Workload Management and Application Placement for the Cray Linux Environment™

S–2496–42

RECORD OF REVISION

S–2496–42 Published April 2013 Supports the Cray Linux Environment (CLE) 4.2 release.

S–2496–4101 Published December 2012 Supports the Cray Linux Environment (CLE) 4.1.UP01 general availability (GA) release.

S–2496–5001 Published November 2012 Supports the Cray Linux Environment (CLE) 5.0.UP01 limited availability (LA) release.

S–2496–41 Published August 2012 Supports the Cray Linux Environment (CLE) 4.1.UP00 limited availability (LA) release.

S–2496–4003 Published March 2012 Supports the Cray Linux Environment (CLE) 4.0.UP03 release.

S–2496–4002 Published December 2011 Supports the Cray Linux Environment (CLE) 4.0.UP02 release.

S–2496–4001 Published September 2011 Supports the Cray Linux Environment (CLE) 4.0.UP01 release.

S–2496–3103 Published March 2011 Supports the Cray Linux Environment (CLE) 3.1.UP03 release.

S–2496–3102 Published December 2010 Supports the Cray Linux Environment (CLE) 3.1.UP02 release.

S–2496–31 Published June 2010 Supports Cray Linux Environment (CLE) 3.1 release.

Changes to this Document

Workload Management and Application Placement for the Cray Linux Environment™ S–2496–42

Revised information:

- Added `-e` option to `aprun` command in Chapter 2, Running Applications on page 15.
- Added a limitation to Chapter 7, Using Checkpoint/Restart on page 69. Single rank MPI applications are not supported with checkpoint/restart.

Contents

System Overviews [1]

1.1 Cray System Features

Cray XE and Cray XK supercomputers are massively parallel processing (MPP) systems. Cray has combined commodity and open source components with custom-designed hardware and software to create a system that can operate efficiently at an immense scale.

Cray systems are based on the Red Storm technology that was developed jointly by Cray Inc. and the U.S. Department of Energy Sandia National Laboratories. Cray systems are designed to run applications that require large-scale processing, high network bandwidth, and complex communications. Typical applications are those that create detailed simulations in both time and space, with complex geometries that involve many different material components. These long-running, resource-intensive applications require a system that is programmable, scalable, reliable, and manageable.

The Cray XE series consists of Cray XE5 and Cray XE6 systems. The Cray XK series consists of Cray XK6 systems. Both systems use the Gemini™ high speed system interconnect. Cray XK6 systems are *hybrid* supercomputers where each node has both an AMD Opteron 6200 series CPU and an NVIDIA GPGPU (General Purpose Graphics Processing Unit) that serves as a highly-threaded coprocessor especially suited for datasets in the SIMD computational domain.

The major features of Cray systems are performance, scalability and resiliency:

- Cray systems are designed to scale to more than 1 million ranks. The ability to scale to such proportions stems from the design of system components and software:
  - The basic component is the *node*. There are two types of nodes. Service nodes provide support functions, such as managing the user's environment, handling I/O, and booting the system. Compute nodes run user applications. Because processors are inserted into standard sockets, customers can upgrade nodes as faster processors become available. Compute nodes consist of subsets called non-uniform memory access (NUMA) nodes, whose boundaries are defined by processor dies. Each NUMA node consists of a set of execution cores and memory. Inter-NUMA node operations within the same compute node will be slower than intra-NUMA node operations—this is the "non-uniformity" of memory access within and between NUMA nodes.

On the Cray XE6 compute blade, consisting of AMD Opteron 6100 and 6200 Series Processors, there are two die per package. Thus, there are four NUMA nodes per compute node. On the Cray XK6 compute blade there is one AMD Opteron 6200 processor, therefore there are two NUMA nodes per compute node.

  - Cray systems use a simple memory model. Every instance of a distributed application has its own processors and local memory. Remote memory is the memory on other nodes that run the associated application instances. However, Cray systems support one-sided programming models such as Chapel and PGAS (Parallel Global Addess Space) languages that allow programs to treat application memory spaces as distributed global memories.
  - The *system interconnection network* links compute and service nodes. This is the data-routing resource that Cray systems use to maintain high communication rates as the number of nodes increases. Cray systems use a full 3D torus network topology. However, Cray XE5m and Cray XE6m use a two-dimensional topology.

- Cray system resiliency features:
  - The Node Health Checker (NHC) performs tests to determine if compute nodes that are allocated to an application are healthy enough to support running subsequent applications. If not, NHC removes any nodes incapable of running an application from the resource pool.
  - Tools that assist administrators to recover from system or node failures, including a hot backup utility, boot node failover, single or multiple compute node reboots, and warm boots.
  - Error correction code (ECC) technology, which detects multiple-bit data storage and transfer errors and corrects most single and some multiple-bit errors.
  - Lustre file system failover. When administrators enable Lustre automatic failover, Lustre services switch to standby services if the primary node fails or Lustre services are temporarily shut down for maintenance.
  - System processor boards (called *blades*) have redundant voltage regulator modules (VRMs or *verties*) or VRMs with redundant circuitry.
  - Multiple redundant RAID controllers, that provide automatic failover capability and multiple Fibre Channel and InfiniBand connections to disk storage.
  - The ability to warm swap system blades.
  - Network link failure detection and automatic rerouting.
  - Application relaunch and reconnect.

The major software components of Cray systems are:

- Application development tools, comprising:
  - Cray Application Development Environment (CADE):
    - Message Passing Toolkit (MPI, SHMEM)
    - Math and science libraries (LibSci, PETSc, ACML, FFTW)
    - Data modeling and management tools (NetCDF, HDF5)
    - GNU debugger (lgdb)
    - GCC C, C++, and Fortran compilers
    - Java (for developing service node programs)
- Application placement tools:
  - Application Level Placement Scheduler (ALPS) application launch and schedule utility.
  - Cluster Compatibility Mode allows users to run cluster-based individual software vendor applications on Cray systems.
  - Checkpoint/restart.
- Optional products:
  - C, C++, and Fortran 95 compilers from PGI and PathScale
  - glibc library (the compute node subset)
  - Chapel
  - Workload management Systems (PBS Professional, Moab and TORQUE, Platform LSF)
  - TotalView debugger
  - DDT debugger
  - Cray Performance Measurement and Analysis Tools
  - Intel Compiler Support
  - Cray Compiling Environment (CCE)
    - Cray C and compilers
    - Cray C++ compiler
    - Fortran 2003 compiler

  - The Cray C compiler supports Unified Parallel C and the Cray Fortran compiler supports co-arrays and several other Fortran 2008 features. All CCE compilers support OpenMP.
  - The CUDA toolkit

- Cray Application Development Supplement (CADES) for stand alone Linux application development platforms
- Operating system services. The operating system, Cray Linux Environment (CLE), is tailored to the requirements of service and compute nodes. A full-featured SUSE Linux operating system runs on service nodes, and a lightweight kernel, CNL, runs on compute nodes. With the compute node root runtime environment compute nodes have a chrooted, read-only view of the shared root file system to allow library linking and other such Linux services that are not included in the compute node kernel.
- Parallel file systems support. Cray supports the Lustre parallel file system. CLE also enables the Cray system to use file systems such as NFS by projecting them to compute nodes using Cray Data Virtualization Services (DVS).
- System management and administration tools:
  - System Management Workstation (SMW), the single point of control for system administration.
  - Hardware Supervisory System (HSS), which monitors the system and handles component failures. HSS is independent of computation and service hardware components and has its own network.
  - Comprehensive System Accounting (CSA), a software package that performs standard system accounting processing. CSA is open-source software that includes changes to the Linux kernel so that the CSA can collect more types of system resource usage data than under standard Fourth Berkeley Software Distribution (BSD) process accounting.

    An additional CSA interface enables the project database to use customer-supplied user, account, and project information that reside on a separate Lightweight Directory Access Protocol (LDAP) server.

Running Applications [2]

The `aprun` utility launches applications on compute nodes. The utility submits applications to the Application Level Placement Scheduler (ALPS) for placement and execution, forwards your login node environment to the assigned compute nodes, forwards signals, and manages the `stdin`, `stdout`, and `stderr` streams.

This chapter describes how to run applications interactively on compute nodes and get application status reports. For a description of batch job processing, see Chapter 4, Using Workload Management Systems on page 43.

2.1 Using the aprun Command

Use the `aprun` command to specify the resources your application requires, request application placement, and initiate application launch.

The format of the `aprun` command is:

```
aprun [-a arch ] [-b ] [-B] [-C ]
[-cc cpu_list|keyword ] [-cp cpu_placement_file_name ] [-d depth ] [-D value ]
[-e env] [-L node_list ] [-l node_list_file] [-m size[h|hs]] [-j num_cpus]
[-n pes ] [-N pes_per_node ] [-F access mode ] [-p protection domain identifier]
[-q] [-r # CPUs] [-R pe_dec] [-S pes_per_numa_node ]
[-sl list_of_numa_nodes ] [-sn numa_nodes_per_node ] [-ss] [-T]
[-t sec ] executable [ arguments_for_executable ]
```

where:

`-b`	Bypasses the transfer of the executable program to the compute nodes. By default, the executable is transferred to the compute nodes during the `aprun` process of launching an application. For an example, see Running Compute Node Commands on page 111.
`-B`	Reuses the width, depth, nppn, and memory request options that are specified with the batch reservation. This option obviates the need to specify `aprun` options `-n`, `-d`, `-N`, and `-m`. `aprun` will exit with errors if these options are specified with the `-B` option.
`-C`	Attempts to reconnect the application-control fan-out tree around failed nodes and complete application execution. To use this option, your application must use a programming model that supports reconnect. Options `-C` and `-R` are mutually exclusive.

`-cc` *cpu_list* | *keyword*

Binds processing elements (PEs) to CPUs. CNL does not migrate processes that are bound to a CPU. This option applies to all multicore compute nodes. The *cpu_list* is not used for placement decisions, but is used only by CNL during application execution. For further information about binding (*CPU affinity*), see Using `aprun` CPU Affinity Options on page 75.

The *cpu_list* is a comma-separated or hyphen-separated list of logical CPU numbers and/or ranges. As PEs are created, they are bound to the CPU in *cpu_list* corresponding to the number of PEs that have been created at that point. For example, the first PE created is bound to the first CPU in *cpu_list*, the second PE created is bound to the second CPU in *cpu_list*, and so on. If more PEs are created than given in *cpu_list*, binding starts over at the beginning of *cpu_list* and starts again with the first CPU in *cpu_list*. The *cpu_list* can also contain an `x`, which indicates that the application-created process at that location in the fork sequence should **not** be bound to a CPU.

If multiple PEs are created on a compute node, the user may optionally specify a *cpu_list* for each PE. Multiple *cpu_lists* are separated by colons (:). This provides the user with the ability to control the placement for PEs that may conflict with other PEs that are simultaneously creating child processes and threads of their own.

```
% aprun -n 2 -d 3 -cc 0,1,2:4,5,6 ./a.out
```

The example above contains two *cpu_lists*. The first (0,1,2) is applied to the first PE created and any threads or child processes that result. The second (4,5,6) is applied to the second PE created and any threads or child processes that result.

Out-of-range *cpu_list* values are ignored unless all CPU values are out of range, in which case an error message is issued. For example, if you want to bind PEs starting with the highest CPU on a compute node and work down from there, you might use this `-cc` option:

```
% aprun -n 8 -cc 10-4 ./a.out
```

If the PEs were placed on Cray XE6 24-core compute nodes, the specified `-cc` range would be valid. However, if the PEs were placed on Cray XK6 eight-core compute nodes, CPUs 10-8 would be out of range and therefore not used.

The following *keyword* values can be used:

- The `cpu` keyword (the default) binds each PE to a CPU within the assigned NUMA node. You do not have to indicate a specific CPU.

If you specify a *depth* per PE (`aprun -d` *depth*), the PEs are constrained to CPUs with a distance of *depth* between them to each PE's threads to the CPUs closest to the PE's CPU.

The `-cc cpu` option is the typical use case for an MPI application.

> **Note:** If you oversubscribe CPUs for an OpenMP application, Cray recommends that you not use the `-cc cpu` default. Test the `-cc none` and `-cc numa_node` options and compare results to determine which option produces the better performance.

- The `numa_node` keyword constrains PEs to the CPUs within the assigned NUMA node. CNL can migrate a PE among the CPUs in the assigned NUMA node but not off the assigned NUMA node.

  If PEs create threads, the threads are constrained to the same NUMA-node CPUs as the PEs. There is one exception. If *depth* is greater than the number of CPUs per NUMA node, once the number of threads created by the PE has exceeded the number of CPUs per NUMA node, the remaining threads are constrained to CPUs within the next NUMA node on the compute node. For example, on 8-core nodes, if *depth* is `5`, threads 0-3 are constrained to CPUs 0-3 and thread 4 is constrained to CPUs 4-7.

- The `none` keyword allows PE migration within the assigned NUMA nodes.

`-cp` *cpu_placement_file_name*

Provides the name of a CPU binding placement file. This option applies to all multicore compute nodes. This file must be located on a file system that is accessible to the compute nodes. The CPU placement file provides more extensive CPU binding instructions than the `-cc` options.

`-p` *protection domain identifier*

Requests use of a protection domain using the user pre-allocated protection identifier. You cannot use this option with protection domains already allocated by system services. Any cooperating set of applications must specify this same `aprun -p` option to have access to the shared protection domain. `aprun` will return an error if either the protection domain identifier is not recognized or if the user is not the owner of the specified protection domain identifier.

`-D` *value*

The `-D` option *value* is an integer bitmask setting that controls debug verbosity, where:

- A *value* of `1` provides a small level of debug messages
- A *value* of `2` provides a medium level of debug messages
- A *value* of `4` provides a high level of debug messages

Because this option is a bitmask setting, *value* can be set to get any or all of the above levels of debug messages. Therefore, valid values are `0` through `7`. For example, `-D 3` provides all small and medium level debug messages.

`-d` *depth*

Specifies the number of CPUs for each PE and its threads. ALPS allocates the number of CPUs equal to *depth* times *pes*. The `-cc` *cpu_list* option can restrict the placement of threads, resulting in more than one thread per CPU.

The default *depth* is `1`.

For OpenMP applications, use **both** the `OMP_NUM_THREADS` environment variable to specify the number of threads **and** the `aprun -d` option to specify the number of CPUs hosting the threads. ALPS creates `-n` *pes* instances of the executable, and the executable spawns `OMP_NUM_THREADS`-1 additional threads per PE. For an OpenMP example, see Running an OpenMP Application on page 95.

Note: For a PathScale OpenMP program, set the `PSC_OMP_AFFINITY` environment variable to `FALSE`.

For Cray systems, compute nodes must have at least *depth* CPUs. For Cray XE5 systems, *depth* cannot exceed 12. For Cray XK6 compute blades, *depth* cannot exceed 16. For Cray XE6 systems, *depth* cannot exceed 32.

`-e` *env*

Sets an environment variable on the compute nodes. The form of the assignment should be of the form *VARNAME=value.*. You can include multiple arguments with multiple, space-separated flag and assignment pairings, e.g. `-e` *VARNAME1=value1* `-e` *VARNAME2=value2*, etc.

`-j` *num_cpus*

Specifies how many CPUs to use per compute unit for an ALPS job. See for an explanation of compute unit affinity in ALPS. For more information on compute unit affinity, see *Using Compute Unit Affinity on Cray Systems*.

`-L` *node_list*

Specifies the candidate nodes to constrain application placement.

The syntax allows a comma-separated list of nodes (such as `-L 32,33,40`), a range of nodes (such as `-L 41-87`), or a combination of both formats. Node values can be expressed in decimal, octal (preceded by `0`), or hexadecimal (preceded by `0x`). The first number in a range must be less than the second number (`8-6`, for example, is invalid), but the nodes in a list can be in any order.

This option is used for applications launched interactively; use the `qsub -lmppnodes=\"`*node_list*`\"` option for batch and interactive batch jobs.

If the placement node list contains fewer nodes than the number required, a fatal error is produced. If resources are not currently available, `aprun` continues to retry.

A common source of node lists is the `cnselect` command. See the `cnselect`(1) man page for details.

`-m` *size*[h|hs]

Specifies the per-PE required Resident Set Size (RSS) memory size in megabytes. K, M, and G suffixes (case insensitive) are supported (16M = 16m = 16 megabytes, for example). If you do not include the `-m` option, the default amount of memory available to each PE equals (compute node memory size) / (number of PEs) calculated for each compute node.

If you want huge pages (2 MB) allocated for an application, use the `h` or `hs` suffix.

Cray XE and Cray XK: The default huge page size is 2 MB. Additional sizes are available: 128KB, 512KB, 8MB, 16MB, and 64MB.

The use of the `-m` option is not required on Cray systems because the kernel allows the dynamic creation of huge pages. However, it is advisable to specify this option and preallocate an appropriate number of huge pages, when memory requirements are known, to reduce operating system overhead. See the `intro_hugepages`(1) man page.

`-m` *size*`h` Requests memory to be allocated to each PE, where memory is preferentially allocated out of the huge page pool. All nodes use as much memory as they are able to allocate and 4 KB base pages thereafter.

`-m` *size*`hs` Requests memory to be allocated to each PE, where memory is allocated out of the huge page pool. If the request cannot be satisfied, an error message is issued and the application launch is terminated.

Note: To use huge pages, you must first link the application with `hugetlbfs`:

```
% cc -c my_hugepages_app.c
% cc -o my_hugepages_app my_hugepages_app.o
-lhugetlbfs
```

Set the huge pages environment variable at run-time:

```
% setenv HUGETLB_MORECORE yes
```

Or

```
% export HUGETLB_MORECORE=yes
```

`-n` *pes* Specifies the number of processing elements (PEs) that your application requires. A PE is an instance of an ALPS-launched executable. You can express the number of PEs in decimal, octal, or hexadecimal form. If *pes* has a leading `0`, it is interpreted as octal (`-n 16` specifies 16 PEs, but `-n 016` is interpreted as 14 PEs). If *pes* has a leading `0x`, it is interpreted as hexadecimal (`-n 16` specifies 16 PEs, but `-n 0x16` is interpreted as 22 PEs). The default value is `1`.

`-N` *pes_per_node*

Specifies the number of PEs to place per node. For Cray systems, the default is the number of available NUMA nodes times the number of cores per NUMA node.

The maximum *pes_per_node* is `32` for systems with Cray XE6 compute blades.

`-F exclusive|share`

`exclusive` mode provides a program with exclusive access to all the processing and memory resources on a node. Using this option with the `cc` option binds processes to those mentioned in the affinity string. `share` mode access restricts the application specific `cpuset` contents to only the application reserved cores and memory on NUMA node boundaries, meaning the application will not have access to cores and memory on other NUMA nodes on that compute node. The `exclusive` option does not need to be specified because exclusive access mode is enabled by default. However, if `nodeShare` is set to *share* in `/etc/alps.conf` then you must use the `-F exclusive` to override the policy set in this file. You can check the value of `nodeShare` by executing `apstat -svv | grep access`.

`-q`
: Specifies quiet mode and suppresses all `aprun`-generated non-fatal messages. Do not use this option with the `-D` (debug) option; `aprun` terminates the application if both options are specified. Even with the `-q` option, `aprun` writes its help message and any ALPS fatal messages when exiting. Normally, this option should not be used.

`-r` *cores*
: Enables core specialization on Cray compute nodes, where the number of *cores* specified is the number of system services cores per node for the application. If the `r` value is greater than one, the system services core will be assigned in a round-robin fashion to each NUMA node in descending order unless the `-cc` *cpu_list* affinity option is specified. In that case, specialized cores are assigned from the highest-order core sans those specified in *cpu_list*.

`-R` *pe_dec*
: Enables application relaunch so that should the application experience certain system failures, ALPS will attempt to relaunch and complete in a degraded manner. *pe_dec* is the processing element (PE) decrement tolerance. If *pe_dec* is non-zero, `aprun` attempts to relaunch with a maximum of *pe_dec* fewer PEs. If *pe_dec* is 0, `aprun` will attempt relaunch with the same number of PEs specified with original launch. Relaunch is supported per `aprun` instance. A decrement count value greater than zero will fail for MPMD launches with more than one element. `aprun` will attempt relaunch with `ec_node_failed` and `ec_node_halt` hardware supervisory system events only. Options `-C` and `-R` are mutually exclusive.

`-S` *pes_per_numa_node*
: Specifies the number of PEs to allocate per NUMA node. You can use this option to reduce the number of PEs per NUMA node, thereby making more resources available per PE.

  For 8-core compute nodes, the default is `4`. For 12-core compute nodes, the default is `6`. For 16-core compute nodes, the default value is `4`. For 24-core compute nodes, the default is `6`. For 32-core compute nodes, the default is `8`. A zero value is not allowed and causes a fatal error. For further information, see Using `aprun` Memory Affinity Options on page 73.

`-sl` *list_of_numa_nodes*

Specifies the NUMA node or nodes (comma separated or hyphen separated) to use for application placement. A space is required between `-sl` and *list_of_numa_nodes*. The *list_of_numa_nodes* value can be `-sl <0,1>` on Cray XE5 and Cray XK6 compute nodes, `-sl <0,1,2,3>` on Cray XE6 compute nodes, or a range such as `-sl 0-1` and `-sl 0-3`. The default is no placement constraints. You can use this option to determine whether restricting your PEs to one NUMA node per node affects performance.

List NUMA nodes in ascending order; `-sl 1-0` and `-sl 1,0` are invalid.

`-sn` *numa_nodes_per_node*

Specifies the number of NUMA nodes per node to be allocated. Insert a space between `-sn` and *numa_nodes_per_node*. The *numa_nodes_per_node* value can be `1` or `2` on Cray XE5 and Cray XK6 compute nodes, or `1`, `2`, `3`, `4` on Cray XE6 compute nodes. The default is no placement constraints. You can use this option to find out if restricting your PEs to one NUMA node per node affects performance.

A zero value is not allowed and is a fatal error.

`-ss`

Specifies strict memory containment per NUMA node. When `-ss` is specified, a PE can allocate only the memory that is local to its assigned NUMA node.

The default is to allow remote-NUMA-node memory allocation to all assigned NUMA nodes. You can use this option to find out if restricting each PE's memory access to local-NUMA-node memory affects performance.

`-T`

Synchronizes the application's `stdout` and `stderr` to prevent interleaving of its output.

`-t` *sec*

Specifies the per-PE CPU time limit in seconds. The *sec* time limit is constrained by your CPU time limit on the login node. For example, if your time limit on the login node is 3600 seconds but you specify a `-t` value of `5000`, your application is constrained to 3600 seconds per PE. If your time limit on the login node is `unlimited`, the *sec* value is used (or, if not specified, the time per-PE is unlimited). You can determine your CPU time limit by using the `limit` command (csh) or the `ulimit -a` command (bash).

Note: For OpenMP or multithreaded applications where processes may have child tasks, the time used in the child tasks accumulates against the parent process. Thus, it may be necessary to multiply the *sec* value by the depth value in order to get a real-time value approximately equivalent to the same value for the PE of a non-threaded application.

`:` (colon) Separates the names of executables and their associated options for Multiple Program, Multiple Data (MPMD) mode. A space is required before and after the colon.

2.1.1 ALPS Application Environment Variables

The following environment variables modify the behavior of `aprun`:

`APRUN_DEFAULT_MEMORY`

Specifies default per PE memory size. An explicit `aprun -m` value overrides this setting.

`APRUN_XFER_LIMITS`

Sets the `rlimit()` transfer limits for `aprun.` If this is set to a non-zero string, `aprun` will transfer the `{get,set}rlimit()` limits to `apinit`, which will use those limits on the compute nodes. If it is not set or set to `0`, none of the limits will be transferred other than `RLIMIT_CORE, RLIMIT_CPU`, and possibly `RLIMIT_RSS`.

`APRUN_SYNC_TTY`

Sets synchronous `tty` for `stdout` and `stderr` output. Any non-zero value enables synchronous `tty` output. An explicit `aprun -T` value overrides this value.

`PGAS_ERROR_FILE`

Redirects error messages issued by the PGAS library (`libpgas`) to standard output stream when set to `stdout`. The default is `stderr`.

`CRAY_CUDA_PROXY`

Overrides the site default for execution in simultaneous contexts on GPU-equipped nodes. Setting `CRAY_CUDA_PROXY` to `1` or `on` will explicitly enable the CUDA proxy. To explicitly disable CUDA proxy, set to `0` or `off`. Debugging and use of performance tools to collect GPU statistics is only supported with the CUDA proxy disabled.

`APRUN_PRINT_APID`

When this variable is set and output is not suppressed with the `-q` option, the APID will be displayed upon launch and/or relaunch.

ALPS will pass values to the following application environment variable:

`ALPS_APP_DEPTH`

Reflects the `aprun -d` value as determined by `apshepherd`. The default is `1`. The value can be different between compute nodes or sets of compute nodes when executing a MPMD job. In that case, an instance of `apshepherd` will determine the appropriate value locally for an executable.

2.1.2 Usage Output String

`utime`, `stime`, `maxrss`, `inblocks` and `outblocks` are printed to `stdout` upon application exit. The values given are approximate as they are a rounded aggregate scaled by the number of resources used. For more information on these values, see the `getrusage`(2) man page.

2.2 Understanding Application Placement

The `aprun` placement options are `-n`, `-N`, `-d`, and `-m`. ALPS attempts to use the smallest number of nodes to fulfill the placement requirements specified by the `-n`, `-N`, `-d`, `-S`, `-sl`, `-sn`, and/or `-m` values. For example, the command:

```
% aprun -n 32 ./a.out
```

places 32 PEs on:

- Cray XE5 dual-socket, quad-core processors on 4 nodes
- Cray XE5 dual-socket, six-core processors on 3 nodes
- Cray XE6 dual-socket, eight-core processors on 2 nodes
- Cray XE6 dual-socket, 12-core processors on 2 nodes
- Cray XE6 dual-socket, 16-core processors on 1 node

Note: Cray XK nodes are populated with single-socket host processors. There is still a one-to-one relationship between PEs and host processor cores.

The above `aprun` command would place 32 PEs on:

- Cray XK single-socket, eight-core processors on 4 nodes.
- Cray XK single-socket, 12-core processors on 3 nodes.
- Cray XK single-socket, 16-core processors on 2 nodes.

The memory and CPU affinity options are **optimization** options, not placement options. You use memory affinity options if you think that remote-NUMA-node memory references are reducing performance. You use CPU affinity options if you think that process migration is reducing performance.

Note: For examples showing how to use memory affinity options, see Using `aprun` Memory Affinity Options on page 104. For examples showing how to use CPU affinity options, see Using `aprun` CPU Affinity Options on page 105.

2.2.1 System Interconnnect Features Impacting Application Placement

ALPS uses interconnect software to make reservations available to workload managers through the BASIL API. The following interconnect features are used through ALPS to allocate system resources and ensure application resiliency using protection and communication domains:

- Node Translation Table (NTT) — assists in addressing remote nodes within the application and enables software to address other NICs within the resource space of the application. NTTs have a value assigned to them called the granularity value. There are 8192 entries per NTT, which represents a granularity value of 1. For applications that use more than 8192 compute nodes, the granularity value will be greater than 1.

- Protection Tag (pTag) — an 8-bit identifier that provides for memory protection and validation of incoming remote memory references. ALPS assigns a pTag-cookie pair to an application. This prevents application interference when sharing NTT entries. This is the default behavior of a private protection domain model. A flexible protection domain model allows users to share memory resources amongst their applications. For more information, see Using the `aprun` Command on page 15.

- Cookies — an application-specific identifier that helps sort network traffic meant for different layers in the software stack.

- Programmable Network Performance Counters — memory mapped registers in the interconnect ASIC that ALPS manages for use with CrayPat (Cray performance analysis tool). Applications can share a one interconnect ASIC, but only one application can have reserved access to performance counters. Thus compute nodes are assigned in pairs to avoid any conflicts.

These parameters interact to schedule applications for placement.

2.2.2 Application Placement Algorithms on Cray Systems

In previous versions of the Cray Linux Environment, applications were placed within the requested compute node resources by numerical node ID (NID) in serial order, as shown in Figure 1. Each color represents a different application, red being the largest. The larger blue spheres indicate the direction of origin in these cabinet views or torus cross-sections. The serial sequence is not necessarily ideal for placement of large applications within the actual torus topology of the Cray system. Cabinets and chassis are usually physically interleaved to reduce the maximum cable lengths. NIDs are numbered in physical order tracking these cabinet placements. While this aids in locating the physical position of the NID in cabinet space, this does not provide for an easy way to track the nodes or their interconnections within two- or three-dimensional topology space. This will likely inhibit optimal performance for larger jobs.

Figure 1. Cabinet View Showing Three Applications in Original Serial Ordering

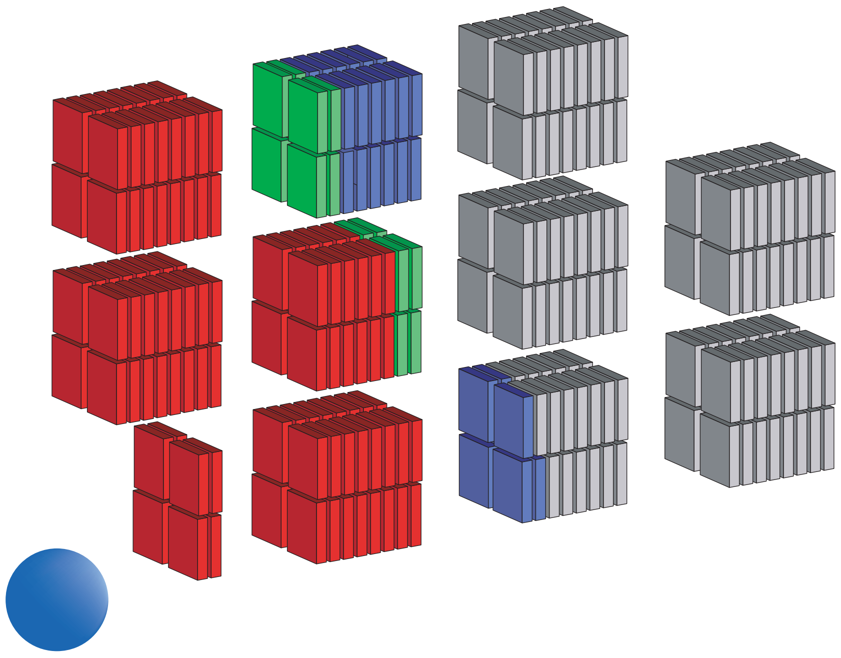


Figure 2 shows the reordered application in the cabinets. The benefit of the new packing will become more obvious in Figure 4.

Figure 2. Cabinet View Showing Three Applications in New Ordering

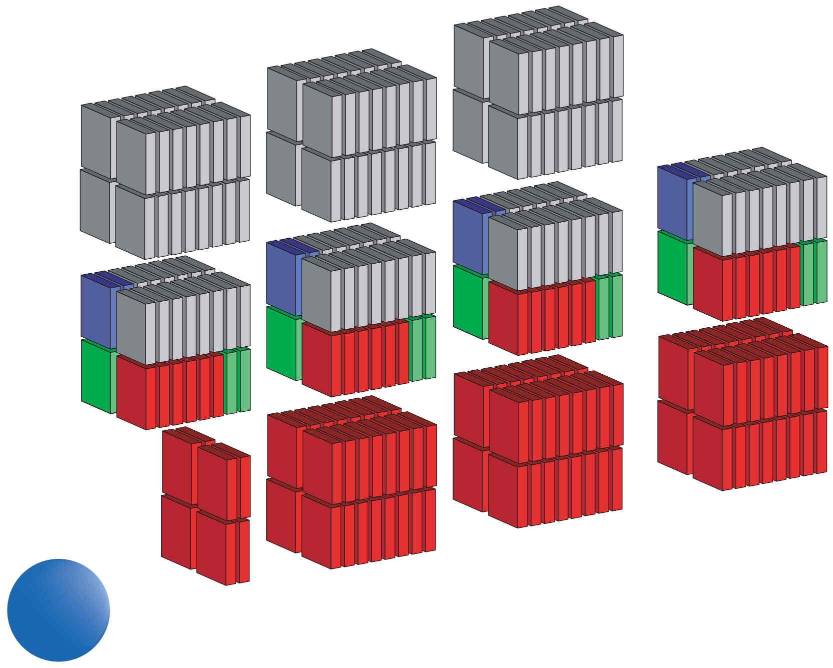
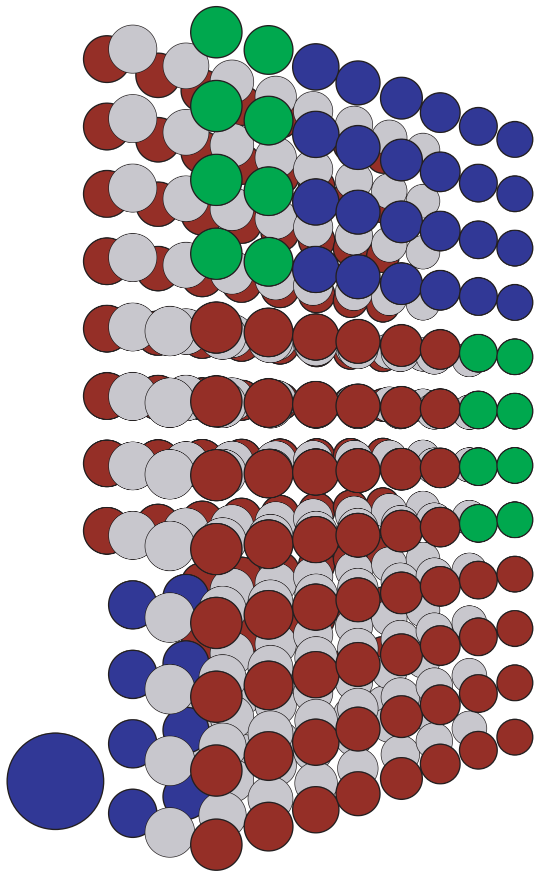
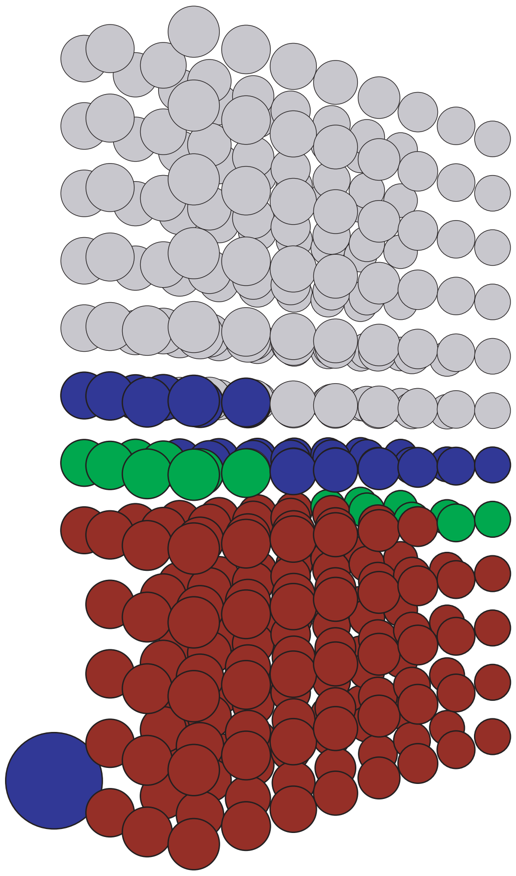


Figure 3. Topology View of Original Application Ordering



A different view: Figure 3 shows the original ordering for the three applications with respect to the topology in a "flattened" cross-section.

Figure 4. Topology View of New Application Ordering



To reduce this type of performance hit for large node count jobs, ALPS introduced an "XYZ" placement method. This method reorders the sequence of the NID numbers used in assigning placements such that they are placed to conform to the mesh or torus topology. An example of this is shown in Figure 4. In an XYZ placement method, jobs are first packed from origin (0, 0, 0) across the x-dimension, then the y-dimension, and finally the z-dimensions assuming these are ascending in size—which may not always be the case. A modification to this is also known as *max-major* ordering: performance is improved for large applications, exploiting the torus bisection bandwidth by packing the minimum dimension first, the next-smallest dimension section second, and the largest dimension last. For example in a 10x4x8 topology, XYZ-ordered node coordinates look like the following: (0, 0, 0)...(0, 3, 0) ... (0, 3, 7), (1, 0, 0). The smallest dimension will vary most quickly.

For Cray systems, *y-major* ordering can be used to exploit the increase in bandwidth over SeaStar due to the doubling of channels in the x- and z-directions. This benefit results from the inclusion of two interconnect chips per package in the Gemini (Figure 5 and Figure 6). In this ordering, the y-dimension is varied last because it has the least bisectional bandwidth of the three axes in the torus.

Figure 5. SeaStar Interconnect Links

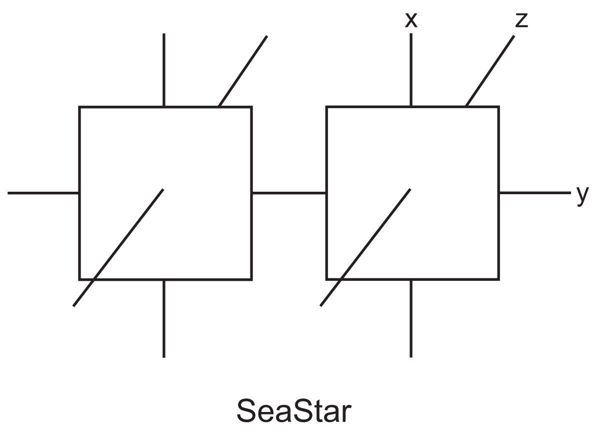
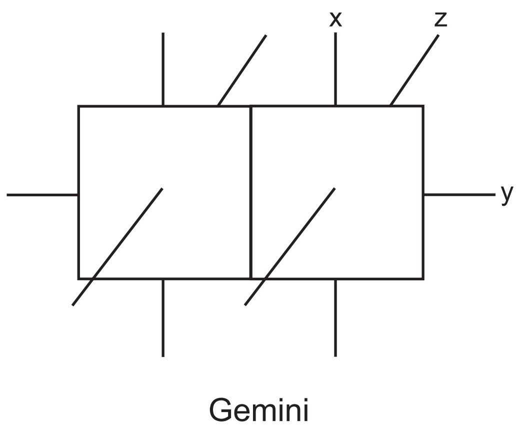


Figure 6. Gemini Interconnect Links



For applications that are considered small node count jobs, the max-major and y-major placement methods may not be optimal. In fact for these types of jobs the original serial NID ordering has shown better bisectional bandwidth if the jobs are confined to a chassis within the Cray system. This effect is compounded by the fact that more applications can fit into the "small node count jobs" category as core density grows with successive processor generations. CLE introduced the hybridized *xyz-by2* NID ordering method to leverage both the communications improvement found with XYZ placement methods and the benefit of the original simple NID ordering for small node count jobs. Cray recommends that sites use this NID ordering for best performance.

The following is the section in `/etc/sysconfig/alps` that describes the selections available to system administrators for NID ordering choice is used on the Cray system. You can also view this file on the login node to view what ordering the system is using:

```
<snip>
...
# The nid ordering option for apbridge can be defined.
# The choices are: (just leave unset) or
#   -On for numerical ordering, i.e. no special order
#   -Ox for max-major dimension ordering
#   -Oy for y-major dimension ordering (for gemini systems of 3+ cabinets)
#   -Or for reverse of max (i.e. min) ordering
#   -Of for field ordering (uses od_allocator_id column)
#  -O2 for 2x2x2 ordering
ALPS_NIDORDER="-Ox"
<snip>
```

If `ALPS_NIDORDER` is not specified, `On` is the default.

- `-On` is the old default option that uses serial ordering; based solely on ascending NID value.
- `-Ox` is max-major NID ordering.
- `-Oy` is y-major dimension ordering, which will order along the y-axis last to exploit the bandwidth in Gemini networks.
- `-Or` is reverse max-major NID ordering. Cray provides this NID ordering for experimental purposes only, there is no evidence it provides a performance improvement, and Cray does not recommend this option for normal use.
- `-Of` gives the system administrator the option to customize NID ordering based on site preferences.
- `-O2` Assigns order of nodes based on the xyz-by2 NID reordering method, which is a merger of the incidental small node packing of the simple NID number method and the inter-application interaction reduction of the "xyz" method.

  Note: Cray recommends this option for Cray XE and Cray XK systems. Use of this option results in better application performance for larger applications running on Cray XE and Cray XK systems.

2.3 Gathering Application Status and Information on the Cray System

Before running applications, you should check the status of the compute nodes.

There are two ways to do this: using the `apstat` and the `xtnodestat` commands.

The `apstat` command provides status information about reservations, compute resources, pending and placed applications, and cores. The format of the `apstat` command is:

`apstat [-a ][-c][-A` *apid* ... `| -R` *resid* ...`][-f` *column list*`][-G][-n|-no|-ng][-P] [-p] [-r] [-s][-v] [-X] [-z]`

You can use `apstat` to display the following types of status information:

- all applications
- placed applications
- applications by application IDs (APIDs)
- applications by reservation IDs (ResIDs)
- protection domain information (e.g., pTags, cookies)
- hardware information such as number of cores, accelerators, and memory
- pending applications
- confirmed and claimed reservations

For example:

```
% apstat -a
Total placed applications: 3
Placed    Apid ResID     User PEs Nodes  Age  State Command
         48062     6     bill   1     1 4h02m run   lsms
         48108  1588      jim   4     1 0h15m run   gtp
         48109  1589      sue   4     2 0h07m run   bench6
```

Adding the `-v` option adds the following output to the above:

```
% apstat -av
...snip...
Application detail
Ap[1]: apid 48062, pagg 0x5201, resId 6, user bill,
       gid 12790, account 0, time 0, normal
  Batch System ID = 171737
  Created at Tue Aug 23 08:17:07 2011
  Originator: aprun on NID 26, pid 21089
  Number of commands 1, control network fanout 32
  Network: pTag 154, cookie 0x878e0000, NTTgran/entries 1/1, hugePageSz 2M
  Cmd[0]: lsms -n 1, 1024MB, XT, nodes 1
  Placement list entries: 1
```

Most of these values were discussed in greater detail in System Interconnnect Features Impacting Application Placement on page 25 but the following items are brief descriptions of the new `apstat` display values:

- pTag — 8-bit protection tag identifier assigned to application
- cookie — 32-bit identifier used to negotiate traffic between software application
- NTTgran/entries — Network Translation Table (NTT) granularity value and number of NTT entries. The NTT contains NIC addresses of compute nodes accessible by this application; ALPS assigns a granularity value of either 1, 2, 4, 8, 16, or 32. The combination of a `pTag` and the NTT creates a unique application identifier and prevents interference between applications.
- hugePageSz — Indicates hugepage size value for the application.

Adding the `-p` option when applications are pending changes the following options:

- PerfCtrs — Indicates that a node considered for placement was not available because it shared a network chip with a node using network performance counters
- pTags — Indicates the application was not able to allocate a free pTag

An APID is also displayed in the `apstat` display after `aprun` execution results. For example:

```
% aprun -n 2 -d 2 ./omp1
Hello from rank 0 (thread 0) on nid00540
Hello from rank 1 (thread 0) on nid00541
Hello from rank 0 (thread 1) on nid00540
Hello from rank 1 (thread 1) on nid00541
Application 48109 resources: utime ~0s, stime ~0s%
```

The `apstat -n` command displays the status of the nodes that are `UP` and core status. Nodes are listed in sequential order:

```
% apstat -n
   NID Arch State HW Rv Pl  PgSz     Avl    Conf  Placed  PEs Apids
    48   XT UP  I  4  1  1    4K 2048000  512000  512000    1 28489
    49   XT UP  I  4  1  1    4K 2048000  512000  512000    1 28490
    50   XT UP  I  4  -  -    4K 2048000       0       0    0
    51   XT UP  I  4  -  -    4K 2048000       0       0    0
    52   XT UP  I  4  1  1    4K 2048000  512000  512000    1 28489
    53   XT UP  I  4  -  -    4K 2048000       0       0    0
    54   XT UP  I  4  -  -    4K 2048000       0       0    0
    55   XT UP  I  4  -  -    4K 2048000       0       0    0
    56   XT UP  I  8  1  1    4K 4096000  512000  512000    1 28490
    58   XT UP  I  8  -  -    4K 4096000       0       0    0
    59   XT UP  I  8  -  -    4K 4096000       0       0    0
Compute node summary
    arch config     up     use    held  avail    down
      XT     20     11       4       0      7       9
```

The `apstat -no` command displays the same information as `apstat -n`, but the nodes are listed in the order that ALPS used to place an application. Site administrators can specify non-sequential node ordering to reduce system interconnect transfer times.

```
% apstat -no
 NID Arch State HW Rv Pl  PgSz     Avl    Conf  Placed  PEs Apids
   14   XT UP  B 24 24  -   4K  8192000 8189952       0    0
   15   XT UP  B 24  1  -   4K  8192000  341248       0    0
   16   XT UP  B 24 24 24   4K  8192000 8189952 8189952   24 290266
   17   XT UP  B 24 24 24   4K  8192000 8189952 8189952   24 290266
   18   XT UP  B 24 24 24   4K  8192000 8189952 8189952   24 290266
   19   XT UP  B 24 24 24   4K  8192000 8189952 8189952   24 290266
   20   XT UP  B 24 24 24   4K  8192000 8189952 8189952   24 290266
   21   XT UP  B 24 24 24   4K  8192000 8189952 8189952   24 290266
   32   XT UP  B 24 24 24   4K  8192000 8189952 8189952   24 290266
   33   XT UP  B 24 24 24   4K  8192000 8189952 8189952   24 290266
   34   XT UP  B 24 24 24   4K  8192000 8189952 8189952   24 290266
   35   XT UP  B 24 24 24   4K  8192000 8189952 8189952   24 290266
   36   XT UP  B 24 24 24   4K  8192000 8189952 8189952   24 290266

...snip...

Compute node summary
    arch config     up    use   held  avail   down
      XT   1124   1123    379    137    607      1
```

where `HW` is the number of cores in the node, `Rv` is the number of cores held in a reservation, and `Pl` is the number of cores being used by an application. If you want to display a `0` instead of a `-` in the `Rv` and `Pl` fields, add the `-z` option to the `apstat` command.

The following `apstat -n` command displays a job using core specialization, demarked by the + sign:

```
% apstat -n
NID Arch State HW Rv Pl  PgSz     Avl    Conf  Placed  PEs Apids
...
84   XT UP  B  8  8  7+   4K 4096000 4096000 4096000    8 1577851
85   XT UP  B  8  2  1+   4K 4096000 4096000 4096000    8 1577851
86   XT UP  B  8  8  8    4K 4096000 4096000 4096000    8 1577854
```

For `apid 1577851`, a total of 10 PEs are placed. On `nid00084`, eight cores are reserved but the `7+` indicates that seven PEs were placed and one core was used for system services. A similar situation appears on `nid00085` three cores are reserved, two application PEs are placed on two cores, and one core is used for system services. For more information, see Core Specialization on page 38.

`apstat -G` will give general information about all nodes that have an accelerator:

```
% apstat -G
GPU Accelerators
 NID Module State Memory(MB)      Family ResId
   6      0    UP       6144 Tesla_X2090 928
   7      0    UP       6144 Tesla_X2090 928
  10      0    UP       6144 Tesla_X2090 928
  11      0    UP       6144 Tesla_X2090 928
```

`Module` is the accelerator module number on the node; `0` is the only valid value. `Memory` is the amount of accelerator memory on the node. `Family` is the name of the particular accelerator product line; in this case it is NVIDIA Tesla.

Using the new custom column output option (`-f`), you can specify which `apstat` display you want to see. For example, to see the NID, Placed, and APID columns one would put the format string in a quote-enclosed comma separated list:

```
apstat -no -f "NID,placed,apids"
NID  Placed   Apids
 28       0
 29       0
  2       0
  3       0
764 8388608 6817081
765 8388608 6817081
738 8388608 6817081
739 8388608 6817081
736 8388608 6817081
737 8388608 6817081
766 8388608 6817081
767 8388608 6817081
576 8388608 6817081
577 8388608 6817081
606 8388608 6817081
607 8388608 6817081
544 8388608 6817081
```

Here's an arbitrarily changed format string displaying compute units:

```
apstat -no -f "NID,apids,CU"
NID Apids CU
 28       24
 29       24
  2       24
  3       24
764       16
765       16
738       16
```

2.3.1 Using the `xtnodestat` Command

The `xtnodestat` command is another way to display the current job and node status. Each character in the display represents a single node. For systems running a large number of jobs, multiple characters may be used to designate a job.

```
% xtnodestat
Current Allocation Status at Tue Aug 23 13:30:16 2011

     C0-0     C1-0     C2-0     C3-0
  n3 -------- ------X- -------- ------A-
  n2 -------- -------- --a----- --------
  n1 -------- -------A -----X-- --------
c2n0 -------- -------- -------- --------
  n3 X------- -------- -------- --------
  n2 -------- -------- -------- --------
  n1 -------- -------- -------- --------
c1n0 -------- ----X--- -------- --------
  n3 S-S-S-S- -e------ --X----X bb-b----
  n2 S-S-S-S- cd------ -------- bb-b----
  n1 S-S-S-SX -g------ -------- bb------
c0n0 S-S-S-S- -f------ -------- bb------
    s01234567 01234567 01234567 01234567

Legend:
   nonexistent node                    S  service node
;  free interactive compute node      -  free batch compute node
A  allocated interactive or ccm node  ?  suspect compute node
W  waiting or non-running job         X  down compute node
Y  down or admindown service node     Z  admindown compute node

Available compute nodes:          0 interactive,        343 batch


Job ID     User       Size   Age        State           command line
--- ------ --------   -----  ---------  --------  ----------------------------------
a   762544 user1      1      0h00m      run        test_zgetrf
b   760520 user2      10     1h28m      run        gs_count_gpu
c   761842 user3      1      0h40m      run        userTest
d   761792 user3      1      0h45m      run        userTest
e   761807 user3      1      0h43m      run        userTest
f   755149 user4      1      5h13m      run        lsms
g   761770 user3      1      0h47m      run        userTest
```

The `xtnodestat` command displays the allocation grid, a legend, and a job listing. The column and row headings of the grid show the physical location of jobs: `C` represents a cabinet, `c` represents a chassis, `s` represents a slot, and `n` represents a node.

Note: If `xtnodestat` indicates that no compute nodes have been allocated for interactive processing, you can still run your job interactively by using the `qsub -I` command. Then launch your application with the `aprun` command.

Use the `xtprocadmin -A` command to display node attributes that show both the logical node IDs (`NID` heading) and the physical node IDs (`NODENAME` heading):

```
% xtprocadmin -A
  NID    (HEX)   NODENAME     TYPE ARCH        OS    CPUS CU  AVAILMEM   PAGESZ CLOCKMHZ GPU SOCKETS DIES C/CU
  1      0x1     c0-0c0s0n1  service  xt (service)   12   6    32768     4096     2500   0      1    1    2
  2      0x2     c0-0c0s0n2  service  xt (service)   12   6    32768     4096     2500   0      1    1    2
  5      0x5     c0-0c0s1n1  service  xt (service)   16   8    32768     4096     2600   0      1    1    2
  6      0x6     c0-0c0s1n2  service  xt (service)   12   6    32768     4096     2500   0      1    1    2
  12     0xc     c0-0c0s3n0  compute  xt       CNL   32  16    32768     4096     2700   0      2    2    2
  13     0xd     c0-0c0s3n1  compute  xt       CNL   32  16    32768     4096     2700   0      2    2    2
  14     0xe     c0-0c0s3n2  compute  xt       CNL   32  16    32768     4096     2700   0      2    2    2
  15     0xf     c0-0c0s3n3  compute  xt       CNL   32  16    32768     4096     2700   0      2    2    2
  20     0x14    c0-0c0s5n0  compute  xt       CNL   32  16    32768     4096     2700   0      2    2    2
  21     0x15    c0-0c0s5n1  compute  xt       CNL   32  16    32768     4096     2700   0      2    2    2
  22     0x16    c0-0c0s5n2  compute  xt       CNL   32  16    32768     4096     2700   0      2    2    2
  23     0x17    c0-0c0s5n3  compute  xt       CNL   32  16    32768     4096     2700   0      2    2    2
  36     0x24    c0-0c0s9n0  compute  xt       CNL   32  16    32768     4096     2700   0      2    2    2
  37     0x25    c0-0c0s9n1  compute  xt       CNL   32  16    32768     4096     2700   0      2    2    2
  38     0x26    c0-0c0s9n2  compute  xt       CNL   32  16    32768     4096     2700   0      2    2    2
  39     0x26    c0-0c0s9n3  compute  xt       CNL   32  16    32768     4096     2700   0      2    2    2
```

For more information, see the `xtnodestat`(1) and `xtprocadmin`(8) man pages.

2.4 Using the `cnselect` Command

The `aprun` utility supports manual and automatic node selection. For manual node selection, first use the `cnselect` command to get a candidate list of compute nodes that meet the criteria you specify. Then, for interactive jobs use the `aprun -L` *node_list* option. For batch and interactive batch jobs, add `-lmppnodes=\"`*node_list*`\"` to the job script or the `qsub` command line.

The format of the `cnselect` command is:

`[-l] [-L` *fieldname*`] [-U] [-D] [-c] [-V] [[-e]` *expression*`]`

where:

- `-l` lists the names of fields in the compute nodes attributes database.

  Note: The `cnselect` utility displays `nodeid`s, sorted by ascending NID number or unsorted. For some sites, node IDs are presented to ALPS in non-sequential order for application placement. Site administrators can specify non-sequential node ordering to reduce system interconnect transfer times.

- `-L` *fieldname* lists the current possible values for a given field.
- `-U` Causes the user-supplied expression to not be enclosed in parentheses but combined with other built-in conditions. This option may be needed if you add other SQL qualifiers (such as `ORDER BY`) to the expression.
- `-V` prints the version number and exits.
- `-c` gives a count of the number of nodes rather than a list of the nodes themselves.
- `[-e]` *expression* queries the compute node attributes database.

You can use `cnselect` to get a list of nodes selected by such characteristics as the number of cores per node (`numcores`), the amount of memory on the node (in megabytes), and the processor speed (in megahertz). For example, to run an application on Cray XK6 16-core nodes with 32 GB of memory or more, use:

```
% cnselect numcores.eq.16 .and. availmem.gt.32000
268-269,274-275,80-81,78-79
% aprun -n 32 -L 268-269 ./app1
```

Note: The `cnselect` utility returns `-1` to `stdout` if the `numcores` criteria cannot be met; for example `numcores.eq.16` on a system that has no 16-core compute nodes.

You can also use `cnselect` to get a list of nodes if a site-defined label exists. For example, to run an application on six-core nodes, you might use:

```
% cnselect -L label1
HEX-CORE
DODEC-CORE
16-Core
% cnselect -e "label1.eq.'HEX-CORE'"
60-63,76,82
% aprun -n 6 -L 60-63,76,82 ./app1
```

If you do not include the `-L` option on the `aprun` command or the `-lmppnodes` option on the `qsub` command, ALPS automatically places the application using available resources.

2.5 Understanding How Much Memory is Available to Applications

When running large applications, you should understand how much memory will be available per node. Cray Linux Environment (CLE) uses memory on each node for CNL and other functions such as I/O buffering, core specialization, and compute node resiliency. The remaining memory is available for user executables; user data arrays; stacks, libraries and buffers; and the SHMEM symmetric stack heap.

The amount of memory CNL uses depends on the number of cores, memory size, and whether optional software has been configured on the compute nodes. For a 24-core node with 32 GB of memory, roughly 28.8 to 30 GB of memory is available for applications.

The default stack size is 16 MB. You can determine the maximum stack size by using the `limit` command (csh) or the `ulimit -a` command (bash).

Note: The actual amount of memory CNL uses varies depending on the total amount of memory on the node and the OS services configured for the node.

You can use the `aprun -m` *size* option to specify the per-PE memory limit. For example, this command launches `xthi` on cores 0 and 1 of compute nodes `472` and `473`. Each node has 8 GB of available memory, allowing 4 GB per PE.

```
% aprun -n 4 -N 2 -m4000 ./xthi | sort
Application 225108 resources: utime ~0s, stime ~0s
PE 0 nid00472 Core affinity = 0,1
PE 1 nid00472 Core affinity = 0,1
PE 2 nid00473 Core affinity = 0,1
PE 3 nid00473 Core affinity = 0,1
% aprun -n 4 -N 2 -m4001 ./xthi | sort
Claim exceeds reservation's memory
```

You can change MPI buffer sizes and stack space from the defaults by setting certain environment variables. For more details, see the `intro_mpi`(3) man page.

2.6 Core Specialization

CLE offers a core-specialization functionality. Core specialization binds a set of Linux kernel-space processes and daemons to one or more cores within a Cray compute node to enable the software application to fully utilize the remaining cores within its `cpuset`. This restricts all possible overhead processing to the specialized cores within the reservation and may improve application performance. To help users calculate the new "scaled-up" width for a batch reservation that uses core specialization, use the `apcount` tool.

Note: `apcount` will work only if your system has uniform compute node types.

See the `apcount`(1) man page for further information.

2.7 Launching an MPMD Application

The `aprun` utility supports multiple-program, multiple-data (MPMD) launch mode. To run an application in MPMD mode under `aprun`, use the colon-separated `-n` *pes executable1* : `-n` *pes executable2* `:` ... format. In the first executable segment, you may use other `aprun` options such as `-cc`, `-cp`, `-d`, `-L`, `-n`, `-N`, `-S`, `-sl`, `-sn`, and `-ss`. If you specify the `-m` option it must be specified in the first executable segment and the value is used for all subsequent executables. If you specify `-m` more than once while launching multiple applications in MPMD mode, `aprun` will return an error. For MPI applications, all of the executables share the same `MPI_COMM_WORLD` process communicator. MPMD mode will not work for system commands and applications require at the least an enclosure within `MPI_Init()` and `MPI_Finalize()` environment management routines.

For example, this command launches 128 instances of `program1` and 256 instances of `program2`:

```
aprun -n 128 ./program1 : -n 256 ./program2
```

A space is required before and after the colon.

Note: MPMD applications that use the SHMEM parallel programming model, either standalone or nested within an MPI program, are not supported on Gemini based systems.

2.8 Managing Compute Node Processors from an MPI Program

MPI programs should call the `MPI_Finalize()` routine at the conclusion of the program. This call waits for all processing elements to complete before exiting. If one of the programs fails to call `MPI_Finalize()`, the program never completes and `aprun` stops responding. There are two ways to prevent this behavior:

- Use the PBS Professional elapsed (wall clock) time limit to terminate the job after a specified time limit (such as `-l walltime=2:00:00`).
- Use the `aprun -t` *sec* option to terminate the program. This option specifies the per-PE CPU time limit in seconds. A process will terminate only if it reaches the specified amount of CPU time (not wallclock time).

  For example, if you use:

  ```
  % aprun -n 8 -t 120 ./myprog1
  ```

  and a PE uses more than two minutes of CPU time, the application terminates.

2.9 About `aprun` Input and Output Modes

The `aprun` utility handles standard input (`stdin`) on behalf of the user and handles standard output (`stdout`) and standard error messages (`stderr`) for user applications.

2.10 About `aprun` Resource Limits

`aprun` utility does not forward its user resource limits to each compute node (except for `RLIMIT_CORE` and `RLIMIT_CPU`, which are always forwarded).

You can set the `APRUN_XFER_LIMITS` environment variable to `1` (`export APRUN_XFER_LIMITS=1` or `setenv APRUN_XFER_LIMITS 1`) to enable the forwarding of user resource limits. For more information, see the `getrlimit`(P) man page.

2.11 About `aprun` Signal Processing

The `aprun` utility forwards the following signals to an application:

- `SIGHUP`
- `SIGINT`
- `SIGQUIT`
- `SIGTERM`
- `SIGABRT`
- `SIGUSR1`
- `SIGUSR2`
- `SIGURG`
- `SIGWINCH`

The `aprun` utility ignores `SIGPIPE` and `SIGTTIN` signals. All other signals remain at default and are not forwarded to an application. The default behaviors that terminate `aprun` also cause ALPS to terminate the application with a `SIGKILL` signal.

2.12 Reserved File Descriptors

The following file descriptors are used by ALPS and should **not** be closed by applications: `100, 102, 108, 110`.

Running User Programs on Service Nodes [3]

To compile a program that you want to run on a login or other service node, call the compiler directly.

- For PGI programs, use the `pgcc`, `pgCC`, or `pgf95` command.
- For GCC programs, use the `gcc`, `g++`, or `gfortran` command.
- For Cray compilers, use the `cc`, `CC`, or `ftn` command.
- For Intel compilers, use the `icc`, `icpc`, `fpp`, or `ifort` command.

These compilers will find the appropriate header files and libraries in their normal Linux locations.

For example, to run program `my_utility` on a service node, first compile the program:

```
% module load pgi
% pgCC -o my_utility my_utility.C
```

Then run `my_utility`:

```
% my_utility
In main(0)
In functionx(0)
Back in main()
```

Using Workload Management Systems [4]

Your Cray system may include the optional PBS Professional, Moab and TORQUE, or Platform LSF workload management system (WMS). If so, your system can be configured with a given number of interactive job processors and a given number of batch processors. A job that is submitted as a batch process can use only the processors that have been allocated to the batch subsystem. If a job requires more processors than have been allocated for batch processing, it remains in the batch queue but never runs.

Note: At any time, the system administrator can change the designation of any node from interactive to batch or vice versa. This change does not affect jobs already running on those nodes. It applies only to jobs already in the queue and jobs submitted later.

The basic process for creating and running batch jobs is to create a job script that includes `aprun` commands, then use the `qsub` command to run the script.

4.1 Creating Job Scripts

A job script may consist of directives, comments, and executable statements:

```
#PBS -N job_name
#PBS -l resource_type=specification
#
command
command
. . .
```

PBS Professional and Moab and TORQUE provide a number of *resource_type* options for specifying, allocating, and scheduling compute node resources, such as `mppwidth` (number of processing elements), `mppdepth` (number of threads per PE), `mppnppn` (number of PEs per node), and `mppnodes` (manual node placement list). See Table 1 and the `pbs_resources`(7B) man page for details.

4.2 Submitting Batch Jobs

To submit a job to the workload management system, load the `pbs`, `moab`, or `xt-lsfhpc` module:

```
% module load pbs
```

Or

```
% module load moab
```

```
% module load xt-lsfhpc
```

Then use the `qsub` command:

```
% qsub [-l resource_type=specification] jobscript
```

where *jobscript* is the name of a job script that includes one or more `aprun` commands.

The `qsub` command scans the lines of the script file for directives. An initial line in the script that has only the characters `#!` or the character `:` is ignored and scanning starts at the next line. A line with `#!/bin/`*shell* invokes *shell* from within the script. Scanning continues until the first executable line. An executable line is not blank, not a directive, and does not start with #). If directives occur on subsequent lines after the first executable line, they are ignored.

When you run the script, `qsub` displays the Job ID. You can use the `qstat` command to check on the status of your job and the `qdel` command to remove a job from the queue.

If a `qsub` option is present in both a directive and on the command line, the command line takes precedence. If an option is present in a directive and not on the command line, that option and its argument, if any, are processed as if you included them on the command line.

Table 1 lists `aprun` options and their counterpart `qsub -l` options:

Table 1. `aprun` Versus `qsub` Versus `bsub` (LSF) Options

`aprun` **Option**	`qsub -l` **Option**	`bsub` **Option**	**Description**
`-n 4`	`-l mppwidth=4`	`-n 4`	Width (number of PEs)
`-d 2`	`-l mppdepth=2`	N/A	Depth (number of CPUs hosting OpenMP threads)
`-N 1`	`-l mppnppn=1`	LSF currently assumes a uniform processor pool	Number of PEs per node
`-L 5,6,7`	`-l mppnodes=\"5,6,7\"`	N/A	Candidate node List
`-m 1000`	`-l mppmem=1000`	`-M 1000`	Memory per PE

For further information about `qsub -l` options, see the `pbs_resources`(7B) man page.

For examples of batch jobs that use `aprun`, see Running a Batch Job Script on page 99.

4.3 Getting Job Status

The `qstat` command displays the following information about all active batch jobs:

- The job identifier (`Job id`) assigned by the WMS
- The job name (`Name`)
- The job owner (`User`)
- CPU time used (`Time Use`)
- The job state `S` is:
  - `C` (job is complete)
  - `E` (job is exiting)
  - `H` (job is held)
  - `Q` (job is in the queue)
  - `R` (job is running)
  - `S` (job is suspended)
  - `T` (job is being moved to a new location)
  - `W` (job is waiting for its execution time)
- The queue (`Queue`) in which the job resides

For example:

```
% qstat
Job id           Name             User             Time Use S Queue
---------------- ---------------- ---------------- -------- - -----
84.nid00003      test_ost4_7      usera            03:36:23 R workq
33.nid00003      run.pbs          userb            00:04:45 R workq
34.nid00003      run.pbs          userb            00:04:45 R workq
35.nid00003      STDIN            userc            00:03:10 R workq
```

If the `-a` option is used, queue information is displayed in an alternative format.

```
% qstat -a
                                                              Req'd  Req'd   Elap
Job ID           Username Queue    Jobname     SessID NDS TSK Memory Time  S Time
---------------- -------- -------- ----------  ------ --- --- ------ ----- - -----
84.nid00003      usera    workq    test_ost4_7    --    1   1    --    --  Q   --
33.nid00003      userb    workq    run.pbs        --    1   1    --    --  Q   --
34.nid00003      userb    workq    run.pbs        --    1   1    --    --  Q   --
35.nid00003      userc    workq    STDIN          --    1   1    --    --  Q   --
```

For details, see the `qstat`(1B) man page.

4.4 Removing a Job from the Queue

The `qdel` command removes a batch job from the queue. As a user, you can remove any batch job that you own. Jobs are removed from the queue in the order they are presented to `qdel`. See the `qdel`(1B) man page for more information.

Dynamic Shared Objects and Libraries (DSLs) [5]

5.1 Introduction

Cray supports dynamically linking applications with shared objects and libraries. Dynamic shared objects allow for use of multiple programs that require the same segment of memory address space to be used during linking and compiling. This functionality enables many previously unavailable applications to run on Cray systems and may reduce executable size and improve optimization of system resources. Also, when shared libraries are changed or upgraded, users do not need to recompile dependent applications. Cray Linux Environment uses Cray Data Virtualization Service (Cray DVS) to project the shared root onto the compute nodes. Thus, each compute node, using its DVS-projected file system transparently, calls shared libraries located at a central location.

5.2 About the Compute Node Root Run Time Environment

CLE facilitates compute node access to the Cray system shared root by projecting it through Cray DVS. DVS is an I/O forwarding mechanism that provides transparent access to remote file systems, while reducing client load. DVS allows users and applications running on compute nodes access to remote POSIX-compliant file systems.

ALPS runs with applications that use read-only shared objects. When a user runs an application, ALPS launches the application to the compute node root. After installation, the compute node root is enabled by default. However, an administrator can define the default case (DSO support enabled or disabled) per site policy. Users can override the default setup by setting an environment variable, `CRAY_ROOTFS`.

5.2.1 DSL Support

CLE supports DSLs for following cases:

- Linking and loading against programming environments supported by Cray
- Use of standard Linux services usually found on service nodes.

Launching terminal shells and other programming language interpreters by using the compute node root are not currently supported by Cray.

5.3 Configuring DSL

The shared root `/etc/opt/cray/cnrte/roots.conf` file contains site-specific values for custom root file systems. To specify a different pathname for `roots.conf`, edit the configuration file `/etc/sysconfig/xt.conf` and change the value for the variable, `CRAY_ROOTFS_CONF`. In the `roots.conf` file, the system default compute node root used is specified by the symbolic name `DEFAULT`. If no default value is specified, `/` will be assumed. In the following example segment of `roots.conf`, the default case uses the root mounted at on the compute nodes at `/dsl`:

```
DEFAULT=/dsl
INITRAMFS=/
DSL=/dsl
```

A user can override the system default compute node root value by setting the environment variable, `CRAY_ROOTFS`, to a value from the `roots.conf` file. This setting effectively changes the compute node root used for launching jobs. For example, to override the use of `/dsl`, a user would enter something similar to the following example at the command line on the login node:

```
% export CRAY_ROOTFS=INITRAMFS
```

If the system default is using `initramfs`, enter something like the following at the command line on the login node to switch to using the compute node root path specified by `DSL`:

```
% export CRAY_ROOTFS=DSL
```

An administrator can modify the contents of this file to restrict user access. For example, if the administrator wants to allow applications to launch only by using the compute node root, the `roots.conf` file would read as follows:

```
% cat /etc/opt/cray/cnrte/roots.conf
DEFAULT=/dsl
```

For more information, see *Managing System Software for Cray XE and Cray XK Systems*.

5.4 Building, Launching, and Workload Management Using Dynamic Objects

5.4.1 Linker Search Order

Search order is an important detail to consider when compiling and linking executables. The dynamic linker uses the following search order when loading a shared object:

- Value of `LD_LIBRARY_PATH` environment variable.
- Value of `DT_RUNPATH` dynamic section of the executable, which is set using the `ld -rpath` command. You can add a directory to the run time library search path using the `ld` command. However, when a supported Cray programming environment is used, the library search path is set automatically. For more information please see the `ld`(1) man page.
- The contents of the human non-readable cache file `/etc/ld.so.cache`. The `/etc/ld.so.conf` contains a list of comma or colon separated path names to which the user can append custom paths.
- The paths `/lib` and `/usr/lib`.

Loading a programming environment module before compiling will appropriately set the `LD_LIBRARY_PATH` environment variable. Conversely, unloading modules will clear the stored value of `LD_LIBRARY_PATH`. Other useful environment variables are listed in the `ld.so`(8) man page. If a programming environment module is loaded when an executable that uses dynamic shared objects is running, it should be the same programming environment used to build the executable. For example, if a program is built using the PathScale compiler, the user should load the module, `PrgEnv-pathscale`, when setting the environment to launch the application.

Example 1. Compiling an application

Compile the following program, `reduce_dyn.c`, dynamically by including the compiler option `dynamic`.

The C version of the program, `reduce_dyn.c`, looks like:

```
/* program reduce_dyn.c */
#include <stdio.h>
#include "mpi.h"

int main (int argc, char *argv[])
{
 int i, sum, mype, npes, nres, ret;
 ret = MPI_Init (&argc, &argv);
 ret = MPI_Comm_size (MPI_COMM_WORLD, &npes);
 ret = MPI_Comm_rank (MPI_COMM_WORLD, &mype);
 nres = 0;
 sum = 0;

 for (i = mype; i <=100; i += npes)
 {
  sum = sum + i;
 }
 (void) printf ("My PE:%d My part:%d\n",mype, sum);
 ret = MPI_Reduce (&sum,&nres,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD);

 if (mype == 0)
 {
  (void) printf ("PE:%d Total is:%d\n",mype, nres);
 }
 ret = MPI_Finalize ();
}
```

Invoke the C compiler using `cc` and the `dynamic` option:

```
% cc -dynamic reduce_dyn.c -o reduce_dyn
```

Alternatively, you can use the environment variable, `XTPE_LINK_TYPE`, without any extra compiler options:

```
% export XTPE_LINK_TYPE=dynamic
% cc reduce_dyn.c -o reduce_dyn
```

You can tell if an executable uses a shared library by executing the `ldd` command:

```
% ldd reduce_dyn
   libsci.so => /opt/xt-libsci/10.3.7/pgi/lib/libsci.so (0x00002b1135e02000)
 libfftw3.so.3 => /opt/fftw/3.2.1/lib/libfftw3.so.3 (0x00002b1146e92000)
 libfftw3f.so.3 => /opt/fftw/3.2.1/lib/libfftw3f.so.3 (0x00002b114710a000)
 libsma.so => /opt/mpt/3.4.0.1/xt/sma/lib/libsma.so (0x00002b1147377000)
 libmpich.so.1.1 => /opt/mpt/3.4.0.1/xt/mpich2-pgi/lib/libmpich.so.1.1 (0x00002b11474a0000)
 librt.so.1 => /lib64/librt.so.1 (0x00002b114777a000)
 libpmi.so => /opt/mpt/3.4.0.1/xt/pmi/lib/libpmi.so (0x00002b1147883000)
 libalpslli.so.0 => /opt/mpt/3.4.0.1/xt/util/lib/libalpslli.so.0 (0x00002b1147996000)
 libalpsutil.so.0 => /opt/mpt/3.4.0.1/xt/util/lib/libalpsutil.so.0 (0x00002b1147a99000)
 libportals.so.1 => /opt/xt-pe/2.2.32DSL/lib/libportals.so.1 (0x00002b1147b9c000)
 libpthread.so.0 => /lib64/libpthread.so.0 (0x00002b1147ca8000)
 libm.so.6 => /lib64/libm.so.6 (0x00002b1147dc0000)
 libc.so.6 => /lib64/libc.so.6 (0x00002b1147f15000)
 /lib64/ld-linux-x86-64.so.2 (0x00002b1135ce6000)
```

There are shared object dependencies listed for this executable. For more information, please consult the `ldd`(1) man page.

Example 2. Running an application in interactive mode

If the system administrator has set up the compute node root run time environment for the default case, then the user executes `aprun` without any further argument:

```
% aprun -n 6 ./reduce_dyn
```

However, if the administrator sets up the system to use `initramfs`, then the user must set the environment variable appropriately:

```
% export CRAY_ROOTFS=DSL
% aprun -n 6 ./reduce_dyn | sort
Application 1555880 resources: utime 0, stime 8
  My PE:0 My part:816
  My PE:1 My part:833
  My PE:2 My part:850
  My PE:3 My part:867
  My PE:4 My part:884
  My PE:5 My part:800
  PE:0 Total is:5050
```

Example 3. Running an application using a workload management system

Running a program interactively using a workload management system such as PBS or Moab and TORQUE with the compute node root is essentially the same as running with the default environment. One exception is that if the compute node root is not the default execution option, you must set the environment variable after you have run the batch scheduler command, `qsub`:

```
% qsub -I -lmppwidth=4
% export CRAY_ROOTFS=DSL
```

Alternatively, you can use `-V` option to pass environment variables to the PBS or Moab and TORQUE job:

```
% export CRAY_ROOTFS=DSL
% qsub -V -I -lmppwidth=4
```

Example 4. Running a program using a batch script

Create the following batch script, `reduce_script`, to launch the `reduce_dyn` executable:

```
#!/bin/bash
#reduce_script
# Define the destination of this job
# as the queue named "workq":
#PBS -q workq
#PBS -l mppwidth=6
# Tell WMS to keep both standard output and
# standard error on the execution host:
#PBS -k eo
cd /lus/nid00008/crayusername
module load PrgEnv-pgi
aprun -n 6 ./reduce_dyn
exit 0
```

Then launch the script using the `qsub` command:

```
% export CRAY_ROOTFS=DSL
% qsub -V reduce_script
1674984.sdb
% cat reduce_script.o1674984
Warning: no access to tty (Bad file descriptor).
Thus no job control in this shell.
My PE:5 My part:800
My PE:4 My part:884
My PE:1 My part:833
My PE:3 My part:867
My PE:2 My part:850
My PE:0 My part:816
PE:0 Total is:5050
Application 1747058 resources: utime ~0s, stime ~0s
```

5.5 Troubleshooting

5.5.1 Error While Launching with `aprun`: "error while loading shared libraries"

If you encounter an error such as:

```
error while loading shared libraries: libsci.so: cannot open shared /
object file: No such file or directory
```

your environment may not be configured to launch applications using shared objects. Set the environment variable `CRAY_ROOTFS` to the appropriate value as prescribed in Example 2.

5.5.2 Running an Application Using a Non-existent Root

If you erroneously set `CRAY_ROOTFS` to a file system not specified in `roots.conf`, `aprun` will exit with the following error:

```
% set CRAY_ROOTFS=WRONG_FS
% aprun -n 4 -N 1 ./reduce_dyn
aprun: Error from DSL library: Could not find shared root symbol WRONG_FS,
   specified by env variable CRAY_ROOTFS, in config file: /etc/opt/cray/cnrte/roots.conf

aprun: Exiting due to errors. Application aborted
```

5.5.3 Performance Implications of Using Dynamic Shared Objects

Using dynamic libraries may introduce delays in application launch times because of shared object loading and remote page faults. Such delays are an inevitable result of the linking process taking place at execution and the relative inefficiency of symbol lookup in DSOs. Likewise, binaries that are linked dynamically may cause a small but measurable performance degradation during execution. If this delay is unacceptable, link the application with static libraries.

Using Cluster Compatibility Mode in CLE [6]

6.1 Cluster Compatibility Mode

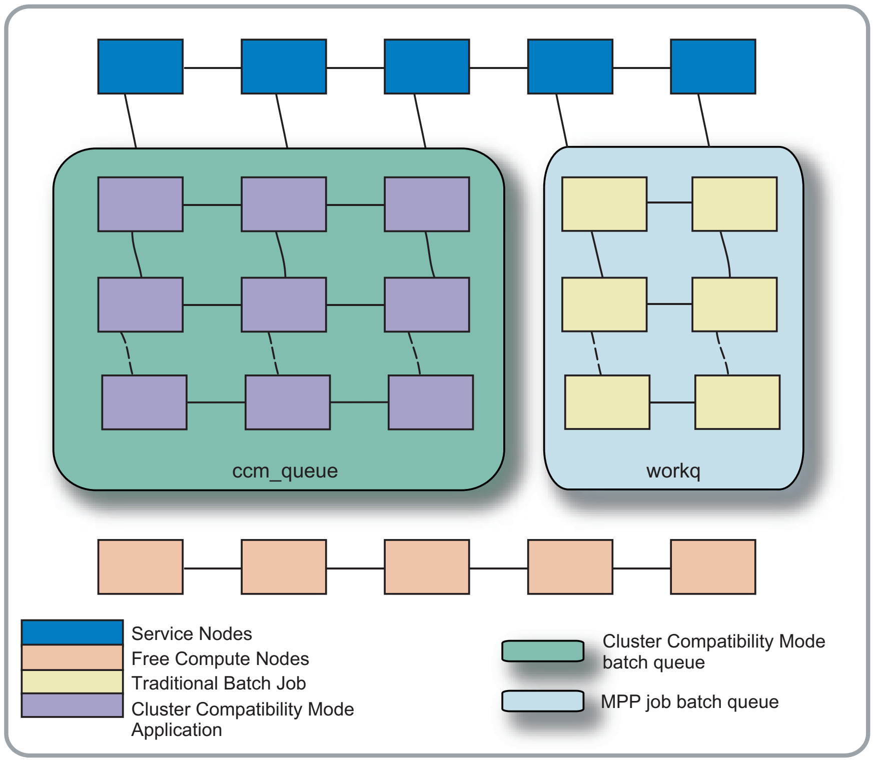
A Cray system is not a cluster but a massively parallel processing (MPP) computer. An MPP is one computer with many networked processors used for distributed computation, and, in the case of Cray system architectures, a high-speed communications interface that facilitates optimal bandwidth and memory operations between those processors. When operating as an MPP machine, the Cray compute node kernel (Cray CNL) typically does not have a full set of the Linux services available that are used in cluster ISV applications.

Cluster Compatibility Mode (CCM) is a software solution that provides the services needed to run most cluster-based independent software vendor (ISV) applications out-of-the-box with some configuration adjustments. It is built on top of the compute node root runtime environment (CNRTE), the infrastructure that provides dynamic library support in Cray systems.

6.1.1 CCM Implementation

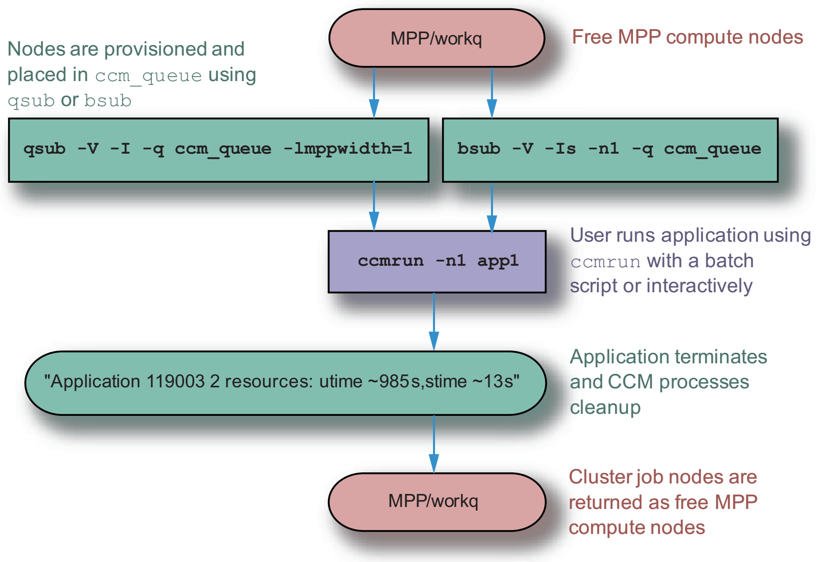
CCM is tightly coupled to the workload management system. It enables users to execute cluster applications together with workload-managed jobs that are running in a traditional MPP batch or interactive queue (see Figure 7). Essentially, CCM uses the batch system to logically designate part of the Cray system as an emulated cluster for the duration of the job.

Figure 7. Cray Job Distribution Cross Section



Users provision the emulated cluster by launching a batch or interactive job in LSF, PBS or Moab and TORQUE using a CCM-specific queue. The user-specified nodes in the batch reservation are no longer available for MPP jobs for the duration of the CCM job. These nodes can be found in the user directory: `$HOME/.crayccm/ccm_nodelist.$PBS_JOBID` or `$HOME/.crayccm/ccm_nodelist.$LSF_JOBID` where the file name suffix is the unique job identification created by the batch reservation. The user launches the application using `ccmrun`. When the job terminates, the applications clean up and the nodes are then returned to the free pool of compute nodes (see Figure 8).

Figure 8. CCM Job Flow Diagram


Nodes are provisioned and placed in ccm_queue using qsub or bsub
MPP/workq
Free MPP compute nodes
qsub -V -I -q ccm_queue -lmppwidth=1
bsub -V -Is -n1 -q ccm_queue
ccmrun -n1 app1
User runs application using ccmrun with a batch script or interactively
"Application 119003 2 resources: utime ~985s,stime ~13s"
Application terminates and CCM processes cleanup
MPP/workq
Cluster job nodes are returned as free MPP compute nodes

6.2 Installation and Configuration of Applications for CCM

Users are encouraged to install programs using their local scratch directory and set paths accordingly to use CCM. However, if an ISV application requires root privileges, the site administrator must install the application on the boot node's shared root in `xtopview`. Compute nodes will then be able to mount the shared root using the compute node root runtime environment and use services necessary for the ISV application.

6.3 Using CCM

6.3.1 CCM Commands

After loading the `ccm` module the user can issue the following two commands: `ccmrun` and `ccmlogin`.

6.3.1.1 `ccmrun`

`ccmrun`, as the name implies, starts the cluster application. The head node is the first node in the emulated cluster where `ccmrun` sets up the CCM infrastructure and propagates the rest of the application. The following shows the syntax for `ccmrun`:

`ccmrun [--help][--ssh][--nossh][--rsh][--norsh][--nscd][--nonscd] [--rsip][--norsip]`*executable* `[`*executable* `arg1] ... [`*executable* `argn]`

where:

`--ssh`	Launches a CCM job with the SSH daemon listening on port 203. This is the default in absence of custom configuration or environment.
`--nosshd`	Launches a CCM job without an SSH daemon.
`--rsh`	Launches a CCM job with `portmap` and `xinetd` daemons on all compute nodes within the CCM reservation. This is the default behavor.
`--norsh`	Launches a CCM job without the `portmap` and `xinetd` daemons. It's possible this option may improve performance if `rsh` is not required by the current job.
`--ncsd`	Launches a CCM job with the name service caching daemon on the CCM compute nodes. This is the default behavior.
`--noncsd`	Launches a CCM job without the name service caching daemon.
`--rsip`	Turns on CCM RSIP (Realm Specific Internet Protocol) Small Port allocation behavior. When you select this option, RSIP allocates bind (`INADDR_ANY`) requests from non-RSIP port ranges. This functionality serves to prevent a CCM application from consuming ports from in the limited RSIP pool. This is the default behavior.
`--norsip`	Disables RSIP for the CCM application. When this option is specified, `bind` (`INADDR_ANY`) requests from non-RSIP port ranges. This pool is not generally recommended in most configurations. Since the number of RSIP ports per host is extremely limited, specifying this option could cause an application to run out of ports. However, this option may be helpful if an application fails in a default environment.
`--help`	Displays `ccmrun` usage statement.

6.3.1.2 `ccmlogin`

`ccmlogin` supports a `-n` *hostname* option. If `-n` option is specified, services are not initiated at startup time, and the user does not need to be in a batch session. `ccmlogin` also supports the `-V` option, which propagates the environment to compute nodes in the same manner as `ssh -V`.

6.3.2 Starting a CCM Batch Job

You can use either PBS, Moab and TORQUE, or Platform LSF (Load Sharing Facility) to reserve the nodes for the cluster by using the `qsub` or `bsub` commands; then launch the application using `ccmrun`. All standard workload management reservation options are supported with `ccmrun`. An example using the application *isv_app* appears below:

Example 5. Launching a CCM application using PBS or Moab and TORQUE

```
% qsub -I -l mppwidth=32 -q ccm_queue
```

```
qsub: waiting for job 434781.sdb to start
qsub: job 434781.sdb ready
Initializing CCM Environment, please wait
```

After the user prompt re-appears, run the application using `ccmrun`:

```
% ccmrun isv_app job=e5 cpus=32
```

An equivalent batch script for this example would look like:

```
#mtscript
#PBS -l mppwidth=32
#PBS -q ccm_queue
#PBS -j oe
#PBS -S /bin/bash
cd $PBS_O_WORKDIR
export PATH=${PATH}:/mnt/lustre_server/ccmuser/isv_app/Commands
ln -s ../e5.inp e5.inp
export TMPDIR=${PBS_O_WORKDIR}/temp
mkdir $TMPDIR
module load ccm
ccmrun isv_app job=e5 cpus=32 interactive
```

To submit the job enter the following at the command prompt:

```
% qsub mtscript
```

Example 6. Launching a CCM application using Platform LSF

For LSF, `bsub` requests use node counts rather than core counts while `ccmrun` still takes the number of cores as its argument:

```
% cnselect -L numcores
24
% module load xt-lsfhpc
% module load ccm
% bsub -n 2 -ext"CRAYXT[]" -q ccm_queue -o
out_file ccmrun isv_app job=e5 cpus=32
```

The equivalent batch script for this example would look like:

```
#lsfscript
#!/bin/bash
#BSUB -n 2
#BSUB -q ccm_queue
#BSUB -o out_file
. /opt/modules/default/init/bash
cd $LS_SUBCWD
export PATH=${PATH}:/mnt/lustre_server/ccmuser/isv_app/Commands
ln -s ../e5.inp e5.inp
export TMPDIR=${LS_SUBCWD}/temp
mkdir $TMPDIR
module load ccm
ccmrun isv_app job=e5 cpus=32 interactive
```

To submit a job to LSF, direct `lsfscript` to the `bsub` command:

```
% bsub < lsfscript
```

6.3.3 X11 Forwarding in CCM

Applications that require X11 forwarding (or tunneling) can use the `qsub -V` option to pass the `DISPLAY` variable to the emulated cluster. Users can then forward X traffic by using `ccmlogin`, as in the following example:

```
ssh -Y login
qsub -V -q=ccm_queue -lmppwidth=1
ccmlogin -V
```

6.3.4 ISV Application Acceleration (IAA)

IAA is a feature that potentially improves application performance by enabling the MPI implementation to directly use the high speed interconnect rather than requiring an additional TCP/IP layer. To MPI, the Aries or Gemini network looks as if it is an Infiniband network that supports the standard OFED (OpenFabrics Enterprise Distribution) API. By default, loading the `ccm` module automatically loads the `cray-isvaccel` module, which sets the general environment options for IAA. However, there are some settings that are specific to implementations of MPI. The method of passing these settings to CCM is highly application-specific. The following serves as a general guide to configuring your application's MPI and setting up the necessary CCM environment for application acceleration with Infiniband over the high speed network. Platform MPI and Open MPI are presently supported.

6.3.4.1 Configuring Platform MPI (HP-MPI) and Launching `mpirun`

Cray recommends you pass the `-IBV` option to `mpirun` to ensure that Platform MPI takes advantage of application acceleration. Without this option, any unexpected problem in application acceleration will cause Platform MPI to fall back to using TCP/IP, resulting in poor performance without explanation.

6.3.4.2 Caveats and Limitations for IAA

You may encounter the following known caveats or limitations when using IAA:

- Only Platform MPI and Open MPI are presently supported.
- IAA supports up to 2048 processing elements per application.
- IAA does not yet support 32-bit applications.
- IAA does not support application code that uses alternately named MPI entry points, such as `PMPI_Init()`.
- Use batch reservation resources efficiently as IAA allocates resources based on the reservation made for CCM. It is possible that an unnecessarily large job reservation will result in memory registration errors application failures.

6.3.4.3 Troubleshooting IAA

- "`Error detected by IBGNI. Subsequent operation may be unreliable`."

  This message indicates that IAA has reported an error to the MPI implementation. Under most conditions, the MPI will properly handle the error and continue. If the job completes successfully, Cray recommends that you disregard the warning messages of this nature. However, if the job aborts, this message can provide important clues about what went wrong.

- "`libibgni:  Could not open /tmp/ccm_alps_info (No such file or directory).`"

  This means that CCM is improperly configured. Contact your system administrator if receiving the message.

- "`lsmod test for MPI_ICMOD_IBV__IBV_MAIN could not find module in list ib_core`."

  This error indicates that Platform MPI is not correctly configured to use Infiniband.

- "`libibverbs:  Fatal:  Couldn't read uverbs ABI version`."

  It is likely that the incorrect version of libibverbs is being linked with the application, which indicates a CLE installation issue. Contact your system administrator when you see this error.

- "`FLEXlm error:  -15,570.  System Error:  19 "Cannot assign requested address`."

This error can occur on systems that use Platform MPI and rely on RSIP for external connectivity. If MPI applications are run in quick succession, the number of ports available to RSIP become exhausted. The solution is to leave more time between MPI runs.

- "`libhugetlbfs [nid000545:5652]:  WARNING: Layout problem with segments 0 and 1.  Segments would overlap.`"

  This is a warning from the huge pages library and will not interrupt execution of the application.

- "`mpid:  IBV requested on node localhost, but not available.`"

  This happens when running Platform MPI in close succession after a `ccmlogin`. The solution is to allow enough time between executions of `mpirun` and `ccmlogin`.

- "`Fatal error detected by IBGNI: Network error is unrecoverable:  SOURCE_SSID_SRSP:MDD_INV`"

  This is a secondary error caused by one or more PEs aborting with subsequent network messages arriving for them. Check earlier in your program output for the primary issue.

- "`mpid:  IBV requested on node localhost, but not available.`"

  If you rerun Platform MPI jobs too close together, it will fail before the IBGNI packet wait timer completes:

  ```
  user@nid00002:~/osu_benchmarks_for_platform> mpirun -np 2 -IBV ./osu_bw
  mpid: IBV requested on node localhost, but not available.
  ```

- "`PAM configuration can cause IAA to fail`"

  The problem results in permission denied errors when IAA tries to access the HSN from compute nodes other than the CCM head node. That happens because the app process is running in a different job container than the one that has permission to the HSN.

  The second job container is created by PAM, specifically the following line in `/etc/pam.d/common-session`:

  ```
  session optional /opt/cray/job/default/lib64/security/pam_job.so
  ```

- "`bind:  Invalid argument`"

  Applications using older versions of MVAPICH may abort with this message due to a bug in the MPI implementation. This bug is present in, at least, MVAPICH version 1.2a1. It is fixed in MVAPICH2-1.8a2.

6.4 Individual Software Vendor (ISV) Example

Example 7. Launching the UMT/pyMPI benchmark using CCM

The UMT/pyMPI benchmark tests MPI and OpenMP parallel scaling efficiency, thread compiling, single CPU performance, and Python functionality.

The following example runs through the UMT/pyMPI benchmark; it can use CCM and presupposes that you have installed it in your user scratch directory. The `runSuOlson.py` Python script runs the benchmark. The `-V` option passes environment variables to the cluster job:

```
module load ccm
qsub -V -q ccm_queue -I -lmppwidth=2 -l mppnodes=471
cd top_of_directory_where_extrated
a=`pwd`
export LD_LIBRARY_PATH=${a}/Teton:${a}/cmg2Kull/sources:${a}/CMG_CLEAN/src:${LD_LIBRARY_PATH}
ccmrun -n2 ${a}/Install/pyMPI-2.4b4/pyMPI python/runSuOlson.py
```

The following runs the UMT test contained in the package:

```
module load ccm
qsub -V -q ccm_queue -I -lmppwidth=2 -l mppnodes=471
qsub: waiting for job 394846.sdb to start
qsub: job 394846.sdb ready

Initializing CCM environment, Please Wait
waiting for jid....
waiting for jid....
CCM Start success, 1 of 1 responses
machine=> cd UMT_TEST
machine=> a=`pwd`
machine=> ccmrun -n2 ${a}/Install/pyMPI-2.4b4/pyMPI python/runSuOlson.py
writing grid file:  grid_2_13x13x13.cmg
Constructing mesh.
Mesh construction complete, next building region, opacity, material, etc.
mesh and data setup complete, building Teton object.
Setup complete, beginning time steps.
CYCLE 1 timerad = 3e-06
TempIters = 3 FluxIters = 3 GTAIters = 0
TrMax =     0.0031622776601684 in Zone 47 on Node 1
TeMax =     0.0031622776601684 in Zone 1239 on Node 1
Recommended time step for next rad cycle = 6e-05


********** Run Time Statistics **********
                  Cycle Advance             Accumulated
                     Time (sec)         Angle Loop Time (sec)
RADTR              = 47.432             39.991999864578

CYCLE 2 timerad = 6.3e-05

...
```

The benchmark continues for several iterations before completing.

6.5 Troubleshooting

6.5.1 CCM Initialization Fails

Immediately after the user enters the `qsub` command line, output appears as in the following example:

```
Initializing CCM environment, Please Wait
 Cluster Compatibility Mode Start failed, 1 of 4 responses
```

This error usually results when `/etc` files (e.g., `nsswitch.conf`, `resolv.conf`, `passwd`, `shadow`, etc.) are not specialized to the `cnos` class view. If you encounter this error, the system administrator must migrate these files from the `login` class view to the `cnos` class view. For more information, see *Managing System Software for the Cray Linux Environment*.

6.5.2 `pam_job.so` Is Incompatible with CCM

The `pam_job.so` module is incompatible with CCM. This can cause symptoms such as failed job cleanup and slow login. PAM jobs should be enabled only for `login` class views, not for the `cnos` class view.

Procedure 1. Disabling CSA Accounting for the cnos class view

1. Enter `xtopview` in the default view and edit `/etc/opt/cray/ccm/ccm_mounts.local` in the following manner:

   ```
   boot:~ # xtopview
   default/:/# vi /etc/opt/cray/ccm/ccm_mounts.local
   /etc/pam.d/common-session-pc.ccm /etc/pam.d/common-session bind 0
   default/:/# exit
   ```

2. Enter `xtopview` in the `cnos` view:

   ```
   boot:~ # xtopview -c cnos -x /etc/opt/cray/sdb/node_classes
   ```

3. Edit `/etc/pam.d/common-auth-pc`:

   ```
   class/cnos:/ # vi /etc/pam.d/common-auth-pc
   ```

   and remove or comment the following line:

```
# session optional        /opt/cray/job/default/lib64/security/pam_job.so
```

4. Edit `/etc/pam.d/common-session` to include:

   ```
   session optional pam_mkhomedir.so skel=/software/skel
   session required pam_limits.so
   session required pam_unix2.so
   session optional pam_ldap.so
   session optional pam_umask.s
   session optional /opt/cray/job/default/lib64/security/pam_job.so
   ```

5. Edit `/etc/pam.d/common-session-pc.ccm` to remove or comment all of the following:

   ```
   session optional pam_mkhomedir.so skel=/software/skel
   session required pam_limits.so
   session required pam_unix2.so
   session optional pam_ldap.so
   ```

6.5.3 `PMGR_COLLECTIVE ERROR`

When you see the error, "`PMGR_COLLECTIVE ERROR: uninitialized MPI task: Missing required environment variable: MPIRUN_RANK`," you are likely trying to run an application compiled with a mismatched version of MPI.

6.5.4 Job Hangs When sa Parameter Is Passed to Platform MPI

The `sa` parameter is provided by Platform MPI to enable MPI messages to continue flowing even when an application is consuming CPU time for long periods. Platform MPI enables a timer that generates signals at regular intervals. The signals interrupt the application and allow Platform MPI to use some necessary CPU cycles.

MP-MPI and Platform MPI 7.*x* have a bug that may cause intermittent hangs when this option is enabled. This issue does not exist with Platform MPI 8.0.

6.5.5 "`MPI_Init: dlopen`" Error(s)

The error message, "`MPI_Init: dlopen /opt/platform_mpi/lib/linux_amd64/plugins/default.so: undefined symbol`," is likely caused by a library search path that includes an MPI implementation which is different from the implementation being used by the application.

6.5.6 Bus Errors In an Application, MPI, or `libibgni`

Sometimes bus errors are due to bugs in the application software. However, the Linux kernel will also generate a bus error if it encounters various errors while handling a page fault. The most likely of those errors is running out of RAM or being unable to allocate a huge page due to memory fragmentation.

6.5.7 `glibc.so` Errors at Start of Application Launch

This error may occur nearly immediately after submission. In certain applications, like FLUENT, glibc errors and a stack trace are appear in `stderr`. This problem typically involves the license server. Be sure to include a line return at the end of your `~/.flexlmrc` file.

6.5.8 "`orted:  command not found`"

This message can appear when using an Open MPI build that is not in the default `PATH`. To avoid the problem, using the `--prefix` command argument to `mpirun` to specify the location of Open MPI.

6.6 Caveats and Limitations for CCM

6.6.1 ALPS Does Not Accurately Reflect CCM Job Resources

Because CCM is transparent to the user application, ALPS utilities such as `apstat` do not accurately reflect resources used by a CCM job.

6.6.2 Open MPI and Moab and TORQUE Integration Not Supported

Open MPI provides native Moab and TORQUE integration. However, CCM does not support this mode or applications that use a shrink-wrapped MPI with this mode. Checking `ompi_info` will reveal if it was built with this integration. It will look like the following:

```
% ompi_info | grep tm
MCA memory: ptmalloc2 (MCA v2.0, API v2.0, Component v1.3.3)
MCA ras: tm (MCA v2.0, API v2.0, Component v1.3.3)
MCA plm: tm (MCA v2.0, API v2.0, Component v1.3.3)
```

You can rebuild Open MPI to disable Moab and TORQUE integration using the following options to the `configure` script:

```
./configure --enable-mca-no-build=plm-tm,ras-tm --disable-mpi-f77 \
   --disable-mpi-f90 \
   --prefix=path_to_install
```

Which should result in no TM API being displayed by `ompi_info`:

```
% ompi_info | grep tm
MCA memory: ptmalloc2 (MCA v2.0, API v2.0, Component v1.3.3)
```

6.6.3 Miscellaneous Limitations

The following limitations apply to supporting cluster queues with CLE 4.1 on Cray systems:

- Applications must fit in the physical node memory because swap space is not supported in CCM.
- Core specialization is not supported with CCM.
- CCM does not support applications that are built in Cray Compiling Environment (CCE) with Fortran 2008 with coarrays or Unified Parallel C (UPC) compiling options, nor any Cray built libraries built with these implementations. Applications built using the Cray SHMEM or Cray MPI libraries are also not compatible with CCM.

Using Checkpoint/Restart [7]

The checkpoint/restart facility enables you to save job state to a checkpoint file and restart the job from its latest checkpoint at a future time. Cray checkpoint/restart is based on Berkeley Lab Checkpoint Restart (BLCR). Moab and TORQUE and PBS Professional are supported workload management systems. Cray checkpoint/restart is only supported on Cray XE systems.

Parallel applications must use MPI, SHMEM and/or Open MP; other parallel programming models are not supported. In general, MPI-2 applications are supported, but MPI process management is not supported. No changes to application source code are required to checkpoint and restart a job. For Open MP applications, you must load `crprep` module to enable checkpoint/restart functionality.

The workload management systems PBS Professional and Moab and TORQUE provide these checkpoint/restart commands:

- `qhold`, which checkpoints a job, releases resources assigned to the job, and places the job in hold state in the job queue.
- `qchkpt`, which checkpoints a job, but the job keeps running.
- `qrls`, which releases a checkpointed job from hold state; the job resumes running.
- `qrerun`, which restarts a previously checkpointed job that has completed, is still queued in the completed state, and has not yet exited the workload management system.

  Note: A system variable sets the amount of time a job can remain in the queue in the completed state. Once a job has been removed from the queue, you can no longer use `qrerun` to restart it.

See the `qhold`(1), `qchkpt`(1), `qrls`(1), and `qrerun`(1) man pages for details about these commands.

Note: Use the WMS commands, not native the BLCR commands. The native BLCR commands are not supported. Also, use the Cray man pages; the BLCR man pages document some features that are not supported on Cray systems.

To use checkpoint/restart, you must load the workload management system module (`moab` or `pbs`) and the `blcr` module. Loading the `blcr` module causes subsequent compilations to link the libraries that make the application checkpointable.

You should be aware of the following checkpoint/restart usage restrictions:

- You cannot checkpoint single (1) rank MPI applications.
- You cannot checkpoint/restart applications that are launched interactively through `aprun`.
- You cannot checkpoint/restart applications that use sockets.
- Files are referenced by name only. Only those files open at checkpoint time have their state saved. At restart time, that state is restored. For example, if a file has grown since the checkpoint, it will be truncated at restart time to the size saved at checkpoint time; the data added to the file will be lost.
- Linux asynchronous I/O is not supported.
- Applications that connect `stdin`, `stdout`, and `stderr` to a TTY are not supported.
- Applications being debugged with an interactive debugger are not supported.
- Any memory locked at the time of checkpoint will not be locked at restart. Applications that depend upon locked memory may not function properly after a checkpoint and restart.
- Unix System V shared memory and IPC (interprocess communication) resources are not supported.
- Tasks and threads created after a restart will not have the correct CPU affinity.
- Cray performance tools shall provide reasonably accurate performance information across a checkpoint/restart. Other performance tools may not give accurate performance information across a checkpoint/restart.
- A writable `/tmp` is required for on-node MPI message transfers.

For an example showing how to create, checkpoint, and restart a job, see Using Checkpoint/Restart Commands on page 107.

Optimizing Applications [8]

8.1 Using Compiler Optimization Options

After you have compiled and debugged your code and analyzed its performance, you can use a number of techniques to optimize performance. For details about compiler optimization and optimization reporting options, see the following manuals:

- *Cray C and C++ Reference Manual, Cray Fortran Reference Manual*
- *PGI User's Guide*
- *Using the GNU Compiler Collection (GCC)*
- *PathScale Compiler Suite User Guide*
- *Intel C++ Compiler Professional Edition for Linux*
- *Intel Fortran Compiler Professional Edition for Linux*

Optimization produces code that is more efficient and runs significantly faster than unoptimized code. Optimization can be performed at the compilation unit level through compiler driver options or to selected portions of code through the use of directives or pragmas. Because optimization may increase compilation time and may make debugging difficult, it is best to use performance analysis data in advance to isolate the portions of code where optimization would provide the greatest benefits.

You also can use `aprun` affinity options to optimize applications.

In the following example, a Fortran matrix multiply subroutine is optimized. The compiler driver option generates an optimization report.

Source code of `matrix_multiply.f90`:

```
subroutine mxm(x,y,z,m,n)
real*8 x(m,n), y(m,n), z(n,n)

do k = 1,n
  do j = 1,n
    do i = 1,m
      x(i,j) = x(i,j) + y(i,k)*z(k,j)
    enddo
  enddo
enddo

end
```

PGI Fortran compiler command:

```
% ftn -c -fast -Minfo matrix_multiply.f90
```

Optimization report:

```
mxm:
     5, Interchange produces reordered loop nest: 7, 5, 9
     9, Generated 3 alternate loops for the inner loop
        Generated vector sse code for inner loop
        Generated 2 prefetch instructions for this loop
        Generated vector sse code for inner loop
        Generated 2 prefetch instructions for this loop
        Generated vector sse code for inner loop
        Generated 2 prefetch instructions for this loop
        Generated vector sse code for inner loop
        Generated 2 prefetch instructions for this loop
```

To generate an optimization report (*loopmark listing*) by using the Cray Fortran compiler, enter:

```
% module swap PrgEnv-pgi PrgEnv-cray
% ftn -ra -c matrix_multiply.f90
```

Optimization report (file `matrix_multiply.lst`):

```
    %%%    L o o p m a r k   L e g e n d    %%%

     Primary Loop Type        Modifiers
     ------- ---- ----        ---------
                              a - vector atomic memory operation
     A  - Pattern matched     b - blocked
     C  - Collapsed           f - fused
     D  - Deleted             i - interchanged
     E  - Cloned              m - streamed but not partitioned
     I  - Inlined             p - conditional, partial and/or computed
     M  - Multithreaded       r - unrolled
     P  - Parallel/Tasked     s - shortloop
     V  - Vectorized          t - array syntax temp used
     W  - Unwound             w - unwound

    1.           subroutine mxm(x,y,z,m,n)
    2.           real*8 x(m,n), y(m,n), z(n,n)
    3.
    4.  D------< do k = 1,n
    5.  D 2----<   do j = 1,n
    6.  D 2 A--<     do i = 1,m
    7.  D 2 A          x(i,j) = x(i,j) + y(i,k)*z(k,j)
    8.  D 2 A-->     enddo
    9.  D 2---->   enddo
   10.  D------> enddo
   11.
   12.           end

ftn-6002 ftn: SCALAR File = matrix_multiply.f90, Line = 4
  A loop starting at line 4 was eliminated by optimization.

ftn-6002 ftn: SCALAR File = matrix_multiply.f90, Line = 5
  A loop starting at line 5 was eliminated by optimization.

ftn-6202 ftn: VECTOR File = matrix_multiply.f90, Line = 6
  A loop starting at line 6 was replaced by a library call.
```

8.2 Using `aprun` Memory Affinity Options

Each Cray compute node has local-NUMA-node memory and remote-NUMA-node memory. Remote-NUMA-node memory references, such as a NUMA node 0 PE accessing NUMA node 1 memory, can adversely affect performance. Cray has added `aprun` memory affinity options to give you run time controls that may optimize memory references.

Applications can use one or all NUMA nodes of a Cray system compute node. If an application is placed using one NUMA node, other NUMA nodes are not used and the application processes are restricted to using local-NUMA-node memory. This memory usage policy is enforced by running the application processes within a *cpuset*. A `cpuset` consists of cores and local memory on a compute node.

When an application is placed using all NUMA nodes, the cpuset includes all node memory and all CPUs. In this case, the application processes allocate local-NUMA-node memory first. If insufficient free local-NUMA-node memory is available, the allocation may be satisfied by using remote-NUMA-node memory. In other words, if there is not enough NUMA node *n* memory, the allocation may be satisfied by using NUMA node *n+1* memory. An exception is the `-ss` (strict memory containment) option. For this option, memory accesses are restricted to local-NUMA-node memory even if both NUMA nodes are available to the application.

The `aprun` memory affinity options are:

- `-S` *pes_per_numa_node*
- `-sn` *numa_nodes_per_node*
- `-sl` *list_of_numa_nodes*
- `-ss`

For details, see Using the `aprun` Command on page 15.

Use these `aprun` options for each element of an MPMD application and vary them with each MPMD element as required.

Compute nodes are considered for the application placement if any of the following conditions is true:

- The `-sn` value is `2`.
- The `-sl` list has more than one entry.
- The `-sl` list is the highest-ordered NUMA node.
- The `-S` value along with a `-N` value requires two NUMA nodes (such as `-N 4 -S 2`).

Use `cnselect numcores.eq.`*number_of_cores* to get a list the Cray system compute nodes.

You can use the `aprun -L` or `qsub -lmppnodes` options to specify those lists or a subset of those lists. For additional information, see the `aprun`(1), `cnselect`(1), and `qsub`(1) man pages.

8.3 Using aprun CPU Affinity Options

CNL can dynamically distribute work by allowing PEs and threads to migrate from one CPU to another within a node. In some cases, moving processes from CPU to CPU increases cache misses and translation lookaside buffer (TLB) misses and therefore reduces performance. Also, there may be cases where an application runs faster by avoiding or targeting a particular CPU. The `aprun` CPU affinity options let you bind a process to a particular CPU or the CPUs on a NUMA node. These options apply to all Cray multicore compute nodes.

Applications are assigned to a `cpuset` and can run only on the CPUs specified by the `cpuset`. Also, applications can allocate memory only on memory defined by the `cpuset`. A `cpuset` can be a compute node (default) or a NUMA node.

The CPU affinity options are:

- `-cc` *cpu-list* `|` *keyword*
- `-cp` *cpu_placement_file_name*

For details, see Using the `aprun` Command on page 15.

These `aprun` options can be used for each element of an MPMD application and can vary with each MPMD element.

8.4 Exclusive Access

The `-F` affinity option for `aprun` provides a program with exclusive access to all the processing and memory resources on a node.

This option assigns all compute node cores and compute node memory to the application's `cpuset`. Used with the `-cc` option, it enables an application programmer to bind processes to those mentioned in the affinity string.

There are two modes: `exclusive` and `share`. The share mode restricts the application specific `cpuset` contents to only the application reserved cores and memory on NUMA node boundaries. For example, if an application requests and is assigned cores and memory on NUMA node `0`, then only NUMA node `0` cores and memory are contained within the application `cpuset`. The application cannot access the cores and memory of the other NUMA nodes on that compute node.

Administrators can modify `/etc/alps.conf` to set a policy for access modes. If `nodeShare` is not specified in this file, the default mode remains `exclusive`; setting to `share` makes the default `share` access mode. Users can override the system-wide policy by specifying `aprun -F exclusive` at the command line or within their respective batch scripts. For additional information, see the `aprun`(1) man page.

8.5 Optimizing Process Placement on Multicore Nodes

Multicore systems can run more tasks simultaneously, which increases overall system performance. The trade-offs are that each core has less local memory (because it is shared by the cores) and less system interconnection bandwidth (which is also shared).

Processes are placed in packed rank-sequential order, starting with the first node. For a 100-core, 50-node job running on dual-core nodes, the layout of ranks on cores is:

	Node 1		**Node 2**		**Node 3**		…	**Node 50**	
Core	0	1	0	1	0	1	…	0	1
Rank	0	1	2	3	4	5	…	98	99

MPI supports multiple interconnect device drivers for a single MPI job. This allows each process (rank) of an MPI job to create the most optimal messaging path to every other process in the job, based on the topology of the given ranks. The SMP device driver is based on shared memory and is used for communication between ranks that share a node. The GNI device driver is used for communication between ranks that span nodes.

To attain the fastest possible run time, try running your program on only one core of each node. (In this case, the other cores are allocated to your job, but are idle.) This allows each process to have full access to the system interconnection network.

For example, you could use the commands:

```
% cnselect numcores.gt.1
20-175
% aprun -n 64 -N 1 -L 20-175 ./prog1
```

to launch `prog1` on one core of each of 64 multicore nodes.

Example Applications [9]

This chapter presents examples that show how to compile, link, and run applications.

Verify that your work area is in a Lustre-mounted directory. Then use the `module list` command to verify that the correct modules are loaded. Each following example lists the modules that have to be loaded.

9.1 Running a Basic Application

This example shows how to compile program `simple.c` and launch the executable.

One of the following modules is required:

```
PrgEnv-cray
PrgEnv-pgi
PrgEnv-gnu
PrgEnv-pathscale
PrgEnv-intel
```

Create a C program, `simple.c`:

```
#include "mpi.h"

int main(int argc, char *argv[])
{
  int rank;
  int numprocs;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);

  printf("hello from pe %d of %d\n",rank,numprocs);
  MPI_Finalize();
}
```

Compile the program:

```
% cc -o simple simple.c
```

Run the program:

```
% aprun -n 6 ./simple
hello from pe 0 of 6
hello from pe 5 of 6
hello from pe 4 of 6
hello from pe 3 of 6
hello from pe 2 of 6
hello from pe 1 of 6
Application 135891 resources: utime ~0s, stime ~0s
```

9.2 Running an MPI Application

This example shows how to compile, link, and run an MPI program. The MPI program distributes the work represented in a reduction loop, prints the subtotal for each PE, combines the results from the PEs, and prints the total.

One of the following modules is required:

```
PrgEnv-cray
PrgEnv-pgi
PrgEnv-gnu
PrgEnv-pathscale
PrgEnv-intel
```

Create a Fortran program, `mpi.f90`:

```
program reduce
include "mpif.h"

integer n, nres, ierr

call MPI_INIT (ierr)
call MPI_COMM_RANK (MPI_COMM_WORLD,mype,ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD,npes,ierr)

nres = 0
n = 0

do i=mype,100,npes
  n = n + i
enddo

print *, 'My PE:', mype, ' My part:',n

call MPI_REDUCE (n,nres,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD,ierr)

if (mype == 0) print *,'   PE:',mype,'Total is:',nres

call MPI_FINALIZE (ierr)

end
```

Compile `mpi.f90`:

```
% ftn -o mpi mpi.f90
```

Run program `mpi`:

```
% aprun -n 6 ./mpi | sort
    PE:            0 Total is:         5050
 My PE:            0  My part:          816
 My PE:            1  My part:          833
 My PE:            2  My part:          850
 My PE:            3  My part:          867
 My PE:            4  My part:          884
 My PE:            5  My part:          800
Application 3016865 resources: utime ~0s, stime ~0s
```

If desired, you could use this C version of the program:

```
/* program reduce */

#include <stdio.h>
#include "mpi.h"

int main (int argc, char *argv[])
{
  int i, sum, mype, npes, nres, ret;
  ret = MPI_Init (&argc, &argv);
  ret = MPI_Comm_size (MPI_COMM_WORLD, &npes);
  ret = MPI_Comm_rank (MPI_COMM_WORLD, &mype);
  nres = 0;
  sum = 0;
  for (i = mype; i <=100; i += npes) {
    sum = sum + i;
  }

  (void) printf ("My PE:%d  My part:%d\n",mype, sum);
  ret = MPI_Reduce (&sum,&nres,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD);
  if (mype == 0)
  {
    (void) printf ("PE:%d  Total is:%d\n",mype, nres);
  }
  ret =  MPI_Finalize ();

}
```

9.3 Using the Cray `shmem_put` Function

This example shows how to use the `shmem_put64()` function to copy a contiguous data object from the local PE to a contiguous data object on a different PE.

One of the following modules is required:

```
PrgEnv-cray
PrgEnv-pgi
PrgEnv-gnu
PrgEnv-pathscale
PrgEnv-intel
```

Source code of C program (`shmem_put.c`):

```
/*
 *      simple put test
 */

#include <stdio.h>
#include <stdlib.h>
#include <mpp/shmem.h>

/* Dimension of source and target of put operations */
#define DIM     1000000

long target[DIM];
long local[DIM];

main(int argc,char **argv)
{
  register int i;
  int my_partner, my_pe;

  /* Prepare resources required for correct functionality
     of SHMEM on XT. Alternatively, shmem_init() could
     be called. */
  start_pes(0);

  for (i=0; i<DIM; i++) {
    target[i] = 0L;
    local[i] = shmem_my_pe() + (i * 10);
  }

  my_pe = shmem_my_pe();

  if(shmem_n_pes()%2) {
    if(my_pe == 0) printf("Test needs even number of processes\n");
    /* Clean up resources before exit. */
    shmem_finalize();
    exit(0);
  }

  shmem_barrier_all();

  /* Test has to be run on two procs. */
  my_partner = my_pe % 2 ? my_pe - 1 : my_pe + 1;

  shmem_put64(target,local,DIM,my_partner);

  /* Synchronize before verifying results. */
  shmem_barrier_all();

  /* Check results of put */
  for(i=0; i<DIM; i++) {
    if(target[i] != (my_partner + (i * 10))) {
      fprintf(stderr,"FAIL (1) on PE %d target[%d] = %d (%d)\n",
        shmem_my_pe(), i, target[i],my_partner+(i*10));
      shmem_finalize();
      exit(-1);
    }
  }

  printf(" PE %d: Test passed.\n",my_pe);

   /* Clean up resources. */
   shmem_finalize();
}
```

Compile `shmem_put.c` and create executable `shmem_put`:

```
% cc -o shmem_put shmem_put.c
```

Run `shmem_put`:

```
% aprun -n 12 -L 56 ./shmem_put
 PE 5: Test passed.
 PE 6: Test passed.
 PE 3: Test passed.
 PE 1: Test passed.
 PE 4: Test passed.
 PE 2: Test passed.
 PE 7: Test passed.
 PE 11: Test passed.
 PE 10: Test passed.
 PE 9: Test passed.
 PE 8: Test passed.
 PE 0: Test passed.

Application 57916 exit codes: 255
Application 57916 resources: utime ~1s, stime ~2s
```

9.4 Using the Cray `shmem_get` Function

This example shows how to use the `shmem_get()` function to copy a contiguous data object from a different PE to a contiguous data object on the local PE.

One of the following modules is required:

```
PrgEnv-pgi
PrgEnv-cray
PrgEnv-gnu
PrgEnv-pathscale
PrgEnv-intel
```

The `cray-shmem` module is also required.

Note: The Fortran module for Cray SHMEM is not supported. Use the `INCLUDE 'mpp/shmem.fh'` statement instead.

Source code of Fortran program (`shmem_get.f90`):

```
program reduction
include 'mpp/shmem.fh'

real values, sum
common /c/ values
real work

call start_pes(0)
values=my_pe()
call shmem_barrier_all! Synchronize all PEs
sum = 0.0
do i = 0,num_pes()-1
  call shmem_get(work, values, 1, i)   ! Get next value
  sum = sum + work     ! Sum it
enddo

print*, 'PE',my_pe(),' computedsum=',sum

call shmem_barrier_all
call shmem_finalize

end
```

Compile `shmem_get.f90` and create executable `shmem_get`:

```
% ftn -o shmem_get shmem_get.f90
```

Run `shmem2`:

```
% aprun -n 6 ./shmem_get
 PE            0  computedsum=    15.00000
 PE            5  computedsum=    15.00000
 PE            4  computedsum=    15.00000
 PE            3  computedsum=    15.00000
 PE            2  computedsum=    15.00000
 PE            1  computedsum=    15.00000
Application 137031 resources: utime ~0s, stime ~0s
```

9.5 Running Partitioned Global Address Space (PGAS) Applications

To run Unified Parallel C (UPC) or Fortran 2008 coarrays applications, use the Cray C compiler. These are not supported for PGI, GCC, PathScale, or Intel C compilers.

This example shows how to compile and run a Cray C program that includes Unified Parallel C (UPC) functions.

Modules required:

```
PrgEnv-cray
```

Check that these additional modules are loaded. These are part of the default modules on the login node loaded with the module `Base-opts`, but you will encounter an error with PGAS applications with these modules unloaded:

```
udreg
ugni
dmapp
```

9.5.1 Running a Unified Parallel C (UPC) Application

The following is the source code of program `upc_cray.c`:

```
#include <upc.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
  int i;
  for (i = 0; i < THREADS; ++i)
    {
      upc_barrier;
      if (i == MYTHREAD)
        printf ("Hello world from thread: %d\n", MYTHREAD);
    }
  return 0;
}
```

Compile `upc_cray.c` and run executable `cray_upc`:

```
% cc -h upc -o upc_cray upc_cray.c
% aprun -n 2 ./upc_cray
Hello world from thread: 0
Hello world from thread: 1
Application 251523 resources: utime ~0s, stime ~0s
```

Note: You need to include the `-h upc` option on the `cc` command line.

9.5.2 Running a Fortran 2008 Application Using Coarrays

The following is the source code of program `simple_caf.f90`:

```
program simple_caf
implicit none

integer :: npes,mype,i
real    :: local_array(1000),total
real    :: coarray[*]

mype = this_image()
npes = num_images()

if (npes < 2) then
 print *, "Need at least 2 images to run"
  stop
end if

do i=1,1000
   local_array(i) = sin(real(mype*i))
end do

coarray = sum(local_array)
sync all

if (mype == 1) then
 total = coarray + coarray[2]
   print *, "Total from images 1 and 2 is ",total
end if

end program simple_caf
```

Compile `simple_caf.f90` and run the executable:

```
% ftn -hcaf -o simple_caf simple_caf.f90
/opt/cray/xt-asyncpe/3.9.39/bin/ftn: INFO: linux target is being used
% aprun -n2 simple_caf
  Total from images 1 and 2 is  1.71800661
  Application 39512 resources: utime ~0s, stime ~0s
```

9.6 Running an Acclerated Cray LibSci Routine

The following sample program displays usage of the `libsci_acc` accelerated libraries to perform LAPACK routines. The program solves a linear system of equations (**AX** = **B**) by computing the LU factorization of matrix **A** in `DGETRF` and completing the solution in `DGETRS`. For more information on auto-tuned LibSci GPU routines, see the `intro_libsci_acc`(3s) man page.

Modules required:

```
PrgEnv-cray
craype-accel-nvidia35
cray-libsci
```

Source of the program

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <libsci_acc.h>

int main ( int argc, char **argv ) {
  double *A, *B; int *ipiv;
  int n, nrhs, lda, ldb, info;
  int i, j;

  n = lda = ldb = 5;
  nrhs = 1;
  ipiv = (int *)malloc(sizeof(int)*n);
  B = (double *)malloc(sizeof(double)*n*nrhs);

  libsci_acc_init();
  libsci_acc_HostAlloc( (void **)&A, sizeof(double)*n*n );

  for ( i = 0; i < n; i++ ) {
   for ( j = 0; j < n; j++ ) {
    A[i*lda+j] = drand48();
   }
  }

  for ( i = 0; i < nrhs; i++ ) {

   for ( j = 0; j < n; j++ ) {
    B[i*ldb+j] = drand48();
   }
  }

  printf("\n\nMatrix A\n");
  for ( i = 0; i < n  ; i++ ) {
   if (i > 0)
    printf("\n");
   for ( j = 0; j < n; j++ ) {
    printf("\t%f",A[i*lda+j]);
   }
   }

  printf("\n\nRHS/B\n");
  for (i=0; i < nrhs; i++) {
   if (i > 0)
      printf("\n");
   for ( j = 0; j < n; j++ ) {
    if (i==0)
     printf("|  %f\n",B[i*ldb+j]);
    else
     printf("  %f\n",B[i*ldb+j]);
   }
   }

 printf("\n\nSolution/X\n");
 dgetrf( n, n, A, lda, ipiv, &info);
 dgetrs('N', n, nrhs, A, lda, ipiv, B, ldb, &info);
 for ( i = 0; i < nrhs; i++ ) {
  printf("\n");
  for ( j = 0; j < n; j++ ) {
   printf("%f\n",B[i*ldb+j]);
   }
 }
 printf("\n");

 libsci_acc_FreeHost ( A );
 free(ipiv);
 free(B);
 libsci_acc_finalize();

}
```

```
% aprun -n1 ./a.out

Matrix A
 0.000000 0.000985 0.041631 0.176643 0.364602
 0.091331 0.092298 0.487217 0.526750 0.454433
 0.233178 0.831292 0.931731 0.568060 0.556094
 0.050832 0.767051 0.018915 0.252360 0.298197
 0.875981 0.531557 0.920261 0.515431 0.810429

RHS/B
|  0.188420
|  0.886314
|  0.570614
|  0.076775
|  0.815274


Solution/X

3.105866
-2.649034
1.836310
-0.543425
0.034012
```

9.7 Running a PETSc Application

This example (Copyright 1995-2004 University of Chicago) shows how to use PETSc functions to solve a linear system of partial differential equations.

Note: There are many ways to use the PETSc solvers. This example is intended to show the basics of compiling and running a PETSc program on a Cray system. It presents one simple approach and may not be the best template to use in writing user code. For issues that are not specific to Cray systems, you can get technical support through `petsc-users@mcs.anl.gov`.

The source code for this example includes a comment about the use of the `mpiexec` command to launch the executable. Use `aprun` instead.

Modules required:

```
petsc
```

and one of the following:

```
PrgEnv-cray
PrgEnv-pgi
PrgEnv-gnu
PrgEnv-pathscale
PrgEnv-intel
```

Source code of program `ex2f.F`:

```
!
!  Description: Solves a linear system in parallel with KSP (Fortran code).
!               Also shows how to set a user-defined monitoring routine.
!
!  Program usage: mpiexec -np  ex2f [-help] [all PETSc options]
!
!/*T
!  Concepts: KSP^basic parallel example
!  Concepts: KSP^setting a user-defined monitoring routine
!  Processors: n
!T*/
!
! -----------------------------------------------------------------------

      program main
      implicit none
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!                    Include files
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!
!  This program uses CPP for preprocessing, as indicated by the use of
!  PETSc include files in the directory petsc/include/finclude.  This
!  convention enables use of the CPP preprocessor, which allows the use
!  of the #include statements that define PETSc objects and variables.
!
!  Use of the conventional Fortran include statements is also supported
!  In this case, the PETsc include files are located in the directory
!  petsc/include/foldinclude.
!
!  Since one must be very careful to include each file no more than once
!  in a Fortran routine, application programmers must exlicitly list
!  each file needed for the various PETSc components within their
!  program (unlike the C/C++ interface).
!
!  See the Fortran section of the PETSc users manual for details.
!
!  The following include statements are required for KSP Fortran programs:
!     petsc.h       - base PETSc routines
!     petscvec.h    - vectors
!     petscmat.h    - matrices
!     petscpc.h     - preconditioners
!     petscksp.h    - Krylov subspace methods
!  Include the following to use PETSc random numbers:
!     petscsys.h    - system routines
!  Additional include statements may be needed if using additional
!  PETSc routines in a Fortran program, e.g.,
!     petscviewer.h - viewers
!     petscis.h     - index sets
!
#include "finclude/petsc.h"
#include "finclude/petscvec.h"
#include "finclude/petscmat.h"
#include "finclude/petscpc.h"
#include "finclude/petscksp.h"
#include "finclude/petscsys.h"
!
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!                   Variable declarations
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!
!  Variables:
!     ksp     - linear solver context
!     ksp      - Krylov subspace method context
!     pc       - preconditioner context
!     x, b, u  - approx solution, right-hand-side, exact solution vectors
!     A        - matrix that defines linear system
!     its      - iterations for convergence
!     norm     - norm of error in solution
!     rctx     - random number generator context
!
!  Note that vectors are declared as PETSc "Vec" objects.  These vectors
!  are mathematical objects that contain more than just an array of
!  double precision numbers. I.e., vectors in PETSc are not just
!        double precision x(*).
!  However, local vector data can be easily accessed via VecGetArray().
!  See the Fortran section of the PETSc users manual for details.
!
      double precision  norm
      PetscInt  i,j,II,JJ,m,n,its
      PetscInt  Istart,Iend,ione
      PetscErrorCode ierr
      PetscMPIInt     rank,size
      PetscTruth  flg
      PetscScalar v,one,neg_one
      Vec         x,b,u
      Mat         A
      KSP         ksp
      PetscRandom rctx

!  These variables are not currently used.
!      PC          pc
!      PCType      ptype
!      double precision tol


!  Note: Any user-defined Fortran routines (such as MyKSPMonitor)
!  MUST be declared as external.

      external MyKSPMonitor,MyKSPConverged

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!                 Beginning of program
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
      call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
      m = 3
      n = 3
      one  = 1.0
      neg_one = -1.0
      ione    = 1
      call PetscOptionsGetInt(PETSC_NULL_CHARACTER,'-m',m,flg,ierr)
      call PetscOptionsGetInt(PETSC_NULL_CHARACTER,'-n',n,flg,ierr)
      call MPI_Comm_rank(PETSC_COMM_WORLD,rank,ierr)
      call MPI_Comm_size(PETSC_COMM_WORLD,size,ierr)

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!      Compute the matrix and right-hand-side vector that define
!      the linear system, Ax = b.
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

!  Create parallel matrix, specifying only its global dimensions.
!  When using MatCreate(), the matrix format can be specified at
!  runtime. Also, the parallel partitioning of the matrix is
!  determined by PETSc at runtime.

      call MatCreate(PETSC_COMM_WORLD,A,ierr)
      call MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,m*n,m*n,ierr)
      call MatSetFromOptions(A,ierr)

!  Currently, all PETSc parallel matrix formats are partitioned by
!  contiguous chunks of rows across the processors.  Determine which
!  rows of the matrix are locally owned.

      call MatGetOwnershipRange(A,Istart,Iend,ierr)

!  Set matrix elements for the 2-D, five-point stencil in parallel.
!   - Each processor needs to insert only elements that it owns
!     locally (but any non-local elements will be sent to the
!     appropriate processor during matrix assembly).
!   - Always specify global row and columns of matrix entries.
!   - Note that MatSetValues() uses 0-based row and column numbers
!     in Fortran as well as in C.

!     Note: this uses the less common natural ordering that orders first
!     all the unknowns for x = h then for x = 2h etc; Hence you see JH = II +- n
!     instead of JJ = II +- m as you might expect. The more standard ordering
!     would first do all variables for y = h, then y = 2h etc.

      do 10, II=Istart,Iend-1
        v = -1.0
        i = II/n
        j = II - i*n
        if (i.gt.0) then
          JJ = II - n
          call MatSetValues(A,ione,II,ione,JJ,v,INSERT_VALUES,ierr)
        endif
        if (i.lt.m-1) then
          JJ = II + n
          call MatSetValues(A,ione,II,ione,JJ,v,INSERT_VALUES,ierr)
        endif
        if (j.gt.0) then
```

```
          JJ = II - 1
          call MatSetValues(A,ione,II,ione,JJ,v,INSERT_VALUES,ierr)
        endif
        if (j.lt.n-1) then
          JJ = II + 1
          call MatSetValues(A,ione,II,ione,JJ,v,INSERT_VALUES,ierr)
        endif
        v = 4.0
        call  MatSetValues(A,ione,II,ione,II,v,INSERT_VALUES,ierr)
 10   continue

!  Assemble matrix, using the 2-step process:
!       MatAssemblyBegin(), MatAssemblyEnd()
!  Computations can be done while messages are in transition,
!  by placing code between these two statements.

      call MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY,ierr)
      call MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY,ierr)

!  Create parallel vectors.
!   - Here, the parallel partitioning of the vector is determined by
!     PETSc at runtime.  We could also specify the local dimensions
!     if desired -- or use the more general routine VecCreate().
!   - When solving a linear system, the vectors and matrices MUST
!     be partitioned accordingly.  PETSc automatically generates
!     appropriately partitioned matrices and vectors when MatCreate()
!     and VecCreate() are used with the same communicator.
!   - Note: We form 1 vector from scratch and then duplicate as needed.

      call VecCreateMPI(PETSC_COMM_WORLD,PETSC_DECIDE,m*n,u,ierr)
      call VecSetFromOptions(u,ierr)
      call VecDuplicate(u,b,ierr)
      call VecDuplicate(b,x,ierr)

!  Set exact solution; then compute right-hand-side vector.
!  By default we use an exact solution of a vector with all
!  elements of 1.0;  Alternatively, using the runtime option
!  -random_sol forms a solution vector with random components.

      call PetscOptionsHasName(PETSC_NULL_CHARACTER,                    &
     &             "-random_exact_sol",flg,ierr)
      if (flg .eq. 1) then
         call PetscRandomCreate(PETSC_COMM_WORLD,rctx,ierr)
         call PetscRandomSetFromOptions(rctx,ierr)
         call VecSetRandom(u,rctx,ierr)
         call PetscRandomDestroy(rctx,ierr)
      else
         call VecSet(u,one,ierr)
      endif
      call MatMult(A,u,b,ierr)

!  View the exact solution vector if desired

      call PetscOptionsHasName(PETSC_NULL_CHARACTER,                    &
     &             "-view_exact_sol",flg,ierr)
      if (flg .eq. 1) then
         call VecView(u,PETSC_VIEWER_STDOUT_WORLD,ierr)
      endif
```

```
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!         Create the linear solver and set various options
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

!  Create linear solver context

      call KSPCreate(PETSC_COMM_WORLD,ksp,ierr)

!  Set operators. Here the matrix that defines the linear system
!  also serves as the preconditioning matrix.

      call KSPSetOperators(ksp,A,A,DIFFERENT_NONZERO_PATTERN,ierr)

!  Set linear solver defaults for this problem (optional).
!   - By extracting the KSP and PC contexts from the KSP context,
!     we can then directly directly call any KSP and PC routines
!     to set various options.
!   - The following four statements are optional; all of these
!     parameters could alternatively be specified at runtime via
!     KSPSetFromOptions(). All of these defaults can be
!     overridden at runtime, as indicated below.

!     We comment out this section of code since the Jacobi
!     preconditioner is not a good general default.

!      call KSPGetPC(ksp,pc,ierr)
!      ptype = PCJACOBI
!      call PCSetType(pc,ptype,ierr)
!      tol = 1.e-7
!      call KSPSetTolerances(ksp,tol,PETSC_DEFAULT_DOUBLE_PRECISION,
!     &     PETSC_DEFAULT_DOUBLE_PRECISION,PETSC_DEFAULT_INTEGER,ierr)

!  Set user-defined monitoring routine if desired

      call PetscOptionsHasName(PETSC_NULL_CHARACTER,'-my_ksp_monitor',  &
     &                    flg,ierr)
      if (flg .eq. 1) then
        call KSPMonitorSet(ksp,MyKSPMonitor,PETSC_NULL_OBJECT,          &
     &                     PETSC_NULL_FUNCTION,ierr)
      endif

!  Set runtime options, e.g.,
!      -ksp_type <type> -pc_type <type> -ksp_monitor -ksp_rtol
!  These options will override those specified above as long as
!  KSPSetFromOptions() is called _after_ any other customization
!  routines.

      call KSPSetFromOptions(ksp,ierr)

!  Set convergence test routine if desired

      call PetscOptionsHasName(PETSC_NULL_CHARACTER,                    &
     &     '-my_ksp_convergence',flg,ierr)
      if (flg .eq. 1) then
        call KSPSetConvergenceTest(ksp,MyKSPConverged,                  &
     &         PETSC_NULL_OBJECT,ierr)
      endif
```

```
!
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!                      Solve the linear system
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

      call KSPSolve(ksp,b,x,ierr)

! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
!                     Check solution and clean up
! - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

!  Check the error

      call VecAXPY(x,neg_one,u,ierr)
      call VecNorm(x,NORM_2,norm,ierr)
      call KSPGetIterationNumber(ksp,its,ierr)
      if (rank .eq. 0) then
        if (norm .gt. 1.e-12) then
           write(6,100) norm,its
        else
           write(6,110) its
        endif
      endif
  100 format('Norm of error ',e10.4,' iterations ',i5)
  110 format('Norm of error < 1.e-12,iterations ',i5)

!  Free work space.  All PETSc objects should be destroyed when they
!  are no longer needed.

      call KSPDestroy(ksp,ierr)
      call VecDestroy(u,ierr)
      call VecDestroy(x,ierr)
      call VecDestroy(b,ierr)
      call MatDestroy(A,ierr)

!  Always call PetscFinalize() before exiting a program.  This routine
!    - finalizes the PETSc libraries as well as MPI
!    - provides summary and diagnostic information if certain runtime
!      options are chosen (e.g., -log_summary).  See PetscFinalize()
!      manpage for more information.

      call PetscFinalize(ierr)
      end

! --------------------------------------------------------------
!
!  MyKSPMonitor - This is a user-defined routine for monitoring
!  the KSP iterative solvers.
!
!  Input Parameters:
!    ksp   - iterative context
!    n     - iteration number
!    rnorm - 2-norm (preconditioned) residual value (may be estimated)
!    dummy - optional user-defined monitor context (unused here)
!
      subroutine MyKSPMonitor(ksp,n,rnorm,dummy,ierr)
      implicit none

#include "finclude/petsc.h"
#include "finclude/petscvec.h"
#include "finclude/petscksp.h"

      KSP              ksp
      Vec              x
      PetscErrorCode ierr
      PetscInt n,dummy
      PetscMPIInt rank
      double precision rnorm

!  Build the solution vector

      call KSPBuildSolution(ksp,PETSC_NULL_OBJECT,x,ierr)

!  Write the solution vector and residual norm to stdout
!   - Note that the parallel viewer PETSC_VIEWER_STDOUT_WORLD
!     handles data from multiple processors so that the
!     output is not jumbled.

      call MPI_Comm_rank(PETSC_COMM_WORLD,rank,ierr)
      if (rank .eq. 0) write(6,100) n
      call VecView(x,PETSC_VIEWER_STDOUT_WORLD,ierr)
      if (rank .eq. 0) write(6,200) n,rnorm

 100  format('iteration ',i5,' solution vector:')
 200  format('iteration ',i5,' residual norm ',e10.4)
      ierr = 0
      end

! -------------------------------------------------------------
!
!  MyKSPConverged - This is a user-defined routine for testing
!  convergence of the KSP iterative solvers.
!
!  Input Parameters:
!    ksp   - iterative context
!    n     - iteration number
!    rnorm - 2-norm (preconditioned) residual value (may be estimated)
!    dummy - optional user-defined monitor context (unused here)
!
      subroutine MyKSPConverged(ksp,n,rnorm,flag,dummy,ierr)

      implicit none

#include "finclude/petsc.h"
#include "finclude/petscvec.h"
#include "finclude/petscksp.h"

      KSP              ksp
      PetscErrorCode ierr
      PetscInt n,dummy
      KSPConvergedReason flag
      double precision rnorm

      if (rnorm .le. .05) then
```

```
    flag = 1
  else
    flag = 0
  endif
  ierr = 0

  end
```

Use the following `makefile.F`:

```
.SUFFIXES:  .mod .o .F

### Compilers, linkers and flags.

FC                   =       ftn
LINKER           =       ftn
FCFLAGS       =
LINKLAGS     =

### Fortran optimization options.

FOPTFLAGS      = -O3

.F.o:
 $(FC) -c ${FOPTFLAGS} ${FCFLAGS} $*.F


all : ex2f
ex2f  :  ex2f.o
 $(LINKER) -o $@ ex2f.o
```

Create and run executable `ex2f`, including the PETSc run time option `-mat_view` to display the nonzero values of the 9x9 matrix `A`:

```
% make -f makefile.F
% aprun -n 2 ./ex2f -mat_view
row 0: (0, 4)  (1, -1)  (3, -1)
row 1: (0, -1)  (1, 4)  (2, -1)  (4, -1)
row 2: (1, -1)  (2, 4)  (5, -1)
row 3: (0, -1)  (3, 4)  (4, -1)  (6, -1)
row 4: (1, -1)  (3, -1)  (4, 4)  (5, -1)  (7, -1)
row 5: (2, -1)  (4, -1)  (5, 4)  (8, -1)
row 6: (3, -1)  (6, 4)  (7, -1)
row 7: (4, -1)  (6, -1)  (7, 4)  (8, -1)
row 8: (5, -1)  (7, -1)  (8, 4)
row 0: (0, 0.25)  (3, -1)
row 1: (1, 0.25)  (2, -1)
row 2: (1, -0.25)  (2, 0.266667)  (3, -1)
row 3: (0, -0.25)  (2, -0.266667)  (3, 0.287081)
row 0: (0, 0.25)  (1, -1)  (3, -1)
row 1: (0, -0.25)  (1, 0.266667)  (2, -1)  (4, -1)
row 2: (1, -0.266667)  (2, 0.267857)
row 3: (0, -0.25)  (3, 0.266667)  (4, -1)
row 4: (1, -0.266667)  (3, -0.266667)  (4, 0.288462)
Norm of error < 1.e-12,iterations     7
Application 155514 resources: utime 0, stime 12
```

9.8 Running an OpenMP Application

This example shows how to compile and run an OpenMP/MPI application.

One of the following modules is required:

```
PrgEnv-cray
PrgEnv-pgi
PrgEnv-gnu
PrgEnv-pathscale
PrgEnv-intel
```

Note: To compile an OpenMP program using a PGI or PathScale compiler, include `-mp` on the compiler driver command line. For a GCC compiler, include `-fopenmp`. For in Intel compiler, include `-openmp`. No option is required for the Cray compilers; `-h omp` is the default.

For a PathScale OpenMP program, set the `PSC_OMP_AFFINITY` environment variable to `FALSE`.

Source code of C program `xthi.c`:

```
#define _GNU_SOURCE

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sched.h>
#include <mpi.h>
#include <omp.h>

/* Borrowed from util-linux-2.13-pre7/schedutils/taskset.c */
static char *cpuset_to_cstr(cpu_set_t *mask, char *str)
{
  char *ptr = str;
  int i, j, entry_made = 0;
  for (i = 0; i < CPU_SETSIZE; i++) {
    if (CPU_ISSET(i, mask)) {
      int run = 0;
      entry_made = 1;
      for (j = i + 1; j < CPU_SETSIZE; j++) {
        if (CPU_ISSET(j, mask)) run++;
        else break;
      }
      if (!run)
        sprintf(ptr, "%d,", i);
      else if (run == 1) {
        sprintf(ptr, "%d,%d,", i, i + 1);
        i++;
      } else {
        sprintf(ptr, "%d-%d,", i, i + run);
        i += run;
      }
      while (*ptr != 0) ptr++;
    }
  }
  ptr -= entry_made;
  *ptr = 0;
  return(str);
}

int main(int argc, char *argv[])
{
  int rank, thread;
  cpu_set_t coremask;
  char clbuf[7 * CPU_SETSIZE], hnbuf[64];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  memset(clbuf, 0, sizeof(clbuf));
  memset(hnbuf, 0, sizeof(hnbuf));
  (void)gethostname(hnbuf, sizeof(hnbuf));
  #pragma omp parallel private(thread, coremask, clbuf)
  {
    thread = omp_get_thread_num();
    (void)sched_getaffinity(0, sizeof(coremask), &coremask);
    cpuset_to_cstr(&coremask, clbuf);
    #pragma omp barrier
    printf("Hello from rank %d, thread %d, on %s. (core affinity = %s)\n",
            rank, thread, hnbuf, clbuf);
  }
  MPI_Finalize();
  return(0);
}
```

Load the `PrgEnv-pathscale` module:

```
% module swap PrgEnv-pgi PrgEnv-pathscale
```

Set the `PSC_OMP_AFFINITY` environment variable to `FALSE`:

```
% setenv PSC_OMP_AFFINITY FALSE
```

Or

```
% export PSC_OMP_AFFINITY=FALSE
```

Compile and link `xthi.c`:

```
% cc -mp -o xthi xthi.c
```

Set the OpenMP environment variable equal to the number of threads in the team:

```
% setenv OMP_NUM_THREADS 2
```

Or

```
% export OMP_NUM_THREADS=2
```

Note: If you are running Intel-compiled code, you must use one of the alternate methods when setting `OMP_NUM_THREADS`:

- Increase the `aprun -d` depth value by one.
- Use the `aprun -cc numa_node` affinity option.

Run program `xthi`:

```
% export OMP_NUM_THREADS=24
% aprun -n 1 -d 24 -L 56 xthi | sort
Application 57937 resources: utime ~1s, stime ~0s
Hello from rank 0, thread 0, on nid00056. (core affinity = 0)
Hello from rank 0, thread 10, on nid00056. (core affinity = 10)
Hello from rank 0, thread 11, on nid00056. (core affinity = 11)
Hello from rank 0, thread 12, on nid00056. (core affinity = 12)
Hello from rank 0, thread 13, on nid00056. (core affinity = 13)
Hello from rank 0, thread 14, on nid00056. (core affinity = 14)
Hello from rank 0, thread 15, on nid00056. (core affinity = 15)
Hello from rank 0, thread 16, on nid00056. (core affinity = 16)
Hello from rank 0, thread 17, on nid00056. (core affinity = 17)
Hello from rank 0, thread 18, on nid00056. (core affinity = 18)
Hello from rank 0, thread 19, on nid00056. (core affinity = 19)
Hello from rank 0, thread 1, on nid00056. (core affinity = 1)
Hello from rank 0, thread 20, on nid00056. (core affinity = 20)
Hello from rank 0, thread 21, on nid00056. (core affinity = 21)
Hello from rank 0, thread 22, on nid00056. (core affinity = 22)
Hello from rank 0, thread 23, on nid00056. (core affinity = 23)
Hello from rank 0, thread 2, on nid00056. (core affinity = 2)
Hello from rank 0, thread 3, on nid00056. (core affinity = 3)
Hello from rank 0, thread 4, on nid00056. (core affinity = 4)
Hello from rank 0, thread 5, on nid00056. (core affinity = 5)
Hello from rank 0, thread 6, on nid00056. (core affinity = 6)
Hello from rank 0, thread 7, on nid00056. (core affinity = 7)
Hello from rank 0, thread 8, on nid00056. (core affinity = 8)
Hello from rank 0, thread 9, on nid00056. (core affinity = 9)
```

The `aprun` command created one instance of `xthi`, which spawned 23 additional threads running on separate cores.

Here is another run of `xthi`:

```
% export OMP_NUM_THREADS=6
% aprun -n 4 -d 6 -L 56 xthi | sort
Application 57948 resources: utime ~1s, stime ~1s
Hello from rank 0, thread 0, on nid00056. (core affinity = 0)
Hello from rank 0, thread 1, on nid00056. (core affinity = 1)
Hello from rank 0, thread 2, on nid00056. (core affinity = 2)
Hello from rank 0, thread 3, on nid00056. (core affinity = 3)
Hello from rank 0, thread 4, on nid00056. (core affinity = 4)
Hello from rank 0, thread 5, on nid00056. (core affinity = 5)
Hello from rank 1, thread 0, on nid00056. (core affinity = 6)
Hello from rank 1, thread 1, on nid00056. (core affinity = 7)
Hello from rank 1, thread 2, on nid00056. (core affinity = 8)
Hello from rank 1, thread 3, on nid00056. (core affinity = 9)
Hello from rank 1, thread 4, on nid00056. (core affinity = 10)
Hello from rank 1, thread 5, on nid00056. (core affinity = 11)
Hello from rank 2, thread 0, on nid00056. (core affinity = 12)
Hello from rank 2, thread 1, on nid00056. (core affinity = 13)
Hello from rank 2, thread 2, on nid00056. (core affinity = 14)
Hello from rank 2, thread 3, on nid00056. (core affinity = 15)
Hello from rank 2, thread 4, on nid00056. (core affinity = 16)
Hello from rank 2, thread 5, on nid00056. (core affinity = 17)
Hello from rank 3, thread 0, on nid00056. (core affinity = 18)
Hello from rank 3, thread 1, on nid00056. (core affinity = 19)
Hello from rank 3, thread 2, on nid00056. (core affinity = 20)
Hello from rank 3, thread 3, on nid00056. (core affinity = 21)
Hello from rank 3, thread 4, on nid00056. (core affinity = 22)
Hello from rank 3, thread 5, on nid00056. (core affinity = 23)
```

The `aprun` command created four instances of `xthi` which spawned five additional threads per instance. All PEs are running on separate cores and each instance is confined to NUMA node domains on one compute node.

9.9 Running an Interactive Batch Job

This example shows how to compile and run an OpenMP/MPI application (see Running an OpenMP Application on page 95) on 16-core Cray X6 compute nodes using an interactive batch job.

Modules required:

```
pbs or moab
```

and one of the following:

```
PrgEnv-cray
PrgEnv-pgi
PrgEnv-gnu
PrgEnv-pathscale
PrgEnv-intel
```

Use the `cnselect` command to get a list of eight-core, dual-socket compute nodes:

```
% cnselect coremask.eq.65535
14-17,128-223,256-351,384-479,512-607,640-715
```

Initiate an interactive batch session:

```
% qsub -I -l mppwidth=8 -l mppdepth=4 -l mppnodes=\"14-15\"
```

Set the OpenMP environment variable equal to the number of threads in the team:

```
% setenv OMP_NUM_THREADS 4
```

Or

```
% export OMP_NUM_THREADS=4
```

Run program `omp`:

```
% aprun -n 8 -d 4 -L14-15 ./xthi | sort
Application 57953 resources: utime ~2s, stime ~2s
Hello from rank 0, thread 0, on nid00014. (core affinity = 0)
Hello from rank 0, thread 1, on nid00014. (core affinity = 1)
Hello from rank 0, thread 2, on nid00014. (core affinity = 2)
Hello from rank 0, thread 3, on nid00014. (core affinity = 3)
Hello from rank 1, thread 0, on nid00014. (core affinity = 4)
Hello from rank 1, thread 1, on nid00014. (core affinity = 5)
Hello from rank 1, thread 2, on nid00014. (core affinity = 6)
Hello from rank 1, thread 3, on nid00014. (core affinity = 7)
Hello from rank 2, thread 0, on nid00014. (core affinity = 8)
Hello from rank 2, thread 1, on nid00014. (core affinity = 9)
Hello from rank 2, thread 2, on nid00014. (core affinity = 10)
Hello from rank 2, thread 3, on nid00014. (core affinity = 11)
Hello from rank 3, thread 0, on nid00014. (core affinity = 12)
Hello from rank 3, thread 1, on nid00014. (core affinity = 13)
Hello from rank 3, thread 2, on nid00014. (core affinity = 14)
Hello from rank 3, thread 3, on nid00014. (core affinity = 15)
Hello from rank 4, thread 0, on nid00015. (core affinity = 0)
Hello from rank 4, thread 1, on nid00015. (core affinity = 1)
Hello from rank 4, thread 2, on nid00015. (core affinity = 2)
Hello from rank 4, thread 3, on nid00015. (core affinity = 3)
Hello from rank 5, thread 0, on nid00015. (core affinity = 4)
Hello from rank 5, thread 1, on nid00015. (core affinity = 5)
Hello from rank 5, thread 2, on nid00015. (core affinity = 6)
Hello from rank 5, thread 3, on nid00015. (core affinity = 7)
Hello from rank 6, thread 0, on nid00015. (core affinity = 8)
Hello from rank 6, thread 1, on nid00015. (core affinity = 9)
Hello from rank 6, thread 2, on nid00015. (core affinity = 10)
Hello from rank 6, thread 3, on nid00015. (core affinity = 11)
Hello from rank 7, thread 0, on nid00015. (core affinity = 12)
Hello from rank 7, thread 1, on nid00015. (core affinity = 13)
Hello from rank 7, thread 2, on nid00015. (core affinity = 14)
Hello from rank 7, thread 3, on nid00015. (core affinity = 15)
```

9.10 Running a Batch Job Script

In this example, a batch job script requests six PEs to run program `mpi`.

Modules required:

```
pbs or moab
```

and one of the following:

```
PrgEnv-cray
PrgEnv-pgi
PrgEnv-gnu
PrgEnv-pathscale
PrgEnv-intel
```

Create `script1`:

```
#!/bin/bash
#
# Define the destination of this job
# as the queue named "workq":
#PBS -q workq
#PBS -l mppwidth=6
# Tell WMS to keep both standard output and
# standard error on the execution host:
#PBS -k eo
cd /lus/nid0008/user1
aprun -n 6 ./mpi
exit 0
```

Set permissions to executable:

```
% chmod +x script1
```

Submit the job:

```
% qsub script1
```

The `qsub` command produces a batch job log file with output from `mpi` (see Running an MPI Application on page 78). The job output is in a `script1.o`*nnnnn* file.

```
% cat script1.o238830 | sort
Application 848571 resources: utime ~0s, stime ~0s
 My PE:           0  My part:          816
 My PE:           1  My part:          833
 My PE:           2  My part:          850
 My PE:           3  My part:          867
 My PE:           4  My part:          884
 My PE:           5  My part:          800
    PE:           0 Total is:         5050
```

9.11 Running Multiple Sequential Applications

To run multiple sequential applications, the number of processors you specify as an argument to `qsub` must be equal to or greater than the **largest number** of processors required by a single invocation of `aprun` in your script. For example, in job script `mult_seq`, the `-l mppwidth` value is `6` because the largest `aprun` `n` value is `6`.

Modules required:

```
pbs or moab
```

and one of the following:

```
PrgEnv-cray
PrgEnv-pgi
PrgEnv-gnu
PrgEnv-pathscale
PrgEnv-intel
```

Create script `mult_seq`:

```
#!/bin/bash
#
# Define the destination of this job
# as the queue named "workq":
#PBS -q workq
#PBS -l mppwidth=6
# Tell WMS to keep both standard output and
# standard error on the execution host:
#PBS -k eo
cd /lus/nid000015/user1
aprun -n 2 ./simple
aprun -n 3 ./mpi
aprun -n 6 ./shmem_put
aprun -n 6 ./shmem_get
exit 0
```

The script launches applications `simple` (see Running a Basic Application on page 77), `mpi` (see Running an MPI Application on page 78), `shmem_put` (see Using the Cray `shmem_put` Function on page 79), and `shmem_get` (see Using the Cray `shmem_get` Function on page 81).

Set file permission to executable:

```
% chmod +x mult_seq
```

Run the script:

```
% qsub mult_seq
```

List the output:

```
% cat mult_seq.o465713
hello from pe 0 of 2
hello from pe 1 of 2
 My PE:           0  My part:         1683
 My PE:           1  My part:         1717
 My PE:           2  My part:         1650
  PE:           0 Total is:         5050
 PE 0: Test passed.
 PE 1: Test passed.
 PE 2: Test passed.
 PE 3: Test passed.
 PE 4: Test passed.
 PE 5: Test passed.
 PE            0  computedsum=    15.00000
 PE            1  computedsum=    15.00000
 PE            2  computedsum=    15.00000
 PE            3  computedsum=    15.00000
 PE            4  computedsum=    15.00000
 PE            5  computedsum=    15.00000
```

9.12 Running Multiple Parallel Applications

If you are running multiple parallel applications, the number of processors must be equal to or greater than the **total** number of processors specified by calls to `aprun`. For example, in job script `mult_par`, the `-l mppwidth` value is `11` because the total of the `aprun` `n` values is `11`.

Modules required:

```
pbs or moab
```

and one of the following:

```
PrgEnv-cray
PrgEnv-pgi
PrgEnv-gnu
PrgEnv-pathscale
PrgEnv-intel
```

Create mult_par:

```
#!/bin/bash
#
# Define the destination of this job
# as the queue named "workq":
#PBS -q workq
#PBS -l mppwidth=11
# Tell WMS to keep both standard output and
# standard error on the execution host:
#PBS -k eo
cd /lus/nid00007/user1
aprun -n 2 ./simple &
aprun -n 3 ./mpi &
aprun -n 6 ./shmem_put &
aprun -n 6 ./shmem_get &
wait
exit 0
```

The script launches applications simple (see Running a Basic Application on page 77), mpi (see Running an MPI Application on page 78), shmem_put (see Using the Cray shmem_put Function on page 79), and shmem_get (see Using the Cray shmem_get Function on page 81).

Set file permission to executable:

```
% chmod +x mult_par
```

Run the script:

```
% qsub mult_par
```

List the output:

```
% cat mult_par.o7231
hello from pe 0 of 2
hello from pe 1 of 2
Application 520255 resources: utime ~0s, stime ~0s
 My PE:           0  My part:         1683
 My PE:           2  My part:         1650
 My PE:           1  My part:         1717
    PE:           0 Total is:         5050
Application 520256 resources: utime ~0s, stime ~0s
 PE 0: Test passed.
 PE 5: Test passed.
 PE 4: Test passed.
 PE 3: Test passed.
 PE 2: Test passed.
 PE 1: Test passed.
Application 520258 exit codes: 64
Application 520258 resources: utime ~0s, stime ~0s
 PE            0  computedsum=    15.00000
 PE            5  computedsum=    15.00000
 PE            4  computedsum=    15.00000
 PE            3  computedsum=    15.00000
 PE            2  computedsum=    15.00000
 PE            1  computedsum=    15.00000
Application 520259 resources: utime ~0s, stime ~0s
```

9.13 Using `aprun` Memory Affinity Options

In some cases, remote-NUMA-node memory references can reduce the performance of applications. You can use the `aprun` memory affinity options to control remote-NUMA-node memory references. For the `-S`, `-sl`, and `-sn` options, memory allocation is satisfied using local-NUMA-node memory. If there is not enough NUMA node 0 memory, NUMA node 1 memory may be used. For the `-ss`, only local-NUMA-node memory can be allocated.

9.13.1 Using the `aprun -S` Option

This example runs each PE on a specific NUMA node 0 CPU:

```
% aprun -n 4 ./xthi | sort
Application 225110 resources: utime ~0s, stime ~0s
PE 0 nid00045 Core affinity = 0
PE 1 nid00045 Core affinity = 1
PE 2 nid00045 Core affinity = 2
PE 3 nid00045 Core affinity = 3
```

This example runs one PE on each NUMA node of nodes `45` and `70`:

```
% aprun -n 4 -S 1 ./xthi | sort
Application 225111 resources: utime ~0s, stime ~0s
PE 0 nid00045 Core affinity = 0
PE 1 nid00045 Core affinity = 4
PE 2 nid00070 Core affinity = 0
PE 3 nid00070 Core affinity = 4
```

9.13.2 Using the `aprun -sl` Option

This example runs all PEs on NUMA node 1:

```
% aprun -n 4 -sl 1 ./xthi | sort
Application 57967 resources: utime ~1s, stime ~1s
Hello from rank 0, thread 0, on nid00014. (core affinity = 4)
Hello from rank 1, thread 0, on nid00014. (core affinity = 5)
Hello from rank 2, thread 0, on nid00014. (core affinity = 6)
Hello from rank 3, thread 0, on nid00014. (core affinity = 7)
```

This example runs all PEs on NUMA node 2:

```
% aprun -n 4 -sl 2 ./xthi | sort
Application 57968 resources: utime ~1s, stime ~1s
Hello from rank 0, thread 0, on nid00014. (core affinity = 8)
Hello from rank 1, thread 0, on nid00014. (core affinity = 9)
Hello from rank 2, thread 0, on nid00014. (core affinity = 10)
Hello from rank 3, thread 0, on nid00014. (core affinity = 11)
```

9.13.3 Using the `aprun -sn` Option

This example runs four PEs on NUMA node 0 of node `45` and four PEs on NUMA node 0 of node `70`:

```
% aprun -n 8 -sn 1 ./xthi | sort
Application 2251114 resources: utime ~0s, stime ~0s
PE 0 nid00045 Core affinity = 0
PE 1 nid00045 Core affinity = 1
PE 2 nid00045 Core affinity = 2
PE 3 nid00045 Core affinity = 3
PE 4 nid00070 Core affinity = 0
PE 5 nid00070 Core affinity = 1
PE 6 nid00070 Core affinity = 2
PE 7 nid00070 Core affinity = 3
```

9.13.4 Using the `aprun -ss` Option

When `-ss` is specified, a PE can allocate only the memory that is local to its assigned NUMA node. The default is to allow remote-NUMA-node memory allocation. For example, by default any PE running on NUMA node 0 can allocate NUMA node 1 memory (if NUMA node 1 has been reserved for the application).

This example runs PEs 0-3 on NUMA node 0, PEs 4-7 on NUMA node 1, PEs 8-11 on NUMA node 2, and PEs 12-15 on NUMA node 3. PEs 0-3 cannot allocate NUMA node 1, 2, or 3 memories, PEs 4-7 cannot allocate NUMA node 0, 2, 3 memories, etc.

```
% aprun -n 16 -sl 0,1,2,3 -ss ./xthi | sort

Application 57970 resources: utime ~9s, stime ~2s
PE 0 nid00014. (core affinity = 0-3)
PE 10 nid00014. (core affinity = 8-11)
PE 11 nid00014. (core affinity = 8-11)
PE 12 nid00014. (core affinity = 12-15)
PE 13 nid00014. (core affinity = 12-15)
PE 14 nid00014. (core affinity = 12-15)
PE 15 nid00014. (core affinity = 12-15)
PE 1 nid00014. (core affinity = 0-3)
PE 2 nid00014. (core affinity = 0-3)
PE 3 nid00014. (core affinity = 0-3)
PE 4 nid00014. (core affinity = 4-7)
PE 5 nid00014. (core affinity = 4-7)
PE 6 nid00014. (core affinity = 4-7)
PE 7 nid00014. (core affinity = 4-7)
PE 8 nid00014. (core affinity = 8-11)
PE 9 nid00014. (core affinity = 8-11)
```

9.14 Using `aprun` CPU Affinity Options

The following examples show how you can use `aprun` CPU affinity options to bind a process to a particular CPU or the CPUs on a NUMA node.

9.14.1 Using the `aprun -cc` *cpu_list* Option

This example binds PEs to CPUs 0-4 and 7 on an 8-core node:

```
% aprun -n 6 -cc 0-4,7 ./xthi | sort
Application 225116 resources: utime ~0s, stime ~0s
PE 0 nid00045 Core affinity = 0
PE 1 nid00045 Core affinity = 1
PE 2 nid00045 Core affinity = 2
PE 3 nid00045 Core affinity = 3
PE 4 nid00045 Core affinity = 4
PE 5 nid00045 Core affinity = 7
```

9.14.2 Using the `aprun -cc` *keyword* Options

Processes can migrate from one CPU to another on a node. You can use the `-cc` option to bind PEs to CPUs. This example uses the `-cc cpu` (default) option to bind each PE to a CPU:

```
% aprun -n 8 -cc cpu ./xthi | sort
Application 225117 resources: utime ~0s, stime ~0s
PE 0 nid00045 Core affinity = 0
PE 1 nid00045 Core affinity = 1
PE 2 nid00045 Core affinity = 2
PE 3 nid00045 Core affinity = 3
PE 4 nid00045 Core affinity = 4
PE 5 nid00045 Core affinity = 5
PE 6 nid00045 Core affinity = 6
PE 7 nid00045 Core affinity = 7
```

This example uses the `-cc numa_node` option to bind each PE to the CPUs within a NUMA node:

```
% aprun -n 8 -cc numa_node ./xthi | sort
Application 225118 resources: utime ~0s, stime ~0s
PE 0 nid00045 Core affinity = 0-3
PE 1 nid00045 Core affinity = 0-3
PE 2 nid00045 Core affinity = 0-3
PE 3 nid00045 Core affinity = 0-3
PE 4 nid00045 Core affinity = 4-7
PE 5 nid00045 Core affinity = 4-7
PE 6 nid00045 Core affinity = 4-7
PE 7 nid00045 Core affinity = 4-7
```

9.15 Using Checkpoint/Restart Commands

To checkpoint and restart a job, first load these modules:

```
moab
blcr
```

This example shows the use of the `qhold` and `qchkpt` checkpoint commands and the `qrls` and `qrerun` restart commands.

Source code of `cr.c`:

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include "mpi.h"
#include <signal.h>
#include <stdlib.h>

static void sig_handler(int);

static unsigned int Cnt = 0;    /* Counter that is
incremented each time app is checkpointed. */

static int me;

int
main (int argc, char *argv[])
{
  int all, ret;
  int sleep_time=100000;
  ret = MPI_Init(&argc, &argv);
  ret = MPI_Comm_rank (MPI_COMM_WORLD, &me);
  ret = MPI_Comm_size(MPI_COMM_WORLD, &all);

  if (me == 0) {

    if (signal(SIGCONT, sig_handler) == SIG_ERR) {
      printf("Can't catch SIGCONT\n");
      ret = MPI_Finalize();
      exit(3);
    }
    printf ("Partition size is = %d\n", all);
  }

  ret = 999;
  while (ret != 0) {

    Cnt += 1;
    ret = sleep(sleep_time);
    if (ret != 0 ) {

      printf("PE %d PID %d interrupted at cnt: %d\n", me, getpid(), Cnt);
      sleep_time = ret;
    }
  }

  printf ("Finished with count at: &d, exiting \n", Cnt);
  ret = MPI_Finalize();
}


static void
sig_handler(int signo)
{
  printf("\n");

}
```

Compile `cr.c`:

```
% cc -o cr cr.c
```

Create script `cr_script`:

```
#!/usr/bin/ksh
#PBS -l mppwidth=2
#PBS -l mppnppn=1
#PBS -j oe
#PBS -l walltime=6:00:00

# cd to directory where job was submitted from:
cd /lus/nid00015/user12/c

export MPICH_VERSION_DISPLAY=1

aprun -n 2 -N 1 ./cr

wait;
```

Launch the job:

```
% qsub cr_script
87151.nid00003
```

The WMS returns the job identifier `87151.nid00003`. Use just the first part (sequence number `87151`) in checkpoint/restart commands.

Check the job status:

```
% qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
87151.nid00003            cr_script        user12          00:00:00 R workq
```

The job is running (`qstat` state `S` is `R`).

Check the status of application `cr`:

```
% apstat
Compute node summary
   arch config     up    use   held  avail   down
     XT     72     72      2      0     70      0

No pending applications are present

Total placed applications: 1
Placed  Apid ResId     User   PEs Nodes    Age   State Command
      331897     6   user12     2     2   0h03m  run   cr
```

The application is running (`State` is `run`).

Checkpoint the job, place it in hold state, and recheck job and application status:

```
% qhold 87151
% qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
87151.nid00003            cr_script        user12          00:00:00 H workq
% apstat
Compute node summary
   arch config     up    use   held  avail   down
     XT     72     72      0      0     72      0

No pending applications are present

No placed applications are present
```

The job is checkpointed and its state changes from run to hold. Application `cr` is checkpointed (`apstat` `State` field is `chkpt`), then stops running.

Note: The `qhold` command checkpointed the job because it was submitted with the `-c enabled` option.

Release the job, get status to verify, then restart it:

```
% qrls 87151
% qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
87151.nid00003            cr_script        user12          00:00:00 R workq
% apstat
Compute node summary
   arch config     up    use   held  avail   down
     XT     72     72      2      0     70      0

No pending applications are present

Total placed applications: 1
Placed  Apid ResId     User   PEs Nodes    Age   State Command
      331899     7   user12     2     2   0h00m  run   cr
```

The job is running (`qstat` `S` field is `R` and application `State` is `run`).

Checkpoint the job but keep it running:

```
% qchkpt 87151
% qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
87151.nid00003            cr_script        user12          00:00:00 R workq
% apstat
Compute node summary
   arch config     up    use   held  avail   down
     XT     72     72      2      0     70      0

No pending applications are present

Total placed applications: 1
Placed  Apid ResId     User   PEs Nodes    Age   State Command
      331899     7   user12     2     2   0h02m  run   cr
```

The `qstat` `S` field changed to `R`, and the application state changed from `chkpt` to `run`.

Use `qdel` to stop the job:

```
% qdel 87151
% qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
87151.nid00003            cr_script        user12          00:00:00 C workq
```

Use the `qrerun` command to restart a completed job previously checkpointed:

```
% qrerun 87151
% qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
87151.nid00003            cr_script        user12          00:00:00 R workq
% apstat
Compute node summary
   arch config     up    use   held  avail   down
     XT     72     72      2      0     70      0

No pending applications are present

Total placed applications: 1
Placed  Apid ResId     User   PEs Nodes    Age   State Command
      331901     8   user12     2     2   0h00m  run   cr
```

You can use `qrerun` to restart a job if the job remains queued in the completed state.

At any step in the checkpoint/restart process, you can use the `qstat -f` option to displays details about the job and checkpoint files:

```
% qstat -f 87151
Job Id: 87151.nid00003
    Job_Name = cr_script
    Job_Owner = user12@nid00004
<snip>
    Checkpoint = enabled
<snip>
    comment = Job 87151.nid00003 was checkpointed and continued to /lus/scratc
        h/BLCR_checkpoint_dir/ckpt.87151.nid00003.1237761585 at Sun Mar 22 17:
        39:45 2009

<snip>
    checkpoint_dir = /lus/scratch/BLCR_checkpoint_dir
    checkpoint_name = ckpt.87151.nid00003.1237761585
    checkpoint_time = Sun Mar 22 17:39:45 2009
    checkpoint_restart_status = Successfully restarted job
```

You can get details about the checkpointed files in `checkpoint_dir`:

```
% cd /lus/scratch/BLCR_checkpoint_dir
% ls -al
<snip>
drwx------  3 user12 dev1   4096 2009-03-22 17:35 ckpt.87151.nid00003.1237761347
drwx------  3 user12 dev1   4096 2009-03-22 17:39 ckpt.87151.nid00003.123776158
% cd ckpt.87151.nid00003.123776158
% ls
331899  cpr.context  info.7828
% cd 331899
% ls
context.0  context.1
```

There is a `context.`*n* file for each width value (`-l mppwidth=2`).

9.16 Running Compute Node Commands

You can use the `aprun -b` option to run compute node BusyBox commands.

The following `aprun` command runs the compute node `grep` command to find references to `MemTotal` in compute node file `/proc/meminfo`:

```
% aprun -b grep MemTotal /proc/meminfo
MemTotal:      8124872 kB
```

9.17 Using the High-level PAPI Interface

PAPI provides simple high-level interfaces for instrumenting applications written in C or Fortran. This example shows the use of the `PAPI_start_counters()` and `PAPI_stop_counters()` functions.

Modules required:

```
xti-papi
```

and one of the following:

```
PrgEnv-cray
PrgEnv-pgi
PrgEnv-gnu
PrgEnv-pathscale
PrgEnv-intel
```

Source of `papi_hl.c`:

```
#include <papi.h>
void main()
{

  int retval, Events[2]= {PAPI_TOT_CYC, PAPI_TOT_INS};
  long_long values[2];

  if (PAPI_start_counters (Events, 2) != PAPI_OK) {
    printf("Error starting counters\n");
    exit(1);
  }

  /* Do some computation here... */

  if (PAPI_stop_counters (values, 2) != PAPI_OK) {
    printf("Error stopping counters\n");
    exit(1);
  }

  printf("PAPI_TOT_CYC = %lld\n", values[0]);
  printf("PAPI_TOT_INS = %lld\n", values[1]);
}
```

Compile `papi_hl.c`:

```
% cc -o papi_hl papi_hl.c
```

Run `papi_hl`:

```
% aprun ./papi_hl
PAPI_TOT_CYC = 4020
PAPI_TOT_INS = 201
Application 520262 exit codes: 19
Application 520262 resources: utime ~0s, stime ~0s
```

9.18 Using the Low-level PAPI Interface

PAPI provides an advanced low-level interface for instrumenting applications. Initialize the PAPI library before calling any of these functions by issuing either a high-level function call or a call to `PAPI_library_init()`. This example shows the use of the `PAPI_create_eventset()`, `PAPI_add_event()`, `PAPI_start()`, and `PAPI_read()` functions.

Modules required:

```
xti-papi
```

and one of the following:

```
PrgEnv-cray
PrgEnv-pgi
PrgEnv-gnu
PrgEnv-pathscale
PrgEnv-intel
```

Source of `papi_ll.c`:

```
#include <papi.h>
void main()
{
  int EventSet = PAPI_NULL;
  long_long values[1];

  /* Initialize PAPI library */
  if (PAPI_library_init(PAPI_VER_CURRENT) != PAPI_VER_CURRENT) {
    printf("Error initializing PAPI library\n");
    exit(1);
  }

  /* Create Event Set */
  if (PAPI_create_eventset(&EventSet) != PAPI_OK) {
    printf("Error creating eventset\n");
  exit(1);
  }

  /* Add Total Instructions Executed to eventset */
  if (PAPI_add_event (EventSet, PAPI_TOT_INS) != PAPI_OK) {
    printf("Error adding event\n");
    exit(1);
  }

  /* Start counting ... */
  if (PAPI_start (EventSet) != PAPI_OK) {
    printf("Error starting counts\n");
    exit(1);
  }

  /* Do some computation here...*/

  if (PAPI_read (EventSet, values) != PAPI_OK) {
    printf("Error stopping counts\n");
    exit(1);
  }
  printf("PAPI_TOT_INS = %lld\n", values[0]);
}
```

Compile `papi_ll.c`:

```
% cc -o papi_ll papi_ll.c
```

Run `papi_ll`:

```
% aprun ./papi_ll
PAPI_TOT_INS = 97
Application 520264 exit codes: 18
Application 520264 resources: utime ~0s, stime ~0s
```

9.19 Using CrayPat

This example shows how to instrument a program, run the instrumented program, and generate CrayPat reports.

Modules required:

```
perftools
```

and one of the following:

```
PrgEnv-cray
PrgEnv-pgi
PrgEnv-gnu
PrgEnv-pathscale
PrgEnv-intel
```

Source code of `pa1.f90`:

```
program main
include 'mpif.h'

  call MPI_Init(ierr)     ! Required
  call MPI_Comm_rank(MPI_COMM_WORLD,mype,ierr)
  call MPI_Comm_size(MPI_COMM_WORLD,npes,ierr)

  print *,'hello from pe',mype,' of',npes

  do i=1+mype,1000,npes   ! Distribute the work
    call work(i,mype)
  enddo

  call MPI_Finalize(ierr) ! Required
end
```

Source code of `pa2.c`:

```
void work_(int *N, int *MYPE)
{
  int n=*N, mype=*MYPE;

  if (n == 42) {
    printf("PE %d: sizeof(long) = %d\n",mype,sizeof(long));
    printf("PE %d: The answer is: %d\n",mype,n);
  }
}
```

Compile `pa2.c` and `pa1.f90` and create executable `perf`:

```
% cc -c pa2.c
% ftn -o perf pa1.f90 pa2.o
```

Run `pat_build` to generate instrumented program `perf+pat`:

```
% pat_build -u -g mpi perf perf+pat
INFO: A trace intercept routine was created for the function 'MAIN_'.
INFO: A trace intercept routine was created for the function 'work_'.
```

The tracegroup (`-g` option) is `mpi`.

Run `perf+pat`:

```
% aprun -n 4 ./perf+pat | sort
CrayPat/X:  Version 5.0 Revision 2635  06/04/09 03:13:22
Experiment data file written:
/mnt/lustre_server/user12/perf+pat+1652-30tdt.xf
Application 582809 resources: utime ~0s, stime ~0s
 hello from pe            0  of            4
 hello from pe            1  of            4
 hello from pe            2  of            4
 hello from pe            3  of            4
PE 1: sizeof(long) = 8
PE 1: The answer is: 42
```

Note: When executed, the instrumented executable creates directory *progname*+`pat`+*PIDkeyletters*, where . *PID* is the process ID that was assigned to the instrumented program at run time.

Run `pat_report` to generate reports `perf.rpt1` (using default `pat_report` options) and `perf.rpt2` (using the `-O calltree` option).

```
% pat_report perf+pat+1652-30tdt.xf > perf.rpt1
pat_report:  Creating file:   perf+pat+1652-30tdt.ap2
Data file 1/1: [....................]
% pat_report -O calltree perf+pat+1652-30tdt.xf > perf.rpt2
pat_report:  Using existing file:   perf+pat+1652-30tdt.ap2
Data file 1/1: [....................]
% pat_report -O calltree -f ap2 perf+pat+1652-30tdt.xf
Output redirected to:  perf+pat+1652-30tdt.ap2
```

Note: The `-f ap2` option is used to create a `*.ap2` file for input to Cray Apprentice2 (see Using Cray Apprentice2 on page 117).

List `perf.rpt1`:

```
CrayPat/X:  Version 5.0 Revision 2635 (xf 2571)  06/04/09 03:13:22

Number of PEs (MPI ranks):      4

Number of Threads per PE:       1

Number of Cores per Processor:  4

<snip>


Table 1:  Profile by Function Group and Function

 Time % |      Time |Imb. Time |   Imb. | Calls |Group
        |           |          | Time % |       | Function
        |          |          |        |       |  PE='HIDE'

 100.0% | 0.000151 |       -- |     -- | 257.0 |Total
|-----------------------------------------------------------
|  98.9% | 0.000150 |       -- |     -- | 253.0 |USER
||----------------------------------------------------------
||  81.0% | 0.000122 | 0.000002 |   2.3% |   1.0 |MAIN_
||  14.5% | 0.000022 | 0.000001 |   4.8% |   1.0 |exit
||   2.1% | 0.000003 | 0.000001 |  20.1% |   1.0 |main
||   1.2% | 0.000002 | 0.000000 |  10.2% | 250.0 |work_
||==========================================================
|   1.1% | 0.000002 |       -- |     -- |   4.0 |MPI
|===========================================================

<snip>

Table 2:  Load Balance with MPI Message Stats

 Time % |     Time |Group
        |          | PE

 100.0% | 0.000189 |Total
|------------------------
|  98.6% | 0.000186 |USER
||-----------------------
||  25.5% | 0.000193 |pe.1
||  24.7% | 0.000187 |pe.0
||  24.3% | 0.000183 |pe.2
||  24.1% | 0.000182 |pe.3
||======================
|   1.4% | 0.000003 |MPI
||-----------------------
||   0.4% | 0.000003 |pe.1
||   0.4% | 0.000003 |pe.2
||   0.3% | 0.000003 |pe.0
||   0.3% | 0.000003 |pe.3
|========================

<snip>

Table 5:  Program Wall Clock Time, Memory High Water Mark

  Process |  Process |PE
     Time |    HiMem |
          | (MBytes) |

 0.033981 |       20 |Total
|-----------------------
| 0.034040 |   19.742 |pe.2
| 0.034023 |   19.750 |pe.3
| 0.034010 |   19.754 |pe.0
| 0.033851 |   19.750 |pe.1
|=======================

=========  Additional details ============================

Experiment:  trace
```

```
<snip>

Estimated minimum overhead per call of a traced function,
  which was subtracted from the data shown in this report
  (for raw data, use the option:  -s overhead=include):
    Time    0.241  microseconds

Number of traced functions: 102
  (To see the list, specify:  -s traced_functions=show)
```

List `perf.rpt2`:

```
CrayPat/X:  Version 5.0 Revision 2635 (xf 2571)  06/04/09 03:13:22

Number of PEs (MPI ranks):      4

Number of Threads per PE:       1

Number of Cores per Processor:  4

<snip>

Table 1:  Function Calltree View

 Time % |     Time | Calls |Calltree
        |          |       | PE='HIDE'

 100.0% | 0.000181 | 657.0 |Total
|-------------------------------------
|  69.7% | 0.000126 | 255.0 |MAIN_
||------------------------------------
||  67.7% | 0.000122 |   1.0 |MAIN_(exclusive)
||   1.0% | 0.000002 | 250.0 |work_
||====================================
|  12.2% | 0.000022 |   1.0 |exit
|   1.8% | 0.000003 |   1.0 |main
|=====================================

=========  Additional details ============================

Experiment:  trace

<snip>

Estimated minimum overhead per call of a traced function,
  which was subtracted from the data shown in this report
  (for raw data, use the option:  -s overhead=include):
    Time    0.241  microseconds

Number of traced functions: 102
  (To see the list, specify:  -s traced_functions=show)
```

9.20 Using Cray Apprentice2

In the CrayPat example (Using CrayPat on page 114), we ran the instrumented program `perf` and generated file `perf+pat+1652-30tdt.ap2`.

To view this Cray Apprentice2 file, first load the `perftools` module.

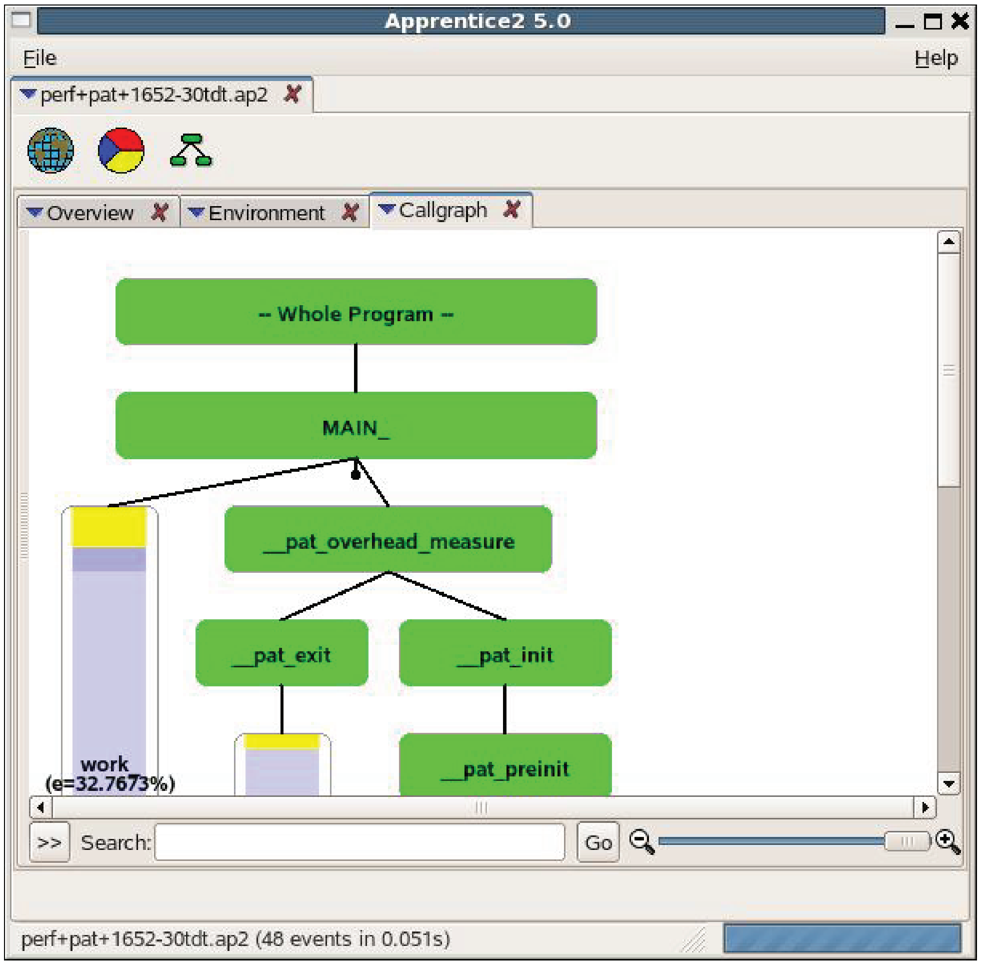
```
% module load perftools
```

Then launch Cray Apprentice2:

```
% app2 perf+pat+1652-30tdt.ap2
```

Display the results in call-graph form:

Figure 9. Cray Apprentice2 Callgraph



Further Information [A]

A.1 Related Publications

Cray systems run with a combination of Cray proprietary, third-party, and open source products, as documented in the following publications.

A.1.1 Publications for Application Developers

- *Cray Application Developer's Environment User's Guide*
- *Cray Application Developer's Environment Installation Guide*
- *Cray Linux Environment (CLE) Software Release Overview*
- *Cray C and C++ Reference Manual*
- *Cray Fortran Reference Manual*
- Cray compiler command options man pages: `craycc`(1), `crayftn`(1)
- *PGI User's Guide*
- *PGI Tools Guide*
- *PGI Fortran Reference*
- PGI compiler command options man pages: `pgcc`(1), `pgCC`(1), `pgf95`(1)
- GCC manuals: http://gcc.gnu.org/onlinedocs/
- GCC compiler command options man pages: `gcc`(1), `g++`(1), `gfortran`(1)
- PathScale manuals: http://www.pathscale.com/docs.html
- PathScale compiler command options man pages: `pathcc`(1), `pathCC`(1), `path95`(1), `eko`(7)
- Cray compiler driver commands man pages: `cc`(1), `CC`(1), `ftn`(1)
- Modules utility man pages: `module`(1), `modulefile`(4)
- Application launch command man page: `aprun`(1)

- Parallel programming models:
  - Cray MPICH2 man pages (read the `intro_mpi`(3) man page first)
  - Cray SHMEM man pages (read the `intro_shmem`(3) man page first)
  - OpenMP documentation: http://www.openmp.org/
  - Cray UPC man pages (read the `intro_upc`(3c) man page first)

    Unified Parallel C (UPC) documents: Berkeley UPC website (http://upc.lbl.gov/docs/).
- Cray scientific library, XT-LibSci, documentation:
  - Basic Linear Algebra Subroutines (BLAS) man pages
  - LAPACK linear algebra man pages
  - ScaLAPACK parallel linear algebra man pages
  - Basic Linear Algebra Communication Subprograms (BLACS) man pages
  - Iterative Refinement Toolkit (IRT) man pages (read the `intro_irt`(3) man page first)
  - SuperLU sparse solver routines guide (*SuperLU Users' Guide*)
- *AMD Core Math Library (ACML)* manual
- FFTW 2.1.5 and 3.1.1 man pages (read the `intro_fftw2`(3) or `intro_fftw3`(3) man page first)
- Portable, Extensible Toolkit for Scientific Computation (PETSc) library, an open source library of sparse solvers. See the `intro_petsc`(3) man page and http://www-unix.mcs.anl.gov/petsc/petsc-as/index.html
- NetCDF documentation (http://www.unidata.ucar.edu/software/netcdf/)
- HDF5 documentation (http://www.hdfgroup.org/HDF5/whatishdf5.html)
- Lustre `lfs`(1) man page
- *PBS Professional 9.0 User's Guide*
- PBS Professional man pages (`qsub`(1B) , `qstat`(1B), and `qdel`(1B))
- Moab and TORQUE documentation (http://www.clusterresources.com/)
- TotalView documentation (http://www.roguewave.com)
- GNU debugger documentation (see the `lgdb`(1) man page and the *GDB User Manual* at http://www.gnu.org/software/gdb/documentation/).
- PAPI man pages (read the `intro_papi`(3) man page first)

- PAPI manuals (see http://icl.cs.utk.edu/papi/)
- *Using Cray Performance Measurement and Analysis Tools*
- CrayPat man pages (read the `intro_craypat`(1) man page first)
- Cray Apprentice2 man page (`app2`(1))
- CLE man pages
- SUSE Linux man pages
- Linux documentation (see the Linux Documentation Project at http://www.tldp.org and SUSE documentation at http://www.suse.com)