Athena is a modular computing system housed in NASA's Modular Supercomputing Facility, located less than a mile from the main NAS building. Athena is configured as follows:

AMD Turin-Based Compute Resources

- 4 racks
- 1,024 Turin nodes
- 262,144 cores
- 786 terabytes (TB) total memory
- 20.132 petaflops (PF) theoretical peak performance

Hostnames

For the Turin based resources, there are 4 racks, 8 chassis per rack, 8 compute blades per chassis and 4 nodes per blade. The hostnames of the Athena Turin nodes are athc[0-3]s[0-7]b[0-7]n[0-3].

Processor, Memory and Network Subsystems Statistics

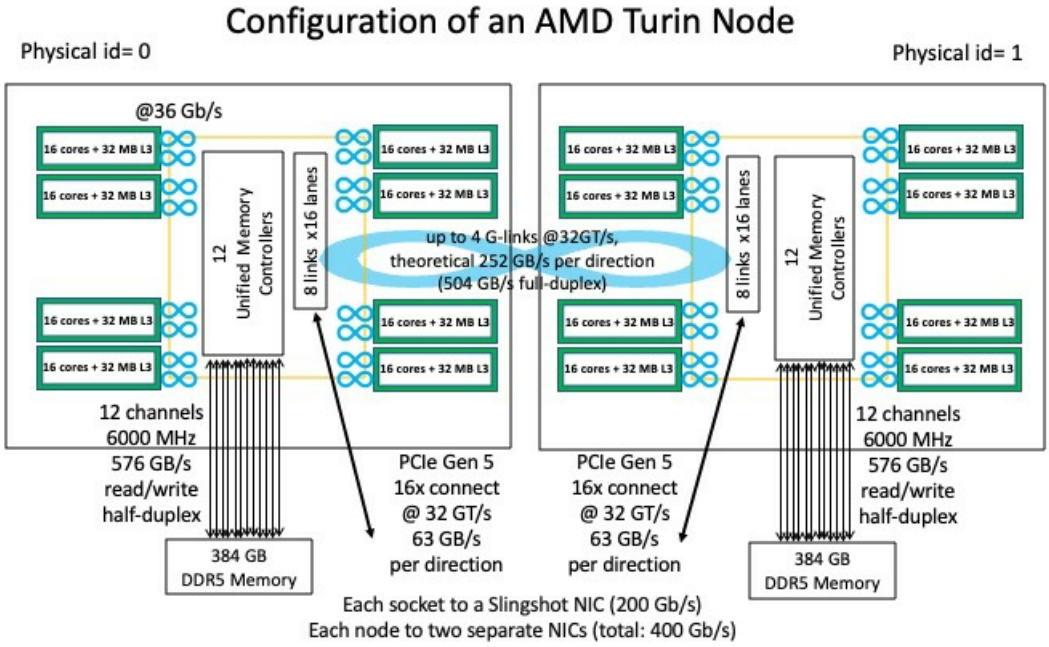
The following table provides detailed configuration statistics for the processor, memory and network subsystems for the Athena compute nodes:

Athena Processor, Memory, and Network Subsystems Statistics

Architecture	HPE Cray EX4000
	Processor
	Turin
CPU	128-core EPYC 9745
Newest Instruction Set	AVX-512
Hyper-Threading	ON
TurboBoost	ON
non-Turbo base CPU-Clock	2.4 GHz
Maximum Double Precision Floating Point Operations per Cycle per Core	32
# of Cores/node	256
Total # of Nodes	1,024
Total # of Cores	262,144
Total Double Precision TFlops	20,132
	Memory
L1 Cache	Local to each core; Instruction cache: 32 KB Data cache: 48 KB; Instruction set associativity: 8; Data set associativity: 12; Cache line size: 64B
L2 Cache	1 MB per core; Associativity: 16; Cache line size: 64B
L3 Cache	32 MB shared among 16 cores in a core complex; 256 MB per socket; Associativity: 16; Cache line size: 64B
TLB	local to each core
Default Page Size	4 KB
Memory/Core	3 GB; DDR5
Total Memory/node	768 GB;
Memory Speed and Bandwidth	6000 MHz; 12 channels; 576 GB/sec read/write half-duplex
Inter-socket Interconnect	Global Memory Interconnect 4 links with 32.0 GT/s per link, or 252 GB/sec per direction
	Inter-node Network

Network Device on node	Cray Slingshot 11 interfaces; 200 Gbits/s per interface; total of 400 Gbit/s per node
Switches between nodes	1 Cray Slingshot 11 switch per 1 interface; 200 Gbits/s per port
Topology	Dragonfly; all-to-all between groups; all-to-all within each group

The AMD Turin nodes of the new Athena cluster are installed in a module located less than a mile from the main NAS building.



This is a simplified configuration of an AMD Turin node with two EPYC 9745 processors (i.e., two sockets) and a total of 768-gigabyte (GB) DDR5 memory with 24 memory channels (12 per socket).

The EPYC 9745 processor uses the AMD Zen 5c microarchitecture. Each processor contains 128 cores in eight Core Complex Dies (CCDs, each shown in a green box). Each CCD contains a single Core Complex (CCX) with 16 cores and a shared 32 MB L3 cache. The cores run with a base clock of 2.4 GHz, an all-core boost clock of 3.45 GHz and a maximum boost clock of 3.7 GHz.

Note: The Zen 5c microarchitecture (manufactured on a 3 nm process) is a compact variant of the Zen 5 microarchitecture (manufactured on a 4 nm process) and therefore can pack more cores per CCX. Zen 5c emphasizes efficient throughput while Zen 5 emphasizes peak performance.

Each processor has one I/O die (IOD, enclosed in a yellow box) which utilizes AMD's Infinity Fabric Technology to interconnect among CCDs, memory and I/O:

- **CCDs:** The IOD has a total of sixteen high-speed (36 Gb/s) Global Memory Interconnect (GMI) links (shown with the small ♾ sign) connecting to the eight CCDs in the processor; 2 links per CCD.
- **Memory:** Twelve Unified Memory Controllers (UMCs) with a total of twelve channels, each running at 6,000 MHz.
- **I/O:** The IOD in the SP5 socket used for Turin has eight combo links (G0-G3 and P0-P3) each with 16 lanes and a transfer rate of 32 GT/s per lane (the same rate as PCIe Gen5); links G0-G3 can be used as External GMI (xGMI) links to connect two sockets together (shown with the large ♾ sign); all eight links support PCIe 5.0 (to connect devices such as graphics cards, storage devices and network cards), among them, P0 - P3 can also be used as Compute eXtress Links 2.0 (CXLs, which link to extended memory).

  Note: The NAS Turin nodes are not configured with extended memory.

Each node connects to two Cray Slingshot-11 Cassini network interface cards (NICs) through a PCIe 5.0 link per socket. Each NIC is connected to a Slighshot-11 Rosetta switch. The NICs and switches run at a rate of 200 Gb/s, which provide an aggregate of 400 Gb/s (per direction) for each node.

Main Differences Between the NAS Turin Processors and Milan Processors:

- **CCD and CPU cores:** Turin has more cores per socket and more advanced features.

  Milan uses the Zen 3 microarchitecture and has 64 cores/socket partitioned in 8 CCDs (each with 8 cores). Turin uses the Zen 5c microarchitecture and has 128 cores/socket partitioned in 8 CCDs (each with 16 cores).

  Turin has more accurate branch prediction, instruction pipeline optimization, and more integer and floating-point units.

  Milan supports Advanced Vector Extensions up to AVX2 (256 bits). Turin adds supports for AVX-512 with full 512-bit floating-point data paths.

- **Cache and Memory:** Turin has larger L1 data cache, larger L2 cache, faster memory speed, more memory channels, and higher per-socket and per-core memory bandwidth than Milan, but smaller L3 cache/core and smaller memory size/core.

	Milan	Turin
L1 data cache	32 KB/core	48 KB/core

L2 cache	512 KB/core	1 MB/core
L3 cache	32 MB/8 core	32 MB/16 cores
Memory Size	256 GB/64 cores	384 GB/128 cores
Memory Speed	3200 MHz	6000 MHz
# of Memory Channels	8	12
Memory Bandwidth	204.8 GB/s /64 cores	576 GB/s / 128 cores

- **Interconnect:** Turin has a faster network to connect the two sockets in a node and to connect outwards.

  The IOD in Milan utilizes PCIe 4.0 technology (running at a rate of 16 GT/s per lane) to provide support for Infinity Fabric within a node and for interconnecting outwards. Turin utilizes PCIe 5.0 technology (running at a rate of 32 GT/s per lane).

Additional information about the NAS Turin nodes is provided below.

CPU Core

Unlike the Zen 4 microarchitecture which supports AVX-512 instructions by processing two 256-bit vectors in sequential clock cycles, Zen 5/5c comes with full 512-bit floating point data paths which allow data to be read into the CPU in a single clock cycle. In addition, a Basic Input/Output System (BIOS) firmware setting can direct the processor to execute AVX-512 instructions using either one 512-bit vector (for better performance) or two 256-bit vectors (for better power efficiency). The NAS AMD Turin nodes are configured to use the 512-bit vector for AVX-512 instructions.

With the use of the full 512-bit data path and floating-point fused multiply-add units, each Zen 5/5c core can deliver 32 double-precision floating-point operations (FLOPs) per clock cycle. Thus, a Turin 9745 processor can deliver 2.4 GHz x 128 cores/processor x 32 Flops = 9.8304 TFLOPS per processor which translates to 19.6608 TFLOPS per node.

The Zen 5/5c microarchitecture brings in some new instructions not available in earlier generations:

- MOVDIRI/MOVD64B: Move 4, 8, or 64 bytes as a direct store (to memory), bypassing caches.
- VP2INTERSECT: AVX-512 vector pair intersection to a pair of mask registers.
- VNNI/VEX: Extend AVX-512 Vector Neural Network Instructions (VNNI) to VEX encoding.
- PREFETCH[I]: Software prefetch of instruction lines into cache hierarchy.

For a complete list of instructions supported by Turin, run the `cat /proc/cpuinfo` command on a Turin node.

Through a BIOS setting, each core can have Simultaneous Multi-Threading (SMT) enabled to allow 2 separate hardware threads which share the corresponding core's L1 and L2 cache. To check the SMT setting on a Turin node, use the `lscpu | grep "Thread(s) per core"` command. When SMT is enabled, `cat /proc/cpuinfo` shows a total of 512 logical cores per NAS Turin node.

Compiler support for the AMD Turin processors:

- Cray compiler:

  `-march=znver5` or `-march=cray-znver5` is available in Cray C compiler versions 19.0.0 and 20.0.0 (but not in Cray Fortran compiler).

- GNU compiler:

  `-march=znver5` is available in GNU compiler version 13.3 and 14.2.

- AOCC compiler:

  `-march=znver5` is available in AOCC compiler version 5.0.

- Intel Classic Fortran Compiler (ifort):

  `-march=core-avx2 -mavx512f -mavx512dq -mavx512bw -mavx512vl`

For more information, read AMD EPYC 9xx5-Series Processors Compiler Options Quick Reference.

Cache Hierarchy

- **op cache:** 6K entries, private to each core; 16-way.

  Note: An op cache is a specialized cache that stores pre-decoded micro-operations (uOps) which are the smallest units of instructions the CPU can execute. There are two op cache pipes; each can deliver 6 uOps simultaneously to the CPU's execution units in a single clock cycle (2 x 6-wide fetch/cycle), resulting in a total of 12 uOps instructions per cycle.

- **L1 instruction (L1I) cache:** 32 kilobytes (KB), private to each core; 64 bytes/line; 8-way; 2 x 32 bytes fetch per cycle.
- **L1 data (L1D) cache:** 48 KB, private to each core; 64 bytes/line; 12-way; support 2 loads and 1 store per cycle for 512-bit operation, otherwise, support 4 loads and 2 stores per cycle to/from the execution units.
- **L2 cache:** 1 MB, private to each core; unified (instruction/data); 64 bytes/line; 16-way; 64 bytes/cycle bandwidth to/from either L1I or L1D cache.
- **L3 cache:** 32 MB shared among 16 cores in a CCX; 64 bytes/line; 16-way; 32 bytes/cycle bandwidth to/from L2 cache.

Memory Subsystem

Turin places twelve unified memory controllers in the centralized I/O die. With a total of 12 memory channels each at the speed of 6,000 GHz, an 8-byte read or write taking place per cycle per channel results in a maximum total memory bandwidth of 576 GB/s per socket.

Each memory channel can be connected with up to two Double Data Rate fifth generation (DDR5) Dual In-line Memory Modules (DIMMs). For the NAS Turin configuration, each channel is connected to a single 32 GB DDR5-6400 registered DIMM (RDIMM) with error correcting code (ECC) support. This one-DIMM-per-channel configuration provides lower latency than the two-DIMM-per-channel configuration. In total, the amount of memory is 384 GB per socket and 768 GB per node.

Through BIOS, the processor and the memory channels in a socket can be set up as 1, 2 or 4 Non-Uniform Memory Access (NUMA) nodes. Currently, the front-end nodes are configured with NPS=1 (i.e., 1 NUMA node per socket, or 2 NUMA nodes per front-end node) and the compute nodes are configured with NPS=2 (i.e., 2 NUMA nodes per socket or 4 NUMA nodes per node).

Inter-Node Network

Currently, a total of 1,024 Turin nodes in the Athena cluster are configured with the HPE Cray EX4000 architecture as follows:

- 4 racks in the cluster (256 nodes/rack; note that racks are also known as cabinets).
- 8 chassis per rack (32 nodes/chassis).
- 8 compute blades per chassis (4 nodes/blade).

The hostname of each node is x100[0-3]c[0-7]s[0-7]b0n[0-3].

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

These 1,024 nodes are connected as a single fabric using the Cray Slingshot-11 High Performance Interconnect (HPI) Network, where each node is connected to two Slingshot-11 Cassini NICs (on two separate dual-NIC Mezzanine cards), which in turn connect to two Slingshot-11 Rosetta Ethernet switches. The HPI possesses these features:

- **HPC Ethernet:** This is a specialized Ethernet network that maintains compatibility with existing Ethernet devices and protocols and is optimized with adaptive routing, congestion management, and quality of service.
- **High radix switches:** These are switches with large number of ports to allow for a high degree of connectivity among devices. Each Slingshot switch provides 25.6 Terabits/s (Tbps) of bi-directional bandwidth with 64 ports (in 32 jacks with 2 ports per jack), each capable of delivering 200 Gbps per direction.
- **Dragonfly topology:** A two-level dragonfly topology is a hierarchical network of switches divided into *groups*. Within a group, it is a local level network (L1), while between groups it is a global level network (L2). For the NAS Athena cluster:
    - A group is a rack. With 4 racks in the system, there are 4 groups.
    - A group contains 32 switches within a rack (4 per chassis).
    - Within a group, an all-to-all topology is used.
        - Each switch uses 31 out of the 64 ports to connect to the other 31 switches in the group (L1, local)

          This results in a fully non-blocking network without internal contention, as every switch has a direct connection to any other switch in the same group. Due to the large number of connections needed at this level, copper cables are used rather than the more expensive optical cables.

        - Each switch uses 16 ports to connect to 16 compute nodes (L0, edge)

          Copper cables are also used to connect switches to NICs in compute nodes.

          Note: Each compute node has two NICs to connect to two switches.

    - Between groups, an all-to-all connection is also formed.
        - The connection between each pair of groups is known as an *arc*.

          At its current configuration with 4 racks, the Athena cluster is designed with 80 Quad Small Form-factor Pluggable - Double Density (QSFP-DD) optical cables per arc. Each cable is connected to a jack with 2 ports and delivers 400 Gbps unidirectional bandwidth (therefore, a 800 Gbps bidirectional bandwidth).

        - The 80 QSFP-DD cables per arc are roughly evenly distributed among the 32 switches in a group.

          Note: When more racks are added to Athena, the number of cables per arc will be reduced due to the limit of available ports for L2 network.

    - The maximum number of switch-to-switch hops is 3 between any pair of nodes in the system.

Connecting Athena to the NAS Filesystems

The Athena cluster is connected via 28 InfiniBand router nodes, to the NAS InfiniBand HDR interconnect, and to the NAS I/O network where the filesystems are located.

References

- AMD EPYC 9005 Processor Architecture Overview
- 5th Gen AMD EPYC Processor Architecture
- AMD EPYC 9005 High Performance Computing (HPC) Tuning Guide
- BIOS and Workload Tuning Guide for AMD EPYC 9005

- AOCL Performance Tuning Guide (63859)
- AMD EPYC 9xx5-Series Processors Compiler Options Quick Reference

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

Athena Front-End Systems

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

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

- Two-step login:

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

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

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

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

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

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

- Then, run the command:

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

Athena Turin Compute Nodes

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

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

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

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

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

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

Requesting CPU Allocations

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

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

Operating System and Software Modules

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

Software packages are managed with the Tool Command Language (Tcl) environment modules system. On Athena, you should use

the module command from sourcing init scripts under /opt/cray/pe/modules/3.2.11.7, not the one under the /usr/local directory. In your login sessions or job scripts, check whether you are using the correct version of the module command by using:

```
% module -V
```

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

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

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

One way to accomplish this is as follows:

For .cshrc:

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

For .bashrc:

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

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

The default $MODULEPATH upon login includes:

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

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

Porting Your Application

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

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

Note: PrgEnv-xxx is case-sensitive.

Cray Programming Environment

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

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

Choosing the Cray Compiler Integrated in PrgEnv-cray

The CPE version 25.09 is installed on Athena. By default, several modules of CPE 25.09 are loaded automatically when you login to

an Athena front-end or compute node.

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

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

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

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

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

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

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

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

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

For example:

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

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

The solution is to (1) comment out sourcing those files, and (2) source /opt/cray/pe/modules/3.2.11.7/init/csh (for csh) or /opt/cray/pe/modules/3.2.11.7/init/bash (for bash) in your system startup scripts (for Athena) and job scripts.

Choosing the Intel compiler integrated in PrgEnv-intel

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

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

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

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

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

Choosing the AMD AOCC compiler integrated in PrgEnv-aocc

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

Choosing the GNU compiler integrated in PrgEnv-gnu

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

Building and Running MPI Applications

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

Building with PrgEnv-cray

athfe01% module load PrgEnv-cray

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

athfe01% module load cray-mpich

Method 1: Use ftn, cc, CC wrappers

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

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

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

Method 2: Use mpifort, mpicc, mpicxx wrappers

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

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

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

Building with PrgEnv-intel

athfe01% module switch PrgEnv-cray PrgEnv-intel

Method 1: Use ftn, cc, CC wrappers

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

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

Method 2: Use mpifort, mpicc, mpicxx wrappers

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

With csh:

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

With bash:

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

Method 3: Explictly invoke compilers and link to MPI libraries

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

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

Running MPI applications

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

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

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

Read man mpiexec for all options and environment variables.

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

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

Building and Running OpenMP or MPI/OpenMP Hybrid Codes

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

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

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

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

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

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

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

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

or

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

Running PBS Jobs on Athena Turin Compute Nodes

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

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

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

Sample PBS Script For Athena Turin Nodes

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

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

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

cd $PBS_O_WORKDIR

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

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

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

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

References

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

Running Jobs on Athena

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

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

Running Jobs on Aitken and Electra

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

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

Running Jobs on Endeavour

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

Running Jobs on Cabeus

To run a PBS job on:

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

PBS Queues

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

```
qstat -Q
```

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

```
qstat -fQ queue_name
```

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

Queues for Athena Users

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

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

Queues for Aitken and Electra Users

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

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

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

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

Queues for Cabeus GPU Users

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

See also:

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

Queues for Endeavour Users

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

See Endeavour PBS Server and Queue Structure for more information.

Queue for Lou Users

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

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

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

See Lou Data Analysis Nodes for more information.

Queues with Restricted Access

The vlong Queue

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

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

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

The wide Queue

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

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

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

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

Various Special Queues

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

Advanced Reservation Queues

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

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

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

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

Cray Message Passing Toolkit (CMPT) is a collection of high-performance software libraries within the Cray Programming Environment (CPE) that enables parallel applications to communicate and exchange data efficiently across nodes in a Cray system with Slingshot interconnect, such as Athena which uses Slingshot 11. CMPT includes MPICH, PMI, OpenSHMEMX, DSMML, and MPIxlate. CPE version 25.09 includes MPICH 8.1.33 and 9.0.1, PMI 6.1.16, OpenSHEMEMX 11.8.0, DSMML 0.3.1, and MPIxlate 1.0.9.

Note: MPICH 8.1.33 implements the MPI-3.1 standard. MPICH 9.0.1 includes support for MPI 4.0 standard features. MPICH 8.1.33 is the current default on Athena.

As the name suggests, Cray MPICH is based on MPICH. If your existing executable was built with a dynamically linked MPI library that is Application Binary Interface (ABI)-compatible with the Cray MPICH library, it might be able run on Athena without recompiling and linking with the Cray MPICH library. For example, Intel MPI is also based on MPICH, and is ABI-compatible with the Cray MPICH library. On the other hand, OpenMPI and HPE/SGI MPT library are not ABI-compatible with MPICH.

This article provides brief information about using the CMPT modules cray-mpich, cray-mpich-abi and cray-mpixlate to build or prepare an MPI executable, together with the cray-pals module from the Cray Parallel Application Launch Service (PALS) to manage job startup and process launch on Athena.

Building Applications with the cray-mpich Module

There are three methods of using cray-mpich to build an application.

Method 1: Explicitly Invoking the Compiler and Linking to the Cray MPICH Library

To explicitly build an executable using the cray-mpich module and a compiler from either PrgEnv-cray, PrgEnv-intel, PrgEnv-aocc or PrgEnv-gnu:

```
module load PrgEnv-cray  (or PrgEnv-intel, PrgEnv-aocc, PrgEnv-gnu)
module load cray-mpich   (or cray-mpich/9.0.1)

name_of_compiler_driver <compiler flags> application_source_file.[f90,c,cpp]
-I/path-to-include-directory-of-other-library
-L/path-to-lib-directory-of-other-library -lname_of_other_library
-I${MPICH_DIR}/include
-L${MPICH_DIR}/lib -lmpi   (for C/C++ applications)
or
-L${MPICH_DIR}/lib -lmpi -lmpifort (for Fortran applications)
```

The name of the compiler driver varies as follows:

```
PrgEnv-cray : crayftn, craycc, crayCC
PrgEnv-intel: ifort, icc, icpc, ifx, icx, icpx
PrgEnv-aocc : flang, clang, clang++
PrgEnv-gnu  : gfortran, gcc, g++
```

The ${MPICH_DIR} can be replaced with ${CRAY_MPICH_PREFIX} or ${CRAY_MPICH_DIR}, since all three of them are defined in cray-mpich and they point to the same directory.

Depending on which PrgEnv is used {-cray, intel, aocc, gnu} and whether cray-mpich/8.1.33 or cray-mpich/9.0.1 are loaded, the actual path of ${MPICH_DIR} varies. For example, if using PrgEnv-cray and cray-mpich/8.1.33, then ${MPICH_DIR} is /opt/cray/pe/mpich/8.1.33/ofi/cray/20.0.

Note: Athena is an HPE Slingshot 11 system that only supports libfabric — a low-level communication library also known as the OpenFabrics Interfaces (OFI). The Unified Communication X (UCX) framework for communication is supported in Slignshot 10 systems, but not Slingshot 11 systems with Cray MPICH.

Method 2: Using the Cray MPICH Wrapper Scripts

Cray MPICH also provides standard MPI wrappers (mpicc, mpicxx, mpic++, mpifort, mpif77, mpif90) for invoking compiler and linking the proper Cray MPICH library in accordance with the compiler that is used (cray, intel, aocc, gnu).

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

Usage:

```
% module load PrgEnv-cray (or PrgEnv-intel, PrgEnv-aocc, PrgEnv-gnu)
% module load cray-mpich  (if not already loaded)
% mpifort <FFLAGS> source.f90
% mpicc <CFLAGS> source.c
% mpicxx <CXXFLAGS> source.cpp
```

If other non-MPI libraries are needed for building the application, they need to be explicitly added to the mpifort, mpicc, mpicxx command line as shown in Method 1.

When using the MPI wrappers, the variables MPICH_FC, MPICH_CC and MPICH_CXX can be used to change the choice of compilers.

For example, with PrgEnv-intel, mpifort, mpicc, and mpicxx default to using ifx, icx and icpx regardless of which of intel, intel-classic, intel-oneapi you have loaded. If you want mpifort, mpicc, and mpicxx to use ifort, icc, icpc, do the following before invoking mpifort, mpicc or mpicxx:

With csh:

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

With bash:

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

Method 3: Using Cray's Compiler/Link Wrapper Scripts

When a cray-mpich module is loaded together with a compiler provided under either PrgEnv-cray, PrgEnv-intel, PrgEnv-aocc, or PrgEnv-gnu, the Cray wrappers ftn, cc, CC will automatically link the Cray MPICH library (-lmpich or simply -lmpi for C/C++ and -lmpi -lmpifort for Fortran) and other Cray libraries (for example, Cray Scientific Library) to ensure compatibility and optimized performance.

The actual -lmpi and -lmpifort linked in depend on the compiler used. For example, if a Cray compiler (crayftn, craycc, crayCC) is used, then -lmpi_cray and -lmpifort_cray are linked in.

Usage:

```
% module load PrgEnv-cray (or PrgEnv-intel, PrgEnv-aocc, PrgEnv-gnu)
% module load cray-mpich  (if not already loaded)
% ftn <FFLAGS> source.f90
% cc <CFLAGS> source.c
% CC <CXXFLAGS> source.cpp
```

If your application needs other libraries not included in the Cray Programming Environment, you must explicitly add them to the ftn, cc, CC command line as shown in Method 1.

Here are a few useful Cray MPICH and LIBFABRIC environment variables to consider:

- **MPICH_ENV_DISPLAY=1**: Causes rank 0 to display all MPICH environment variables and their current settings at MPI initialization time.
- **MPICH_VERSION_DISPLAY=1**: Causes MPICH to display the HPE Cray MPI version number as well as build date information; helps to track issues with different MPI versions.
- **MPICH_RANK_REORDER_DISPLAY=1**: Causes rank 0 to display which node each MPI rank resides in. The rank order can be manipulated via the MPICH_RANK_REORDER_METHOD environment variable.
- **MPICH_MEMORY_REPORT=*n***:
  - *n*=1: prints a summary of the min/max high water mark and associated rank to stderr.
  - *n*=2: outputs each rank's high water mark to a file as specified using MPICH_MEMORY_REPORT_FILE.
  - *n*=3: does both 1 and 2.
- **MPICH_OFI_VERBOSE=1**: Helps for debugging by displaying more verbose output during MPI_init to verify which libfabric provider has been selected, along with the name and address of the NIC being used.
- **MPICH_OFI_USE_PROVIDER**: On Slighshot 11 systems, the default is the cxi provider. Other libfabric providers (such as "sockets") can be requested for debugging purposes.

  Note: cxi stands for Cassini fabric provider in libfabric.
- **FI_PROVIDER**: Controls what providers to load.
- **FI_LOG_LEVEL=warn & FI_LOG_PROV=cxi**: Causes the Cassini provider to log messages which help to debug issues.
- **FI_CXI_RX_MATCH_MODE**: Specifies the receive message matching mode to be utilized; possible choices are hardware, software, hybrid.

  Set this variable to hybrid or software if your application fails to run and the error message includes a recommendation to set this variable.
- **MPICH_OFI_NIC_MAPPING**: Specifies the precise rank-to-NIC mapping to use on each node. This is evaluated only if the MPICH_OFI_NIC_POLICY variable is set to USER.
- **MPICH_OFI_NIC_POLICY**: Selects the rank-to-NIC assignment policy used by Cray MPI. Each MPI rank will be assigned to exactly one NIC. There are five available options: [BLOCK | ROUND-ROBIN | NUMA | GPU | USER].
- **MPICH_OFI_NIC_VERBOSE**: If set to 1, verbose information pertaining to NIC selection is printed at the start of the job.

- **MPICH_OFI_NUM_NICS**: Specifies the number of NICs the job can use on a per-node basis.

For more information, read **man intro_mpi** and the libfabric **man fabric** and **man fi_cxi**.

Running Applications with the cray-pals Module

Loading a cray-mpich module provides access to scripts such as mpifort, mpif77, mpif90, mpicc, mpicxx, mpic++ as well as various libmpi* for building MPI applications. However, cray-mpich does not provide mpiexec for use during runtime. A cray-pals module is needed during runtime to access the mpiexec executable in the parallel application launch service (PALS).

Note: Make sure cray-pals is loaded before launching your application with mpiexec.

Usage:

```
% module load cray-pals    (if not already loaded)
% mpiexec [options] your_executable
```

Here are some useful mpiexec options to consider:

- **-n, -np, --np:** Number of processes to start. Default is 1.
- **-ppn, --ppn:** Processes/ranks per compute node.
- **-hostfile, --hostfile:** File containing list of host names. Default for <*hostfile*.txt> is $PBS_NODEFILE. For example:

  ```
  mpiexec [-hostfile|--hostfile] <hostfile.txt> <application>
  ```

- **--cpu-bind [verbose],[core,depth,none,numa,socket,core,thread,list,mask]:** Bind ranks to CPUs. Default is thread. For example:

  ```
  mpiexec --cpu-bind core -n 8 <application>
  ```

- **--mem-bind [none, local, list, mask]:** Bind ranks to NUMA nodes.
- **-env** or **--env:** Set an environment variable. For example:

  ```
  mpiexec [-env|--env]  <ENV_VAR_NAME>="ENV_VAR_VALUE" <aplication>
  ```

- **-l, --label:** Add node and rank labels output.
- **--line-buffer** or **--no-line-buffer:** Line buffered mode for stdio, to avoid interleaving lines from multiple ranks. Default is --no-line-buffer to allow fully buffered stdio.
- **--restart:** mpiexec to restart application if it detects that an application aborted. Default value for --restart is 16.
- **-v, --verbose:** More verbose output.

Running Applications Built with Intel MPI Without Rebuilding the Executable

An existing executable built with the Intel MPI library might be able to run on Athena without having to rebuild it with the Cray MPICH library. To try this, use the cray-mpich-abi module. For example:

1. Load all modules and/or set any variables required to run the application, except the Intel MPI module.
2. Load the Cray MPICH module cray-mpich-abi instead of cray-mpich.
3. Load the cray-pals module.
4. Launch the application.

Note: If the existing executable was built with an Intel MPI version older than version 5.0 (released around 2014), use cray-mpich-abi-pre-intel-5.0 instead of cray-mpich-abi.

For example:

```
% module switch PrgEnv-cray PrgEnv-intel
   (if you executable needs some libraries from Intel compiler)

% setenv LD_LIBRARY_PATH \
    "/nasa/intel/oneapi/mkl/2023.2.0/lib/intel64:$LD_LIBRARY_PATH"
    (if MKL is needed. Note that the command line is too long to be formatted
as one line, so it is broken with a backslash [\].)
% module switch cray-mpich cray-mpich-abi
% module load cray-pals
% mpiexec [options] executable-built-with-intel-mpi
```

WARNING: Your existing executable may not be able to achieve the best performance that the Athena cluster can offer. For example, an executable built with Intel compiler on other NAS systems and/or linked with the Intel MKL library likely does not support the AVX512 capability of the AMD Turin processor. For best performance, consider rebuilding your application and linking with library

such as the Cray Scientific Library or the AMD Optimizing CPU Libraries (AOCL) instead of the Intel MKL library.

Translating Executables Built with OpenMPI or HPE/SGI MPT to Cray MPICH

MPIxlate, which stands for MPI Translate, enables C and C++ applications compiled using an MPI library that is not binary compatible with Cray MPICH to be run without recompilation on supported HPE Cray systems.

WARNING: MPIxlate does not provide support for Fortran applications.

MPIxlate transparently translates OpenMPI ABI and HPE/SGI MPT ABI to MPICH ABI at runtime.

Usage:

1. Load all modules and/or set any variables required to run the application, except the OpenMPI or HPE/SGI MPT module.
2. Load the Cray MPICH module `cray-mpich`.
3. Load the MPIxlate module `cray-mpixlate`.
4. Load the `cray-pals` module.
5. Launch the application using MPIxlate as a pass-through intermediary.

For example, to translate an application built with OpenMPI with an ABI major number 40 (`libmpi.so.40`) to Cray MPICH with an ABI major number 12 (`libmpi_cray.so.12`), do:

```
% mpiexec [mpiexec options] mpixlate -s ompi.40 -t cmpich.12 a.out ...
```

To translate an application built with HPE/SGI MPT 2.25 to Cray MPICH with an ABI major number 12, do:

```
% mpiexec [mpiexec options] mpixlate -s sgi.225 -t cmpich.12 a.out ...
```

Note: With MPIxlate 1.0.9, only `ompi.40` and `sgi.225` are supported. If your C/C++ executable was built with HPE/SGI MPT 2.30, try `-s sgi.225`.

For more information about MPIxlate and its limitations, see **man mpixlate**.

The Cray Programming Environment (CPE) includes the Cray Performance Measurement & Analysis Tools (CPMAT) suite. This article provides brief descriptions of the two CPMAT tool sets, Cray Perftools (a set of performance data collection tools) and Apprentice2 or Apprentice3 (a visualization tool), together with the associated Perftools modules, usage examples, and sample performance analysis results in text and graphics formats.

Cray Perftools

Cray Perftools (also known as PAT or CrayPat) is a core data capture tool within CPMAT. It can analyze whole program performance across many nodes, use Performance API (PAPI) to interface with Cray system's hardware performance counters, and perform either sampling or tracing experiments.

Note: Sampling collects data periodically based on expiration of a timer or overflow of a hardware counter. Tracing records detailed time-sequenced information of instrumented functions or regions.

TIP: Tracing a large, long-running program can incur considerable overhead.

Instrumentation Approaches

This section describes three ways of instrumenting applications (namely, at run-time, compile-time or link-time) to enable performance data collections with CrayPat. Different instrumentations require loading one or more of the Perftools modulefiles shown in this module avail perftools output:

```
% module use /opt/cray/pe/perftools/25.09.0/modulefiles
% module avail perftools
--------- /opt/cray/pe/perftools/25.09.0/modulefiles ---------
perftools             perftools-lite-gpu    perftools-preload
perftools-lite        perftools-lite-hbm
perftools-lite-events perftools-lite-loops


--------- /opt/cray/pe/modulefiles ---------------------------
perftools-base/25.09.0(default)
```

See the Perftools Modules section for a brief description of each of these modules.

Note: To avoid interference with CrayPat, do not use other tools that also do performance analysis (for example, the NAS-developed mpiprof tool) at the same time as CrayPat.

Run-Time Instrumentation

This is CrayPat's simplest and recommended approach for starters. It hides the complexity of Craypat's classic approach which uses the pat_build utility for a link-time instrumentation. With this approach, you can choose to use your existing un-instrumented executable without recompiling and relinking and will be able to perform some basic sampling performance analysis via the pat_run utility accessible through the perftools-base module.

On the other hand, you can also choose to relink your executable with the perftools-preload module loaded which allows the resulting executable to facilite the use of the LD_PRELOAD environment variable during execution with the pat_run utility and access nearly all the performance analysis features that would typically require explicit instrumentation via pat_build.

In addition, pat_run supports Python tracing and sampling experiments while pat_build does not. Read **man pat_python** for more information. Some high-level differences between pat_build and pat_run are included in **man pat_run**.

TIP: We recommend that you build your executable using the -g compiler flag in order to allow the source line numbers to be shown in the performance analysis report.

There are multiple options for the pat_run utility to choose different sampling or tracing experiments. A few of the pat_run options are listed here. Read **man pat_run** for more information and see this example for a demonstration.

Compile-Time Instrumentation

This is CrayPat's simplified (compared to the classic link-time approach), lightweight and automatic reporting approach. Recompiling/linking is needed to create an instrumented executable.

WARNING: If you have an existing un-instrumented executable with the same filename, it will be overwritten.

This approach requires loading *only one* of the following instrumentation modules (in addition to perftools-base) before compiling the application.

perftools-lite
: For default time sampling profile.

perftools-lite-loops
: For loop analysis where the performance data are to be used with Apprentice2/3 (must be used with the Cray Compiler).

perftools-lite-events

For event profile, such as tracing top functions, providing I/O information, and suggestion for rank reordering.

Note: The following two perftools modules do not apply to the NAS Turin nodes since they do not contain GPUs or high bandwidth memory (HBM):

perftools-lite-gpu
: For tracing experiments that focus on the program's use of GPUs.

perftools-lite-hbm
: For identifying program locations responsible for high bandwidth from memory (must be used with the Cray Compiler and Intel processors with HBM).

See this example for a demonstration. For limitations of using this approach, see the WARNINGS section of **man perftools-lite**.

Link-Time Instrumentation

This is CrayPat's classic approach. Recompiling is not needed if all the.o files exist, but relinking is required with the modules perftools-base and perftools loaded. In addition, the Cray compiler wrappers ftn, cc, and CC must be used during the link step. After instrumentation using the pat_build utility, it creates a uniquely instrumented stand-alone executable. The instrumented executable includes +pat appended to the end of the executable name. This approach provides more in-depth performance analysis and greater flexibility in controlling data collection.

A few of the pat_build options are listed here. See this example for a demonstration. For more information, read **man pat_build** for options and environment variables.

Notes:

- If the application is instrumented as a tracing experiment, you must re-instrument the application to trace different or additional functions.
- The pat_build utility does not support instrumenting programs written in Python.

CrayPat also provides an Application Program Interface (API) where CrayPat functions can be inserted into an application's source code. See **man pat_api** for more information.

Performance Reports

Depending on the method of instrumentation, options and run-time environment variables used (such as various PAT_RT_*XXX* variables), various reports or graphs can be generated by running pat_report or Apprentice2/3 (app2/app3) on data collected under an experiment_data_directory. The experiment_data_directory may include multiple subdirectories and files:

xf-files
: A directory containing raw data written by the instrumented executable; processing thexf-files requires access to the instrumented executable. To save disk space, we recommend deletiing this subdirectory after the ap2-files subdirectory has been generated.

ap2-files
: A directory containing data to be used with Apprentice2 or Apprentice3 (orpat_report to generate a text report). Automatically generated after first invocation of pat_report on the experiment_data_directory.

rpt-files
: A directory containing a copy of the text report; automatically generated at the end of a run if the executable was instrumented at (i) run-time using pat_run with the -r option of pat_run, (ii) compile-time using one of the perftools-lite-xxx modules, and (iii) link-time using pat_build with the -D report=y option of pat_build.

build-options.apa
: A file generated when performing an Automatic Profiling Analysis (for applications instrumented with pat_build, which by default includes -O apa) with results processed by pat_report; contains the pat_build arguments recommended for further performance analysis with pat_build -O experiment_data_directory/build-options.apa.

(One or more) MPICH_RANK_ORDER_FILE files
: Contain suggestions for optimizing MPI rank placement.

In addition, pat_report, a first-level data analysis tool available through the perftools-base module, can be used to process the xf-files, create text reports, and export data for additional analysis. See a sample text report below. Read pat_report -h and **man pat_report** to learn its many options and environment variables. The utility pat_info is used to generate a quick summary of the contents of the experiment_data_directory.

Cray Apprentice2/3

Apprentice2 or Apprentice3 (app2 or app3) is a graphical user interface (GUI) for visualizing and analyzing performance data. It is a second-level data analysis tool available through the perftools-base or the perftools module. Compared to Apprentice2, Apprentice3 provides enhanced scalability, an improved user interface, and advanced metrics for more detailed and efficient performance analysis. Apprentice3 can help to identify load imbalance, excessive serialization, and excessive communication and network contention. It can also provide call-graph-based profiling information with source code mapping and time-based tracing.

Note: The Cray Programming Environment version 25.09 installed on Athena provides only Apprentice3.

Performance raw data in .xf format cannot be processed by app2/3. Use pat_report to convert .xf to .ap2 format first.

To analyze *.ap2 file(s), use:

```
% module load perftools-base
% app3 experiment_data_directory
or
% app3 filename.ap2
```

See some sample graphs generated by app3.

To reduce latency, download one of the following Cray Apprentice3 clients from /opt/cray/pe/perftools/25.09.0/share/desktop_installers on Athena and install on your local machine:

- Apprentice3Installer-25.09.0-5.dmg (for MacOS)
- Apprentice3Installer-25.09.0-5.exe (for Windows OS)

Then, use the Apprentice3 GUI locally to analyze performance data collected on the NAS compute nodes.

Perftools Modules

This section provides a brief description of some of the Perftools modules that are located in /opt/cray/pe/perftools/25.09.0/modulefiles.

- **perftools-base**

  This is CrayPat's fundamental module. Loading it provides access to:

  - Man pages (such as app3, intro_craypat, intro_papi, papi_counters, pat_api, pat_build, pat_report, pat_run, pat_info, pat_help, pat_view, pat_python, perftools-base, perftools-lite, etc.).
  - help systems, such as pat_help.
  - PAPI.
  - Utilities such as pat_run, pat_build, pat_report and many other pat_*xxx*.
  - Apprentice3.
  - Additional instrumentation modules such as perftools and perftools-lite-*xxx* visible through the command module avail perftools.

  This module does not add compiler flags and will not alter the program's behavior. It can be left loaded even when building and running programs without CrayPat instrumentation. It can remain loaded for access to Apprentice3.

  WARNING: In addition to perftools-base, only one of the other perftools module described below should be loaded.
- **perftools-preload**

  Loading this module will use the environment variable LD_PRELOAD to instrument and collect performance data at runtime using the pat_run utility.

- **perftools-lite** and other **perftools-lite-*xxx***

  Load one of these modules prior to compiling the program. When execution is completed, a text report is written to the stdout file in addition to data under the experiment_data_directory.

  perftools-lite
  : Performs a sampling experiment with a text report including the execution time, aggregate MFLOP count, top time-consuming functions and routines, and MPI behavior in the user's program.

  perftools-lite-events
  : Performs a tracing experiment with a text report of a profile of the top functions traced and node observation, and possible rank order suggestions.

  perftools-lite-loops
  : Performs a tracing experiment focusing on estimating loop performance. After program execution completes, a performance data file (.ap2) is produced for use with Apprentice3.

- **perftools**

  This includes a collection of CrayPat libraries for collecting and recording performance data. It enables instrumentation with full access to CrayPat functionality. Load this module, then use pat_build to instrument and pat_report to process the data collected and generate reports.

Perftools Usage Examples

To access man pages or the help system:

```
% module load perftools-base
% man intro_craypat
% pat_help
% pat_help FAQ
% pat_help all . > get_help_and_exit.txt
```

To use the pat_run approach—which allows a run-time instrumentation using either your existing executable or a new executable created during link-step with the perftools-preload module loaded—do the following:

Note: If you want the performance analysis report to include source line numbers, your executable must be built with the-g compiler flag.

```
% module use /opt/cray/pe/perftools/25.09.0/modulefiles
```

```
% module load perftools-base

Optional:
% module load perftools-preload          # for relinking to create a new executable

- Do the link step (assuming you have the *.o)
  to allow the new executable to access nearly all
  the performance analysis features.
  Without this step, pat_run can still be used, but it can
  only provide a limited set of instrumentation features,
  primarily basic sampling.
End_of_optional

% module use /opt/cray/pals/modulefiles
% module load cray-pals
% mpiexec [mpiexec options] pat_run [pat_run options] your_executable
CrayPat/X:  Version 25.09.0 ...
...
Experiment data directory written:
/path/to/experiment_data_directory        # xf-files subdirectory generated under this directory

% pat_report experiment_data_directory    # This generates ap2-files subdirector
```

The following are some pat_run options. Without any option, the default is to do a sample call stack over time (samp_cs_time) experiment.

-E *events*
: Specifies performance counters. For example:

- -E "PAPI_TOT_CYC,PAPI_TOT_INS,PAPI_TLB_DM,PAPI_FP_OPS", -E default, or -E 0, which have the same effect of collecting data for the PAPI_TOT_CYC,PAPI_TOT_INS,PAPI_TLB_DM,PAPI_FP_OPS counters.
- -E fp_stats collects data for the PAPI_TOT_CYC,PAPI_TOT_INS,RETIRED_FP_OPS_BY_TYPE:SCALAR_ALL, RETIRED_FP_OPS_BY_TYPE:VECTOR_ALL
- -E "PAPI_TOT_CYC,RETIRED_FP_OPS_BY_WIDTH:PACK128_UOPS_RETIRED, RETIRED_FP_OPS_BY_WIDTH:PACK256_UOPS_RETIRED,RETIRED_FP_OPS_BY \ _WIDTH:PACK512_UOPS_RETIRED", to see whether the retired floating-point operations in your run are dominated by 128, 256, or 512 bit vector operations.

Notes:

- One of the -E lines listed above is too long to be formatted as one line, so it is broken with a backslash (\).
- You can also set the environment variable PAT_RT_PERFCTR instead of using the -E option.

-g *group-names*
: Traces all functions that belong to group names, such as mpi, omp, blas, lapack, fftw, hdf5, netcdf, heap, io, lustre . See -g option of **man pat_build** for a full list.
TIP: After the experiment has been completed, use pat_report -s traced_functions=show experiment_data_directory to see the traced functions called.

-r
: Automatically generates a text report (under the rpt-files subdirectory of experimen_data_directory) at the end of a run.

-s *sampling-spec*
: Performs a sampling experiment with the given specification. For example, -s cs,time will capture the entire call stack using a time interval to trigger a sampling interrupt, while -s pc,time will capture the program address instead of the call stack.

-t
: Records a full trace of performance data, whether the data is collected by sampling or tracing. See **man intro_craypat** for an explanation of the terminology. This option provides a convenient way to set PAT_RT_SUMMARY to 0.

-w
: Performs a tracing experiment.

TIP: To do a tracing experiment of a group of functions, such as mpi, and avoid summarizing performance data that leads to losing detailed trace information, use -w -t -g mpi.

To do a lightweight time sampling profiling, which does a compile-time instrumentation:

```
% module use /opt/cray/pe/perftools/25.09.0/modulefiles
% module load perftools-base
% module load perftools-lite
% module load PrgEnv-xxx
      # Load the module with your choice of compiler
```

Compile and link the program to generate an instrumented executable. For example:

```
% ftn [options] -o your_executable your_program.f
% module use /opt/cray/pals/modulefiles
% module load cray-pals
% mpiexec [mpiexec options] ./your_executable
CrayPat/X:  Version 25.09.0 ...
..
# A text report is generated together with experiment_data_directory.
#######################################
```

```
#                                      #
# CrayPat-lite Performance Statistics #
#                                      #
########################################
....
For a complete report with expanded tables and notes, run:
pat_report /path/to/experiment_data_directory

% pat_report /path/to/experiment_data_directory

% module unload perftools-lite
    # to prevent further instrumentation in a later build
```

To use the pat_build approach, which does a link-time instrumentation:

```
% module use /opt/cray/pe/perftools/25.09.0/modulefiles
% module load perftools-base
% module load perftools
```

If the *.o files of your application are no longer available, recompile with the -c option. For example:

```
% ftn -c [compiler options] your_program.f
    # This generates your_program.o
% ftn -o your_executable your_program.o
    # This must be done with the module perftools loaded
```

Note: The Cray compiler wrappers (ftn, cc, CC) must be used during the link step in order to use with pat_build. Otherwise, pat_build will fail with this message:

```
ERROR: Missing required ELF section '.note.link' from the program ...
```

```
% pat_build [pat_build options] your_executable # This instruments your_executable
                                    # and generates your_executable+pat
% module use /opt/cray/pals/modulefiles
% module load cray-pals
% mpiexec [mpiexec options] ./your_executable+pat
CrayPat/X:  Version 25.09.0 ...
...
Experiment data directory written:
/path/to/experiment_data_directory              # The directory contains xf-files

% pat_report /path/to/experiment_data_directory # This generates a text performance output
                                    # and creates the ap2-files directory
                                    # may also create a *.apa file
% app3 /path/to/experiment_data_directory       # Use app3 GUI to analyze performance
```

If the experiment_data_directory contains a build-options.apa file that includes recommended arguments for further performance analysis, then use the following to generate your_executable+**apa**:

```
% pat_build -O /path/to/experiment_data_directory/build-options.apa
```

With this new instrumented executable, you can perform further analysis with the following:

```
% mpiexec [mpiexec options] ./your_executable+apa
```

Here are some pat_build options to consider:

-g *group_names*
: Traces all functions that belong to *group_names*.

-t *trace-file*
: Traces all functions listed in the *trace-file*.

-u
: Traces functions defined by user in the program source files. This analysis is not available with pat_run.

-w
: Makes tracing the default experiment.

-D report=y
: Automatically generates reports at the end of the run.

-O apa
: *apa* stands for automatic profiling analysis; this is the default if neither a sampling nor a tracing experiment is specified. It first profiles the application for time usage, and it generates output in a template file (*.apa) which shows suggested pat_build options for subsequent performance analysis.

-S
: Runs a sampling experiment; good for finding what functions are most time consuming. This option can not be used together with other options that do tracing experiments, such as -g, -t, -u, -w, -T.

-T *function_name*
: Instruments a specific function. This analysis is not available with pat_run.

Some environment variables to consider for pat_build:

- **PAT_RT_EXPERIMENT**

  For example, export PAT_RT_EXPERIMENT=samp_cs_time lets you perform a sampling of call stack over time. Note that sampling

the call stack reveals the sequence of funtion calls (by capturing a series of memory addresses) that led to that location. Other choices, as seen in the output of the pat_help experiment topic, include:

samp_pc_time
: This experiment samples the Program Counter (PC) at specific time intervals, where a program counter is a special CPU register that holds the memory address of the next instruction to be executed.

samp_cs_ovfl
: This experiment captures call stack information when a hardware counter overflows. Setting a counter and an overflow threshold is required for this experiment. Below are two examples:

```
export PAT_RT_PERFCTR=PERF_COUNT_HW_CPU_CYCLES@1000000
or
export PAP_RT_PERFCTR=PAPI_FP_OPS@10000
```

samp_pc_ovfl
: This experiment captures data based on the overflow of PC.

trace
: This experiment requires the program be instrumented to trace a specific set of functions.

- **PAT_RT_PERFCTR**

  Run papi_avail to see the PAPI preset events. Up to four counters can be specified. For example:

```
export PAT_RT_PERFCTR="PAPI_FP_OPS,PAPI_TOT_INS"
   ## lets you instrument floating-point operations and total instructions executed.
   ## You can also
   ## choose a predefined counter number.

export PAT_RT_PERFCTR=0
   ## will instrument PAPI_TOT_CYC, PAPI_TOT_INS, PAPI_TLB_DM, and PAPI_FP_OPS

export PAT_RT_PERFCTR=1
   ## will instrument PAPI_TOT_CYC, PAPI_TOT_INS, PAPI_BR_INS, and PAPI_BR_TKN

export PAT_RT_PERFCTR=fp_stats
   ## will instrument PAPI_TOT_CYC, PAPI_TOT_INS, RETIRED_FP_OPS_BY_TYPE:SCALAR_ALL, and
   ## RETIRED_FP_OPS_BY_TYPE:VECTOR_ALL
```

  Read **man hwpc** for more information.

- **PAT_RT_SUMMARY**

  Controls runtime summarization. When set to 1 (default), it summarizes/aggregates data, leading to smaller files. When set to 0, it disables summarization to capture detailed trace data.

- **PAT_RT_EXPFILE_PES**

  Specifies the individual PEs from which to collect and record data. Default: All PEs.

- **PAT_RT_TRACE_DEPTH**

  This variable lets you set the default maximum depth of the runtime call stack for traced function during runtime summarization. The default is 512.

For more choices of environment variables, see the output of the pat_help environment option.

Sample Text Report

This example demonstrates using a run-time instrumentation approach with pat_run. Specifically:

- An executable named bt.B.x from the NAS Parallel Benchmarks (NPB) 3.4.2 was built.
- The environment used during compile/link time was PrgEnv-cray with perftools-base loaded but not perftools-preload, perftools-lite-xxx or perftools.
- The driver and compiler flag used were ftn -g.
- Two nodes were used to run eight MPI processes per node.
- The MPI processes were pinned using the --cpu-bind option of mpiexec.
- The run was profiled with pat_run -r which will automatically provide a text summary file. This avoids the explicit step of using pat_report to process the raw data directory xf-files.

  ```
  mpiexec -np 16 --ppn 8 --cpu-bind list:0,32,64,96,128,160,192,224 pat_run -r bt.B.x
  ```

- An experiment_data_directory named bt.B.x+*NNNN-NNNN*s was generated which contains the automatically generated directories xf-files, ap2-files, and rpt-files. *NNNN-NNNN*s is a sequence of numbers used to uniquely identify the run. The s at the end of the string indicates that this is from a sampling experiment.

  Note: For tracing experiments, the name of the experiment_data_directory ends with t.

- A high-level text summary of the performance analysis result was included in the file RUNTIME.rpt under the rpt-files directory.

The text file RUNTIME.rpt contains some information of the experiment details, such as the hardware, programming environment,

number of MPI ranks or threads, how the executable was instrumented, runtime environment variables, and the experiment type and sampling interval:

```
Experiment:  samp_cs_time

Sampling interval:  10000 microsecs
```

The default text summary report of the samp_cs_time (sample call stack over time) experiment provides a glimpse of where the run spent most of its time, what functions and source line were heavily used, the imbalance between ranks, some statistics of four hardware counters, some I/O statistics, memory usage, process time, and memory high water mark by rank. For this run, high-level profiling results were divided into seven tables:

1. samp_profile
2. profile_max
3. samp_profile+src
4. hwpc
5. write_stats
6. himem
7. program_time

Some tables are truncated in this display to save space. No table for read_stats was generated—probably because no read samples were collected.

Here are the seven tables reported for this run:

```
Table 1:  Sample Profile by Function

  Samp% |  Samp | Imb. |  Imb. | Group
        |       | Samp | Samp% |  Function
        |       |      |       |  PE=HIDE

 100.0% | 636.3 |   -- |    -- | Total
|-----------------------------------------------
|  92.4% | 588.1 |   -- |    -- | USER
||----------------------------------------------
||  28.1% | 178.9 | 20.1 | 10.8% | binvcrhs_
||  15.7% |  99.9 | 19.1 | 17.1% | z_solve_cell_
||  15.1% |  95.9 | 13.1 | 12.8% | y_solve_cell_
||  13.9% |  88.4 | 15.6 | 16.0% | x_solve_cell_
||   6.2% |  39.5 |  8.5 | 18.9% | matmul_sub_
||   5.3% |  33.7 |  3.3 |  9.5% | compute_rhs_
||   2.6% |  16.5 |  7.5 | 33.3% | matvec_sub_
||   2.3% |  14.4 |  2.6 | 16.5% | x_solve_
||   1.0% |   6.2 |  3.8 | 40.7% | y_solve_
||==============================================
|   7.6% |  48.1 |   -- |    -- | MPI
||----------------------------------------------
||   5.8% |  36.9 | 12.1 | 26.4% | MPI_WAIT
||   1.0% |   6.1 |  5.9 | 52.2% | MPI_WAITALL
|===============================================
```

Note: This table shows only lines with Samp% > 0.95.

```
Table 2:  Profile of maximum function times

  Samp% |  Samp | Imb. |  Imb. | Function
        |       | Samp | Samp% |  PE=[max,min]
|-----------------------------------------------------
| 100.0% | 199.0 | 20.1 | 10.8% | binvcrhs_
||----------------------------------------------------
|| 100.0% | 199.0 |   -- |    -- | pe.4
||  82.9% | 165.0 |   -- |    -- | pe.14
||====================================================
|  59.8% | 119.0 | 19.1 | 17.1% | z_solve_cell_
||----------------------------------------------------
||  59.8% | 119.0 |   -- |    -- | pe.14
||  39.7% |  79.0 |   -- |    -- | pe.4
||====================================================
|  54.8% | 109.0 | 13.1 | 12.8% | y_solve_cell_
||----------------------------------------------------
||  54.8% | 109.0 |   -- |    -- | pe.15
||  41.7% |  83.0 |   -- |    -- | pe.9
||====================================================
|  52.3% | 104.0 | 15.6 | 16.0% | x_solve_cell_
||----------------------------------------------------
||  52.3% | 104.0 |   -- |    -- | pe.5
||  37.2% |  74.0 |   -- |    -- | pe.8
||====================================================
|  24.6% |  49.0 | 12.1 | 26.4% | MPI_WAIT
||----------------------------------------------------
||  24.6% |  49.0 |   -- |    -- | pe.2
...
```

Note: This table shows only lines with Samp% > 0.95. PE refers to an MPI rank.

Table 3: Sample Profile by Group, Function, and Line

```
  Samp% |  Samp | Imb. |  Imb. | Group
        |       | Samp | Samp% |  Function
        |       |      |       |   Source
        |       |      |       |    Line
        |       |      |       |     PE=HIDE

 100.0% | 636.3 |   -- |    -- | Total
|-------------------------------------------------------------------------
|  92.4% | 588.1 |   -- |    -- | USER
||------------------------------------------------------------------------
||  28.1% | 178.9 |   -- |    -- | binvcrhs_
3|        |       |      |        |  NPB3.4.2/NPB3.4-MPI/BT/solve_subs.f90
||||---------------------------------------------------------------------
4|||   3.5% |  22.5 |  7.5 | 26.7% | line.287
4|||   1.6% |  10.4 |  4.6 | 32.9% | line.332
4|||   1.1% |   6.9 |  4.1 | 39.4% | line.382
4|||   2.6% |  16.3 |  7.7 | 34.2% | line.440
4|||   2.2% |  13.9 |  5.1 | 28.4% | line.453
||||=====================================================================
||  15.7% |  99.9 |   -- |    -- | z_solve_cell_
3|        |       |      |        |  NPB3.4.2/NPB3.4-MPI/BT/z_solve.f90
||  15.1% |  95.9 |   -- |    -- | y_solve_cell_
3|        |       |      |        |  NPB3.4.2/NPB3.4-MPI/BT/y_solve.f90
||  13.9% |  88.4 |   -- |    -- | x_solve_cell_
3|        |       |      |        |  NPB3.4.2/NPB3.4-MPI/BT/x_solve.f90
||   6.2% |  39.5 |   -- |    -- | matmul_sub_
3|        |       |      |        |  NPB3.4.2/NPB3.4-MPI/BT/solve_subs.f90
||||---------------------------------------------------------------------
4|||   2.6% |  16.3 |  8.7 | 37.1% | line.71
4|||   1.0% |   6.6 |  6.4 | 52.3% | line.121
4|||   1.0% |   6.2 |  6.8 | 55.4% | line.171
||||=====================================================================
||   5.3% |  33.7 |   -- |    -- | compute_rhs_
...
```

Note: This table shows only lines with Samp% > 0.95.

Table 4: Program HW Performance Counter Data

PE=HIDE

```
=====================================================
  Total
-----------------------------------------------------
  Thread Time             6.413347 secs
  PAPI_TOT_CYC       23,747,805,751
  PAPI_TOT_INS       69,857,826,048
  PAPI_TLB_DM            29,748,174
  PAPI_FP_OPS        42,218,093,310
=====================================================
```

Note: This table shows the average of all ranks.

The four counters sampled in a samp_cs_time experiment for this run are:

PAPI_TOT_CYC
: The total CPU cycles executed in user mode

PAPI_TOT_INS
: The total instructions completed

PAPI_TLB_DM
: Translation Lookaside Buffer Data Misses

PAPI_FP_OPS
: The total number of floating-point operations (FLOPS) executed

Note: You can estimate the average FLOPS rate per rank by number of PAPI_FP_OPS / number of Thread Time.

Table 5: File Output Stats by Filename

```
      Avg |      Avg |  Write Rate | Number |     Avg | Bytes/ | File Name=!x/^/(proc|sys)/
    Write |    Write | MiBytes/sec |     of | Writes |   Call |  PE=HIDE
 Time per |  MiBytes |             | Writer |    per |        |
   Writer |      per |             |  Ranks | Writer |        |
     Rank |   Writer |             |        |   Rank |        |
          |     Rank |             |        |        |        |
|-------------------------------------------------------------------------
| 0.000161 | 0.001130 |    7.019092 |     16 |   19.3 |  61.33 | _UnknownFile_
| 0.000011 | 0.002022 |  182.686324 |      1 |   56.0 |  37.86 | stdout
```

```
|============================================================================
```

Table 6: Memory High Water Mark by Numa Node

```
  Process |     HiMem |     HiMem |     HiMem |     HiMem | Numanode
    HiMem | Numa Node | Numa Node | Numa Node | Numa Node |  PE=HIDE
 (MiBytes) |         0 |         1 |         2 |         3 |
         | (MiBytes) | (MiBytes) | (MiBytes) | (MiBytes) |
|---------------------------------------------------------------------------
|     131.2 |     103.3 |      19.2 |       4.6 |       4.1 | numanode.0
|     130.3 |       8.5 |     112.5 |       4.9 |       4.4 | numanode.1
|     131.8 |       9.2 |       5.9 |      98.8 |      17.9 | numanode.2
|     130.7 |      22.9 |       7.4 |       4.8 |      95.6 | numanode.3
|===========================================================================
```

The value shown for **Process HiMem** is calculated from information in the /proc/self/numa_maps files captured near the end of the program. It is the total size of all pages (including huge pages) that were actually mapped into physical memory from both private and shared memory segments.

Table 7: Wall Clock Time, Memory High Water Mark

```
  Process |   Process | PE=[mmm]
     Time |     HiMem |
          | (MiBytes) |

 6.431709 |     131.0 | Total
|-------------------------------
| 6.436082 |     129.7 | pe.1
| 6.430993 |     130.5 | pe.6
| 6.427668 |     134.1 | pe.9
|===============================
```

Note: This table shows only the maximum, median, and minimum PE entries, sorted by **Process Time**.

The data shown in the tables of a default output do not show rank-by-rank results. If a row does not display a PE (process) number, the data shown are the average of all ranks. If a PE number is displayed for a row, the data shown are either the maximum, minimum, or median of all ranks. To get rank-by-rank data for a specific table, you can run the following:

```
% pat_report -O samp_profile_all experiment_data_directory
```

or

```
% pat_report -O samp_profile -s pe=ALL experiment_data_directory
```

If you want rank-to-rank data for more than one tables, you can use a comma-separated (without space) list. For example:

```
% pat_report -O samp_profile_all,samp_profile+src_all,hwpc_all experiment_data_directory
```

or

```
% pat_report -O samp_profile,samp_profile+src,hwpc -s pe=ALL experiment_data_directory
```

If you want rank-to-rank data for all tables, use:

```
% pat_report -s pe=ALL experiment_data_directory
```

To get a more detailed explanation of each table, add -v after pat_run. For example:

```
% pat_report -v -O samp_profile experiment_data_directory
```

Read pat_report -s -h, pat_report -d -h, and pat_report -b -h to learn more on the output generated with pat_report -v.

The data collected in experiment_data_directory may contain more information beyond what is shown in the default summary report. You can try pat_report -O -h to see what options are available then try out some of the options. Keep in mind that not all options of -O are applicable to the data collected for a run. For example:

```
% pat_report -O -h
pat_report: Help for -O option:
Available option values are in left column, a prefix can be specified:

D1_D2_observation       D1 + D2 cache utilization
D1_D2_util              Functions with low D1+D2 cache hit ratio
D1_observation          D1 cache utilization
D1_util                 Functions with low D1 cache hit ratio
...
calltree                Calltree View by Function
calltree+hwpc           Calltree View by Function with Counters
calltree+src            Calltree View with Callsite Line Numbers
calltree+src+hwpc       Calltree View with Line Numbers and Counters
...
mpi_callers             MPI Message Stats by Caller
mpi_coll_callers        MPI Collective Message Stats by Caller
mpi_dest_bytes          MPI Sent Message Stats by Distance Sorted by Bytes
```

```
mpi_dest_counts          MPI Sent Message Stats by Distance Sorted by Counts
mpi_hy_rank_order        MPI Hybrid Rank Order
mpi_observation          MPI utilization
mpi_p2p_bytes            MPI P2P Message Bytes, Src to Dest
mpi_p2p_map              Mapping of MPI P2P Source PE and callstack to number of destination PEs
mpi_p2p_map_observation  MPI Collective Recommendations
mpi_p2p_recv_time        MPI P2P Message Receive Time, Src to Dest
mpi_p2p_send_time        MPI P2P Message Send Time, Src to Dest
mpi_rank_order           Metric-Based Rank Order
...
p2p                      Abbreviation for:  mpi_p2p_bytes,mpi_p2p_send_time,mpi_p2p_recv_time
...
```

For this bt.B.x example, the following two commands with the options calltree and p2p did generate reports which provide a calltree view by function and statistics of p2p, which include data on mpi_p2p_bytes, mpi_p2p_send_time, mpi_p2p_recv_time.

```
% pat_report -O calltree experiment_data_directory
% pat_report -O p2p experiment_data_directory
```

On the other hand, the experiment_data_directory from this experiment does not have data on functions with low D1 cache hit ratio; therefore, the following command produced an error:

```
pat_report -O D1_util dir.a

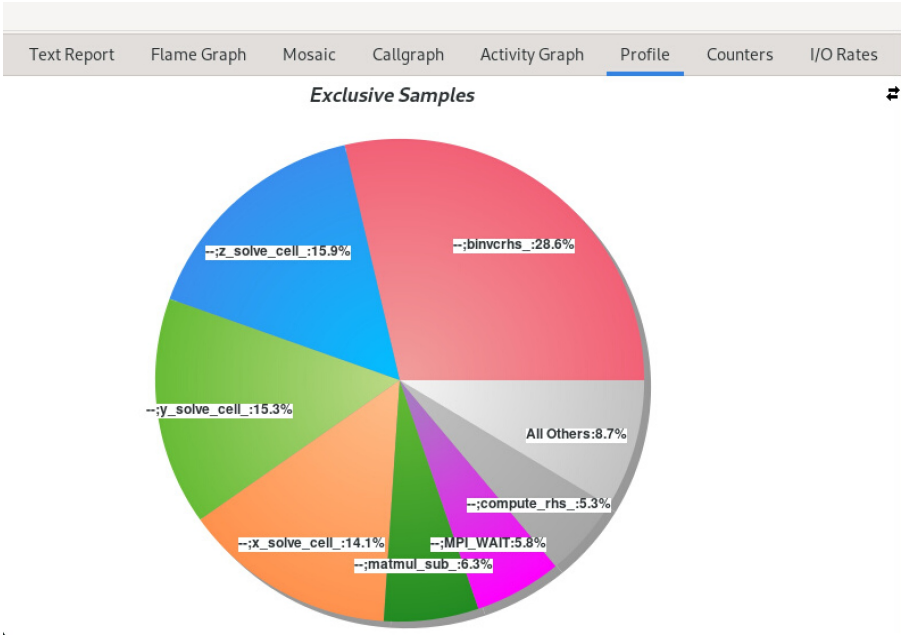
### Error: 'D1_cache_hit_ratio not valid in -d option, or has no data'
```

TIP: Instead of trying out different options displayed by the command pat_report -O -h, it is easier to use the app3 tool which automatically generates texts or graphs of all data collected from the run.

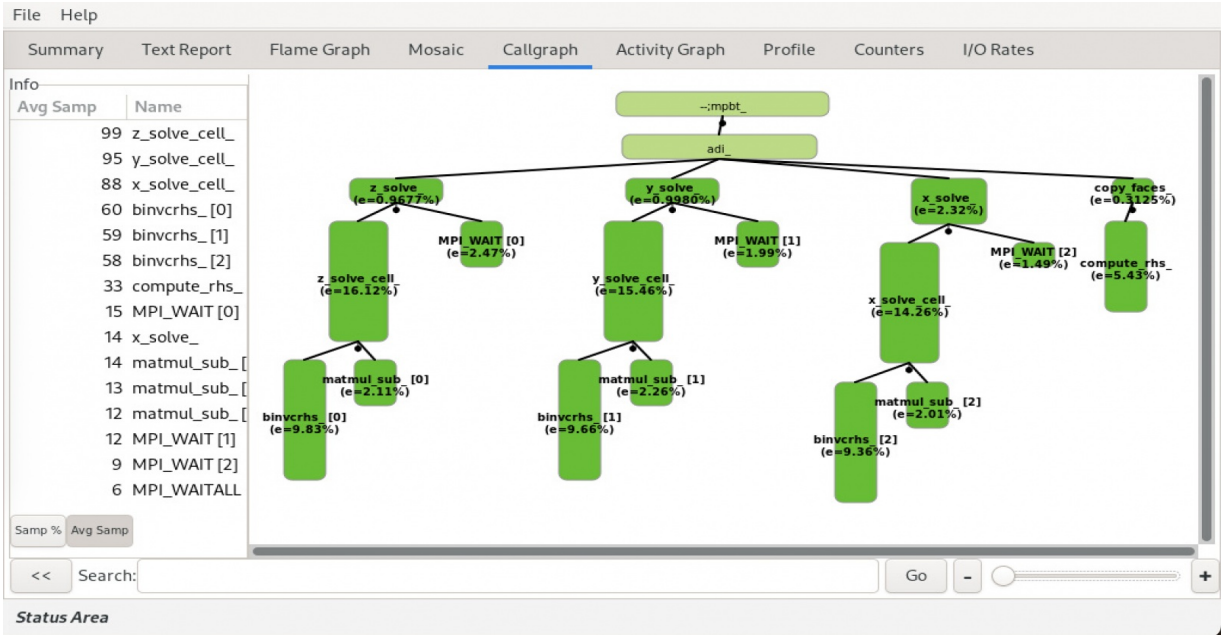
Sample Graphs from Apprentice3

The Apprentice3 visualization tool allows you to visualize, explore, and analyze performance data captured. From the **Text Report** pulldown menu, you can select text reports for various options (such as those shown in the seven tables above and also some which are not included in the default). In addition, graphical presentation of some reports may be available from other pulldown menus.

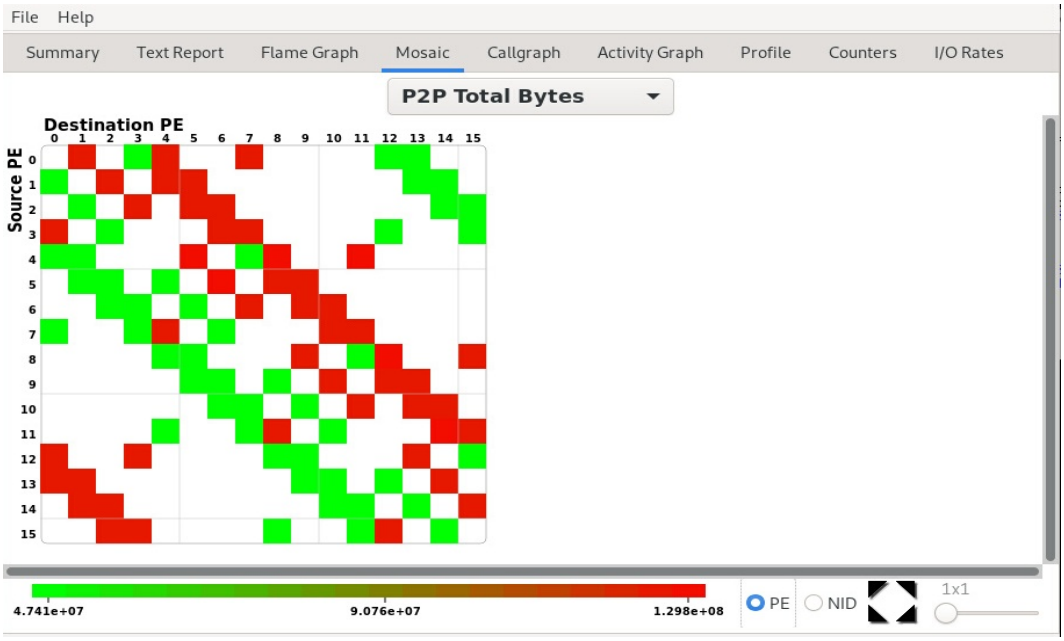
For example, for the same run shown in the Sample Text Report, this image displays the percentage of samples collected from various functions in a pie chart from the **Profile** pulldown menu:



This image illustrates a callgraph (with the percentage of samples collected in each function) from the **CallGraph** pulldown menu:



Another image shows the P2P Total Bytes from the **Mosaic** pulldown menu:



References

- [Cray Performance and Analysis Tools]

In the Using Cray Message Passing Toolkit on Athena article, a method is described that allows applications built with Intel MPI to run on Athena, and take advantage of the Cray Slingshot 11 interconnect through the use of the cray-mpich-abi and cray-pals modules. In this article, we describe an alternative approach that doesn't use Cray MPICH while still benefiting from the high-performance Cray Slingshot 11 interconnect.

Specifically, this approach relies on setting a few Intel MPI environment variables to use Cray's Libfabric, PMI2 libraries, and the OpenFabrics Interface (OFI) CXI provider, and launching with the Cray Parallel Application Launch Service (PALS).

More information is included in the comments of the following sample script.

Note: The sample script below assumes that you're using the **csh** shell. For **bash**, replace the setenv command syntax with the export command syntax, as follows:

```
#for csh:
setenv variable_name value

#for bash:
export variable_name=value
```

Sample Script

```
## This example runs an Intel MPI app on 2 nodes with 256 processes each.
#PBS -lselect=2:ncpus=256:mpiprocs=256:model=tur_ath
#PBS -lwalltime=2:00:00
#PBS -q normal
#PBS -j oe
#PBS -koed

## The executable will be launched with mpiexec from Cray PALS instead
## of from Intel MPI. This is due to the need to satisfy
## Virtual Network Interface (VNI) enforcement and security.
## Load Intel MPI before Cray PALS to ensure CRAY PALS mpiexec is used.

## If any, also load other modules needed by the application.

module use -a /nasa/modulefiles/testing
module purge
module load mpi-intel/2021.16
module load cray-pals

## Check that the mpiexec is from Cray Pals, not Intel MPI.

which mpiexec
mpiexec --version

## Use Cray's PMI2 instead of PMI to allow proper interaction with Intel MPI.

setenv I_MPI_PMI pmi2
setenv I_MPI_PMI_LIBRARY /opt/cray/pe/pmi/default/lib/libpmi2.so

## The Open Fabrics Interfaces (OFI) cxi provider is not distributed
## in the libfabric of Intel MPI 2021.17 and earlier versions.
## Direct Intel MPI not to use libfabric from Intel MPI and instead use
## the libfabric library from Cray and use the cxi provider.

setenv I_MPI_OFI_LIBRARY_INTERNAL 0
setenv I_MPI_OFI_LIBRARY /opt/cray/libfabric/2.2.0rc1/lib64/libfabric.so.1
setenv I_MPI_OFI_PROVIDER cxi

## (Optional) Direct Intel MPI to check the OFI provider used.

#setenv I_MPI_DEBUG 1

## Direct Cray PALS mpiexec to use the PMI2 interface.

mpiexec -np 512 --ppn 256 \
--pmi=pmi2   \
executable_built_with_intel_mpi
```