NAME

aprun - Launches an application

SYNOPSIS

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

IMPLEMENTATION

Cray Linux Environment (CLE)

DESCRIPTION

To run an application on compute nodes, use the Application Level Placement Scheduler (ALPS) aprun command. The aprun command specifies application resource requirements, requests application placement, and initiates application launch.

The aprun utility provides user identity and environment information as part of the application launch so that your login node session can be replicated for the application on the assigned set of compute nodes. This information includes the aprun current working directory, which must be accessible from the compute nodes.

Before running aprun, ensure that your working directory has a file system accessible from the compute nodes. This will likely be a Lustre-mounted directory, such as /lus/nid00007/user1/.

Do not suspend aprun; it is the local representative of the application that is running on compute nodes. If aprun is suspended, the application cannot communicate with ALPS, such as sending exit notification to aprun that the application has completed.

Cray system compute node cores are paired to memory in NUMA (Non-Uniform Memory Access) nodes. Local NUMA node access is defined as memory accesses within the same NUMA node while remote NUMA node access is defined as memory accesses between separate NUMA nodes in a Cray compute node. Remote NUMA node accesses will have more latency as a result of this configuration. Cray XE5 and Cray XK6 compute nodes

have two NUMA nodes while Cray XE6 compute nodes have four NUMA nodes.

MPMD Mode

You can use aprun to launch applications in Multiple Program, Multiple Data (MPMD) mode. The command format is:

```
  aprun -n pes [other_aprun_options] executable1
[arguments_for_executable1] :
        -n pes [other_aprun_options] executable2
[arguments_for_executable2] :
        -n pes [other_aprun_options] executable3
[arguments_for_executable3] :
    ...
```

such as:

```
  % aprun -n 12 ./app1 : -n 8 -d 2 ./app2 : -n 32 -N 2 ./app3
```

A space is required before and after each colon.

On compute nodes, other_aprun_options can be -a, -cc, -cp, -d, -L, -n, -N, -S, -sl, -sn, and -ss. If you specify the -m option it must be specified in the first executable segment and the value is used for all subsequent executables. If you specify -m more than once while launching multiple applications in MPMD mode, aprun will return an error. System commands are not supported in MPMD mode and will return an error when used. For more information about MPMD mode, see Workload Management and Application Placement for the Cray Linux Environment.

Compute Nodes and NUMA

The terms CPU , integer core or core are synonymous: they represent one single block of processor logic that serves as an execution engine. A core currently has a one-to-one relationship to processing elements. The term processor refers to the hardware package that goes in a single socket.

For eight-core Cray XE6 processors, NUMA nodes 0 through 3 have four cores each (logical CPUs 0-3, 4-7, 8-11, and 12-15, respectively). For 12-core Cray XE6 processors, NUMA nodes 0 through 3 have six cores each (logical CPUs 0-5, 6-11, 12-17, and 18-23, respectively).

For quad-core Cray XE5 processors, NUMA node 0 has four cores (logical CPUs 0-3), and NUMA node 1 has four cores (logical CPUs 4-7). For six-core Cray XE5 processors, NUMA node 0 has six cores (logical CPUs 0-5), and NUMA node 1 has six cores (logical CPUs 6-11).

Two types of operations ,Äî remote-NUMA-node memory accesses and process
migration ,Äî can reduce performance. The aprun command provides memory
affinity and CPU affinity options that allow you to control these operations. For more information, see the Memory Affinity and CPU Affinity NOTES sections.

Note: Having a compute node reserved for your job does not guarantee that you can use all NUMA nodes. You have to request sufficient resources through qsub -l resource options and aprun placement options (-n, -N, -d, and/or -m) to be able to use all NUMA nodes. See the aprun option descriptions and the EXAMPLES section for more information.

aprun Options

The aprun command accepts the following options:

-b
Bypasses the transfer of the application executable to compute nodes. By default, the executable is transferred to the compute nodes as part of the aprun process of launching an application. You would likely use the -b option only if the executable to be launched was part of a file system accessible from the compute node. For more information, see the EXAMPLES section.

-B
Tells ALPS to reuse the width, depth, nppn and memory requests specified with the corresponding batch reservation. This option obviates the need to specify aprun options -n, -d, -N, and -m and aprun will exit with an error if the user specifies these with the -B option.

-cc cpu_list | keyword
Binds processing elements (PEs) to CPUs. CLE 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 CLE during application execution. For further information about binding (CPU affinity), see the CPU Affinity NOTES section.

The cpu_list is a comma-separated or hyphen-separated list of logical CPU numbers and/or ranges. As PEs are created, they are bound to the CPU in cpu_list corresponding to the number of PEs that have been created at that point. For example, the first PE created is bound to the first CPU in

cpu_list, the second PE created is bound to the second CPU in cpu_list, and so on. If more PEs are created than given in cpu_list, binding starts over at the beginning of cpu_list and starts again with the first CPU in cpu_list. The cpu_list can also contain an x, which indicates that the application-created process at that location in the fork sequence should not be bound to a CPU. If a PE creates any threads or child processes, those threads or processes will be bound to a CPU from the cpu_list in the same manner as PEs.

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

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

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

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

```
% aprun -n 8 -cc 7-0 ./a.out
```

See Example 4: Binding PEs to CPUs (-cc cpu_list options).

The following keyword values can be used:

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

  If you specify a depth per PE (aprun -d depth), the PEs are constrained to CPUs with a distance of depth between them so each PE's threads can be constrained to the CPUs

closest to the PE's CPU.

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

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

¬∑ The numa_node keyword causes a PE to be constrained to the CPUs within the assigned NUMA node. CLE can migrate a PE among the CPUs in the assigned NUMA node but not off the assigned NUMA node. For example, if PE 2 is assigned to NUMA node 0, CLE can migrate PE 2 among CPUs 0-3 but not among CPUs 4-7.

If PEs create threads, the threads are constrained to the same NUMA-node CPUs as the PEs. There is one exception. If depth is greater than the number of CPUs per NUMA node, once the number of threads created by the PE has exceeded the number of CPUs per NUMA node, the remaining threads are constrained to CPUs within the next NUMA node on the compute node. For example, 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.

For more information about the -cc keywords, see Example 5: Binding PEs to CPUs (-cc keyword options).

-cp cpu_placement_file_name
(Deferred implementation) Provides the name of a CPU binding placement file. This option applies to all multicore compute nodes. This file must be located on a file system accessible from the compute nodes. The CPU placement file provides more extensive CPU binding instructions than the -cc options.

-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.

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

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

See Example 3: OpenMP threads (-d option).

-L node_list
Specifies the candidate nodes to constrain application placement. The syntax allows a comma-separated list of nodes (such as -L 32,33,40), a range of nodes (such as -L 41-87), or a combination of both formats. Node values can be expressed in decimal, octal (preceded by 0), or hexadecimal (preceded by 0x). The first number in a range must be less than the second number (8-6, for example, is invalid), but the nodes in a list can be in any order. See Example 12: Using node lists (-L option).

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

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

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

-m size[h|hs]

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

For example, given Cray XE5 compute nodes with 32 GB of memory and 8 CPUs, the default per-PE memory size is 32 GB / 8 CPUs = 4 GB. See Example 10: Memory per PE (-m option).

If you want huge pages allocated for a Cray XE application, use the h or hs suffix.

The default huge page size for Cray XE systems is 2 MB. On Cray XE systems, additional sizes are available: 128KB, 512KB, 8MB, 16MB, and 64MB.

The use of the -m option is not required on the Cray XE system because the kernel allows the dynamic creation of huge pages. However, it is advisable to specify this option and preallocate an appropriate number of huge pages, when memory requirements are known, to reduce operating system overhead. See Hugepages for Cray Systems.

-m sizeh	Requests memory size to be allocated to each PE, where memory is preferentially allocated out of the huge page pool. All nodes use as much huge page memory as they are able to allocate and 4 KB pages thereafter. See the NOTES section and Example 11: Using huge pages (-m h and hs suffixes).

-m sizehs Requests memory size 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. See Example 11: Using huge pages (-m h and hs suffixes).

Note: To use huge pages, you must first load the huge pages library during the linking phase, such as:

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

Then set the huge pages environment variable:

```
% setenv HUGETLB_MORECORE yes
```

or

```
% export HUGETLB_MORECORE=yes
```

-n pes Specifies the number of processing elements (PEs) needed for your application. A PE is an instance of an ALPS-launched executable. The number of PEs can be expressed 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 is 1. See Example 1: PE placement (-n option).

-N pes_per_node
Specifies the number of PEs to place per node. You can use this option to reduce the number of PEs per node, thereby making more resources available per PE. For Cray systems, the default is the number of CPUs available on a node.

For Cray systems, the maximum pes_per_node is 24.

-F exclusive|share
exclusive mode specifies affinity options to provide a program with exclusive access to all the processing and

memory resources on a node. Using this option along with the cc option will bind 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 as exclusive access mode is enabled by default. However, if nodeShare is set to share in /etc/alps.conf then you must use the -F exclusive to override the policy set in this file. You can check the value of nodeShare by executing apstat -svv | grep access.

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

-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 fatal ALPS message when exiting. Normally, this option should not be used.

-r cores Enables core specialization on Cray compute nodes, where the number of cores specified is the number of system services cores per node for the application.

-S pes_per_numa_node
Specifies the number of PEs to allocate per NUMA node. This option applies to both Cray XE5 and Cray XE6 compute nodes. You can use this option to reduce the number of PEs per NUMA node, thereby making more resources available per PE. The pes_per_numa_node value can be 1-6. For eight-core Cray XE5 nodes, the default is 4. For 12-core Cray XE5 and 24-core Cray XE6 nodes, the default is 6. A zero value is not allowed and is a fatal error. For more information, see the Memory Affinity NOTES section and Example 6: Optimizing NUMA-node memory references (-S option).

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

List NUMA nodes in ascending order; -sl 1-0 and -sl 1,0 are invalid. For more information, see the Memory Affinity NOTES section and Example 7: Optimizing NUMA-node memory references (-sl option).

-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. This option applies to Cray XE5 and Cray XE6 compute nodes. The numa_nodes_per_node value can be 1 or 2 on Cray XE5 compute nodes, or 1, 2, 3, 4 on Cray XE6 compute nodes. The default is no placement constraints. You can use this option to find out if restricting your PEs to one NUMA node per node affects performance.

A zero value is not allowed and is a fatal error. For more information, see the Memory Affinity NOTES section and Example 8: Optimizing NUMA node-memory references (-sn option).

-ss Specifies strict memory containment per NUMA node. This option applies to Cray XE5 and Cray XE6 compute nodes. When -ss is specified, a PE can allocate only the memory local to its assigned NUMA node.

The default is to allow remote-NUMA-node memory allocation to all assigned NUMA nodes. You can use this option to find out if restricting each PE's memory access to local-NUMA-node memory affects performance. For more information, see the Memory Affinity NOTES section.

-T Synchronizes the application's stdout and stderr to prevent interleaving of its output.

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

: 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.

NOTES

Standard I/O

When an application has been launched on compute nodes, aprun forwards stdin only to PE 0 of the application. All of the other application PEs have stdin set to /dev/null. An application's stdout and stderr messages are sent from the compute nodes back to aprun for display.

Signal Processing

The aprun command forwards the following signals to an application:

¬∑ SIGHUP

¬∑ SIGINT

¬∑ SIGQUIT

¬∑ SIGTERM

¬∑ SIGABRT

¬∑ SIGUSR1

¬∑ SIGUSR2

¬∑ SIGURG

¬∑ SIGWINCH

User Environment Variables

The following environment variables modify the behavior of aprun:

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

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

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

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

Output Environment Variables
ALPS will pass values to the following application environment variables:

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.

Memory Affinity
Cray XE5 compute blades use dual-socket quad-core or dual-socket, six-core compute nodes. The Cray XE6 compute blades use dual-socket twelve-core or eight-core compute nodes. Cray XK6 compute blades use single G34-socket host processors accompanied by a guest GPU in one compute node, however the NUMA designation is only applicable to the host. Because Cray systems can run more tasks simultaneously, this can increase overall performance. However, remote-NUMA-node memory references, such as a process running on NUMA node 0 accessing NUMA node 1 memory, can adversely affect performance. To give you run time controls that can optimize memory references, Cray has added the following aprun memory affinity options:

¬∑ -S pes_per_numa_node

¬∑ -sl list_of_numa_nodes

¬∑ -sn numa_nodes_per_node

¬∑ -ss

Hugepages for Cray Systems

When memory usage, specifically memory which is mapped through the high speed network, exceeds 2GB on a single node, an application should be linked with the libhugetlbfs library to use the larger address range available with huge pages. At run time, set HUGETLB_ELFMAP=W to map static data to huge pages and set HUGETLB_MORECORE=yes to map the private heap to huge pages.

Please see intro_hugepages(1) for more information.

CPU Affinity

CPU affinity options enable you to bind a PE or thread to a particular CPU or a subset of CPUs on a node. These options apply to all Cray multicore compute nodes.

The compute node kernel can dynamically distribute work by allowing PEs and threads to migrate from one CPU to another within a node. In some cases, moving PEs or threads from CPU to CPU increases cache 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.

Cray systems support the following aprun CPU affinity options: -cc cpu_list | keyword.

> Note: On Cray compute nodes, your application can access only the resources you request on the aprun or qsub command (or default values). Your application does not have automatic access to all of a compute node's resources. For example, if you request four or fewer CPUs per dual-socket, quad-core compute node and you are not using the aprun -m option, your application can access only the CPUs and memory of a single NUMA node per node. If you include CPU affinity options that reference the other NUMA node's resources, the kernel either ignores those options or causes the application's termination. For more information, see Example 4: Binding PEs to CPUs (-cc cpu_list options) and the Workload Management and Application Placement for the Cray Linux Environment.

Core Specialization

When you use the -r option, cores are assigned to system services associated with your application. Using this option may improve the performance of your application. The width parameter of the batch reservation (e.g. mppwidth) that you use may be affected. To help you calculate the appropriate width when using core specialization, you can use apcount. For more information, see the apcount(1) manpage.

Resolving Claim exceeds reservation's node-count" Errors"

If your aprun command requests more nodes than were reserved by the qsub command, ALPS displays the Claim exceeds reservation's node-count error message. For batch jobs, the number of nodes reserved is set when the qsub command is successfully processed. If you subsequently request additional nodes through aprun affinity options, apsched issues the error message and aprun exits. For example, on a Cray system , the following qsub command reserves two nodes (290 and 294):

```
% qsub -I -lmppwidth=4 -lmppnppn=2
% aprun -n 4 -N 2 ./xthi | sort
Application 225100 resources: utime ~0s, stime ~0s
Hello from rank 0, thread 0, on nid00290. (core affinity = 0)
Hello from rank 1, thread 0, on nid00290. (core affinity = 1)
Hello from rank 2, thread 0, on nid00294. (core affinity = 0)
Hello from rank 3, thread 0, on nid00294. (core affinity = 1)
```

In contrast, the following aprun command fails because the -S 1 option constrains placement to one PE per NUMA node. Two additional nodes are required:

```
% aprun -n 4 -N 2 -S 1 ./xthi | sort
Claim exceeds reservation's CPUs
```

ERRORS

If all application processes exit normally, aprun exits with zero. If there is an internal aprun error or a fatal message is received from ALPS on a compute node, aprun exits with 1. Otherwise, the aprun exit code is 128 plus the termination signal number of an application process that was abnormally terminated, or the aprun exit code is the exit code of an application process that exited abnormally. The ordering of exit signals and exit codes is arbitrary, and aprun retains and displays only four application signals and exit codes.

LIMITATIONS

Cray systems currently do not support running more than one

application on a compute node.

EXAMPLES

Example 1: PE placement (-n option)

ALPS uses the smallest number of nodes available to fulfill the -n requirements. For example, the command:

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

places 32 PEs on:

¬∑ Cray XE5 dual-socket, quad-core processors on 4 nodes.

¬∑ Cray XE5 dual-socket, six-core processors on 3 nodes.

¬∑ Cray XE6 dual-socket, eight-core processors on 2 nodes.

¬∑ Cray XE6 dual-socket, 12-core processors on 2 nodes.

¬∑ Cray XE6 dual-socket, 16-core processors on 1 node.

Note: Cray XK6 nodes are populated with single-socket host processors. There is still a one-to-one relationship between PEs and host processor cores.

The above aprun command would place 32 PEs on:

¬∑ Cray XK6 single-socket, eight-core processors on 4 nodes.

¬∑ Cray XK6 single-socket, 12-core processors on 3 nodes.

¬∑ Cray XK6 single-socket, 16-core processors on 2 nodes.

The following example runs 12 PEs on three quad-core compute nodes (nodes 28-30):

```
% cnselect coremask.eq.15
28-95,128-207
% qsub -I -lmppwidth=12 -lmppnodes=\"28-95,128-207\"
% aprun -n 12 ./xthi | sort
Application 1071056 resources: utime ~0s, stime ~0s
Hello from rank 0, thread 0, on nid00028. (core affinity = 0)
Hello from rank 0, thread 1, on nid00028. (core affinity = 0)
Hello from rank 1, thread 0, on nid00028. (core affinity = 1)
Hello from rank 1, thread 1, on nid00028. (core affinity = 1)
Hello from rank 10, thread 0, on nid00030. (core affinity = 2)
```

```
Hello from rank 10, thread 1, on nid00030. (core affinity = 2)
Hello from rank 11, thread 0, on nid00030. (core affinity = 3)
Hello from rank 11, thread 1, on nid00030. (core affinity = 3)
Hello from rank 2, thread 0, on nid00028. (core affinity = 2)
Hello from rank 2, thread 1, on nid00028. (core affinity = 2)
Hello from rank 3, thread 0, on nid00028. (core affinity = 3)
Hello from rank 3, thread 1, on nid00028. (core affinity = 3)
Hello from rank 4, thread 0, on nid00029. (core affinity = 0)
Hello from rank 4, thread 1, on nid00029. (core affinity = 0)
Hello from rank 5, thread 0, on nid00029. (core affinity = 1)
Hello from rank 5, thread 1, on nid00029. (core affinity = 1)
Hello from rank 6, thread 0, on nid00029. (core affinity = 2)
Hello from rank 6, thread 1, on nid00029. (core affinity = 2)
Hello from rank 7, thread 0, on nid00029. (core affinity = 3)
Hello from rank 7, thread 1, on nid00029. (core affinity = 3)
Hello from rank 8, thread 0, on nid00030. (core affinity = 0)
Hello from rank 8, thread 1, on nid00030. (core affinity = 0)
Hello from rank 9, thread 0, on nid00030. (core affinity = 1)
Hello from rank 9, thread 1, on nid00030. (core affinity = 1
```

The following example runs 12 PEs on one dual-socket, six-core compute node:

```
% cnselect coremask.eq.4095
168-171, 172-175, 176-179
% qsub -I -lmppwidth=12 -lmppnodes=\"168-171\"
% aprun -n 12 ./xthi | sort
Application 225101 resources: utime ~0s, stime ~0s
Hello from rank 0, thread 0, on nid00170. (core affinity = 0)
Hello from rank 10, thread 0, on nid00170. (core affinity = 10)
Hello from rank 11, thread 0, on nid00170. (core affinity = 11)
Hello from rank 1, thread 0, on nid00170. (core affinity = 1)
Hello from rank 2, thread 0, on nid00170. (core affinity = 2)
Hello from rank 3, thread 0, on nid00170. (core affinity = 3)
Hello from rank 4, thread 0, on nid00170. (core affinity = 4)
Hello from rank 5, thread 0, on nid00170. (core affinity = 5)
Hello from rank 6, thread 0, on nid00170. (core affinity = 6)
Hello from rank 7, thread 0, on nid00170. (core affinity = 7)
Hello from rank 8, thread 0, on nid00170. (core affinity = 8)
Hello from rank 9, thread 0, on nid00170. (core affinity = 9)
```

Example 2: PEs per node (-N option)

If you want more compute node resources available for each PE, you can

use the -N option. For example, the following command used on a quad-core system runs all PEs on one compute node:

```
% cnselect coremask.eq.15
25-88
% qsub -I -lmppwidth=4 -lmppnodes=\"25-88\"
% aprun -n 4 ./xthi | sort
Application 225102 resources: utime ~0s, stime ~0s
Hello from rank 0, thread 0, on nid00028. (core affinity = 0)
Hello from rank 1, thread 0, on nid00028. (core affinity = 1)
Hello from rank 2, thread 0, on nid00028. (core affinity = 2)
Hello from rank 3, thread 0, on nid00028. (core affinity = 3)
```

In contrast, the following commands restrict placement to 1 PE per node:

```
% qsub -I -lmppwidth=4 -lmppnppn=1 -lmppnodes=\"25-88\"
% aprun -n 4 -N 1 ./xthi | sort
Application 225103 resources: utime ~0s, stime ~0s
Hello from rank 0, thread 0, on nid00028. (core affinity = 0)
Hello from rank 1, thread 0, on nid00029. (core affinity = 0)
Hello from rank 2, thread 0, on nid00030. (core affinity = 0)
Hello from rank 3, thread 0, on nid00031. (core affinity = 0)
```

Example 3: OpenMP threads (-d option)

For OpenMP applications, use the OMP_NUM_THREADS environment variable to specify the number of OpenMP threads and the -d option to specify the depth (number of CPUs) to be reserved for each PE and its threads.

> Note: If you are using a PathScale compiler, set the PSC_OMP_AFFINITY environment variable to FALSE before compiling:
>
> ```
> % setenv PSC_OMP_AFFINITY FALSE
> ```
>
> or:
>
> ```
> % export PSC_OMP_AFFINITY=FALSE
> ```

ALPS creates -n pes instances of the executable, and the executable spawns OMP_NUM_THREADS-1 additional threads per PE.

For example, if we use dual-socket, quad-core compute nodes, set OMP_NUM_THREADS to 4, request four PEs, and use the default depth(-d

1), then each PE spawns three additional threads:

```
% cnselect coremask.eq.255
28-95
% qsub -I -lmppwidth=4 -lmppnodes=\"28-95\"
% setenv OMP_NUM_THREADS 4
% aprun -n 4 ./xthi | sort
Application 1304346 resources: utime ~0s, stime ~0s
Hello from rank 0, thread 0, on nid00092. (core affinity = 0)
Hello from rank 0, thread 1, on nid00092. (core affinity = 0)
Hello from rank 0, thread 2, on nid00092. (core affinity = 0)
Hello from rank 0, thread 3, on nid00092. (core affinity = 0)
Hello from rank 1, thread 0, on nid00092. (core affinity = 1)
Hello from rank 1, thread 1, on nid00092. (core affinity = 1)
Hello from rank 1, thread 2, on nid00092. (core affinity = 1)
Hello from rank 1, thread 3, on nid00092. (core affinity = 1)
Hello from rank 2, thread 0, on nid00092. (core affinity = 2)
Hello from rank 2, thread 1, on nid00092. (core affinity = 2)
Hello from rank 2, thread 2, on nid00092. (core affinity = 2)
Hello from rank 2, thread 3, on nid00092. (core affinity = 2)
Hello from rank 3, thread 0, on nid00092. (core affinity = 3)
Hello from rank 3, thread 1, on nid00092. (core affinity = 3)
Hello from rank 3, thread 2, on nid00092. (core affinity = 3)
Hello from rank 3, thread 3, on nid00092. (core affinity = 3)
```

Because we used the default depth, each PE (rank) and its threads execute on one CPU of a single compute node.

By setting the depth to 4, each PE and its threads run on separate CPUs:

```
% cnselect coremask.eq.255
28-95
% qsub -I -lmppwidth=4 -lmppdepth=4 -lmppnodes=\"28-95\"
% setenv OMP_NUM_THREADS 4
% aprun -n 4 -d 4 ./xthi | sort
Application 225105 resources: utime ~0s, stime ~0s
Hello from rank 0, thread 0, on nid00028. (core affinity = 0)
Hello from rank 0, thread 1, on nid00028. (core affinity = 1)
Hello from rank 0, thread 2, on nid00028. (core affinity = 2)
Hello from rank 0, thread 3, on nid00028. (core affinity = 3)
Hello from rank 1, thread 0, on nid00028. (core affinity = 4)
Hello from rank 1, thread 1, on nid00028. (core affinity = 5)
Hello from rank 1, thread 2, on nid00028. (core affinity = 6)
Hello from rank 1, thread 3, on nid00028. (core affinity = 7)
```

```
  Hello from rank 2, thread 0, on nid00029. (core affinity = 0)
  Hello from rank 2, thread 1, on nid00029. (core affinity = 1)
  Hello from rank 2, thread 2, on nid00029. (core affinity = 2)
  Hello from rank 2, thread 3, on nid00029. (core affinity = 3)
  Hello from rank 3, thread 0, on nid00029. (core affinity = 4)
  Hello from rank 3, thread 1, on nid00029. (core affinity = 5)
  Hello from rank 3, thread 2, on nid00029. (core affinity = 6)
  Hello from rank 3, thread 3, on nid00029. (core affinity = 7)
```

If you want all of a compute node's cores and memory available for one PE and its threads, use -n 1 and -d depth. In the following example, one PE and its threads run on cores 0-11 of a 12-core Cray XE5 compute node:

```
  % setenv OMP_NUM_THREADS 12
  % aprun -n 1 -d 12 ./xthi | sort
  Application 286315 resources: utime ~0s, stime ~0s
  Hello from rank 0, thread 0, on nid00514. (core affinity = 0)
  Hello from rank 0, thread 10, on nid00514. (core affinity = 10)
  Hello from rank 0, thread 11, on nid00514. (core affinity = 11)
  Hello from rank 0, thread 1, on nid00514. (core affinity = 1)
  Hello from rank 0, thread 2, on nid00514. (core affinity = 2)
  Hello from rank 0, thread 3, on nid00514. (core affinity = 3)
  Hello from rank 0, thread 4, on nid00514. (core affinity = 4)
  Hello from rank 0, thread 5, on nid00514. (core affinity = 5)
  Hello from rank 0, thread 6, on nid00514. (core affinity = 6)
  Hello from rank 0, thread 7, on nid00514. (core affinity = 7)
  Hello from rank 0, thread 8, on nid00514. (core affinity = 8)
  Hello from rank 0, thread 9, on nid00514. (core affinity = 9)
```

Example 4: Binding PEs to CPUs (-cc cpu_list options)

This example uses the -cc option to bind the PEs to CPUs 0-2:

```
  % aprun -n 6 -cc 0-2 ./xthi | sort
  Application 225107 resources: utime ~0s, stime ~0s
  Hello from rank 0, thread 0, on nid00028. (core affinity = 0)
  Hello from rank 1, thread 0, on nid00028. (core affinity = 1)
  Hello from rank 2, thread 0, on nid00028. (core affinity = 2)
  Hello from rank 3, thread 0, on nid00028. (core affinity = 0)
  Hello from rank 4, thread 0, on nid00028. (core affinity = 1)
  Hello from rank 5, thread 0, on nid00028. (core affinity = 2)
```

Normally, if the -d option and the OMP_NUM_THREADS values are equal, each PE and its threads will run on separate CPUs. However, the -cc cpu_list option can restrict the dynamic placement of PEs and threads:

```
% setenv OMP_NUM_THREADS 5
% aprun -n 4 -d 4 -cc 2,4 ./xthi | sort
Application 225108 resources: utime ~0s, stime ~0s
Hello from rank 0, thread 0, on nid00028. (core affinity = 2)
Hello from rank 0, thread 1, on nid00028. (core affinity = 2)
Hello from rank 0, thread 2, on nid00028. (core affinity = 4)
Hello from rank 0, thread 3, on nid00028. (core affinity = 2)
Hello from rank 0, thread 4, on nid00028. (core affinity = 2)
Hello from rank 1, thread 0, on nid00028. (core affinity = 4)
Hello from rank 1, thread 1, on nid00028. (core affinity = 4)
Hello from rank 1, thread 2, on nid00028. (core affinity = 4)
Hello from rank 1, thread 3, on nid00028. (core affinity = 2)
Hello from rank 1, thread 4, on nid00028. (core affinity = 4)
Hello from rank 2, thread 0, on nid00029. (core affinity = 2)
Hello from rank 2, thread 1, on nid00029. (core affinity = 4)
Hello from rank 2, thread 2, on nid00029. (core affinity = 4)
Hello from rank 2, thread 3, on nid00029. (core affinity = 2)
Hello from rank 2, thread 4, on nid00029. (core affinity = 4)
Hello from rank 3, thread 0, on nid00029. (core affinity = 4)
Hello from rank 3, thread 1, on nid00029. (core affinity = 2)
Hello from rank 3, thread 2, on nid00029. (core affinity = 2)
Hello from rank 3, thread 3, on nid00029. (core affinity = 2)
Hello from rank 3, thread 4, on nid00029. (core affinity = 4)
```

If depth is greater than the number of CPUs per NUMA node, once the number of threads created by the PE exceeds the number of CPUs per NUMA node, the remaining threads are constrained to CPUs within the next NUMA node on the compute node. In the following example, all threads are placed on NUMA node 0 except thread 6, which is placed on NUMA node 1:

```
% setenv OMP_NUM_THREADS 7
% aprun -n 2 -d 7 ./xthi | sort
Application 286320 resources: utime ~0s, stime ~0s
Hello from rank 0, thread 0, on nid00514. (core affinity = 0)
Hello from rank 0, thread 1, on nid00514. (core affinity = 1)
Hello from rank 0, thread 2, on nid00514. (core affinity = 2)
Hello from rank 0, thread 3, on nid00514. (core affinity = 3)
Hello from rank 0, thread 4, on nid00514. (core affinity = 4)
Hello from rank 0, thread 5, on nid00514. (core affinity = 5)
Hello from rank 0, thread 6, on nid00514. (core affinity = 6)
```

```
Hello from rank 1, thread 0, on nid00262. (core affinity = 0)
Hello from rank 1, thread 1, on nid00262. (core affinity = 1)
Hello from rank 1, thread 2, on nid00262. (core affinity = 2)
Hello from rank 1, thread 3, on nid00262. (core affinity = 3)
Hello from rank 1, thread 4, on nid00262. (core affinity = 4)
Hello from rank 1, thread 5, on nid00262. (core affinity = 5)
Hello from rank 1, thread 6, on nid00262. (core affinity = 6)
```

Example 5: Binding PEs to CPUs (-cc keyword options)

By default, each PE is bound to a CPU (-cc cpu). For a Cray XE5 application, each PE runs on a separate CPU of NUMA nodes 0 and 1. In the following example, each PE is bound to a CPU of a 12-core Cray XE5 compute node:

```
% aprun -n 12 -cc cpu ./xthi | sort
Application 286323 resources: utime ~0s, stime ~0s
Hello from rank 0, thread 0, on nid00514. (core affinity = 0)
Hello from rank 10, thread 0, on nid00514. (core affinity = 10)
Hello from rank 11, thread 0, on nid00514. (core affinity = 11)
Hello from rank 1, thread 0, on nid00514. (core affinity = 1)
Hello from rank 2, thread 0, on nid00514. (core affinity = 2)
Hello from rank 3, thread 0, on nid00514. (core affinity = 3)
Hello from rank 4, thread 0, on nid00514. (core affinity = 4)
Hello from rank 5, thread 0, on nid00514. (core affinity = 5)
Hello from rank 6, thread 0, on nid00514. (core affinity = 6)
Hello from rank 7, thread 0, on nid00514. (core affinity = 7)
Hello from rank 8, thread 0, on nid00514. (core affinity = 8)
Hello from rank 9, thread 0, on nid00514. (core affinity = 9)
```

In the following example, each PE is bound to the CPUs within a NUMA node; CLE can migrate PEs among the CPUs in the assigned NUMA node but not off the assigned NUMA node:

```
% cnselect coremask.eq.255
28-95
% qsub -I -lmppwidth=8 -lmppnodes=\"28-95\"
% aprun -n 8 -cc numa_node ./xthi | sort
Application 225113 resources: utime ~0s, stime ~0s
Hello from rank 0, thread 0, on nid00028. (core affinity = 0-3)
Hello from rank 1, thread 0, on nid00028. (core affinity = 0-3)
Hello from rank 2, thread 0, on nid00028. (core affinity = 0-3)
Hello from rank 3, thread 0, on nid00028. (core affinity = 0-3)
Hello from rank 4, thread 0, on nid00028. (core affinity = 4-7)
```

```
  Hello from rank 5, thread 0, on nid00028. (core affinity = 4-7)
  Hello from rank 6, thread 0, on nid00028. (core affinity = 4-7)
  Hello from rank 7, thread 0, on nid00028. (core affinity = 4-7)
```

The following command specifies no binding; CLE can migrate threads among all the CPUs of node 28:

```
  % aprun -n 8 -cc none ./xthi | sort
  Application 225116 resources: utime ~0s, stime ~0s
  Hello from rank 0, thread 0, on nid00028. (core affinity = 0-7)
  Hello from rank 1, thread 0, on nid00028. (core affinity = 0-7)
  Hello from rank 2, thread 0, on nid00028. (core affinity = 0-7)
  Hello from rank 3, thread 0, on nid00028. (core affinity = 0-7)
  Hello from rank 4, thread 0, on nid00028. (core affinity = 0-7)
  Hello from rank 5, thread 0, on nid00028. (core affinity = 0-7)
  Hello from rank 6, thread 0, on nid00028. (core affinity = 0-7)
  Hello from rank 7, thread 0, on nid00028. (core affinity = 0-7)
```

In the following example, multiple cpu_lists are specified. Each PE is bound to the first CPU on a NUMA node, each PE creates one thread, and all odd-numbered CPUs are skipped:

```
   % aprun -n 4 -cc 0,2:4,6:8,10:12,14 ./xthi
  Hello from rank 0, thread 0, on nid00028. (core affinity = 0)
  Hello from rank 0, thread 1, on nid00028. (core affinity = 2)
  Hello from rank 1, thread 0, on nid00028. (core affinity = 4)
  Hello from rank 1, thread 1, on nid00028. (core affinity = 6)
  Hello from rank 2, thread 0, on nid00028. (core affinity = 8)
  Hello from rank 2, thread 1, on nid00028. (core affinity = 10)
  Hello from rank 3, thread 0, on nid00028. (core affinity = 12)
  Hello from rank 3, thread 1, on nid00028. (core affinity = 14)
```

Example 6: Optimizing NUMA-node memory references (-S option)

This example uses the -S option to restrict placement of PEs to one per NUMA node. Two compute nodes are required, with one PE on NUMA node 0 and one PE on NUMA node 1:

```
  % aprun -n 4 -S 1 ./xthi | sort
  Application 225117 resources: utime ~0s, stime ~0s
  Hello from rank 0, thread 0, on nid00043. (core affinity = 0)
  Hello from rank 1, thread 0, on nid00043. (core affinity = 4)
  Hello from rank 2, thread 0, on nid00044. (core affinity = 0)
```

```
  Hello from rank 3, thread 0, on nid00044. (core affinity = 4)
```

Example 7: Optimizing NUMA-node memory references (-sl option)

This example runs all PEs on NUMA node 0; the PEs cannot allocate remote NUMA node memory:

```
  % aprun -n 8 -sl 0 ./xthi | sort
  Application 225118 resources: utime ~0s, stime ~0s
  Hello from rank 0, thread 0, on nid00028. (core affinity = 0)
  Hello from rank 1, thread 0, on nid00028. (core affinity = 1)
  Hello from rank 2, thread 0, on nid00028. (core affinity = 2)
  Hello from rank 3, thread 0, on nid00028. (core affinity = 3)
  Hello from rank 4, thread 0, on nid00029. (core affinity = 0)
  Hello from rank 5, thread 0, on nid00029. (core affinity = 1)
  Hello from rank 6, thread 0, on nid00029. (core affinity = 2)
  Hello from rank 7, thread 0, on nid00029. (core affinity = 3)
```

This example runs all PEs on NUMA node 1:

```
  % aprun -n 8 -sl 1 ./xthi | sort
  Application 225119 resources: utime ~0s, stime ~0s
  Hello from rank 0, thread 0, on nid00028. (core affinity = 4)
  Hello from rank 1, thread 0, on nid00028. (core affinity = 5)
  Hello from rank 2, thread 0, on nid00028. (core affinity = 6)
  Hello from rank 3, thread 0, on nid00028. (core affinity = 7)
  Hello from rank 4, thread 0, on nid00029. (core affinity = 4)
  Hello from rank 5, thread 0, on nid00029. (core affinity = 5)
  Hello from rank 6, thread 0, on nid00029. (core affinity = 6)
  Hello from rank 7, thread 0, on nid00029. (core affinity = 7)
```

Example 8: Optimizing NUMA node-memory references (-sn option)

This example runs four PEs on NUMA node 0 of node 28 and four PEs on NUMA node 0 of node 29:

```
  % aprun -n 8 -sn 1 ./xthi | sort
  Application 225120 resources: utime ~0s, stime ~0s
  Hello from rank 0, thread 0, on nid00028. (core affinity = 0)
  Hello from rank 1, thread 0, on nid00028. (core affinity = 1)
  Hello from rank 2, thread 0, on nid00028. (core affinity = 2)
  Hello from rank 3, thread 0, on nid00028. (core affinity = 3)
  Hello from rank 4, thread 0, on nid00029. (core affinity = 0)
```

```
  Hello from rank 5, thread 0, on nid00029. (core affinity = 1)
  Hello from rank 6, thread 0, on nid00029. (core affinity = 2)
  Hello from rank 7, thread 0, on nid00029. (core affinity = 3)
```

Example 9: Optimizing NUMA-node memory references (-ss option)

When the -ss option is used, a PE can allocate only the memory local to its assigned NUMA node. The default is to allow remote-NUMA-node memory allocation to all assigned NUMA nodes. For example, by default any PE running on NUMA node 0 can allocate NUMA node 1 memory.

This example runs PEs 0-3 on NUMA node 0 and PEs 4-7 on NUMA node 1. PEs 0-3 cannot allocate NUMA node 1 memory, and PEs 4-7 cannot allocate NUMA node 0 memory.

```
  % aprun -n 8 -ss ./xthi | sort
  Application 225121 resources: utime ~0s, stime ~0s
  Hello from rank 0, thread 0, on nid00028. (core affinity = 0)
  Hello from rank 1, thread 0, on nid00028. (core affinity = 1)
  Hello from rank 2, thread 0, on nid00028. (core affinity = 2)
  Hello from rank 3, thread 0, on nid00028. (core affinity = 3)
  Hello from rank 4, thread 0, on nid00028. (core affinity = 4)
  Hello from rank 5, thread 0, on nid00028. (core affinity = 5)
  Hello from rank 6, thread 0, on nid00028. (core affinity = 6)
  Hello from rank 7, thread 0, on nid00028. (core affinity = 7)
```

Example 10: Memory per PE (-m option)

The -m option can affect application placement. This example runs all PEs on node 43. The amount of memory available per PE is 4000 MB:

```
  % aprun -n 8 -m4000m ./xthi | sort
  Application 225122 resources: utime ~0s, stime ~0s
  Hello from rank 0, thread 0, on nid00043. (core affinity = 0)
  Hello from rank 1, thread 0, on nid00043. (core affinity = 1)
  Hello from rank 2, thread 0, on nid00043. (core affinity = 2)
  Hello from rank 3, thread 0, on nid00043. (core affinity = 3)
  Hello from rank 4, thread 0, on nid00043. (core affinity = 4)
  Hello from rank 5, thread 0, on nid00043. (core affinity = 5)
  Hello from rank 6, thread 0, on nid00043. (core affinity = 6)
  Hello from rank 7, thread 0, on nid00043. (core affinity = 7)
```

In this example, node 43 does not have enough memory to fulfill the

request for 4001 MB per PE. PE 7 runs on node 44:

```
% aprun -n 8 -m4001 ./xthi | sort
Application 225123 resources: utime ~0s, stime ~0s
Hello from rank 0, thread 0, on nid00043. (core affinity = 0)
Hello from rank 1, thread 0, on nid00043. (core affinity = 1)
Hello from rank 2, thread 0, on nid00043. (core affinity = 2)
Hello from rank 3, thread 0, on nid00043. (core affinity = 3)
Hello from rank 4, thread 0, on nid00043. (core affinity = 4)
Hello from rank 5, thread 0, on nid00043. (core affinity = 5)
Hello from rank 6, thread 0, on nid00043. (core affinity = 6)
Hello from rank 7, thread 0, on nid00044. (core affinity = 0)
```

Example 11: Using huge pages (-m h and hs suffixes)

This example requests 4000 MB of huge pages per PE:

```
% cc -o xthi xthi.c -lhugetlbfs
% HUGETLB_MORECORE=yes aprun -n 8 -m4000h ./xthi | sort
%
Application 225124 resources: utime ~0s, stime ~0s
Hello from rank 0, thread 0, on nid00043. (core affinity = 0)
Hello from rank 1, thread 0, on nid00043. (core affinity = 1)
Hello from rank 2, thread 0, on nid00043. (core affinity = 2)
Hello from rank 3, thread 0, on nid00043. (core affinity = 3)
Hello from rank 4, thread 0, on nid00043. (core affinity = 4)
Hello from rank 5, thread 0, on nid00043. (core affinity = 5)
Hello from rank 6, thread 0, on nid00043. (core affinity = 6)
Hello from rank 7, thread 0, on nid00043. (core affinity = 7)
```

The following example requests 4000 MB of hugepages per PE, and also specifies that a hugepage size of 16 MB is to be used.

```
% HUGETLB_DEFAULT_PAGE_SIZE=16m aprun -n 8 -m4000h ./xthi | sort
Application 225124 resources: utime ~0s, stime ~0s
Hello from rank 0, thread 0, on nid00043. (core affinity = 0)
Hello from rank 1, thread 0, on nid00043. (core affinity = 1)
Hello from rank 2, thread 0, on nid00043. (core affinity = 2)
Hello from rank 3, thread 0, on nid00043. (core affinity = 3)
Hello from rank 4, thread 0, on nid00043. (core affinity = 4)
Hello from rank 5, thread 0, on nid00043. (core affinity = 5)
Hello from rank 6, thread 0, on nid00043. (core affinity = 6)
Hello from rank 7, thread 0, on nid00043. (core affinity = 7)
```

The following example terminates because the required 4000 MB of huge pages per PE are not available:

```
% aprun -n 8 -m4000hs ./xthi | sort
[NID 00043] 2009-04-09 07:58:28 Apid 379231: unable to acquire
enough huge memory: desired 32000M, actual 31498M
```

Example 12: Using node lists (-L option)

You can specify candidate node lists through the aprun -L option for applications launched interactively and through the qsub -lmppnodes option for batch and interactive batch jobs.

For an application launched interactively, use the cnselect command to get a list of all Cray XE5 compute nodes. Then use aprun -L option to specify the candidate list:

```
% cnselect coremask.eq.255
28-95
% aprun -n 4 -N 2 -L 28-95 ./xthi | sort
Application 225127 resources: utime ~0s, stime ~0s
Hello from rank 0, thread 0, on nid00028. (core affinity = 0)
Hello from rank 1, thread 0, on nid00028. (core affinity = 1)
Hello from rank 2, thread 0, on nid00029. (core affinity = 0)
Hello from rank 3, thread 0, on nid00029. (core affinity = 1)
```

Example 13: Bypassing binary transfer (-b option)

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

```
% aprun -b /bin/ash -c "cat /proc/meminfo |grep MemTotal"
MemTotal:     32909204 kB
```

For further information about the commands you can use with the aprun -b option, see Workload Management and Application Placement for the Cray Linux Environment.

SEE ALSO

intro_alps(1), apkill(1), apstat(1), cnselect(1), qsub(1)

CC(1), cc(1), ftn(1)

Workload Management and Application Placement for the Cray Linux Environment

Cray Application Developer's Environment User's Guide