Chapter 5. Data Placement Tools

This chapter describes data placement tools you can use on an SGI Altix system. It covers the following topics:

Data Placement Tools Overview

On an symmetric multiprocessor (SMP) machine, all data is visible from all processors. NonUniform Memory Access (NUMA) machines also have a shared address space. In both cases, there is a single shared memory space and a single operating system instance. However, in an SMP machine, each processor is functionally identical and has equal time access to every memory address. In contrast, a NUMA system has a shared address space, but the access time to memory vary over physical address ranges and between processing elements. The Intel Xeon 7500 series processor (Nehalem i7 architecture) is an example of NUMA architecture. Each processor has its own memory and can address the memory attached to another processor through the Quick Path Interconnect (QPI).

The Altix UV 1000 series is a family of multiprocessor distributed shared memory (DSM) computer systems that initially scale from 32 to 4,096 Intel processor cores as a cache-coherent single system image (SSI). The Altix UV 100 series is a family of multiprocessor distributed shared memory (DSM) computer systems that initially scale from 16 to 768 Intel processor cores as a cache-coherent SSI.

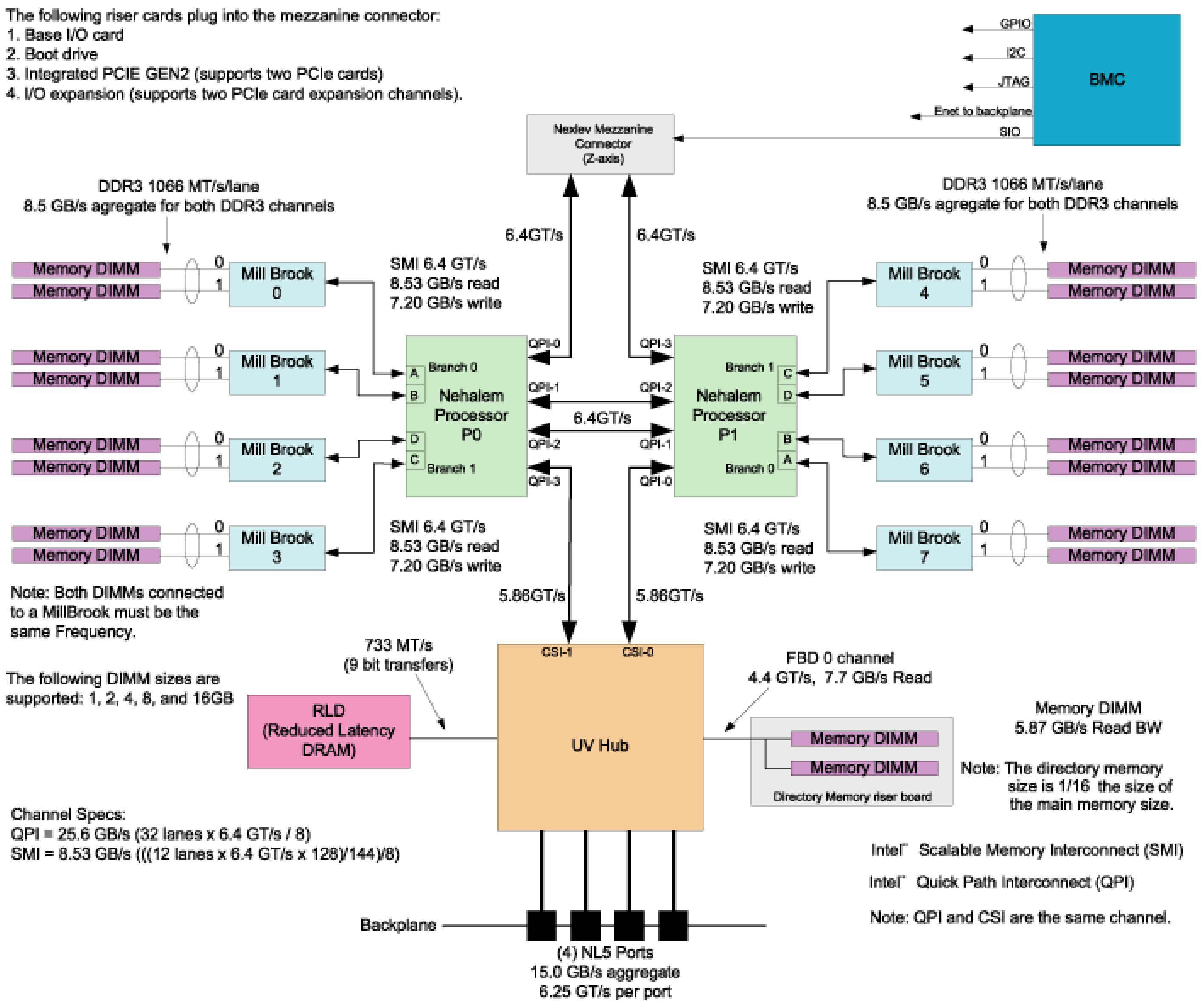
In both Altix UV series systems, there are two levels of NUMA: intranode managed by the QPI and internode managed through the HUB ASIC and NUMAlink 5. This section covers the following topics:

Distributed Shared Memory (DSM)

In the Altix UV 100 or Altix UV 1000 series server, memory is physically distributed both within and among the IRU enclosures (compute/memory/I/O blades); however, it is accessible to and shared by all devices connected by NUMAlink within the single–system image (SSI). This is to say that all components connected by NUMAlink sharing a single Linux operating system, operate and share the memory "fabric" of the system. Memory latency is the amount of time required for a processor to retrieve data from memory. Memory latency is lowest when a processor accesses local memory. Note the following sub–types of memory within a system:

- If a processor accesses memory that it is connected to on a compute node blade, the memory is referred to as the node's local memory. Figure 5–1 shows a conceptual block diagram of the blade's memory, compute and I/O pathways.
- If processors access memory located in other blade nodes within the IRU, (or other NUMAlink IRUs) the memory is referred to as remote memory.
- The total memory within the NUMAlink system is referred to as global memory.

Figure 5–1. Blade Node Block Diagram


The following riser cards plug into the mezzanine connector:
1. Base I/O card
2. Boot drive
3. Integrated PCIE GEN2 (supports two PCIe cards)
4. I/O expansion (supports two PCIe card expansion channels).
GPIO
I2C
JTAG
Enet to backplane
SIO
BMC
Nexlev Mezzanine Connector (Z-axis)
DDR3 1066 MT/s/lane
8.5 GB/s agregate for both DDR3 channels
6.4GT/s
SMI 6.4 GT/s
8.53 GB/s read
7.20 GB/s write
Memory DIMM
Mill Brook 0
Mill Brook 1
Mill Brook 2
Mill Brook 3
Mill Brook 4
Mill Brook 5
Mill Brook 6
Mill Brook 7
QPI-0
QPI-1
QPI-2
QPI-3
Branch 0
Branch 1
Nehalem Processor P0
Nehalem Processor P1
Note: Both DIMMs connected to a MillBrook must be the same Frequency.
5.86GT/s
733 MT/s
(9 bit transfers)
The following DIMM sizes are supported: 1, 2, 4, 8, and 16GB
RLD (Reduced Latency DRAM)
CSI-1
CSI-0
UV Hub
FBD 0 channel
4.4 GT/s, 7.7 GB/s Read
Memory DIMM
5.87 GB/s Read BW
Directory Memory riser board
Note: The directory memory size is 1/16 the size of the main memory size.
Channel Specs:
QPI = 25.6 GB/s (32 lanes x 6.4 GT/s / 8)
SMI = 8.53 GB/s (((12 lanes x 6.4 GT/s x 128)/144)/8)
Intel® Scalable Memory Interconnect (SMI)
Intel® Quick Path Interconnect (QPI)
Note: QPI and CSI are the same channel.
Backplane
(4) NL5 Ports
15.0 GB/s aggregate
6.25 GT/s per port

ccNUMA Architecture

As the name implies, the cache–coherent non–uniform memory access (ccNUMA) architecture has two parts, cache coherency and nonuniform memory access, which are discussed in the sections that follow.

Cache Coherency

The Altix UV 100 and Altix 1000 server series use caches to reduce memory latency. Although data exists in local or remote memory, copies of the data can exist in various processor caches throughout the system. Cache coherency keeps the cached copies consistent.

To keep the copies consistent, the ccNUMA architecture uses directory–based coherence protocol. In directory–based coherence protocol, each block of memory (128 bytes) has an entry in a table that is referred to as a directory. Like the blocks of memory that they represent, the directories are distributed among the compute/memory blade nodes. A block of memory is also referred to as a cache line.

Each directory entry indicates the state of the memory block that it represents. For example, when the block is not cached, it is in an unowned state. When only one processor has a copy of the memory block, it is in an exclusive state. And when more than one processor has a copy of the block, it is in a shared state; a bit vector indicates which caches may contain a copy.

When a processor modifies a block of data, the processors that have the same block of data in their caches must be notified of the modification. The Altix UV 100 and Altix UV 1000 server series uses an invalidation method to maintain cache coherence. The invalidation method purges all unmodified copies of the block of data, and the processor that wants to modify the block receives exclusive ownership of the block.

Non–uniform Memory Access (NUMA)

In DSM systems, memory is physically located at various distances from the processors. As a result, memory access times (latencies) are different or "non–uniform." For example, it takes less time for a processor blade to reference its locally installed memory than to reference remote memory.

Data Placement Practices

For cc–NUMA systems like the Altix UV 100 or Altix UV 1000, there is a performance penalty to access remote memory versus local memory. Because the Linux operating system has a tendency to migrate processes, the importance of using a placement tool becomes more apparent. Various data placement tools are described in this section.

For a detailed overview of Altix UV system architecture, see the *SGI Altix UV 100 System User's Guide* or *SGI Altix UV 1000 System User's Guide*

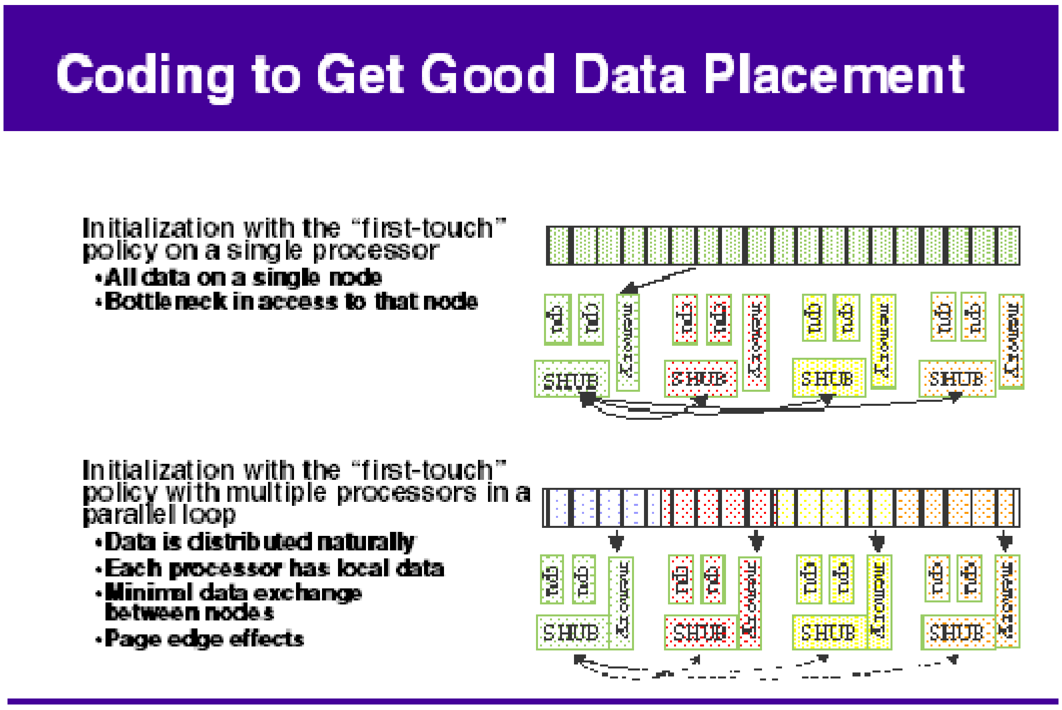
Special optimization applies to SGI Altix systems to exploit multiple paths to memory, as follows:

- By default, all pages are allocated with a “first touch” policy.
- The initialization loop, if executed serially, will get pages from single node.
- In the parallel loop, multiple processors will access that one memory.

So, perform initialization in parallel, such that each processor initializes data that it is likely to access later for calculation.

Figure 5–2, shows how to code to get good data placement.

Figure 5-2. Coding to Get Good Data Placement


Coding to Get Good Data Placement
Initialization with the "first-touch" policy on a single processor
•All data on a single node
•Bottleneck in access to that node
Initialization with the "first-touch" policy with multiple processors in a parallel loop
•Data is distributed naturally
•Each processor has local data
•Minimal data exchange between nodes
•Page edge effects
cpu
memory
SHUB

Placement facilities include cpusets, `taskset(1)`, and `dplace(1)`, all built upon CpuMemSets API:

- cpusets -- Named subsets of system cpus/memories, used extensively in batch environments.
- `taskset` and `dplace` -- Avoid poor data locality caused by process or thread drift from CPU to CPU.
  - `taskset` restricts execution to the listed set of CPUs (see the `taskset -c --cpu-list` option); however, processes are still free to move among listed CPUs.
  - `dplace` binds processes to specified CPUs in round-robin fashion; once pinned, they do not migrate. Use this for high performance and reproducibility of parallel codes.

For more information on CpuMemSets and cpusets, see chapter 4, "CPU Memory Sets and Scheduling" and chapter 5, "Cpuset System", respectively, in the *Linux Resource Administration Guide*.

`taskset` Command

The `taskset(1)` command retrieves or sets a CPU affinity of a process, as follows:

```
taskset [options] mask command [arg]...
taskset [options] -p [mask] pid
```

The `taskset` command is used to set or retrieve the CPU affinity of a running process given its PID or to launch a new command with a given CPU affinity. CPU affinity is a scheduler property that "bonds" a process to a given set of CPUs on the system. The Linux scheduler will honor the given CPU affinity and the process will not run on any other CPUs. Note that the Linux scheduler also supports natural CPU affinity; the scheduler attempts to keep processes on the same CPU as long as practical for performance reasons. Therefore, forcing a specific CPU affinity is useful only in certain applications.

The CPU affinity is represented as a bitmask, with the lowest order bit corresponding to the first logical CPU and the highest order bit corresponding to the last logical CPU. Not all CPUs may exist on a given system but a mask may specify more CPUs than are present. A retrieved mask will reflect only the bits that correspond to CPUs physically on the system. If an invalid mask is given (that is, one that corresponds to no valid CPUs on the current system) an error is returned. The masks are typically given in hexadecimal. For example:

`0x00000001`
: is processor #0

`0x00000003`
: is processors #0 and #1

`0xFFFFFFFF`
: is all processors (#0 through #31)

When `taskset` returns, it is guaranteed that the given program has been scheduled to a legal CPU.

The `taskset` command does not pin a task to a specific CPU. It only restricts a task so that it does not run on any CPU that is not in the `cpulist`. For example, if you use `taskset` to launch an application that forks multiple tasks, it is possible that multiple tasks will initially be assigned to the same CPU even though there are idle CPUs that are in the `cpulist`. Scheduler load balancing software will eventually distribute the tasks so that CPU bound tasks run on different CPUs. However, the exact placement is not predictable and can vary from run-to-run. After the tasks are evenly distributed (assuming that happens), nothing prevents tasks from jumping to different CPUs. This can affect memory latency since pages that were node-local before the jump may be remote after the jump.

If you are running an MPI application, SGI recommends that you do not use the `taskset` command. The `taskset` command can pin the MPI shepherd process (which is a waste of a CPU) and then putting the remaining working MPI rank on one of the CPUs that already had some other rank running on it. Instead of `taskset`, SGI recommends using the `dplace(1)` (see "dplace Command") or the environment variable `MPI_DSM_CPULIST` . The following example assumes a job running on eight CPUs. For example:

```
# mpirun -np 8 dplace -s1 -c10,11,16-21 myMPIapplication ...
```

To set `MPI_DSM_CPULIST` variable, perform a command similar to the following:

```
setenv MPI_DSM_CPULIST 10,11,16-21 mpirun -np 8 myMPIapplication ...
```

If they are using a batch scheduler that creates and destroys cpusets dynamically, you should use `MPI_DSM_DISTRIBUTE` environment variable instead of either `MPI_DSM_CPULIST` environment variable or the `dplace` command.

For more detailed information, see the `taskset(1)` man page.

To run an executable on CPU 1 (the cpumask for CPU 1 is 0x2), perform the following:

```
# taskset 0x2 executable name
```

To move pid 14057 to CPU 0 (the cpumask for cpu 0 is 0x1), perform the following:

```
# taskset -p 0x1 14057
```

To run an MPI Abaqus/Std job on an Altix system with eight CPUs, perform the following:

```
# taskset -c 8-15 ./runme < /dev/null &
```

The stdin is redirected to `/dev/null` to avoid a `SIGTTIN` signal for MPT applications.

The following example uses the `taskset` command to lock a given process to a particular CPU (CPU5) and then uses the `profile(1)` command to profile it. It then shows how to use `taskset` to move the process to another CPU (CPU3).

```
# taskset -p -c 5 16269
pid 16269's current affinity list: 0-15
pid 16269's new affinity list: 5
```

```
# taskset -p 16269 -c 3
pid 16269's current affinity list: 5
pid 16269's new affinity list: 3
```

`dplace` Command

You can use the `dplace(1)` command to bind a related set of processes to specific CPUs or nodes to prevent process migration. This can improve the performance of your application since it increases the percentage of memory accesses that are local.

Using the `dplace` Command

The `dplace` command allows you to control the placement of a process onto specified CPUs, as follows:

```
dplace [-e] [-c cpu_numbers] [-s skip_count] [-n process_name] \
             [-x skip_mask] [-r [l|b|t]] [-o log_file] [-v 1|2] \
             command [command-args]
dplace [-p placement_file] [-o log_file] command [command-args]
dplace [-q] [-qq] [-qqq]
```

Scheduling and memory placement policies for the process are set up according to `dplace` command line arguments.

By default, memory is allocated to a process on the node on which the process is executing. If a process moves from node to node while it running, a higher percentage of memory references are made to remote nodes. Because remote accesses typically have higher access times, process performance can be diminished. CPU instruction pipelines also have to be reloaded.

You can use the `dplace` command to bind a related set of processes to specific CPUs or nodes to prevent process migrations. In some cases, this improves performance since a higher percentage of memory accesses are made to local nodes.

Processes always execute within a CpuMemSet. The CpuMemSet specifies the CPUs on which a process can execute. By default, processes usually execute in a CpuMemSet that contains all the CPUs in the system (for detailed information on CpusMemSets, see the *Linux Resource Administration Guide*).

The `dplace` command invokes an SGI kernel hook (module called `numatools`) to create a placement container consisting of all the CPUs (or a or a subset of CPUs) of a cpuset. The `dplace` process is placed in this container and by default is bound to the first CPU of the cpuset associated with the container. Then `dplace` invokes `exec` to execute the command.

The command executes within this placement container and remains bound to the first CPU of the container. As the command forks child processes, they inherit the container and are bound to the next available CPU of the container.

If you do not specify a placement file, `dplace` binds processes sequentially in a round-robin fashion to CPUs of the placement container. For example, if the current cpuset consists of physical CPUs 2, 3, 8, and 9, the first process launched by `dplace` is bound to CPU 2. The first child process forked by this process is bound to CPU 3, the next process (regardless of whether it is forked by parent or child) to 8, and so on. If more processes are forked than there are CPUs in the cpuset, binding starts over with the first CPU in the cpuset.

For more information on `dplace(1)` and examples of how to use the command, see the `dplace(1)` man page.

The `dplace(1)` command accepts the following options:

- `-c` *`cpu_numbers`*: Specified as a list of cpus, optionally strided cpu ranges, or a striding pattern. Example: "`-c 1`", "`-c 2-4`", "`-c 1,4-8,3`", "`-c 2-8:3`", " `-c CS`", "`-c BT`". The specification " `-c 2-4`" is equivalent to "–`c 2,3,4`" and " `-c 2-8:3`" is equivalent to `2,5,8`. Ranges may also be specified in reverse order: "–c `12-8`" is equivalent to `12,11,10,9,8`. CPU numbers are **NOT** physical cpu numbers. They are logical cpu number that are relative to the cpus that are in the set of allowed cpus as specified by the current cpuset. A cpu value of "`x`" (or " `*`"), in the argument list for `-c` option, indicates that binding should not be done for that process."x" should be used only if the `-e` option is also used. Cpu numbers start at 0. For striding patterns any subset of the characters (B)lade, (S)ocket, (C)ore, (T)hread may be used and their ordering specifies the nesting of the iteration. For example "SC" means to iterate all the cores in a socket before moving to the next CPU socket, while "CB" means to pin to the first core of each blade, then the second core of every blade, etc. For best results, use the `-e` option when using stride patterns. If the `-c` option is not specified, all cpus of the current cpuset are available. The command itself (which is `exec`'d by `dplace`) is the first process to be placed by the `-c` *cpu_numbers* .

- `-e`: Exact placement. As processes are created, they are bound to cpus in the exact order that the cpus are specified in the cpu list. Cpu numbers may appear multiple times in the list. A cpu value of "`x`" indicates that binding should not be done for that process. If the end of the list is reached, binding starts over at the beginning of the list.

- `-o`: Write a trace file to Write a trace file to <*log file*> that decribes the placement actions that were made for each `fork`, `exec`, and so on. Each line contains a timestamp, process id:thread number, cpu that task was executing on, taskname | placement action. Works with version 2 only.

- `-s` *`skip_count`*: Skips the first *skip_count* processes before starting to place processes onto CPUs. This option is useful if the first *skip_count* processes are “shepherd" processes that are used only for launching the application. If *skip_count* is not specified, a default value of 0 is used.

- `-n` *`process_name`*: Only processes named *process_name* are placed. Other processes are ignored and are not explicitly bound to CPUs.

  The *process_name* argument is the basename of the executable.

- `-r`: Specifies that text should be replicated on the node or nodes where the application is running. In some cases, replication will improve performance by reducing the need to make offnode memory references for code. The replication option applies to all programs placed by the dplace command. See the `dplace`(5) man page for additional information on text replication. The replication options are a string of one or more of the following characters:

  `l`

  Replicate library text

  `b`

  Replicate binary (a.out) text

  `t`

  Thread round-robin option

- `-x` *`skip_mask`*: Provides the ability to skip placement of processes. The *skip_mask* argument is a bitmask. If bit N of `skip_mask` is set, then the N+1th process that is forked is not placed. For example, setting the mask to 6 prevents the second and third processes from being placed. The first process (the process named by the command) will be assigned to the first CPU. The second and third processes are not placed. The fourth process is assigned to the second CPU, and so on. This option is useful for certain classes of threaded applications that spawn a few helper processes that typically do not use much CPU time.

  Note: OpenMP with Intel applications should be placed using the `-x` option with a `skip_mask` of `2` (`-x2`). This could change in future versions of OpenMP. For applications compiled using the Native Posix Thread Library (NPTL), use the `-x2` option.

- `-v`: Provides the ability to run in version 1 or version 2 compatibility mode if the kernel support is available. If not specified, version 2 compatibility is selected. See COMPATIBILITY section of the `dplace`(1) man page for more details. Note: version 1 requires kernel support for PAGG.

- `-p` *`placement_file`*: Specifies a placement file that contains additional directives that are used to control process placement.
- `command [`*`command-args`*`]`: Specifies the command you want to place and its arguments.
- `-q`: Lists the global count of the number of active processes that have been placed (by `dplace`) on each CPU in the current cpuset. Note that CPU numbers are logical CPU numbers within the cpuset, **not** physical CPU numbers. If specified twice, lists the current `dplace` jobs that are running. If specified three times, lists the current `dplace` jobs and the tasks that are in each job.

Example 5-1. Using the `dplace` command with MPI Programs

You can use the `dplace` command to improve placement of MPI programs on NUMA systems and verify placement of certain data structures of a long running MPI program by running a command such as the following:

```
mpirun -np 64 /usr/bin/dplace -s1 -c 0-63 ./a.out
```

You can then use the `dlook(1)` command to verify placement of certain data structures of a long running MPI program by using the `dlook` command in another window on one of the slave thread PIDs to verify placement. For more information on using the `dlook` command, see “dlook Command” and the `dlook(1)` man page.

Example 5-2. Using `dplace` command with OpenMP Programs

To run an OpenMP program on logical CPUs 4 through 7 within the current cpuset, perform the following:

```
%efc -o prog -openmp -O3 program.f
%setenv OMP_NUM_THREADS 4
%dplace -x6 -c4-7 ./prog
```

The `dplace(1)` command has a static load balancing feature so that you do not necessarily have to supply a CPU list. To place `prog1` on logical CPUs 0 through 3 and `prog2` on logical CPUs 4 through 7, perform the following:

```
%setenv OMP_NUM_THREADS 4
%dplace -x6 ./prog1 &
%dplace -x6 ./prog2 &
```

You can use the `dplace -q` command to display the static load information.

Example 5-3. Using the `dplace` command with Linux commands

The following examples assume that the command is executed from a shell running in a cpuset consisting of physical CPUs 8 through 15.

Command	Run Location
`dplace -c2 date`	Runs the `date` command on physical CPU 10.
`dplace make linux`	Runs `gcc` and related processes on physical CPUs 8 through 15.
`dplace -c0-4,6 make linux`	Runs `gcc` and related processes on physical CPUs 8 through 12 or 14.
`taskset 4,5,6,7 dplace app`	The `taskset` command restricts execution to physical CPUs 12 through 15. The `dplace` command sequentially binds processes to CPUs 12 through 15.

To use the `dplace` command accurately, you should know how your placed tasks are being created in terms of the `fork`, `exec`, and `pthread_create` calls. Determine whether each of these worker calls are an MPI rank task or are they groups of pthreads created by rank tasks? Here is an example of two MPI ranks, each creating three threads:

```
cat <<EOF > placefile
firsttask cpu=0
exec name=mpiapp cpu=1
fork   name=mpiapp cpu=4-8:4 exact
thread name=mpiapp oncpu=4 cpu=5-7 exact thread name=mpiapp oncpu=8
cpu=9-11 exact EOF

#  mpirun is placed on cpu 0 in this example #  the root mpiapp is
placed on cpu 1 in this example

# or, if your version of dplace supports the "cpurel=" option:
# firsttask cpu=0
# fork   name=mpiapp cpu=4-8:4 exact
# thread name=mpiapp oncpu=4 cpurel=1-3 exact


# create 2 rank tasks, each will pthread_create 3 more # ranks will be
on 4 and 8
#  thread children on 5,6,7   9,10,11
dplace -p placefile mpirun -np 2 ~cpw/bin/mpiapp -P 3 -l


exit
```

You can use the debugger to determine if it is working. It should show two MPI rank applications, each with three pthreads, as follows:

```
>> pthreads | grep mpiapp
px *(task_struct *)e00002343c528000   17769   17769   17763  0       mpiapp
      member task: e000013817540000   17795   17769   17763  0     5 mpiapp
      member task: e000013473aa8000   17796   17769   17763  0      6 mpiapp
      member task: e000013817c68000   17798   17769   17763  0        mpiapp
px *(task_struct *)e0000234704f0000   17770   17770   17763  0        mpiapp
      member task: e000023466ed8000   17794   17770   17763  0      9 mpiapp
      member task: e00002384cce0000   17797   17770   17763  0        mpiapp
      member task: e00002342c448000   17799   17770   17763  0        mpiapp
```

And you can use the debugger, to see a root application, the parent of the two MPI rank applications, as follows:

```
>> ps | grep mpiapp
0xe00000340b300000   1139  17763  17729      1  0xc800000    -  mpiapp
0xe00002343c528000   1139  17769  17763      0  0xc800040    -  mpiapp
0xe0000234704f0000   1139  17770  17763      0  0xc800040    8  mpiapp
```

Placed as specified:

```
>> oncpus e00002343c528000 e000013817540000 e000013473aa8000
>> e000013817c68000 e0
000234704f0000 e000023466ed8000 e00002384cce0000 e00002342c448000
task: 0xe00002343c528000  mpiapp cpus_allowed: 4
task: 0xe000013817540000  mpiapp cpus_allowed: 5
task: 0xe000013473aa8000  mpiapp cpus_allowed: 6
task: 0xe000013817c68000  mpiapp cpus_allowed: 7
task: 0xe0000234704f0000  mpiapp cpus_allowed: 8
task: 0xe000023466ed8000  mpiapp cpus_allowed: 9
task: 0xe00002384cce0000  mpiapp cpus_allowed: 10
task: 0xe00002342c448000  mpiapp cpus_allowed: 11
```

`dplace` for Compute Thread Placement Troubleshooting Case Study

This section describes common reasons why compute threads do not end up on unique processors when using commands such a `dplace(1)` or `profile.pl` (see “Profiling with PerfSuite” in Chapter 3).

In the example that follows, a user used the `dplace -s1 -c0-15` command to bind 16 processes to run on 0–15 CPUs. However, output from the `top(1)` command shows only 13 CPUs running with CPUs 13, 14, and 15 still idle and CPUs 0, 1 and 2 are shared with 6 processes.

```
263 processes: 225 sleeping, 18 running, 3 zombie, 17 stopped
CPU states:  cpu    user    nice  system    irq  softirq  iowait    idle
             total  1265.6%    0.0%   28.8%   0.0%    11.2%    0.0%  291.2%

             cpu00  100.0%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

             cpu01   90.1%    0.0%    0.0%   0.0%     9.7%    0.0%    0.0%

             cpu02   99.9%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

             cpu03   99.9%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

             cpu04  100.0%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

             cpu05  100.0%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%
```

```
              cpu06  100.0%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

              cpu07   88.4%    0.0%   10.6%   0.0%     0.8%    0.0%    0.0%

              cpu08  100.0%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

              cpu09   99.9%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

              cpu10   99.9%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

              cpu11   88.1%    0.0%   11.2%   0.0%     0.6%    0.0%    0.0%

              cpu12   99.7%    0.0%    0.2%   0.0%     0.0%    0.0%    0.0%

              cpu13    0.0%    0.0%    2.5%   0.0%     0.0%    0.0%   97.4%

              cpu14    0.8%    0.0%    1.6%   0.0%     0.0%    0.0%   97.5%

              cpu15    0.0%    0.0%    2.4%   0.0%     0.0%    0.0%   97.5%
 Mem:  60134432k av, 15746912k used, 44387520k free,       0k shrd,
672k buff
         351024k active,            13594288k inactive

  Swap: 2559968k av,       0k used, 2559968k free
  2652128k cached

    PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME CPU COMMAND

   7653 ccao      25   0  115G 586M  114G R    99.9  0.9   0:08   3 mocassin

  7656 ccao      25   0  115G 586M  114G R    99.9  0.9   0:08   6 mocassin

   7654 ccao      25   0  115G 586M  114G R    99.8  0.9   0:08   4 mocassin

   7655 ccao      25   0  115G 586M  114G R    99.8  0.9   0:08   5 mocassin

   7658 ccao      25   0  115G 586M  114G R    99.8  0.9   0:08   8 mocassin

   7659 ccao      25   0  115G 586M  114G R    99.8  0.9   0:08   9 mocassin

   7660 ccao      25   0  115G 586M  114G R    99.8  0.9   0:08  10 mocassin

   7662 ccao      25   0  115G 586M  114G R    99.7  0.9   0:08  12 mocassin

   7657 ccao      25   0  115G 586M  114G R    88.5  0.9   0:07   7 mocassin

   7661 ccao      25   0  115G 586M  114G R    88.3  0.9   0:07  11 mocassin

   7649 ccao      25   0  115G 586M  114G R    55.2  0.9   0:04   2 mocassin

   7651 ccao      25   0  115G 586M  114G R    54.1  0.9   0:03   1 mocassin

   7650 ccao      25   0  115G 586M  114G R    50.0  0.9   0:04   0 mocassin

   7647 ccao      25   0  115G 586M  114G R    49.8  0.9   0:03   0 mocassin

   7652 ccao      25   0  115G 586M  114G R    44.7  0.9   0:04   2 mocassin

   7648 ccao      25   0  115G 586M  114G R    35.9  0.9   0:03   1 mocassin
```

An application can start some threads executing for a very short time yet the threads still have taken a token in the CPU list. Then, when the compute threads are finally started, the list is exhausted and restarts from the beginning. Consequently, some threads end up sharing the same CPU. To bypass this, try to eliminate the "ghost" thread creation, as follows:

- Check for a call to the "system" function. This is often responsible for the placement failure due to unexpected thread creation.

- When all the compute processes have the same name, you can do this by issuing a command, such as the following:

```
dplace -c0-15 -n compute-process-name ...
```

- You can also run `dplace -e -c0-32` on 16 CPUs to understand the pattern of the thread creation. If by chance, this pattern is the same from one run to the other (unfortunately race between thread creation often occurs), you can find the right flag to `dplace`. For example, if you want to run on CPU 0-3, with `dplace -e -C0-16` and you see that threads are always placed on CPU 0, 1, 5, and 6, then `dplace -e -c0,1,x,x,x,2,3` or `dplace -x24 -c0-3` (24 =11000, place the 2 first and skip 3 before placing) should place your threads correctly.

`dlook` Command

You can use `dlook(1)` to find out where in memory the operating system is placing your application's pages and how much system and user CPU time it is consuming.

Using the dlook Command

The `dlook(1)` command allows you to display the memory map and CPU usage for a specified process as follows:

```
dlook [-a] [-p] [-h] [-l] [-n] [-o outfile] [-s secs] command [command-args]
dlook [-a] [-p] [-h] [-l] [-n] [-o outfile] [-s secs] pid
```

For each page in the virtual address space of the process, `dlook(1)` prints the following information:

- The object that owns the page, such as a file, SYSV shared memory, a device driver, and so on.
- The type of page, such as random access memory (RAM), FETCHOP, IOSPACE, and so on.
- If the page type is RAM memory, the following information is displayed:
    - Memory attributes, such as, SHARED, DIRTY, and so on
    - The node on which the page is located
    - The physical address of the page (optional)
- Optionally, the `dlook(1)` command also prints the amount of user and system CPU time that the process has executed on each physical CPU in the system.

Two forms of the `dlook(1)` command are provided. In one form, `dlook` prints information about an existing process that is identified by a process ID (PID). To use this form of the command, you must be the owner of the process or be running with root privilege. In the other form, you use `dlook` on a command you are launching and thus are the owner.

The `dlook(1)` command accepts the following options:

- `-a`: Shows the physical addresses of each page in the address space.
- `-h`: Explicitly lists holes in the address space.
- `-l`: Shows libraries.
- `-p`: Show raw hardware page table entries.
- `-o`: Outputs to file name (*outfile*). If not specified, output is written to stdout.
- `-s`: Specifies a sample interval in seconds. Information about the process is displayed every second (`secs`) of CPU usage by the process.

An example for the `sleep` process with a PID of 4702 is as follows:

Note: The output has been abbreviated to shorten the example and bold headings added for easier reading.

```
dlook 4702

Peek:  sleep
Pid: 4702       Thu Aug 22 10:45:34 2002

Cputime by cpu (in seconds):
                  user    system
  TOTAL          0.002     0.033
  cpu1           0.002     0.033

Process memory map:
  2000000000000000-2000000000030000 r-xp 0000000000000000 04:03 4479 /lib/ld-2.2.4.so
       [2000000000000000-200000000002c000]        11 pages on node   1  MEMORY|SHARED

  2000000000030000-200000000003c000 rw-p 0000000000000000 00:00 0
       [2000000000030000-200000000003c000]         3 pages on node   0  MEMORY|DIRTY

                                          ...

  2000000000128000-2000000000370000 r-xp 0000000000000000 04:03 4672       /lib/libc-2.2.4.so
       [2000000000128000-2000000000164000]        15 pages on node   1  MEMORY|SHARED
       [2000000000174000-2000000000188000]         5 pages on node   2  MEMORY|SHARED
       [2000000000188000-2000000000190000]         2 pages on node   1  MEMORY|SHARED
       [200000000019c000-20000000001a8000]         3 pages on node   1  MEMORY|SHARED
       [20000000001c8000-20000000001d0000]         2 pages on node   1  MEMORY|SHARED
       [20000000001fc000-2000000000204000]         2 pages on node   1  MEMORY|SHARED
       [200000000020c000-2000000000230000]         9 pages on node   1  MEMORY|SHARED
       [200000000026c000-2000000000270000]         1 page  on node   1  MEMORY|SHARED
       [2000000000284000-2000000000288000]         1 page  on node   1  MEMORY|SHARED
       [20000000002b4000-20000000002b8000]         1 page  on node   1  MEMORY|SHARED
       [20000000002c4000-20000000002c8000]         1 page  on node   1  MEMORY|SHARED
       [20000000002d0000-20000000002d8000]         2 pages on node   1  MEMORY|SHARED
       [20000000002dc000-20000000002e0000]         1 page  on node   1  MEMORY|SHARED
       [2000000000340000-2000000000344000]         1 page  on node   1  MEMORY|SHARED
       [200000000034c000-2000000000358000]         3 pages on node   2  MEMORY|SHARED

                                               ....

  20000000003c8000-20000000003d0000 rw-p 0000000000000000 00:00 0
       [20000000003c8000-20000000003d0000]         2 pages on node   0  MEMORY|DIRTY
```

The `dlook` command gives the name of the process (`Peek: sleep`), the process ID, and time and date it was invoked. It provides total user and system CPU time in seconds for the process.

Under the heading **Process memory map**, the `dlook` command prints information about a process from the `/proc/`*`pid`*`/cpu` and `/proc/ `*`pid`*`/maps` files. On the left, it shows the memory segment with the offsets below in decimal. In the middle of the output page, it shows the type of access, time of execution, the PID, and the object that owns the memory (in this case, `/lib/ld-2.2.4.so`). The characters `s` or `p` indicate whether the page is mapped as sharable (`s`) with other processes or is private (`p`). The right side of the output page shows the number of pages of memory consumed and on which nodes the pages reside. A page is 16, 384 bytes. *Dirty memory* means that the memory has been modified by a user.

In the second form of the `dlook` command, you specify a command and optional command arguments. The `dlook` command issues an `exec` call on the command and passes the command arguments. When the process terminates, `dlook` prints information about the process, as shown in the following example:

```
dlook date

Thu Aug 22 10:39:20 CDT 2002
_______________________________________________________________________________
Exit:  date
Pid: 4680       Thu Aug 22 10:39:20 2002


Process memory map:
  2000000000030000-200000000003c000 rw-p 0000000000000000 00:00 0
        [2000000000030000-200000000003c000]         3 pages on node   3  MEMORY|DIRTY

  20000000002dc000-20000000002e4000 rw-p 0000000000000000 00:00 0
        [20000000002dc000-20000000002e4000]         2 pages on node   3  MEMORY|DIRTY

  2000000000324000-2000000000334000 rw-p 0000000000000000 00:00 0
        [2000000000324000-2000000000328000]         1 page  on node   3  MEMORY|DIRTY

  4000000000000000-400000000000c000 r-xp 0000000000000000 04:03 9657220    /bin/date
        [4000000000000000-400000000000c000]         3 pages on node   1  MEMORY|SHARED

  6000000000008000-6000000000010000 rw-p 0000000000008000 04:03 9657220    /bin/date
        [600000000000c000-6000000000010000]         1 page  on node   3  MEMORY|DIRTY

  6000000000010000-6000000000014000 rwxp 0000000000000000 00:00 0
        [6000000000010000-6000000000014000]         1 page  on node   3  MEMORY|DIRTY

  60000fff80000000-60000fff80004000 rw-p 0000000000000000 00:00 0
        [60000fff80000000-60000fff80004000]         1 page  on node   3  MEMORY|DIRTY

  60000ffffff4000-60000fffffffc000 rwxp ffffffffffffc000 00:00 0
        [60000ffffff4000-60000fffffffc000]         2 pages on node   3  MEMORY|DIRTY
```

If you use the `dlook` command with the `-s` *`secs`* option, the information is sampled at regular internals. The output for the command **`dlook -s 5 sleep 50`** is as follows:

```
Exit:  sleep
Pid: 5617       Thu Aug 22 11:16:05 2002


Process memory map:
  2000000000030000-200000000003c000 rw-p 0000000000000000 00:00 0
        [2000000000030000-200000000003c000]            3 pages on node   3  MEMORY|DIRTY
  2000000000134000-2000000000140000 rw-p 0000000000000000 00:00 0

  20000000003a4000-20000000003a8000 rw-p 0000000000000000 00:00 0
        [20000000003a4000-20000000003a8000]            1 page  on node   3  MEMORY|DIRTY

  20000000003e0000-20000000003ec000 rw-p 0000000000000000 00:00 0
        [20000000003e0000-20000000003ec000]            3 pages on node   3  MEMORY|DIRTY

  4000000000000000-4000000000008000 r-xp 0000000000000000 04:03 9657225    /bin/sleep
        [4000000000000000-4000000000008000]            2 pages on node   3  MEMORY|SHARED

  6000000000004000-6000000000008000 rw-p 0000000000004000 04:03 9657225    /bin/sleep
        [6000000000004000-6000000000008000]            1 page  on node   3  MEMORY|DIRTY

  6000000000008000-600000000000c000 rwxp 0000000000000000 00:00 0
        [6000000000008000-600000000000c000]            1 page  on node   3  MEMORY|DIRTY

  60000fff80000000-60000fff80004000 rw-p 0000000000000000 00:00 0
        [60000fff80000000-60000fff80004000]            1 page  on node   3  MEMORY|DIRTY

  60000ffffff4000-60000fffffffc000 rwxp ffffffffffffc000 00:00 0
        [60000ffffff4000-60000fffffffc000]            2 pages on node   3  MEMORY|DIRTY
```

You can run a Message Passing Interface (MPI) job using the `mpirun` command and print the memory map for each thread, or redirect the ouput to a file, as follows:

Note: The output has been abbreviated to shorten the example and bold headings added for easier reading.

```
mpirun -np 8 dlook -o dlook.out ft.C.8

Contents of dlook.out:
_______________________________________________________________________________
Exit:  ft.C.8
Pid: 2306       Fri Aug 30 14:33:37 2002


Process memory map:
  2000000000030000-200000000003c000 rw-p 0000000000000000 00:00 0
        [2000000000030000-2000000000034000]            1 page  on node  21  MEMORY|DIRTY
        [2000000000034000-200000000003c000]            2 pages on node  12  MEMORY|DIRTY|SHARED

  2000000000044000-2000000000060000 rw-p 0000000000000000 00:00 0
        [2000000000044000-2000000000050000]            3 pages on node  12  MEMORY|DIRTY|SHARED
                                            ...
_______________________________________________________________________________
_______________________________________________________________________________
Exit:  ft.C.8
Pid: 2310       Fri Aug 30 14:33:37 2002


Process memory map:
  2000000000030000-200000000003c000 rw-p 0000000000000000 00:00 0
        [2000000000030000-2000000000034000]            1 page  on node  25  MEMORY|DIRTY
        [2000000000034000-200000000003c000]            2 pages on node  12  MEMORY|DIRTY|SHARED

  2000000000044000-2000000000060000 rw-p 0000000000000000 00:00 0
        [2000000000044000-2000000000050000]            3 pages on node  12  MEMORY|DIRTY|SHARED
        [2000000000050000-2000000000054000]            1 page  on node  25  MEMORY|DIRTY

                                              ...
_______________________________________________________________________________
_______________________________________________________________________________
Exit:  ft.C.8
```

```
Pid: 2307       Fri Aug 30 14:33:37 2002


Process memory map:
  2000000000030000-200000000003c000 rw-p 0000000000000000 00:00 0
        [2000000000030000-2000000000034000]            1 page  on node  30  MEMORY|DIRTY
        [2000000000034000-200000000003c000]            2 pages on node  12  MEMORY|DIRTY|SHARED

  2000000000044000-2000000000060000 rw-p 0000000000000000 00:00 0
        [2000000000044000-2000000000050000]            3 pages on node  12  MEMORY|DIRTY|SHARED
        [2000000000050000-2000000000054000]            1 page  on node  30  MEMORY|DIRTY
                                              ...
________________________________________________________________________________
________________________________________________________________________________
Exit:  ft.C.8
Pid: 2308       Fri Aug 30 14:33:37 2002


Process memory map:
  2000000000030000-200000000003c000 rw-p 0000000000000000 00:00 0
        [2000000000030000-2000000000034000]            1 page  on node   0  MEMORY|DIRTY
        [2000000000034000-200000000003c000]            2 pages on node  12  MEMORY|DIRTY|SHARED

  2000000000044000-2000000000060000 rw-p 0000000000000000 00:00 0
        [2000000000044000-2000000000050000]            3 pages on node  12  MEMORY|DIRTY|SHARED
        [2000000000050000-2000000000054000]            1 page  on node   0  MEMORY|DIRTY
                                             ...
```

For more information on the `dlook` command, see the `dlook` man page.

`omplace` Command

The `omplace(1)` command is a tool for controlling the placement of MPI processes and OpenMP threads.

The `omplace` command causes the successive threads in a hybrid MPI/OpenMP job to be placed on unique CPUs. The CPUs are assigned in order from the effective CPU list within the containing cpuset.

This command is a wrapper script for `dplace (1)` that can be used with MPI, OpenMP, pthreads, and hybrid MPI/OpenMP and MPI/pthreads codes. It generates the proper `dplace` placement file syntax automatically. It also supports some unique options like block-strided CPU lists.

The CPU placement is performed by dynamically generating a placement file and invoking `dplace` with the MPI job launch. For example, the threads in a 2-process MPI program with 2 threads per process would be placed, as follows:

```
rank 0 thread 0 on CPU 0
rank 0 thread 1 on CPU 1
rank 1 thread 0 on CPU 2
rank 1 thread 1 on CPU 3
```

For more information, see the `omplace(1)` man page and “Run-Time Tuning” chapter in the *Message Passing Toolkit (MPT) User's Guide*.

`numactl` Command

The `numactl(8)` command runs processes with a specific NUMA scheduling or memory placement policy. The policy is set for command and inherited by all of its children. In addition it can set persistent policy for shared memory segments or files. For more information, see the `numactl(8)` man page.

Installing NUMA Tools

To use the `dlook(1)`, `dplace(1)`, and `topology (1)` commands, you must load the `numatools` kernel module. Perform the following steps:

1. To configure `numatools` kernel module to be started automatically during system startup, use the `chkconfig(8)` command as follows:

   ```
   chkconfig --add numatools
   ```

2. To turn on `numatools`, enter the following command:

   ```
   /etc/rc.d/init.d/numatools start
   ```

   This step will be done automatically for subsequent system reboots when `numatools` are configured on by using the `chkconfig(8)` utility.

The following steps are required to disable `numatools`:

1. To turn off `numatools`, enter the following:

   ```
   /etc/rc.d/init.d/numatools stop
   ```

2. To stop `numatools` from initiating after a system reboot, use the `chkconfig (8)` command as follows:

   ```
   chkconfig --del numatools
   ```

An Overview of the Advantages Gained by Using Cpusets

The cpuset facility is primarily a workload manager tool permitting a system administrator to restrict the number of processor and memory resources that a process or set of processes may use. A *cpuset* defines a list of CPUs and memory nodes. A process contained in a cpuset may only execute on the CPUs in that cpuset and may only allocate memory on the memory nodes in that cpuset. Essentially, cpusets provide you with a CPU and memory containers or “soft partitions” within which you can run sets of related tasks. Using cpusets on an SGI Altix UV system improves cache locality and memory access times and can substantially improve an application's performance and runtime repeatability. Restraining all other jobs from using any of the CPUs or memory resources assigned to a critical job minimizes interference from other jobs on the system. For example, Message Passing Interface (MPI) jobs frequently consist of a number of threads that communicate using message passing interfaces. All threads need to be executing at the same time. If a single thread loses a CPU, all threads stop making forward progress and spin at a barrier.

Cpusets can eliminate the need for a gang scheduler, provide isolation of one such job from other tasks on a system, and facilitate providing equal resources to each thread in a job. This results in both optimum and repeatable performance.

In addition to their traditional use to control the placement of jobs on the CPUs and memory nodes of a system, cpusets also provide a convenient mechanism to control the use of Hyper–Threading Technology.

Cpusets are represented in a hierarchical virtual file system. Cpusets can be nested and they have file–like permissions.

The `sched_setaffinity`, `mbind`, and `set_mempolicy` system calls allow you to specify the CPU and memory placement for individual tasks. On smaller or limited–use systems, these calls may be sufficient.

The kernel cpuset facility provides additional support for system–wide management of CPU and memory resources by related sets of tasks. It provides a hierarchical structure to the resources, with filesystem–like namespace and permissions, and support for guaranteed exclusive use of resources.

You can have a *boot* cpuset running the traditional daemon and server tasks and a second cpuset to hold interactive `telnet`, `rlogin` and/or secure shell (SSH) user sessions called the *user* cpuset.

Creating a user cpuset provides additional isolation between interactive user login sessions and essential system tasks. For example, a user process in the user cpuset consuming excessive CPU or memory resources will not seriously impact essential system services in the boot cpuset.

This section covers the following topics:

Linux 2.6 Kernel Support for Cpusets

The Linux 2.6 kernel provides the following support for cpusets:

- Each task has a link to a cpuset structure that specifies the CPUs and memory nodes available for its use.
- Hooks in the `sched_setaffinity` system call, used for CPU placement, and in the `mbind` system call, used for memory placement, ensure that any requested CPU or memory node is available in that task's cpuset.
- All tasks sharing the same placement constraints reference the same cpuset.
- Kernel cpusets are arranged in a hierarchical virtual file system, reflecting the possible nesting of "soft partitions".
- The kernel task scheduler is constrained to only schedule a task on the CPUs in that task's cpuset.
- The kernel memory allocation mechanism is constrained to only allocate physical memory to a task from the memory nodes in that task's cpuset.
- The kernel memory allocation mechanism provides an economical, per–cpuset metric of the aggregate memory pressure of the tasks in a cpuset. *Memory pressure* is defined as the frequency of requests for a free memory page that is not easily satisfied by an available free page.
- The kernel memory allocation mechanism provides an option that allows you to request that memory pages used for file I/O (the kernel page cache) and associated kernel data structures for file inodes and directories be evenly spread across all the memory nodes in a cpuset. Otherwise, they are preferentially allocated on whatever memory node that the task first accessed the memory page.
- You can control the memory migration facility in the kernel using per–cpuset files. When the memory nodes allowed to a task by cpusets changes, any memory pages no longer allowed on that node may be migrated to nodes now allowed. For more information, see "Safe Job Migration and Cpusets".

Cpuset Facility Capabilities

A cpuset constrains the jobs (set of related tasks) running in it to a subset of the system's memory and CPUs. The cpuset facility allows you and your system service software to do the following:

- Create and delete named cpusets.
- Decide which CPUs and memory nodes are available to a cpuset.
- Attach a task to a particular cpuset.
- Identify all tasks sharing the same cpuset.
- Exclude any other cpuset from overlapping a given cpuset, thereby, giving the tasks running in that cpuset exclusive use of those CPUs and memory nodes.
- Perform bulk operations on all tasks associated with a cpuset, such as varying the resources available to that cpuset or hibernating those tasks in temporary favor of some other job.
- Perform sub–partitioning of system resources using hierarchical permissions and resource management.

Initializing Cpusets

The kernel, at system boot time, initializes one cpuset, the root cpuset, containing the entire system's CPUs and memory nodes. Subsequent user space operations can create additional cpusets.

Mounting the cpuset virtual file system (VFS) at `/dev/cpuset` exposes the kernel mechanism to user space. This VFS allows for nested resource allocations and the associated hierarchical permission model.

You can initialize and perform other cpuset operations, using any of the these three mechanisms, as follows:

- You can create, change, or query cpusets by using shell commands on `/dev/cpuset`, such as `echo(1)`, `cat(1)`, `mkdir(1)`, or `ls(1)`.
- You can use the `cpuset(1)` command line utility to create or destroy cpusets or to retrieve information about existing cpusets and to attach processes to existing cpusets.
- You can use the `libcpuset` C programming application programming interface (API) functions to query or change them from within your application. You can find information about `libcpuset` at `/usr/share/doc/packages/libcpuset/libcpuset.html` .

How to Determine if Cpusets are Installed

You can issue several commands to determine whether cpusets are installed on your system, as follows:

1. Use the `grep(1)` command to search the`/proc/filesystems` for cpusets, as follows:

```
% grep cpuset /proc/filesystems
nodev   cpuset
```

2. Determine if cpuset `tasks` file is present on your system by changing directory to `/dev/cpuset` and listing the content of the directory, as follows:

```
% cd /dev/cpuset
Directory: /dev/cpuset

% ls
cpu_exclusive  cpus  mem_exclusive  mems  notify_on_release
pagecache_list  pagecache_local  slabcache_local  tasks
```

3. If the `/dev/cpuset/tasks` file is not present on your system, it means the cpuset file system is not mounted (usually, it is automatically mounted when the system was booted). As root, you can mount the cpuset file system, as follows:

```
% mount -t cpuset cpuset /dev/cpuset
```

Fine-grained Control within Cpusets

Within a single cpuset, use facilities such as `taskset(1)`, `dplace(1)`, first-touch memory placement, pthreads, `sched_setaffinity` and `mbind` to manage processor and memory placement to a more fine-grained level.

The user-level bitmask library supports convenient manipulation of multiword bitmasks useful for CPUs and memory nodes. This bitmask library is required by and designed to work with the cpuset library. You can find information on the bitmask library on your system at `/usr/share/doc/packages/libbitmask/libbitmask.html` .

Cpuset Interaction with Other Placement Mechanism

The Linux 2.6 kernel supports additional processor and memory placement mechanisms, as follows:

Note: Use the `uname`(1) command to print out system information to make sure you are running the Linux 2.6.x sn2 kernel, as follows:

```
% uname -r -s
Linux 2.6.27.19-5-default
```

- The `sched_setaffinity(2)` and `sched_getaffinity(2)` system calls set and get the CPU affinity mask of a process. This determines the set of CPUs on which the process is eligible to run. The `taskset(1)` command provides a command line utility for manipulating the CPU affinity mask of a process using these system calls. For more information, see the appropriate man page.
- The `set_mempolicy` system call sets the NUMA memory policy of the current process to *policy*. A NUMA machine has different memory controllers with different distances to specific CPUs. The memory *policy* defines in which node memory is allocated for the process.

  The `get_mempolicy`(2) system retrieves the NUMA policy of the calling process or of a memory address, depending on the setting of *flags*. The `numactl`(8) command provides a command line utility for manipulating the NUMA memory policy of a process using these system calls.
- The `mbind`(2) system call sets the NUMA memory policy for the pages in a specific range of a task's virtual address space.

Cpusets are designed to interact cleanly with other placement mechanisms. For example, a batch manager can use cpusets to control the CPU and memory placement of various jobs; while within each job, these other kernel mechanisms are used to manage placement in more detail. It is possible for a batch manager to change a job's cpuset placement while preserving the internal CPU affinity and NUMA memory placement policy, without requiring any special coding or awareness by the affected job.

Most jobs initialize their placement early in their timeslot, and jobs are rarely migrated until they have been running for a while. As long a batch manager does **not** try to migrate a job at the same time as it is adjusting its own CPU or memory placement, there is little risk of interaction between cpusets and other kernel placement mechanisms.

The CPU and memory node placement constraints imposed by cpusets always override those of these other mechanisms.

Calls to the `sched_setaffinity`(2) system call automatically mask off CPUs that are not allowed by the affected task's cpuset. If a request results in all the CPUs being masked off, the call fails with errno set to `EINVAL`. If some of the requested CPUs are allowed by the task's cpuset, the call proceeds as if only the allowed CPUs were requested. The unallowed CPUs are silently ignored. If a task is moved to a different cpuset, or if the CPUs of a cpuset are changed, the CPU affinity of the affected task or tasks is lost. If a batch manager needs to preserve the CPU affinity of the tasks in a job that is being moved, it should use the `sched_setaffinity`(2) and `sched_getaffinity`(2) calls to save and restore each affected task's CPU affinity across the move, relative to the cpuset. The `cpu_set_t` mask data type supported by the C library for use with the CPU affinity calls is different from the `libbitmask` bitmasks used by `libcpuset`, so some coding will be required to convert between the two, in order to calculate and preserve cpuset relative CPU affinity.

Similar to CPU affinity, calls to modify a task's NUMA memory policy silently mask off requested memory nodes outside the task's allowed cpuset, and will fail if that results in requested an empty set of memory nodes. Unlike CPU affinity, the NUMA memory policy system calls to not support one task querying or modifying another task's policy. So the kernel automatically handles preserving cpuset relative NUMA memory policy when either a task is attached to a different cpuset, or a cpusets `mems` value setting is changed. If the old and new `mems` value sets have the same size, the cpuset relative offset of affected NUMA memory policies is preserved. If the new `mems` value is smaller, the old `mems` value relative offsets are folded onto the new `mems` value, modulo the size of the new `mems`. If the new `mems` value is larger, then just the first N nodes are used, where N is the size of the old `mems` value.

Cpusets and Thread Placement

If your job uses the placement mechanisms described in "Cpuset Interaction with Other Placement Mechanism" and operates under the control of a batch manager, you **cannot** guarantee that a migration will preserve placement done using the mechanisms. These placement mechanisms use system wide numbering of CPUs and memory nodes, not cpuset relative numbering and the job might be migrated without its knowledge while it is trying to adjust its placement. That is, between the point where an application computes the CPU or memory node on which it wants to place a thread and the point where it issues the `sched_setaffinity`(2), `mbind`(2) or `set_mempolicy`(2) call to direct such a placement, the thread might be migrated to a different cpuset, or its cpuset changed to different CPUs or memory nodes, invalidating the CPU or memory node number it just computed.

The `libcpuset` library provides the following mechanisms to support cpuset relative thread placement that is robust even if the job is being migrated using a batch scheduler.

If your job needs to pin a thread to a single CPU, you can use the convenient `cpuset_pin` function. This is the most common case.

If your job needs to implement some other variation of placement, such as to specific memory nodes, or to more than one CPU, you can use the following functions to safely guard such code from placement changes caused by job migration, as follows:

- `cpuset_get_placement`
- `cpuset_equal_placement`

- `cpuset_free_placement`

Safe Job Migration and Cpusets

Jobs that make use of cpuset aware thread pinning described in "Cpusets and Thread Placement" can be safely migrated to a different cpuset or have the CPUs or memory nodes of the cpuset safely changed without destroying the per-thread placement done within the job.

Procedure 5-1. Safe Job Migration Between Cpusets

To safely migrate a job to a different cpuset, perform the following steps:

1. Suspend the tasks in the job by sending their process group a `SIGSTOP` signal.
2. Use the `cpuset_init_pidlist` function and related `pidlist` functions to determine the list of tasks in the job.
3. Use `sched_getaffinity`(2) to query the CPU affinity of each task in the job.
4. Create a new cpuset, under a temporary name, with the new desired CPU and memory placement.
5. Invoke `cpuset_migrate_all` function to move the job's tasks from the old cpuset to the new cpuset.
6. Use `cpuset_delete` to delete the old cpuset.
7. Use `rename`(2) on the `/dev/cpuset` based path of the new temporary cpuset to rename that cpuset to the to the old cpuset name.
8. Convert the results of the previous `sched_getaffinity`(2) calls to the new cpuset placement, preserving cpuset relative offset by using the `cpuset_c_rel_to_sys_cpu` and related functions.
9. Use `sched_setaffinity`(2) to reestablish the per-task CPU binding of each thread in the job.
10. Resume the tasks in the job by sending their process group a `SIGCONT` signal.

The `sched_getaffinity`(2) and `sched_setaffinity` (2) C library calls are limited by C library internals to systems with 1024 CPUs or less. To write code that will work on larger systems, you should use the `syscall`(2) indirect system call wrapper to directly invoke the underlying system call, bypassing the C library API for these calls.

The suspend and resume operation are required in order to keep tasks in the job from changing their per thread CPU placement between steps three and six. The kernel automatically migrates the per-thread memory node placement during step four. This is necessary, because there is no way for one task to modify the NUMA memory placement policy of another task. The kernel does not automatically migrate the per-thread CPU placement, as this can be handled by the user level process doing the migration.

Migrating a job from a larger cpuset (more CPUs or nodes) to a smaller cpuset will lose placement information and subsequently moving that cpuset back to a larger cpuset will **not** recover that information. This loss of CPU affinity can be avoided as described above, using `sched_getaffinity`(2) and `sched_setaffinity`(2) to save and restore the placement (affinity) across such a pair of moves. This loss of NUMA memory placement information cannot be avoided because one task (the one doing the migration) cannot save nor restore the NUMA memory placement policy of another. So if a batch manager wants to migrate jobs without causing them to lose their `mbind`(2) or `set_mempolicy`(2) placement, it should only migrate to cpusets with at least as many memory nodes as the original cpuset.

For detailed information about using cpusets, see The “Cpusets on Linux” chapter in the *Linux Resource Administration Guide* at at http://docs.sgi.com .

Application Performance on Large Altix UV Systems

This section describes cpuset settings you should pay particular attention to when running applications on large SGI Altix UV 1000 series systems.

`mem_exclusive`

Flag (0 or 1). If set (1), the cpuset has exclusive use of its memory nodes (no sibling or cousin may overlap). Also if set (1), the cpuset is a hardwall cpuset. See “Hardwall” section in the `cpuset(7)` man page for more information. By default, this is off (0). Newly created cpusets also initially default this to off (0).

`mem_spreadpage`

Flag (0 or 1). If set (1), pages in the kernel page cache (file-system buffers) are uniformly spread across the cpuset. By default, this is off (0) in the top cpuset, and inherited from the parent cpuset in newly created cpusets. See the “Memory Spread” section in the `cpuset(7)` man page for more information.

`mem_hardwall`

Flag (0 or 1). If set (1), the cpuset is a Hardwall cpuset. See “Hardwall” section in the `cpuset(7)` man page for more information. Unlike `mem_exclusive`, there is no constraint on whether cpusets marked `mem_hardwall` may have overlapping memory nodes with sibling or cousin cpusets. By default, this is off (0). Newly created cpusets also initially default this to off (0).