CLE User Application Placement Guide

S-2496-5204

Contents

About CLE User Application Placement in CLE

CLE User Application Placement in CLE is the replacement publication for *Workload Management and Application Placement for the Cray Linux Environment*. The title was changed to more accurately reflect the content, which no longer includes WLM specific information. Also, it has a new format that allows it to be easily published as a PDF and also on the new Cray Publication Portal: *http://pubs.cray.com*.

Scope and Audience

This publication is intended for Cray system programmers and users.

Release Information

This publication includes user application placement information for Cray software release CLE5.2.UP04 and supports Cray XE, Cray XK, and Cray XC Series systems. Changes included in this document include:

Added Information

- Added the error message "Fatal error detected by libibgni.so" and explanation to the *Troubleshooting IAA* on page 39 section.

Revised Information

- ISV Application Acceleration (IAA) now supports up to 4096 processing elements per application.
- Modified the suggestions for running Intel-compiled code in the *Run an OpenMP Application* on page 65 example.
- An updated example was placed in *Optimize Process Placement on Multicore Nodes* on page 45.
- The example in *Run an Accelerated Cray LibSci Routine* on page 55 was corrected to call `libsci_acc_HostFree()`.

Deleted Information

- Specific details on creating a job script for any particular workload manager have been deleted.
- Removed `-cp` as an `aprun` CPU affinity option.
- Removed all references to the PathScale compiler, which is no longer supported.

Typographic Conventions

`Monospace`	`Monospaced` text indicates program code, reserved words, library functions, command-line prompts, screen output, file names, path names, and other software constructs.
`Monospaced Bold`	**`Bold monospaced`** text indicates commands that must be entered on a command line or in response to an interactive prompt.

Oblique or *Italics*	*Oblique* or *italicized* text indicates user-supplied values in commands or syntax definitions.
Proportional Bold	**Proportional bold** text indicates a graphical user interface window or element.
\ (backslash)	A backslash at the end of a command line is the Linux® shell line continuation character; the shell parses lines joined by a backslash as though they were a single line. Do not type anything after the backslash or the continuation feature will not work correctly.
`Alt-Ctrl-f`	`Monospaced` hyphenated text typically indicates a keyboard combination.

Feedback

Please provide feedback by visiting *http://pubs.cray.com* and clicking the *Contact Us* button in the upper-right corner, or by sending email to *pubs@cray.com*.

Run Applications Using the aprun Command

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

Use the `aprun` command to specify required resources, request application placement, and initiate application launch. The basic format of the `aprun` command is as follows:

```
aprun [global_options] [command_options] cmd1 [: [command_options] cmd2
[: ...] ] [--help] [--version]
```

Use the colon character (`:`) to separate the different options for separate binaries when running in MPMD (Multiple Program Multiple Data) mode. Use the `--help` option to display detailed `aprun` command line usage information. Use the `--version` option to display the ALPS version information.

The `aprun` command supports two general sets of arguments: global options and command options. The global options apply to the execution command line as a whole and are as follows:

`-b | --bypass-app-transfer`	Bypass application transfer to compute node
`-B | --batch-args`	Get values from Batch reservation for `-n`, `-N`, `-d`, and `-m`
`-C | --reconnect`	Reconnect fanout control tree around failed nodes
`-D | --debug level`	Debug level bitmask (0-7)
`-e | --environment-override` ***`env`***	Set an environment variable on the compute nodes Must use format *`VARNAME=value`* Set multiple environment variables using multiple `-e` arguments
`-m | --memory-per-pe` ***`size`***	Per PE memory limit in megabytes (default node memory/number of processors) K|M|G suffix supported (16 == 16M == 16 megabytes) Add an 'h' suffix to request per PE huge page memory Add an 's' to the 'h' suffix to make the per PE huge page memory size strict (required)
`-P | --pipes` ***`pipes`***	Write[,read] pipes (not applicable for general use)
`-p | --protection-domain` ***`pdi`***	Protection domain identifier
`-q, --quiet`	Quiet mode; suppress `aprun` non-fatal messages
`-R | --relaunch` ***`max_shrink`***	Relaunch application; *`max_shrink`* is zero or more maximum PEs to shrink for a relaunch
`-T, --sync-output`	Use synchronous TTY

-t, --cpu-time-limit *sec*	Per PE CPU time limit in seconds (default unlimited)

The command options apply to individual binaries and can be set differently for each binary when operating in MPMD mode. The command options are as follows:

-a | --architecture *arch*	Architecture type
--cc | --cpu-binding *cpu_list*	CPU binding list or keyword ([cpu#[,cpu# | cpu1-cpu2] | x]...] | keyword)
--cp | --cpu-binding-file *file*	CPU binding placement filename
-d | --cpus-per-pe *depth*	Number of CPUs allocated per PE (number of threads)
-E | --exclude-node-list *node_list*	List of nodes to exclude from placement
--exclude-node-list-file *node_list_file*	File with a list of nodes to exclude from placement
-F | --access-mode *flag*	Exclusive or share node resources flag
-j | --cpus-per-cu *CPUs*	CPUs to use per Compute Unit (CU)
-k | --knc-autonomous	Place application for execution on Xeon Phi (KNC - Cray XC Series only)
-L | --node-list *node_list*	Manual placement list (node[,node | node1-node2]...)
-l | --node-list-file *node_list_file*	File with manual placement list
-N | --pes-per-node *pes*	PEs per node
-n | --pes *width*	Number of PEs requested
--p-governor *governor_name*	Specify application performance governor
--p-state *pstate*	Specify application p-state in kHz
-r | --specialized-cpus *CPUs*	Restrict this many CPUs per node to specialization
-S | --pes-per-numa-node *pes*	PEs per NUMA node
--sl | --numa-node-list *numa_node_list*	List of NUMA nodes to use (numa_node[,numa_node | numa_node1-numa_node2]...)
--sn | --numa-nodes-per-node *numa_nodes*	Maximum number of NUMA nodes used on compute node
--ss | --strict-memory-containment	Strict memory containment per NUMA node

In more detail, the `aprun` options are as follows:

-a arch

Specifies the architecture type of the compute node on which the application will run; *arch* is `xt`. If using `aprun` to launch a compiled and linked executable, do not include the `-a` option; ALPS can determine the compute node architecture type from the ELF header (see the `elf(5)` man page).

-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.

-B

Reuses the `width`, `depth`, `nppn`, and `memory` request options that were 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, the 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 *The aprun CPU Affinity Option* on page 44.

The *`cpu_list`* is a comma-separated list of logical CPU numbers and/or hyphen-separated CPU ranges. It controls the cpu affinity of each PE task, and each descendent thread or task of each PE, as they are created. (Collectively, "app tasks".) Upon the creation of each app task, it is bound to the CPU in *`cpu_list`* corresponding to the number of app tasks 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`* (assuming the first PE has not created any children or threads first). If more app tasks are created than given in *`cpu_list`*, binding starts over at the beginning of *`cpu_list`*.

Instead of a CPU number, an `x` may be specified in any position or positions in *`cpu_list`*. The app task that corresponds to this position will not be bound to any CPU.

The above behavior can result in undesireable and/or unpredictable behavior when more than one PE on a node creates children or threads without synchronizing between themselves. Because the app tasks are bound to CPUs in *`cpu_list`* in the order in which they are created, unpredictable creation order will lead to unpredictable binding. To prevent this, the user may specify a *`cpu_list`* per PE. Multiple *`cpu_list`*s are separated by colons (:).

```
% 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, to bind PEs starting with the highest CPU on a compute node and work down from there, 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.

Instead of a *`cpu_list`*, the argument to the `-cc` option may be one of the following keywords:

- The `cpu` keyword (the default) binds each PE to a CPU within the assigned NUMA node. Indicating a specific CPU is not necessary.

- If a *depth* per PE (`aprun -d depth`) is specified, 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.

  TIP: If CPUs are oversubscribed for an OpenMP application, Cray recommends not using 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 `depth` keyword can improve MPI rank and thread placement on Cray XC30 nodes by assigning to a PE and its children a cpumask with `-d` (depth) bits set. If the `-j` option is also used, only `-j` PEs will be assigned per compute unit.
- 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, when 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 an 8-core XK node where CPUs 0-3 are on NUMA node 0 and CPUs 4-7 are on NUMA node 1, 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.

-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 *Run an OpenMP Application* on page 65.

The maximum permissible *depth* value depends on the types of CPUs installed on the Cray system.

-e env

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

-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 `alps.conf` then use the `-F exclusive` to override the policy set in this file. Check the value of `nodeShare` by executing `apstat -svv | grep access`.

-j *num_cpus*

Specifies how many CPUs to use per compute unit for an ALPS job. For more information on compute unit affinity, see Using Compute Unit Affinity on Cray Systems (S-0030).

-k

(Cray XC Series only) Indicates that the application should be placed for execution on an Intel Xeon Phi co-processor.

Note that executable programs must be built specially for execution on a Xeon Phi. Otherwise, the following message will occur when attempting to run a program that was not built specially for Xeon Phi: `aprun: Binary not built for Xeon Phi. Cross-compile your application or use -b to run a command.` Commands which are already present on the Xeon Phi may be run by adding the `-b` switch to bypass copying the binary to the compute node.

Attempting to place an Xeon Phi application on a node that does not have a Xeon Phi coprocessor will cause the following message: `apsched: in user NIDs request exceeds max resources`

-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 hyphen-separated 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.

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.

The `cnselect` command is a common source of node lists. 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 the `-m` option is not included, the default amount of memory available to each PE equals (compute node memory size) / (number of PEs) calculated for each compute node.

Use the `h` or `hs` suffix to allocate huge pages (2 MB) for an application.

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.

-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.

To use huge pages, 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
```

See the `intro_hugepages(1)` man page for further details.

-n *pes*

Specifies the number of processing elements (PEs) that the application requires. A PE is an instance of an ALPS-launched executable. 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 cores on the node.

-p *protection domain identifier*

Requests use of a protection domain using the user pre-allocated protection identifier. If protection domains are already allocated by system services, this option cannot be used. 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.

--p-governor *governor_name*

`--p-governor` sets a performance governor on compute nodes used by the application. Choices are `performance`, `powersave`, `userspace`, `ondemand`, `conservative`. See `/usr/src/linux/Documentation/cpu-freq/governors.txt` for details. `--p-governor` cannot by used with `--p-state`.

--p-state *pstate*

Specifies the CPU frequency used by the compute node kernel while running the application. `--p-state` cannot be used with `--p-governor`.

-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*

When *cores* > `0`, core specialization is enabled. On each compute node, *cores* CPUs will be dedicated to system tasks, and system tasks will not run on the CPUs on which the the application is placed.

Whenever core specialization is enabled, the highest-numbered CPU will be used as one of these system CPUs. When *cores* > `1`, the additional system CPUs will be chosen from the CPUs not selected for the application by the usual affinity options.

It is an error to specify *cores* > `0` and to include the highest-numbered CPU in the `-cc` *cpu_list* option. It is an error to specify *cores* and a `-cc` *cpu_list* where the number of CPUs in the *cpu_list* plus *cores* is greater than the number of CPUs on the node.

-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` attempts 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` attempts 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. Use this option to reduce the number of PEs per NUMA node, thereby making more resources (such as memory) available per PE.

The allowable values for this option vary depending on the types of CPUs installed on the system. A zero value is not allowed and causes a fatal error. For further information, see *The aprun Memory Affinity Options* on page 44.

-sl *list_of_numa_nodes*

Specifies the NUMA node or nodes (comma- or hyphen-separated list) to use for application placement. A space is required between `-sl` and *list_of_numa_nodes*.

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. A space is required between `-sn` and *numa_nodes_per_node*.

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

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.

Additional aprun Information

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.

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.

aprun Resource Limits

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

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

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.

Reserved File Descriptors

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

ALPS Environment Variables

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

APRUN_DEFAULT_MEMORY Specifies the 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_MPS Overrides the site default for execution in simultaneous contexts on GPU-equipped nodes. Setting `CRAY_CUDA_MPS` 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.

How Application Placement Works

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 16 ./a.out
```

places 16 PEs on:

- one Cray XC30 compute node
- one Cray XE6 dual-socket, 16-core compute node
- two Cray XK7 single-socket, 12-core compute nodes

  Note that 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 memory and CPU affinity options are **optimization** options, not placement options. Use memory affinity options if there is a concern that remote-NUMA-node memory references are reducing performance. Use CPU affinity options if there is a concern that process migration is reducing performance.

For examples showing how to use memory affinity options, see *Use aprun Memory Affinity Options* on page 71. For examples showing how to use CPU affinity options, see *Use aprun CPU Affinity Options* on page 73.

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:

- (Cray XE/XK systems) 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.
- (Cray XE/XK systems) 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 *Run Applications Using the aprun Command* on page 6.
- (Cray XC Series systems) Two 16-bit Protection Keys (pKeys) are allocated instead of one pTag, and the pKey values are part of the cookie.

- 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 (2 nodes in Gemini, 4 nodes in Aries). Other applications may run on the ASIC if they do not need performance counters.

  (Cray XC Series systems) The Aries ASIC network performance counters operate as described in Using the Aries Hardware Counters (S-0045) and the `nwpc(5)` man page.

Display Application Status and Cray System Information using apstat

The `apstat` utility produces tables of status information about the Cray system, the computing resources on it, and pending and placed applications. The `apstat` utility does not report dynamic run time information about any given application, so the display does not imply anything about the running state of an application. The `apstat` display merely shows that an application has been placed, and that the `aprun` claim against the reserved resources has not yet been released.

The `apstat` command syntax is as follows.

```
apstat [OPTION...]
```

The `apstat` command supports the following options.

Table 1. Table Output

Option	Action
`-a, --applications`	Displays applications
`-C, --cleanup`	Displays cleanup information
`-G, --accelerators`	Displays GPU accelerators
`-K, --co-processors`	Displays Xeon Phi co-processors
`-n, --nodes`	Displays nodes
`-p, --pending-applications`	Displays pending applications
`-P, --protection-domains`	Displays protection domains
`-r, --reservations`	Displays reservations

Table 2. Node Table Options (with -n)

Option	Action
`-c, --cpus-per-node`	In Compute Node Summary, break down architecture column by CPUs per node
`--no-summary`	Do not print the node summary table
`-o, --placement-order`	Do not force sorted NIDs in output
`-z, --zeros`	Replace '-' with '0' in cells

Table 3. Common Options

Option	Action
`-A, --apid-list` *`apid_list`*	Filter all table rows unrelated to the given space-separated list of apids. Must be the final argument provided.
`-f, --format` *`format`*	Specifies columns to output in a table as a comma separated list. For example, 'apstat -n -f 'NID,HW,PEs'
`--no-headers`	Don't print table headers
`-N, --node-list` *`node_list`*	Filter all table rows unrelated to the given space-separated list of nodes. Must be the final argument provided.
`-R, --resid-list` *`resid_list`*	Filter all table rows unrelated to the given space-separated list of resids. Must be the final argument provided.
`--tab-delimit`	Output tab delimiter separated values. Use --no-headers to omit column headers.
`-v, --verbose`	Provide more detailed output (on a global basis). Repeated usage may result in even more verbose output.
`-X, --allow-stale-appinfo`	Do not try to detect stale data.

Table 4. Other Options

Option	Action
`-s, --statistics`	Displays scheduler statistics
`--help`	Shows this help message
`--version`	Displays version information

The `apstat` utility only reports information about the current state of the system and does not change anything. Therefore, it is safe to experiment with the options. For example, enter these commands:

`apstat -a` To see who is currently running what applications on the system and what resources the applications are using.

`apstat -n` To see the current status of every compute node on the system.

`apstat -p` To see all pending applications.

`apstat -r` To see the current node reservations.

`apstat -G` To see the current status of every node on the system that is equipped with a GPU accelerator, and to see which accelerator GPUs are installed.

`apstat -K` To see the current status of every node on the system that is equipped with an Intel Xeon Phi Coprocessor.

The contents of the reports produced by the `apstat` utility vary, depending on the report selected and the hardware features of the Cray system. For more information about the information reported and the definitions of report column headers, see the `apstat(1)` man page.

Display Job and Node Status using `xtnodestat`

The `xtnodestat` utility is another way to display current job and node status. The output is a simple two-dimensional text grid representing each node and its current 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. The following was executed on a Cray XE system. Output on a Cray XC series system is slightly different; in particular, the `s` (slot) numbers along the bottom are hexadecimal.

```
% xtnodestat
Current Allocation Status at Tue Jul 21 13:30:16 2015

     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` utility reports 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.

The `xtnodestat` command line supports three options. The `-d` and `-m` options suppress the listing of current jobs. The `xtnodestat -j` command is equivalent to the `apstat -a` command. For more information, see the `xtnodestat(1)` man page.

Manual Node Selection 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 specified criteria. Then, for interactive jobs use the `aprun -L` *`node_list`* option.

The format of the `cnselect` command is:

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

Where:

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

  The `cnselect` utility displays NIDs either sorted in ascending order 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 other SQL qualifiers (such as `ORDER BY`) are added 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.

Example 1

`cnselect` can 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
```

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.

Example 2

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

```
% 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 the `-L` option is not included on the `aprun` command, ALPS automatically places the application using available resources.

How much Memory is Available to Applications?

When running large applications, it is important to 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. The maximum stack size is determined using the `limit` command (csh) or the `ulimit -a` command (bash). Note that 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.

Use the `aprun -m` *`size`* option to specify the per-PE memory limit. For example, the following 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
```

The MPI buffer sizes and stack space defaults are changed by setting certain environment variables. For more details, see the `intro_mpi`(3) man page.

Core Specialization

CLE (Cray Linux Enviornment) offers core-specialization functionality. Core specialization binds sets of Linux kernel-space processes and daemons to one or more cores within a Cray compute node. This enables the software application to fully utilize the remaining cores within its `cpuset`. All possible overhead processing is resticted to the specialized cores within the reservation. This may improve application performance.

To calculate the new "scaled-up" width for a batch reservation using core specialization, use the `apcount` tool. `apcount` only works if the system has uniform compute node types. See the `apcount(1)` man page for further information.

Launch 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 other `aprun` options such as `-cc`, `-cp`, `-d`, `-L`, `-n`, `-N`, `-S`, `-sl`, `-sn`, and `-ss` are valid. The `-m` option (if used) must be specified in the first executable segment and the value is used for all subsequent executables. If `-m` is specified 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.

For example, this command launches 128 instances of `program1` and 256 instances of `program2`. Note that a space is required before and after the colon.

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

(Cray XE/XK systems) MPMD applications that use the SHMEM parallel programming model, either standalone or nested within an MPI program, are not supported.

Manage 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 workload manager's elapsed (wall clock) time limit to terminate the job after a specified time limit (such as `-l walltime=HH:MM:SS` for PBS).
- 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 the application is run using:

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

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

Batch Systems and Program Execution

At most sites, access to the compute node resources is managed by a batch control system, typically PBS Pro, Moab HPC Suite, TORQUE Resource Manager, Platform LSF, or Slurm. Users run jobs either by using the `qsub` command to submit a job script (or the equivalent command for their batch control system), or else by using the `qsub` command (or its equivalent) to request an interactive session within the context of the batch system, and then using the `aprun` command to run the application within the interactive session. Details for creating and submitting jobs through a specific workload manager are available through the vendor.

User applications are always launched on compute nodes using the application launcher, `aprun`, which submits applications to the Application Level Placement Scheduler (ALPS) for placement and execution. The ALPS service is both very powerful and highly flexible, and a thorough discussion of it is beyond the scope of this manual. For more detailed information about ALPS and `aprun`, see the `intro_alps(1)` and `aprun(1)` man pages, and *Application Placement for the Cray Linux Environment*.

Running an application typically involves these steps.

1. Determine what system resources are needed. Generally, this means deciding how many cores and/or compute nodes are needed for the job.
2. Use the `apstat` command to determine whether the resources needed are available. This is very important when planning to run in an interactive session. This is not as important if submitting a job script, as the batch system will keep the job script in the queue until the resources become available to run it.
3. Translate the resource request into the appropriate batch system and `aprun` command options, which are not necessarily the same. If running a batch job, modify the script accordingly.
4. For batch job submission, use the batch command (e.g., `qsub`) to submit the job script that launches the job.
5. For interactive job submission, there are two ways to reserve the needed resources and launch an application:
   - First, use the appropriate batch command with interactive option (e.g., `qsub -I`) to explicitly reserve resources. Then, enter the `aprun` command to launch your application.
   - Omit explicit reservation through `qsub -I`. Using `aprun` assumes interactive session and resources are implicitly reserved based on `aprun` options.

Dynamic Shared Objects and Libraries (DSLs)

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. Also, when shared libraries are changed or upgraded, users do not need to recompile dependent applications. Cray Linux Environment (CLE) 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.

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`.

DSL Support

CLE supports DSLs for the 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.
- Allowing sites to install DSLs provided by ISVs, or others, into the shared root in order to project them to the compute nodes for users

Configure Dynamic Shared Libraries (DSL) on CLE

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` 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 dynamic shared libraries, `/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`, 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 the following command 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 *CLE System Administration Guide*.

Build, Launch, and Workload Management Using Dynamic Objects

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:

1. Value of `LD_LIBRARY_PATH` environment variable.
2. Value of `RPATH` in the header of the executable, which is set using the `ld -rpath` command. A user 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.
3. The contents of the human non-readable cache file `/etc/ld.so.cache`. The `/etc/ld.so.conf` contains a list of colon, space, tab, newline, or comma-separated path names to which the user can append custom paths. For complete details, see the `ldconfig`(8) man page.
4. The paths `/lib` and `/usr/lib`.

When there are entries in `LD_LIBRARY_PATH` and `RPATH`, those directory paths are searched first for each library that is referenced by the run time application. This affects the run time for applications, particularly at higher node counts. For this reason, the default programming environment does not use `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 Cray compiler, the user should load the module, `PrgEnv-cray`, when setting the environment to launch the application.

Example 1: Compile an application

Dynamically compile the following program, `reduce_dyn.c`, 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, use the environment variable, `XTPE_LINK_TYPE`, without any extra compiler options:

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

To determine if an executable uses a shared library, execute 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. Fore more information, consult the `ldd`(1) man page.

Example 2: Run 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: Run 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, the user must set the environment variable after running the batch scheduler command, `qsub`:

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

Alternatively, 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
```

Troubleshooting DSLs

Some of the more common issues are described here.

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

If an error similar to the following occurs, the 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: Run an application in interactive mode* on page 32.

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

Running an Application Using a Non-existent Root

If `CRAY_ROOTFS` is erroneously set 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
```

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.

Cluster Compatibility Mode in CLE

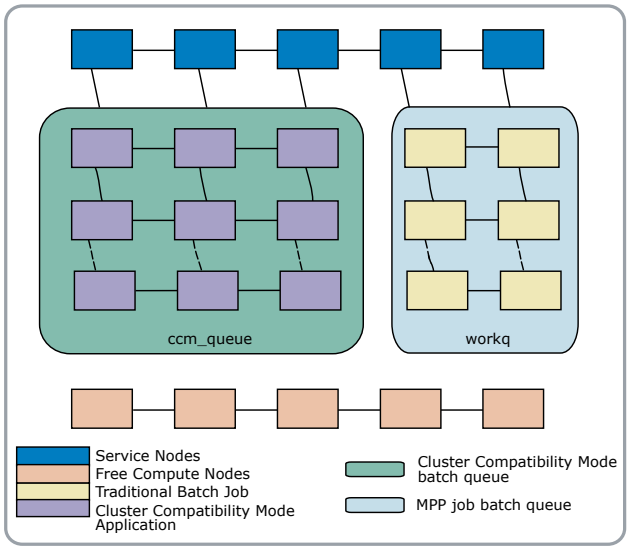
A Cray system is not a cluster but a massively parallel processing (MPP) computer. An MPP is one computer with many networked processors used for distributed computation, and, in the case of Cray system architectures, a high-speed communications interface that facilitates optimal bandwidth and memory 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.

CCM Implementation

CCM may be 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. Essentially, CCM uses the batch system to logically designate part of the Cray system as an emulated cluster for the duration of the job.

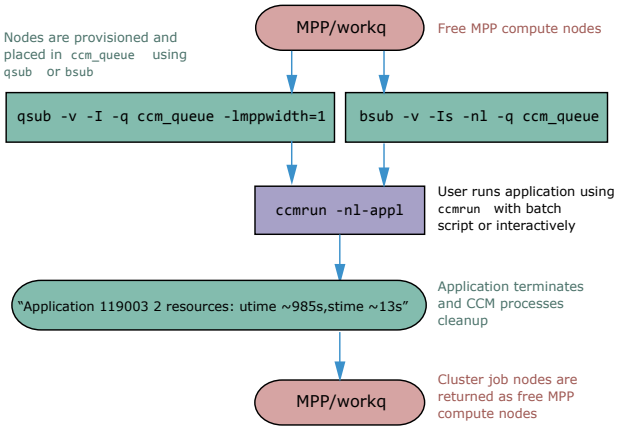
Figure 1. Cray System Job Distribution Cross-section



Users provision the emulated cluster by launching a batch or interactive job in LSF, PBS, Moab and TORQUE, or Slurm 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.

Figure 2. CCM Job Flow Diagram



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.

Cluster Compatibility Mode Commands

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

The ccmrun Command

The ccmrun command, 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] [--nscd] [--nonscd] [--norsh] [--norsip] [--nosshd] [--rsh] [--
rsip] [--ssh] executable [executable arg1] ... [executable argn]
```

Where:

--help Displays ccmrun usage statement.

--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.

--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.

--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.

--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 behavior.

--rsip Turns on CCM RSIP (Realm Specific Internet Protocol) Small Port allocation behavior. When selected, 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.

--ssh Launches a CCM job with the SSH daemon listening on port 203. This is the default in absence of custom configuration or environment.

The ccmlogin Command

The `ccmlogin` command supports a `-n` *`host`* 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`.

If `-n` *`host`* is not specified, `ccmlogin` will `ssh` to the first node in `/var/run/crayccm/ccm_nodelist.$CCM_JOBID`.

Start a CCM Batch Job

Use either PBS, Moab and TORQUE, Platform LSF or Slurm 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:

Launch a CCM application using qsub

```
% qsub -I -l mmpwidth=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 reappears, run the application using `ccmrun`:

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

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 -X login
% qsub -V -q=ccm_queue -lmppwidth=1
% ccmlogin -V
```

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 an application's MPI and setting up the necessary CCM environment for application acceleration with Infiniband over the high speed network. Platform MPI, Open MPI, Intel MPI, and MVAPICH2 are supported.

Configure Platform MPI (HP-MPI) and Launch `mpirun`

Cray recommends passing 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.

Caveats and Limitations for IAA

The following are known caveats or limitations when using IAA:

- IAA supports up to 4096 processing elements per application.
- IAA does not yet support 32-bit applications.
- 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.

Individual Software Vendor (ISV) Example

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 it is installed it a user's 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.

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 disregarding 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 the system administrator if receiving the message.

- "`Fatal error detected by libibgni.so`"

This indicates that IAA encountered a condition that causes it to abort. Further information is provided in 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, indicating a CLE installation issue. Contact the system administrator when this error occurs.

- "`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 the program output for the primary issue.

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

  If a user reruns Platform MPI jobs too close together, one 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.

Troubleshooting Cluster Compatibility Mode Issues

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`) are not specialized to the `cnos` class view. If this error occurs, the system administrator must migrate these files from the `login` class view to the `cnos` class view. For more information, see *CLE System Management Guide*.

PMGR_COLLECTIVE ERROR

The error "`PMGR_COLLECTIVE ERROR: uninitialized MPI task: Missing required environment variable: MPIRUN_RANK`," typically means that the user is trying to run an application compiled with a mismatched version of MPI.

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.

Follow the procedure *Disable CSA Accounting for the cnos Class View* on page 42.

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.

"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 that is different from the implementation being used by the application.

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.

glibc.so Errors at Start of Application Launch

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

"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.

Disable CSA Accounting for the cnos Class View

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.

1. Start `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. Start `xtopview` in the `cnos` view:

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

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

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

   and remove or comment out the following line:

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

4. Edit the `/etc/pam.d/common-session` file 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
```

Caveats and Limitations for CCM

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.

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)
```

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
```

This 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)
```

Miscellaneous Limitations

The following limitations apply to supporting cluster queues with CLE 5.2 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.

The aprun Memory Affinity Options

A compute node's memory and CPUs are divided into one or more NUMA nodes. References from a CPU on one NUMA node to memory on another NUMA node can adversely affect performance. Cray has added `aprun` memory affinity options to give the user run time controls that may optimize memory references. Most nodes with an NVIDIA GPU or a KNC co-processor have only a single NUMA node; Cray XK compute nodes with a GPU can have more than one NUMA node.

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 NUMA-node memory on the node on which the application is placed. 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, each application process allocates memory from the NUMA node on which it is running. If insufficient free NUMA-node memory is available on the process node, the allocation may be satisfied by using NUMA-node memory from another node. 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 allocations are restricted to local-NUMA-node memory even if all remote-NUMA-node memory is available to the application. Therefore, `-ss` requests fail if there is insufficient memory on the local NUMA node regardless of how much memory is available elsewhere on the compute node.

The `aprun` memory affinity options are:

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

Use these `aprun` options for each element of an MPMD application and vary them with each MPMD element as required. For complete details, see *Run Applications Using the aprun Command* on page 6.

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

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.

The aprun CPU Affinity Option

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 enables a user to 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 option is:

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

This option can be used for each element of an MPMD application and can vary with each MPMD element.

For details, see *Run Applications Using the aprun Command* on page 6.

Exclusive Access to a Node's Processing and Memory Resources

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/opt/cray/alps/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.

Optimize 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, 5-node job running on dual-core nodes, the layout of ranks on cores is:

	Node 1	Node 2	Node 3	Node 4	Node 5
Core	0 - 23	0 - 23	0 - 23	0 - 23	0 - 4
Rank	0 - 23	24 - 47	48 - 71	72 - 95	96 - 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.

Run 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-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
```

Run 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-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
```

Rum 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
```

Here is a 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 ();
}
```

Use 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-intel

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

```
/*
 *      simple put test
 */

#include <stdio.h>
#include <stdlib.h>
#include <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. */
  shmem_init();

  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));
      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
```

Use 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-cray
PrgEnv-pgi
PrgEnv-gnu
PrgEnv-intel

The `cray-shmem` module is also required.

> **IMPORTANT:** The Fortran module for Cray SHMEM is not supported. Use the `INCLUDE 'shmem.fh'` statement instead.

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

```
program reduction
include 'shmem.fh'

real values, sum
common /c/ values
real work

call shmem_init()
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 `shmem_get`:

```
% 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
```

Run 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, or Intel C compilers.

Run a Unified Parallel C (UPC) Application

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 an error will occur with PGAS applications if these modules are not loaded:

udreg
ugni
dmapp

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`. Note that it is necessary to include the `-h upc` option on the `cc` command line.

```
% 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
```

Run 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
```

Run an Accelerated 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_HostFree ( 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
```

Run a PETSc Application

This example shows how to use PETSc functions to solve a linear system of partial differential equations.

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,technical support is available 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 - `cray-petsc` and one of the following:

PrgEnv-cray
PrgEnv-pgi
PrgEnv-gnu
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.
!
!
!/*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/petsc/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:
!     petscsys.h       - base PETSc routines
!     petscvec.h    - vectors
!     petscmat.h    - matrices
!     petscpc.h     - preconditioners
!     petscksp.h    - Krylov subspace methods
!  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 <petsc/finclude/petscsys.h>
#include <petsc/finclude/petscvec.h>
#include <petsc/finclude/petscmat.h>
#include <petsc/finclude/petscpc.h>
#include <petsc/finclude/petscksp.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.
!
      PetscReal  norm
      PetscInt  i,j,II,JJ,m,n,its
      PetscInt  Istart,Iend,ione
      PetscErrorCode ierr
      PetscMPIInt     rank,size
      PetscBool   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
!      PetscReal 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)
      call MatSetUp(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) 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) 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,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_REAL,
!     &     PETSC_DEFAULT_REAL,PETSC_DEFAULT_INTEGER,ierr)

!  Set user-defined monitoring routine if desired

      call PetscOptionsHasName(PETSC_NULL_CHARACTER,'-my_ksp_monitor',  &
     &                    flg,ierr)
      if (flg) 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 <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) then
        call KSPSetConvergenceTest(ksp,MyKSPConverged,                  &
```

```
     &          PETSC_NULL_OBJECT,PETSC_NULL_FUNCTION,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 ',e11.4,' 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 <petsc/finclude/petscsys.h>
```

```
#include <petsc/finclude/petscvec.h>
#include <petsc/finclude/petscksp.h>

      KSP              ksp
      Vec              x
      PetscErrorCode ierr
      PetscInt n,dummy
      PetscMPIInt rank
      PetscReal 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 ',e11.4)
      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 <petsc/finclude/petscsys.h>
#include <petsc/finclude/petscvec.h>
#include <petsc/finclude/petscksp.h>

      KSP              ksp
      PetscErrorCode ierr
      PetscInt n,dummy
      KSPConvergedReason flag
      PetscReal rnorm

      if (rnorm .le. .05) then
        flag = 1
      else
        flag = 0
      endif
      ierr = 0
```

```
      end
```

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

```
% 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
```

Run 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-intel

To compile an OpenMP program using the PGI 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.

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-cray` module:

```
% module swap PrgEnv-pgi PrgEnv-cray
```

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
```

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

- Increase the `aprun -d` depth value by one. This will reserve one extra CPU per process, increasing the total number of CPUs required to run the job.

- Use the `aprun -cc depth` affinity option. Setting the environment variable `KMP_AFFINITY=compact` may increase performance (see "*User and Reference Guide for the Intel® C++ Compiler*" for more information).

  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.

Run an Interactive Batch Job

This example shows how to compile and run an OpenMP/MPI application (see *Run an OpenMP Application* on page 65) on 16-core Cray X6 compute nodes using an interactive batch job.

Modules required: pbs or moaband one of the following:

PrgEnv-cray
PrgEnv-pgi
PrgEnv-gnu
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 4Or
% 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)
```

Use aprun Memory Affinity Options

In some cases, remote-NUMA-node memory references can reduce the performance of applications. 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.

Use 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
```

Use 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)
```

Use 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
```

Use 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)
```

Use aprun CPU Affinity Options

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

Use 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
```

Use the `aprun -cc keyword` Options

Processes can migrate from one CPU to another on a node. 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
```