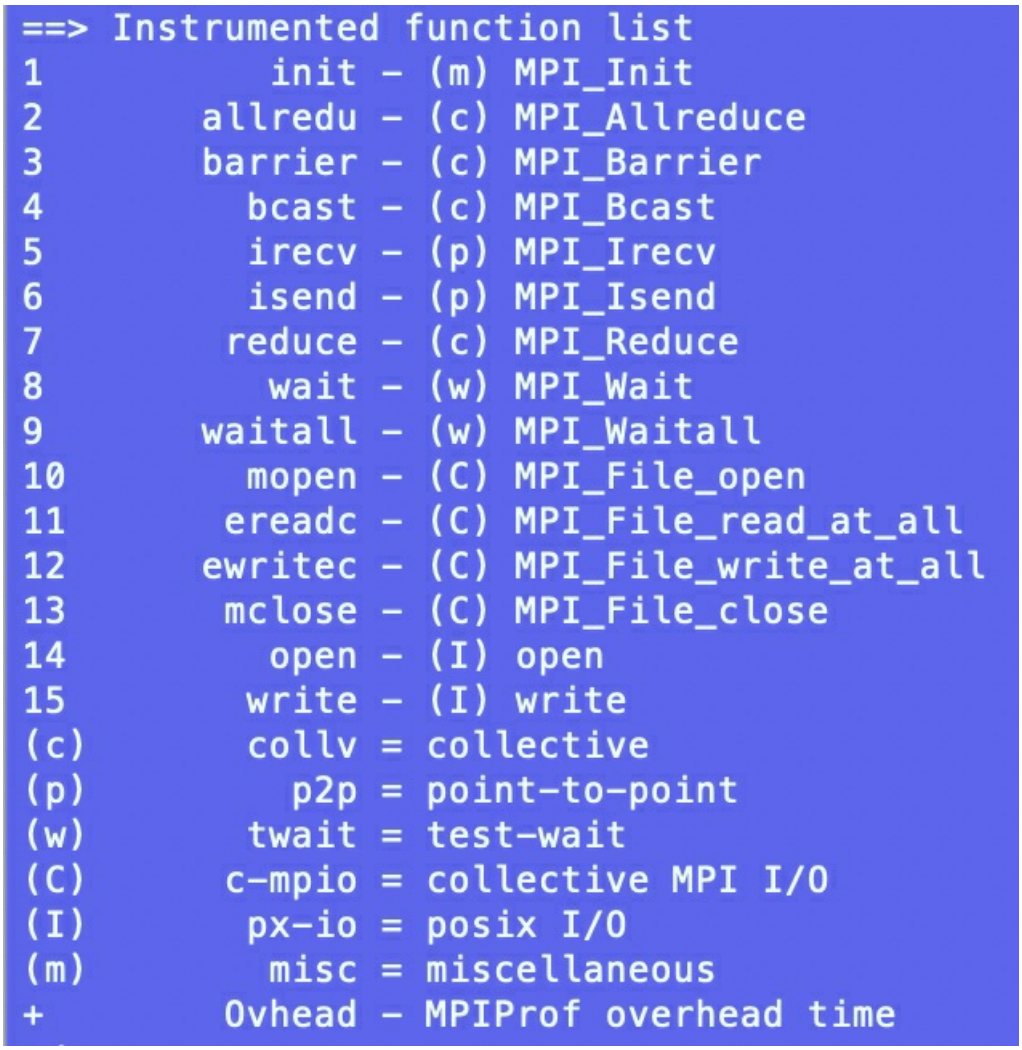
For illustration purposes, the lists below compare the instrumented function list using the above three methods for one of the NAS Parallel Benchmarks (NPB) BTIO benchmarks.

The instrumented functions are enumerated as shown in the first column of each list. The second column provides a short name of each instrumented function. For example, the short name for the function MPI_Init is init. Use of these short names allows a concise presentation of statistics in other sections of the output file. Each instrumented function is also denoted with one of { (c), (p), (w), (C), (N), (I), (<), (:), (m) } to indicate its nature, for example, (c), (p), and (w) for various MPI communication functions, (C) for collective MPI I/O functions, and (N) for non-collective MPI I/O functions. Be aware that MPIProf separates the statistics for (I) POSIX I/O functions not called by MPI I/O functions, (<) POSIX I/O from collective MPI I/O functions, and (:) POSIX I/O from non-collective MPI I/O functions.

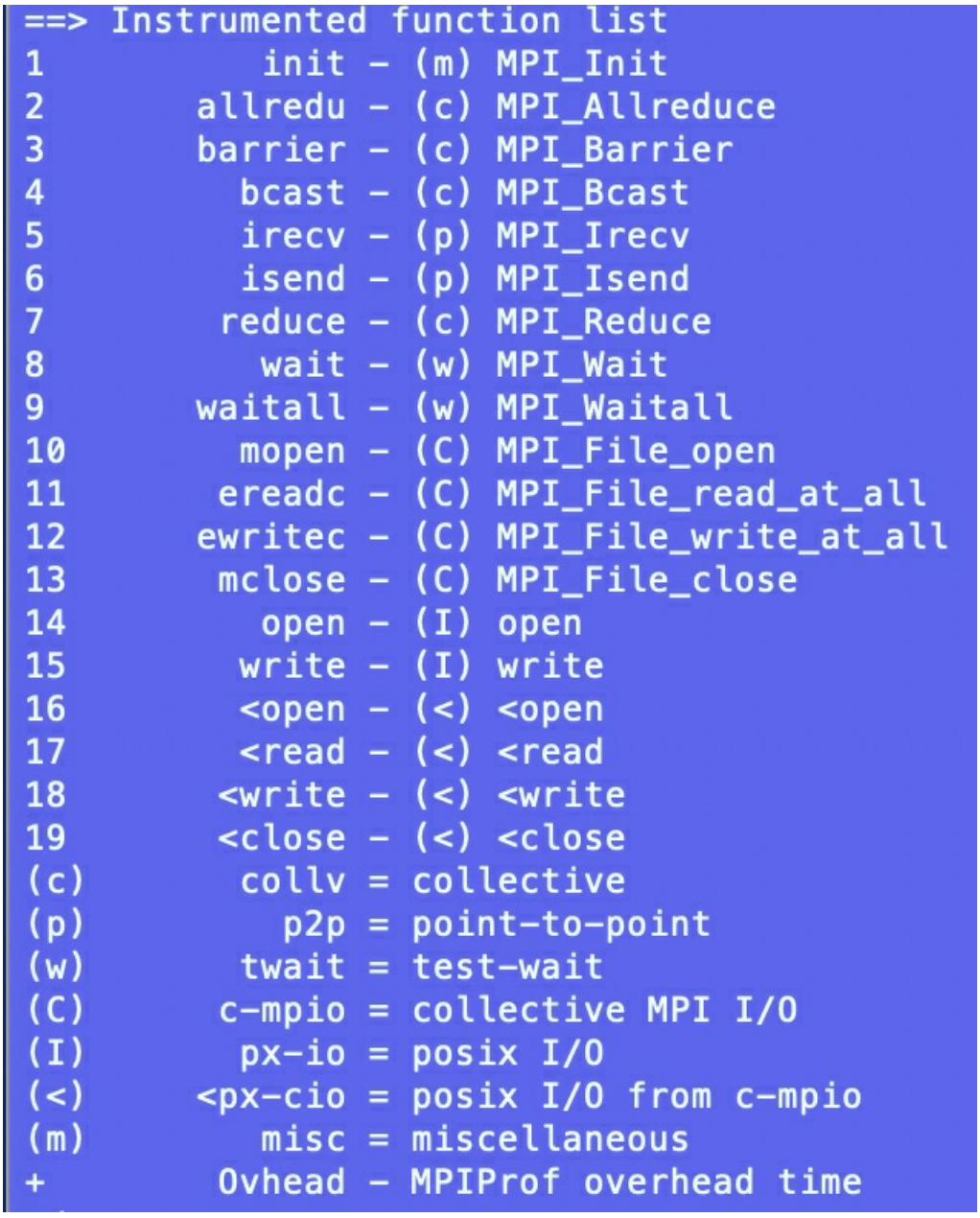
Note: The BTIO benchmark used for illustration does not call non-collective MPI I/O functions, so there are no functions denoted with (N) or (:) in the instrumented function list.

TIP: Use this section to find: what functions are instrumented in your run; the mapping of the short names and the original function names; and whether the function is an MPI communication, collective or non-collective MPI I/O function, or a POSIX I/O function.

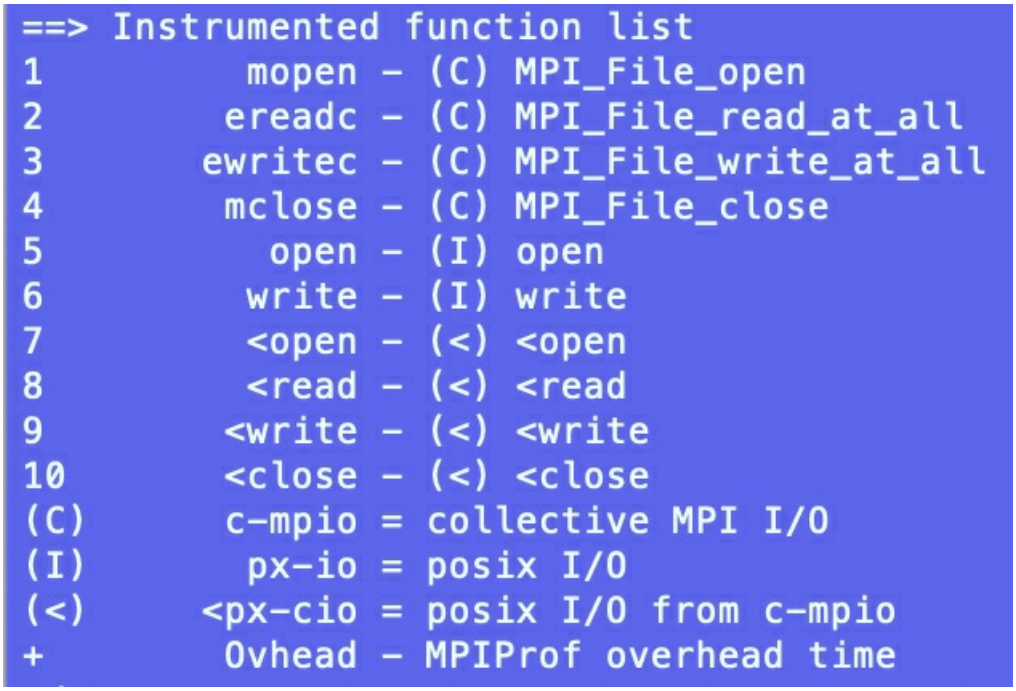
default instrumented function list



with -mio instrumented function list



with -ios instrumented function list



==> Per function summary

This section provides statistics (time spent and percentage time, number and sizes) of each instrumented function summing over all MPI ranks. For the runs with the -mio or -ios option, note that the time reported for <px-cio (that is, summing over all POSIX I/O from collective MPI I/O) and <px-nio (that is, summing over all POSIX I/O from non-collective MPI I/O) is already included in the time spent in the corresponding MPI I/O function groups (c-mpio and nc-mpio).

TIP: Use this section to identify which instrumented function(s) take most time in the application.

==> Message and data size histograms

For runs performed with -ios, the title of this section is "Data size historgrams" since MPI communication is not profiled.

This section provides the counts (summed over all MPI ranks) of send, receive, read and write in different size ranges. Note the unit of size is represented in decimal. For example, 16.384K is 16384 bytes.

TIP: High counts of large read and write sizes are more likely seen in applications with more effective I/O.

Multiple Sections Where Per-Rank Statistics are Reported

MPIProf reports per-rank statistics in multiple sections. For instance,

- ==> Summary - Timing in seconds
- ==> Summary - Function calls, node map, and memory bytes
- ==> Summary - Message sent or data written in bytes, rate in Mbytes/sec
- ==> Summary - Message received or data read in bytes, rate in Mbytes/sec

Some sections provide both per-rank and per-function statistics. For example,

- ==> Timing of functions in seconds
- ==> Counts of function calls
- ==> Message sent or data written in bytes
- ==> Message received or data read in bytes

For runs profiled with -ios, the 'Message sent', 'Message received' in the title are removed since no MPI communication is profiled.

In the ==> Summary - Timing in seconds section, total I/O time is broken down into time spent in c-mpio, nc-mpio, px-io, <px-cio, and <px-nio . As stated in the description of the ==> Per function summary above, time spent by each rank in <px-cio and <px-nio has already been included in the time spent in c-mpio and nc-mpio.

In the ==> Summary - Timing of functions in seconds section, the total I/O time is broken down into the specific MPI I/O function or POSIX I/O function being profiled. Note that the time reported for the underlying POSIX I/O calls from MPI I/O (such as <open, <read, <write, <close, <ncread, and <ncwrite) has already been included in the time reported for the MPI I/O functions (such as mopen, ereadc, ewritec, mclose, eread, and ewrite).

TIP: Examine the statistics reported for each rank in these sections to help identify which ranks perform the dominant POSIX I/O operations and whether the dominance is due to POSIX functions called or not called from MPI I/O functions.

NVIDIA Nsight Systems is a low-overhead statistical sampling profiler with tracing features to check overall performance of your GPU applications. The Feature Table in Get Started with Nsight Systems provides a glimpse of its capabilities, including but not limited to: profiling both the CPUs and the GPUs; investigating CPU-GPU interactions; tracing CUDA Application Programming Interface (API) and CUDA libraries; correlating API and GPU workload; and identifying GPU idle and sparse usage.

Nsight Systems provides both a command line interface (CLI), called nsys, and a graphical user interface (GUI), called nsys-ui. Using the CLI facilitates a quick way to collect performance data on a target (the CPU, the GPU, or both). The collected data can then be copied to any system where Nsight Systems is installed for analysis through the Nsight Systems CLI and/or GUI.

Note: Starting with Nsight Systems version 2024.4, no source code line number is available in the profiling data.

This article provides basic information primarily to help you use the Nsight Systems CLI on NAS systems. The examples use the Cabeus front ends (CFEs) for GPUs with x86_64-based CPUs (mil_a100).

gh200 nodes: If you are using GPUs with aarch64-based CPUs (gh200), please be aware of the following differences from use of the GPUs with x86_64-based CPUs:

- Access to these nodes is via the Cabeus Grace Hopper front ends, cghfe01 or cghfe02.
- The available nvhpc modulefiles may be different.
- The location of the Nsight Systems Python scripts is different.
- HPE MPT is unavailable.

Loading a nvhpc Modulefile

Nsight Systems is included in the NVIDIA HPC Software Development Kit (SDK). To access the kit, load one of the HPC SDK modulefiles on a NAS GPU compute node either through an interactive PBS job or a batch job script. The example below starts an interactive PBS session on a mil_a100 node through a qsub command from a CFE (cfe01 or cfe02), loads an nvhpc module, checks the version of nsys, prints its help page, and returns info about the system's profiling environment:

```
cfe% qsub -I -q gpu_devel -lselect=1:ngpus=1:model=mil_a100
a_GPU_node> source /usr/local/lib/global.profile  # if using bash
a_GPU_node> module use -a /nasa/modulefiles/testing
a_GPU_node> module load nvhpc/24.3
a_GPU_node> nsys --version   # or nsys -v
a_GPU_node> nsys --help      # or nsys -h
a_GPU_node> nsys status -e
```

Notes:

- All of the current NAS NVIDIA GPU nodes use a CUDA driver version =/> 535.104.12, which uses driver 535 or newer to improve GPU profiling stability.
- Use the latest available nsys version, which includes more features, by loading the latest available nvhpc modulefile.

Profiling an Application

- For Single or Multi-Threaded non-MPI Applications:

  nsys profile [profile switches] application [application options]

- For Single-Node MPI Applications:

  nsys profile [profile switches] mpirun [mpirun options] application [application options]

- For Multi-Node MPI Applications:

  mpirun [mpirun options] nsys profile [profile switches] application [application options]

All available profile switches (including their description, possible parameters and default setting) are listed in theNsight Systems User Guide or through the nsys -h profile or nsys profile -h help page.

Profiling a Non-MPI Application

Note: If you want to profile an MPI application, see the Profiling and Analyzing MPI Applications section.

The simplest way to profile an application is by running:

GPU_node> nsys profile *your_application*

This will launch *your_application*, start collection immediately, and end collection when*your_application* stops. It will also trace CUDA, OpenGL, NVTX, and OS runtime libraries APIs, and collect CPU sampling information and thread scheduling information. At the end, a profiling output file with the suffix .nsys-rep is generated.

Note: The NVIDIA Tools Extension (NVTX) library is a set of functions that a developer can use to annotate the source code for

profiling time line, customizing their appearance and assigning names to resources.

By default, the first profile output filename in a directory is report1.nsys-rep. A new output generated in the same directory will be named report2.nsys-rep, followed by report3.nsys-rep, and so on. You can add the option --output=filename (long notation, use of "=" is optional) or -o filename (short notation) to name the output file filename.nsys-rep. You can also add the option --force-overwrite=true or -f true to force overwriting existing *xxx*.nsys-rep with same output filename.

Instead of tracing the default APIs (CUDA, OPENGL, NVTX, OS Runtime), using -t or --trace switch allows you to choose what API(s) to trace. To keep the overhead small, try to narrow down to the set of features you specifically want to trace and only trace a smaller number of iterations. See the nsys profile -h help page for possible -t API options.

For example:

GPU_node> nsys profile --trace=cublas,openacc *your_application*

See the Example Single Command Lines section of the Nsight Systems User Guide for many more nsys profile examples.

Generating Profiling Statistics Reports

Nsight Systems includes many python scripts that read data from a SQLite database to generate various reports. You can find these scripts in (for example) the /nasa/nvidia/hpc_sdk/toss4/Linux_x86_64/24.3/profilers/Nsight_Systems/host-linux-64/reports directory. Therefore, the *xxx*.nsys-rep file needs to be exported to a *xxx*.sqlite file first before reports can be generated.

There are a few choices:

- **Profiling and generating statistics summary in one command**

  GPU_node> nsys profile --stats=true *your_application*

  This will profile your_application to create an .nsys-rep file and also generate an SQLite database (.sqlite). Furthermore, statistics summary reports in text format will be shown to the terminal (when using interactive PBS session) or the PBS standard output file (when using batch job).

- **Generating the SQLite database and the statistics summary in text format from an existing .nsys-rep file**

  nsys stats *xxx*.nsys-rep

  In this case, nsys will look for an accompanying SQLite file and use it. If no SQLite file exists, one will be exported and created.

- **Generating the statistics summary in text format from an existing .sqlite file**

  nsys stats *xxx*.sqlite

Notes:

- Post-processing an existing *xxx*.nsys-rep or *xxx*.sqlite file can be done on a GPU node or a NAS Cabeus front end (cfe01 or cfe02).
- By default, the statistics summary reports (such as nvtx_sum, osrt_sum, cuda_api_sum, etc.) generated are a subset of all available summary reports. See the --report section of the nsys stats -h printout for more information. In addition to summary reports, there are a few tracing reports (such as cuda_api_trace, cuda_gpu_trace, cuda_kern_exec_race, mpi_event_trace, etc.) one can generate. For a complete list, see nsys stats --help-report printout. To generate a tracing report for, say, cuda_api_trace, perform the following:

  nsys stats --report=cuda_api_trace *xxx*.nsys-rep
  or
  nsys stats -r cuda_api_trace *xxx*.sqlite

Identifying Optimization Opportunities

The nsys command option analyze can be used to get a high-level diagnosis of possible issues and suggestions for performance improvement in the areas of synchronization, data movement, overlap and parallelization.

nsys analyze *xxx*.nsys-rep
or
nsys analyze *xxx*.sqlite

Using the Nsight Systems GUI

For more detailed analysis of your application (such as looking into the timeline and correlation of events happening on different CPU cores, GPU devices, or threads), use the Nsight Systems GUI nsys-ui. For example,

nsys-ui *xxx*.nsys-rep

To learn and gain experience with nsys-ui, watch the videos listed in the reference section below.

Notes:

- Using a Cabeus front end (cfe01 or cfe02) through aVNC session is recommended for launching the Nsight Systems GUI.
- Beyond the use of Nsight Systems for application-level overall performance analysis, use Nsight Compute for detailed CUDA

kernel performance analysis and seek optimization opportunities in GPU utilization, kernel implementation, and memory access.

Profiling and Analyzing MPI Applications

Nsight Systems supports the use of OpenMPI. The description here should work for OpenMPI.

- Single-node MPI applications

  nsys profile [profile switches] mpirun [mpirun options] application [application options]

  will generate a single xxx.nsys-rep file, which contains profiling data from all MPI ranks.

- Multi-node MPI applications

  mpirun [mpirun options] nsys profile [profile switches] application [application options]

  will generate one *xxx*.nsys-rep for each MPI rank. For example, if the directory already has a report1.nsys-rep file, then by default profiling a 4-rank MPI run will generate report2.nsys-rep, report3.nsys-rep, report4.nsys-rep, and report5.nsys-rep.

  In order to better correlate each .nsys-rep file with the corresponding MPI rank, it is recommended to add the OpenMPI rank identifier OMPI_COMM_WORLD_RANK into the .nsys-rep files. For example,

  For bash:

  mpirun [mpirun options] nsys profile -o application_%q{OMPI_COMM_WORLD_RANK} application [application_options]

  For csh:

  mpirun [mpirun options] nsys profile -o application_%q\{OMPI_COMM_WORLD_RANK\} application [application options]

  TIP: Not all OpenMPI modulefiles at NAS are PBS-aware. When running across nodes, you may need to use:

  mpirun -np *X* -npernode *Y* --hostfile $PBS_NODEFILE -x PATH -x LD_LIBRARY_PATH nsys profile .... or mpirun -np *X* --map-by ppr:Y:node --hostfile $PBS_NODEFILE -x PATH -x LD_LIBRARY_PATH nsys profile ....

  where *X* is the total number of MPI ranks and *Y* is the number of MPI ranks on each node.

  This will generate application.0.nsys-rep, application.1.nsys-rep, application.2.nsys-rep, and application.3.nsys-rep for a 4-rank MPI run.

  The -o option of nsys profile also allows the use of %h and %p to include the hostname and process id into the .nsys-rep output filename.

  To analyze multiple .nsys-rep files from multiple MPI ranks using Nsight Systems GUI nsys-ui in a single 'Timeline View', perform the following:

  - Start nsys-ui
  - From the File pulldown, select New multi-report view. You should then see:

  - Click the icon indicated by the red arrow to select and add multiple *xxx*.nsys-rep files one by one. After the first file is added, the **Apply** button indicated by the blue arrow will be enabled.
  - After all files you want to analyze together have been added, click **Apply**.

  WARNING: There may be too much data collected from all of the MPI ranks. Consider profile only a single or a few MPI processes to reduce the amount of data to be analyzed. See Profile a Single Process or a Subset of Processes for more information.

Use of Nsight Systems with HPE MPT to profile GPU applications may work but is not guaranteed. To proceed, load anvhpc-nompi modulefile followed by mpi-hpe/mpt. For example:

```
source /usr/local/lib/global.profile    # if using bash
module use -a /nasa/modulefiles/testing
module load nvhpc-nompi/24.3
module load mpi-hpe/mpt
```

Besides setting the CUDA_VISIBLE_DEVICES variable properly, you will also need to set two environment variables before launching HPE MPT mpiexec.

For bash:

```
export MPI_SHEPHERD=1
export MPI_USE_CUDA=1
```

mpiexec -perhost *Y* -np *X* nsys profile -o application_%q{MPT_MPI_RANK} application ..

For csh:

```
setenv MPI_SHEPHERD 1
setenv MPI_USE_CUDA 1

mpiexec -perhost Y -np X nsys profile -o application_%q\{MPT_MPI_RANK\} application ..
```

Note: The option -perhost can be replaced with -ppn. Both -perhost and -ppn must come before the -np option.

References

- Get Started With Nsight Systems
- Nsight Systems User Guide - Profiling from the CLI
- CUDA Pro Tip: Generate Custom Application Profile Timelines with NVTX
- Video recording: Performance Analysis with the Nsight Systems Timeline
- Video recording: What the Profiling is Telling You: How to Get the Most Performance out of Your Hardware

Nsight Compute is a tool that collects metrics via hardware counters and software instrumentations for deep-dive profiling and guided performance analysis of the CUDA kernels. GPU architectures supported by Nsight Compute started with the NVIDIA Volta processors.

To understand the profile results from Nsight Compute, you need some knowledge of the GPU hardware.

TIP: Before using Nsight Compute, consider using Nsight Systems to analyze the overall performance on both the CPU and GPU of your application.

Nsight Compute has a command-line interface (CLI), called ncu, for profiling in non-interactive mode; and a graphical user interface (GUI), called ncu-ui, for profiling in interactive mode. This article only provides basic information on using ncu. For detailed information about ncu and ncu-ui, refer to NVIDIA documentation listed in the References section.

Depending on the nature of the application, some CUDA kernels may be launched multiple times during a run (for example, a kernel called within a loop). For each kernel launch, the name of the kernel function (e.g., vectorAdd_A in the example below) and the progress of data collection is shown in the standard output (STDOUT). To collect all requested profile information for the many metrics that are included for profiling, you might need to replay the kernels multiple times. When the collection is completed, it will also show the number of replay passes of that kernel. For example:

==PROF== Profiling "vectorAdd_A" - 0: 0%....50%....100% - 46 passes

WARNING: Collecting performance data using ncu can incur significant runtime overhead on the application, as discussed here.

Nsight Compute is included in the CUDA Toolkit, which in turn is included in the NVIDIA HPC software development kit (SDK) distributions (for example, NVHPC). The latest available version of ncu is recommended, as it likely provides more features for profiling.

Command-Line Options

Some of the ncu options to consider when profiling are listed below. For all options and more details, read output from the ncu -h command after loading a nvhpc module.

- -o [--export] *report_name*

  Saves profile results in a binary report file called *report_name*.ncu-rep. For example, if you use -o my_ncu_output, the profile results are saved in the my_ncu_output.ncu-rep file. After profiling, the binary report file can be imported for analysis via ncu or ncu-ui. Without this option, a profile result file is created temporarily with some results written in text format to STDOUT. The profile result file is then removed.

- -k [--kernel_name] *kernel_name*

  Chooses a specific kernel, with exact match to kernel_name, to profile. For example, -k vectorAdd_A, which is an exact match to the kernel listed after ==PROF== shown above.

- -c [--launch-count] *a_number*

  Limits the number of collected profile results to *a_number*. For example, -c 2.

- -s [--launch-skip] *a_number*

  Skips a given number of kernel launches before data collection starts. The default is 0.

- --set *set_identifier*

  Chooses among the pre-defined profiling sets: basic, detailed, full, nvlink, pmsampling, roofline . The default is basic, which is less comprehensive and incurs less profiling overhead in runtime and memory usage.

  The basic set includes four sections: LaunchStats, Occupancy, SpeedOfLight, WorkloadDistribution . The detailed set and the full set include the same four sections as in the basic set, plus additional sections. Run the ncu --list-sets command to see what sections are included in each pre-defined set.

  Note: The WorkloadDistribution section is not available in nvhpc/24.3 and earlier.

- --section *section_identifier*

  Chooses a specific section to profile. For example, --section ComputeWorkloadAnalysis.

  Run the ncu --list-sections command to see the list of available sections. Each section includes a group of metrics to be profiled, as pre-defined in Nsight_Compute_installation_location/sections/section-identifier.section.

- --metrics *arg*

  Specify all metrics to be profiled, separated by comma. For example, --metrics sm__cycles_active.avg,sm__cycles_elapsed.sum.

  Run the ncu --list-metrics and ncu --query-metrics commands on a GPU node where you intend to run your application to see the

available metrics.

Profile Your Application

The examples shown in this article use the Cabeus front ends (CFEs) for the GPUs with x86_64 based CPUs (mil_a100).

For GPUs with aarch64-based CPUs, such as gh200, please be aware of the following:

- You should use one of the following front ends: cghfe01 or cghfe02.
- The module path and available nvhpc modulefiles may be different.

In order to avoid version incompatibility, before profiling, it is recommended to build your application using the compilers, math, and CUDA libraries included in the same nvhpc version you plan to use ncu for profiling.

Within a PBS job, run the module use and module avail commands as shown below to find all available nvhpc versions, then load the latest version. The following example assumes the latest version is 24.7.

```
CFE% qsub -I -q gpu_devel -lselect=1:ngpus=4:model=mil_a100
a_GPU_node> source /usr/local/lib/global.profile   #if using bash
a_GPU_node> module avail nvhpc
a_GPU_node> module load nvhpc/24.7    # with OpenMPI version 4
or
a_GPU_node> module load nvhpc/24.7_openmpi3  # with OpenMPI version 3
```

For Single-Process Applications

Without any additional options, ncu launches and instruments your application, for example, a.out, and collects the basic set of profile results of all CUDA kernels in the application launched during execution.

```
ncu -o report_name [other ncu options] a.out [a.out options]
```

An output file *report_name*.ncu-rep containing the profile results is saved.

For Single-Node Multi-Process Applications

The following example will save profile results from all ranks into a single report file, *report_name*.ncu_rep.

```
ncu --target-processes all -o report_name [other ncu options] \
mpirun [mpirun options] a.out [a.out options]
```

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

For Multi-Node Multi-Process Applications

The following will save profile results with one file per rank, e.g., *report_name*_0.ncu_rep, *report_name*_1.ncu_rep, and so on.

```
For bash,
mpirun [mpirun options] ncu -o report_name_%q{OMPI_COMM_WORLD_RANK} \
[other ncu options] a.out [a.out options]

For csh,
mpirun [mpirun options] ncu -o report_name_%q\{OMPI_COMM_WORLD_RANK\} \
[other ncu options] a.out [a.out options]
```

Note: The mpirun command lines above are too long to be formatted as one line, so they are broken with a backslash (\).

Profiling Just One Rank

NVIDIA suggests using the following script, wrap.sh, for profiling just one rank (e.g., rank 0):

(For bash):

```
#!/bin/bash
if [[ $OMPI_COMM_WORLD_RANK == 0 ]]; then
   ncu -o report-name_${OMPI_COMM_WORLD_RANK}  --target-processes all "$@"
else
   "$@"
fi
```

(For csh):

```
#!/bin/csh
if ( $OMPI_COMM_WORLD_RANK == 0 ) then
ncu -o report-name_${OMPI_COMM_WORLD_RANK} --target-processes all $*
else
$*
endif
```

To profile and then run the script, do:

```
mpirun [mpi options] ./wrap.sh a.out [a.out options]
```

View Profile Report

This step can be performed on either a GPU compute node or a front-end node where Nsight Compute is available.

To get a quick text summary from the binary report, do:

```
ncu --import report_name.ncu-rep
or
ncu -i report_name.ncu-rep
```

If the profiling was done with the basic set, this will generate four sections, including GPU Speed Of Light Throughput, Launch Statistics, Occupancy, and GPU and Memory Workload Distribution, for each profiled kernel. The output may also provide additional explanation on the profile results and optimization advice based on rules created by NVIDIA engineers.

You may also add the --print-details all option to show more details.

```
ncu --print-details all --import report-name.ncu-rep
```

Use of Nsight Compute ncu-ui may provide more comprehensive analysis, including the source line information if the application was compiled with -gpu=lineinfo for nvfortran, nvc, and nvc++ and --generate-line-info or -lineinfo for nvcc. However, the front-ends cfe[01,02] and cghfe[01,02] do not yet have the required Qt software installed to enable the use of ncu-ui. If you have Nsight Compute and Qt installed on your local workstation, you may copy your report-name.ncu-rep there for analysis.

References

- NVIDIA Nsight Compute
- Nsight Compute Documentation (latest); includes chapters on Kernel Profiling Guide, Nsight Compute, Nsight Compute CLI, and more
- Nsight Compute CLI User Manual (PDF, version 2023.3.1)
- Intro to NVIDIA Nsight Compute (video)
- Speed of Light Analysis with Nsight Compute (video)
- Introduction to Kernel Performance Analysis with NVIDIA Nsight Compute (video)
- Memory Analysis with NVIDIA Nsight Compute (video)

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.

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.

File Transfer

If you are moving large files, use the shiftc command instead of cp or scp. An online NAS service can help diagnose your remote network connection issues, and our network experts can work with your specific file transfer problems.

For fastest file transfer between NAS /nobackup and Lou, log into Lou and use shiftc, cxfscp, or mcp. A simple cp or tar will also work, but at slower speeds.

Moving large amounts of data efficiently to or from NAS across the network can be challenging. Often, minor system, software, or network configuration changes can increase network performance an order of magnitude or more.

If you are experiencing slow transfer rates, try these quick tips:

- NAS /nobackup are mounted on Lou, enabling disk-to-disk copying, which should give the highest transfer rates. You can use the shiftc, cp, or mcp commands to copy files or even make tar files directly from NAS /nobackup to your Lou home directory.
- If using the scp command, make sure you are using OpenSSH version 5 or later. Older versions of SSH have a hard limit on transfer rates and are not designed for WAN transfers. You can check your version of SSH by running the command ssh -V.
- For large files that are a gigabyte or larger, we recommend using shiftc. This application allows for transferring simultaneous streams of data and, when used without the --secure option, doesn't have the overhead associated with encrypting all the data (authentication is still encrypted).
- Another reliable option for large file transfers is through the Shift transfer tool, which includes options specific to the NAS environment, such as checking to see whether files residing on Lou are also on tape.

One-on-One Help

If you would like further assistance, contact the NAS Control Room at support@nas.nasa.gov, and a network expert will work with you or your local administrator one-on-one to identify methods for increasing your transfer rates.

To learn about other network-related support areas see End-to-End Networking Services.

On Lou, commands that should finish quickly may occasionally take a long time. This problem is usually due to slow retrieval of files from tape to disk.

When you run the ls command on Lou, the output shows all your Lou files on disk. However, most of the files are actually written to tape using the Data Migration Facility (DMF).

One reason for slow file retrieval is that for some multiple file transfers—for example, if you do an scp transfer with a list of files—Linux feeds each file to DMF one at a time, and DMF does not deal well with retrieving one file at a time from a long list of files. This means that the tape(s) containing the files is constantly being loaded and unloaded, which is very slow (and is bad for the tape and tape drives). As the list of files gets longer (through the use of "*" or moving a "tree" of files), the problem grows to where it can take hours to transfer a set of files that would only take a few minutes if they were on disk. This can be particularly problematic when several people do these types of file transfers at the same time.

The methods described below can help you avoid these problems.

Note: For more information about the commands in this section, see Data Migration Facility (DMF) Commands.

Optimizing File Retrieval

You can fetch files to disk as a group by running the dmget command before running your file transfer. dmget reads the tape once and gets all the requested files in a single pass.

Run dmget on the same list of files you are about to transfer. Then, after the dmget operation completes, you can transfer the files using scp/ftp/cp as you had originally intended. Or, you can put dmget in the background and run your transfer while dmget is working. If any files are already on disk, dmget sees this and doesn't try to get them from tape.

DMF also provides the dmfind command, which enables you to walk a file tree to find offline files to give to dmget.

Note: Be sure you are in the correct directory before running dmfind. Use the pwd command to determine your current directory.

Please check to make sure too much data isn't brought back online at once, either by using du with the --apparent-size option or by using /usr/local/bin/dmfdu. For example:

```
lou% /usr/local/bin/dmfdu filename
   filename
         13 MB regular          340 files
       1114 MB dual-state        1920 files
      74633 MB offline          2833 files
         13 MB small            340 files
      75761 MB total            5093 files
```

File transfer rates vary depending on the load on the system and how many users are transferring files at the same time. Typically, scp transfers between Lou and NAS compute systems on the /nobackup file system run between 30-120 MB/s for files larger than 100 MB, using the 10-gigabit network interface.

Example 1:

```
lou%  dmget *.data &
lou%  scp -qp *.data  myhost.jpl.nasa.gov:/home/user/dir_name
```

Example 2:

```
lou%  dmfind  /u/username/FY2000 -state OFL -print | dmget &
lou%  scp -rqp /u/username/FY2000 hostname:/nobackup/username/dir_name
```

You can see the state of a file by running dmls -l instead of ls -l.

Maximum Amount of Data to Retrieve Online

The online disk space for Lou is considerably smaller than its tape storage capacity, and it is impossible to retrieve all files to online storage at the same time. Using the Shift tool for file transfers automatically ensures that files on Lou are retrieved in batches and released afterwards so there is no need to manually split up the transfer. If you do not use Shift, however, then you should confirm whether there is enough disk space before you retrieve a large amount of data.

The df command shows the amount of free space in a filesystem. The Lou script dmfdu reports how much total (online and offline) data exists in a directory. To use dmfdu, simply cd into the directory you want to check, and execute the script.

If you would like to know the total amount of data under your home directory on Lou, simply cd to your home directory and use dmfdu *, which will show use for each file or directory.

Lou's archive filesystems are between 254 TB and 1.3 PB in size, but the available space typically floats between 10% to 30%. In

Example 3, 29% of space is unused.

It is best to retrieve no more than 10 TB at a time. As shown in Example 3, it is best to release the space (dmput -r) after using the retrieved files (scp, edit, compile, etc), then retrieve the next group of files, use them, and release the space again, and so on.

Example 3:

To retrieve one directory's data from tape, copy the data to a remote host, release the data blocks, and then retrieve more data from tape:

```
lou% df -lh .
Filesystem             Size  Used Avail Use% Mounted on
/dev/cxvm/la1          1.3P  1.1P 225T  83%  /lou/la1


lou% dmfdu project1 project2
project1
         2 MB regular            214 files
        13 MB dual-state           1 files
   2229603 MB offline            101 files
         2 MB small              214 files
   2229606 MB total              315 files

project2
         7 MB regular            245 files
      4661 MB dual-state          32 files
   2218999 MB offline             59 files
         7 MB small              245 files
   2223668 MB total              336 files


lou% cd project1

lou% dmfind . -state OFL -print | dmget &

lou% scp -rp /u/username/project1 remote_host:/nobackup/username

##(Verify that the data has successfully transferred)

lou% dmfind . -state DUL -print | dmput -rw

lou% df -lh .

lou% cd ../project2

lou% dmfind . -state OFL -print | dmget &

lou% scp -rpq /u/username/project2 remote_host:/nobackp/username

lou% dmfind . -state DUL -print  | dmput -rw
```

You can maximize your wide-area network bulk data transfer performance by tuning the TCP settings on your local host. This article shows some common configuration tasks for enabling high-performance data transfers on your system.

Note that making changes to your system should only be done by a lead system administrator or someone who is authorized to make changes.

Linux

1. Edit the file sysctl.conf located under the /etc directory, and add the following lines:

   ```
   net.core.wmem_max = 4194304
   net.core.rmem_max = 4194304
   ```

2. Then have them loaded by running `sysctl -p`

Windows

We recommend using a tool like Dr. TCP.

1. Set the "Tcp Receive Window" to at least 4000000
2. Turn on "Window Scaling," "Selective Acks,"and "Time Stamping"

Other options for tuning Windows XP TCP are the SG TCP Optimizer or using Windows Registry Editor to edit the registry, but the latter is only recommended for Windows users who are already familiar with registry parameters.

Mac OS 10.4

Note that these changes require root access.

In order to allow the Mac operating system to retain the parameters after a reboot, edit the following variables in /etc/sysctl.conf:

1. Set maximum TCP window sizes to 4 megabytes

   ```
   net.inet.tcp.sendspace= 4194304
   net.inet.tcp.recvspace= 4194304
   ```

2. Set maximum Socket Buffer sizes to 4 megabytes

   ```
   kern.ipc.maxsockbuf= 4194304
   ```

Mac OS 10.5 and Later

Use the sysctl command for the following variable:

```
sysctl -w net.inet.tcp.win_scale_factor=8
```

If you follow these steps and are still getting less than your expected throughput, please contact the NAS network group at support@nas.nasa.gov (attn: Networks). We will work with you on tuning your system to optimize file transfers.

You can also try the additional steps outlined in the related articles listed below.

Optional Advanced Tuning for Linux

This document describes additional TCP settings that can be tuned on high-performance Linux systems. This is intended for 10-Gigabit hosts, but can also be applied to 1-Gigabit hosts. The following steps should be taken in addition to the steps outlined in TCP Performance Tuning for WAN transfers.

Configure the following /etc/sysctl.conf settings for faster TCP

1. Set maximum TCP window sizes to 12 megabytes:

   ```
   net.core.rmem_max = 11960320
   net.core.wmem_max = 11960320
   ```

2. Set minimum, default, and maximum TCP buffer limits:

   ```
   net.ipv4.tcp_rmem = 4096 524288 11960320
   net.ipv4.tcp_wmem = 4096 524288 11960320
   ```

3. Set maximum network input buffer queue length:

   ```
   net.core.netdev_max_backlog = 30000
   ```

4. Disable caching of TCP congestion state (Linux Kernel version 2.6 *only*). Fixes a bug in some Linux stacks:

   ```
   net.ipv4.tcp_no_metrics_save = 1
   ```

5. Use the BIC TCP congestion control algorithm instead of the TCP Reno algorithm (Linux Kernel versions 2.6.8 to 2.6.18):

   ```
   net.ipv4.tcp_congestion_control = bic
   ```

6. Use the CUBIC TCP congestion control algorithm instead of the TCP Reno algorithm (Linux Kernel versions 2.6.18 and newer):

   ```
   net.ipv4.tcp_congestion_control = cubic
   ```

7. Set the following to 1 (should default to 1 on most systems):

   ```
   net.ipv4.tcp_window_scaling =1
   net.ipv4.tcp_timestamps = 1
   net.ipv4.tcp_sack = 1
   ```

A reboot will be needed for changes to /etc/sysctl.conf to take effect, or you can attempt to reload sysctl settings (as root) with `sysctl -p`.

For additional information visit the Energy Science Network website.

If you have a 10-Gb system or if you follow these steps and are still getting less than your expected throughput, please contact NAS Control Room staff at support@nas.nasa.gov, and we will work with you on tuning your system to optimize file transfers.

Streamlining PBS Job File Transfers to Lou

Some users prefer to streamline the storage of files (created during a job run) to Lou, within a PBS job. Because direct access to the Lou storage nodes from the NAS compute nodes has been disabled, all file transfers to Lou within a PBS job must first go through one of the appropriate front-end systems (for example, PFEs for bro_ele, sky_ele, cas_ait, rom_ait).

Here is an example of what you can add to your PBS script to accomplish this:

1. ssh to a PFE (for example, pfe*21*) and create a directory on lou where the files are to be copied.

   ssh -q pfe*21* "ssh -q lou mkdir -p $SAVDIR"

   Here, $SAVDIR is assumed to have been defined earlier in the PBS script. Note the use of -q for quiet-mode, and double quotes so that shell variables are expanded prior to the ssh command being issued.

2. Use scp via a PFE to transfer the files.

   ssh -q pfe*21* "scp -q $RUNDIR/* lou:$SAVDIR"

   Here, $RUNDIR is assumed to have been defined earlier in the PBS script.

File Transfers Tips

The following quick and easy techniques may improve your performance rates when transferring files remotely to or from NAS.

This can increase your transfer rates by 5x, compared to older methods such as 3des.

- Transfer files from the /nobackup filesystem, which is often faster than the locally mounted disks.
- If you are using scp and your data is compressible, try adding the -C option to enable file compression, which can sometimes double your performance rates:

  % scp -C *filename user@remote_host.com*:

- For SCP transfers, use a low-process-overhead cipher such as aes128-gcm@openssh.com or arcfour:

  % scp -c -aes128-gcm@openssh.com *filename user@remote_host.com*:

- If you are transferring files from Lou, make sure they are online, rather than on the tape archive, before you perform the transfer operation.

  Note: If you use the shiftc command to transfer your files, it will automatically bring any files that are on the tape archive online before it transfers them. If you are not using shiftc, use the following DMF commands to determine the location of your files and bring them online if necessary:

  % dmls -al *filename* # show the status of your file.
  % dmget *filename* # retrieve your file from tape prior to transferring.

  For a full list of DMF commands, see DMF commands.

- If you are transferring many small files, try using the tar command to compress them into a single file prior to transfer. Copying one large file is faster than transferring many small files.
- For files larger than a gigabyte, we recommended using the Shift Transfer Tool, which can achieve much faster rates than single-stream applications such as scp or rsync.

To improve your performance by modifying your system, see TCP Performance Tuning for WAN Transfers.

If you continue experiencing slow transfers and want to work with a network engineer to help improve file transfers, please contact the NAS Control Room at support@nas.nasa.gov.

Troubleshooting SCP File Transfer Failure with Protocol Error

To address security issues with the scp command, we are in the process of adding checks to ensure that the files returned by a remote server match the files requested by the user. In some cases, the checks implemented in scp to address this issue may cause requested files to be rejected with the following error message:

protocol error: filename does not match request

This error can occur when you use quotation marks to escape special characters (such as a space) on the remote server, or when you use wildcard characters. The safest way to avoid this issue is to use the sftp command instead of scp to retrieve files. This avoids complications due to interpretation of the requested file names by the shell on the remote server.

For file retrieval, the syntax and command-line options for sftp are very similar to those for scp. For example, to retrieve files matching test*.c from a remote server to the directory *somedir*, use the following command line:

?$ sftp *somehost*:'test*.c' *somedir*/

Alternatively, you can add the -T option to the scp command line to disable checking of the file names returned by the remote server. Following the example above, the scp command line would be:

$ scp -T *somehost*:'test*.c' *somedir*/

If you need further help, please contact the NAS Control Room at (800) 331-8737 or (650) 604-4444.

PBS Jobs

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

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

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

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

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

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

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

The Queue Has Reached Its Maximum Run Limit

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

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

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

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

Your Job is Waiting for Resources

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

FY2024 SBU cost == $0.22/SBU

Your Mission Share Has Run Out

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

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

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

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

```
pfe25% qstat -W shares=-

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

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

The System is Going into Dedicated Time

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

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

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

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

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

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

Scheduling is Turned Off

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

+++Scheduling turned off.

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

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

Your Home Filesystem or Default /nobackup Filesystem is Down

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

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

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

Your Hard Quota Limit for Disk Space Has Been Exceeded

NAS quotas have hard limits and soft limits. Hard limits should never be exceeded. Soft limits can be exceeded temporarily, for a grace period of 14 days. If your data remains over the soft limit for more than 14 days, the soft limit is enforced as a hard limit.

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

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

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

Example

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

runscript

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

cd $PBS_O_WORKDIR

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

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

my_hello.scr

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

```
#!/bin/csh -f

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

set case=$1
set nprocs=5

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

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

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

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

Job Submission

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

Output

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

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

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

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

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

Viewing Soft and Hard Limits

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

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

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

For csh:

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

For bash:

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

To view the hard limits, run:

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

Modifying Soft Limits

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

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

Common Issues with the Default Limits

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

- stacksize (stack size)

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

- maxproc (maximum user processes)

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

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

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

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

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

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

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

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

- coredumpsize (core file size)

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

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

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

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

Example 1

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

gnu_serial1.pbs

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

cd $PBS_O_WORKDIR

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

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

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

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

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

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

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

myscript.csh

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

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

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

In this above sample script executed by the parallel command:

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

Potential Modifications to Example 1

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

Example 2

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

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

gnu_serial2.pbs

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

cd $PBS_O_WORKDIR

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

Example 3

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

gnu_openmp.pbs

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

cd $PBS_O_WORKDIR

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

myopenmpscript.csh

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

setenv OMP_NUM_THREADS 16

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

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

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

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

Checking Page Cache Usage

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

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

Examples

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

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

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

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

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

Managing Page Cache

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

Assigning Rank 0 to a Single Node

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

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

Limiting Page Cache Usage

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

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

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

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

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

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

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

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

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

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

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

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

Creating a Job Array

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

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

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

mkdir dir.$PBS_ARRAY_INDEX
cd    dir.$PBS_ARRAY_INDEX

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

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

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

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

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

When you develop your script, note the following:

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

Checking Status

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

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

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

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

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

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

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

Deleting a Job Array or Sub-Job

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

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

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

More Information

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

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).

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.

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

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

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

PBS Script Examples

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

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

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

Example 1

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

serial1.pbs

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

module load mpi-hpe/mpt
setenv MPI_SHEPHERD true

cd $PBS_O_WORKDIR

mpiexec -np 64 wrapper.csh 0
```

wrapper.csh

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

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

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

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

Example 2

The following PBS script provides flexibility in situations where the total number of serial jobs may not be the same as the total number of processors requested in a PBS job. The serial jobs are divided into a few batches, and the batches are processed sequentially. Again, the wrapper script is used where multiple versions of the a.out program in a batch are run in parallel.

serial2.pbs

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

module load mpi-hpe/mpt
setenv MPI_SHEPHERD true

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

cd $PBS_O_WORKDIR

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

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

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

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

set batch_count=`wc -l < $PBS_NODEFILE`

set count=0

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

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

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

Use A Processor With More Memory

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

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

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

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

to one of the following:

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

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

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

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

Run Fewer Processes per Node

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

For example, change:

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

to the following:

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

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

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

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

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

to the following:

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

Use the Endeavour System

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

Report Bad Nodes

If you suspect that certain nodes have less total physical memory than normal, report them to the NAS Control Room staff at (800)

331-8737, (650) 604-4444, or by email at support@nas.nasa.gov. The nodes can be taken offline while we resolve any issues, preventing them from being used before they are fixed.

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
   ```

Known Issues

Under Investigation

Problem

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: xmpi_net_accept_timeo/accept() timeout
- MPT ERROR: could not run executable.

In the past, the following types of errors have also occurred:

- MPT ERROR: inet_connect to host xxx port yyy failed 113
- MPT ERROR: Assertion failed at daemon.c:272: "len == rv"
- MPT ERROR: Assertion failed at ibdev_multirail.c:7090: "context != ((void *)0)"

Status: Under Investigation

Workaround

For instructions, see MPT Startup Failures: Workarounds.

Resolved

Problem

Occasional data loss or corruption occurred in some output files that were sent to the /nobackupnfs2 filesystem.

Status: Resolved

We made a number of configuration changes to the /nobackupnfs2 filesystem to address the problem. Subsequently, the problem has not recurred.

Background

Due to a server issue, any data sent to the /nobackupnfs2 filesystem within a certain period (30 minutes to several hours) before the server hung/rebooted was being held in memory and not written to disk. After the server rebooted, the data in memory was lost, resulting in data loss or corruption.

Normally, data is continuously written to disk, so rebooting the server causes minimal if any data loss; writes to the filesystem are held during the reboot and the data can resume writing when the server is back up.