An NUMA API for Linux

Andi Kleen
SUSE Labs
ak@suse.de

Aug 2004

1 What is NUMA API?

CC/NUMA (*Cache Coherent/Non Uniform Memory Access*) machines are becoming more common. Examples are AMD Opteron, IBM Power5 or SGI Altix. This is opposite to SMP (*Symmetric Multi Processing*) where all CPUs in the system have the same access to all memory.

A word on terminology: a node is a set of CPUs (Central Processing Unit) that all have equal fast access to some memory using a memory controller. On a system where CPUs have integrated memory controllers a node consists of a single CPU, possibly with multiple cores or virtual threads. On other more traditional NUMA systems like an SGI Altix or a HP Superdome bigger nodes with 2 to 4 CPUs are sharing memory.

In addition to the local memory NUMA machines have remote memory. This is memory located on other nodes, connected over a fast interconnect. From the software point of view this remote memory can be used in the same way as local memory; it is fully cache coherent. The only difference is that accessing it is slower because the interconnect is slower than the local memory bus of the node. In addition to these big server systems some embedded architectures also have memory with different performance.

NUMA policy is concerned about putting memory allocations on specific nodes to let programs access it as fast as possible. The primary way to do this is to allocate memory for a thread on its local node and keep the thread running there (*node affinity*) This gives the best latency for memory and minimizes traffic over the global interconnect.

SMP Systems try to optimize in similar ways to optimize use of the per CPU caches (*cache affinity*). However there is an important difference: on a SMP system when a thread moves between CPUs its cache contents will eventually move with it. On a NUMA system once a memory area is committed to a specific node it stays there and a thread running on a different node that accesses it will always add traffic to the interconnect and see higher latency.

However the scheduler can not always optimize purely for node affinity. The problem is that not using a CPU in the system would be even worse than a process using remote memory and seeing lower memory performance. In cases where memory performance is more important than even use of all CPUs in the system the application or the system administrator can override the default decisions of the scheduler and the memory allocator. This allows to optimize better for specific workloads.

Linux traditionally had ways to bind threads to specific CPUs (using the *sched_set_affinity(2)* system call and *schedutils*) NUMA API extends this to allow programs to specify on which node memory should be allocated.

The NUMA API described here separates placement of threads to CPUs and placement of memory. Primarily it is concerned about the placement of memory. In addition the application can configure CPU affinity separately. NUMA API is currently available on SUSE Linux Enterprise Server 9 for AMD64 and for IA64.

2 Optimizing Bandwidth

Memory access performance of programs can be optimized for latency or for bandwidth. Most programs seem to prefer lower latency, but there are a few exceptions that want bandwidth.

Using node local memory has the best latency. To get more bandwidth the memory controllers of multiple nodes can be used in parallel. This is similar to how RAID can improve disk IO performance spreading IO operations over multiple hard disks. NUMA API can use the MMU (*Memory Management Unit*) in the CPU to interleave blocks of memory from different memory controllers. This means each consecutive page[1] in such a mapping will come from a different node.

When the applications does a large streaming memory access to such an interleaved area it will see the bandwidth of the memory controllers of multiple nodes combined. How well this works depends on the particular NUMA architecture, in particular on the performance of the interconnect and the latency difference between local and remote memory. On some systems it only works effectively on a subset of neighboring nodes.

Some NUMA systems like Opteron can be configured by firmware to interleave all memory across all nodes on a page basis. This is called "Node Interleaving" by most firmware. "Node Interleaving" is similar to the interleaving mode offered by the NUMA API; however they differ in important ways. Node Interleaving applies to *all* memory while NUMA API interleaving can be configured for each process or thread individually. If Node Interleaving is enabled by firmware, then NUMA policy is disabled. With NUMA API each application can individually policy memory areas for latency or bandwidth.

3 Parts of implementation: Kernel/System calls, libnuma, numactl

NUMA API consists of different sub components: It has a kernel part that manages memory policy for processes or specific memory mappings. This kernel part can be controlled by three new system calls.

There is a user space shared library called libnuma that can be linked to applications. libnuma is the recommended interface to use NUMA policy. It offers a more user friendly and more abstracted interface than using the system calls directly. The following paper will only describe this higher level interface. It also has a command line utility *numactl* that can be used to control policy for an unmodified application and its children.

The user libraries and applications are included in the numactl rpm, which is included distribution since SUSE Linux Enterprise Server 9 [2] In addition there are some utility programs like *numastat* to collect statistics about the memory allocation and *numademo* to show the effect of different policies on the system. The package also contains man pages for all functions and programs.

[1] page is a 4K unit on an AMD64 or PPC64 system, normally 16k on IA64

[2] It was already included in SUSE Linux 9.1, but it is recommended to use the newer version from SLES9.

4 Policies

The main task of NUMA API is to manage policies. Policies can be applied to processes or to memory areas.

NUMA API currently supports four different policies:

Name	Description
default	Allocate on the local node.
bind	Allocate on a specific set of nodes.
interleave	Interleave memory allocations on a set of nodes.
preferred	Try to allocate on a node first.

The difference between *bind* and *preferred* is that bind will fail the allocation when the memory cannot be allocated on the specified nodes. *preferred* falls back to other nodes. Using bind can lead to earlier running out of memory and delays due to more swapping. In libnuma *preferred* and *bind* are folded into one and can be changed per thread with the *numa_set_strict* libnuma function. The default is non strict preferred allocation

Policies can be per process (p̂rocess policy) or per memory region Children inherit the process policy of the parent on fork. The process policy is applied to all memory allocations done in the context of the process. This includes kernel internal allocations done in system calls and the file cache. Interrupts always allocate on the current node. Process policy always applies when a page of memory is allocated by the kernel.

Per memory region policies (also called *VMA policies* [3]) allow a process to set a policy for a block of memory in its address space. Memory region policies have higher priority than the process policy. The main advantage of memory region policies is that they can be set up before an allocation happens. Currently they are only supported for anonymous process memory, SYSV shared memory, shmem/tmpfs mappings or hugetlbfs files. The region policy for shared memory is persistent until the shared memory segment or file is deleted.

5 Some simple numactl examples

numactl is a command line tool to run processes with a specific NUMA policy. Using it is useful to set policies for programs that cannot be modified and recompiled.

Here are some simple examples on how to use numactl

```
numactl --cpubind=0 --membind=0,1 program
```

Run program bound to the CPUs of node 0 and only allocating memory from node 0 or 1. Please note that cpubind uses node numbers, not CPU numbers. On a system with multiple CPUs per node these can differ.

```
numactl --preferred=1 numactl --show
```

Set preferred policy to node 1 and show the resulting state

```
numactl --interleave=all numbercruncher
```

Run memory bandwidth intensive number cruncher with memory interleaved over all available nodes.

```
numactl --offset=1G --length=1G --membind=1 --file /dev/shm/A --touch
```

Bind the second gigabyte in the tmpfs file /dev/shm/A to node 1.

[3] VMA stands for Virtual Memory Area and is region of virtual memory in a process.

```
numactl --localalloc /dev/shm/file
```

Reset the policy for the shared memory file /dev/shm/file.

```
numactl --hardware
```

Print an overview over the available nodes.

6 numactl: important command line switches

Here is a quick overview of the important command line switches of numactl.

Many of these switches need a *node mask* as argument. Each node in the system has a unique number. A node mask can be a comma separated list of node numbers, a range of nodes (*node1-node2*) or *all*. See *numactl –hardware* for the nodes available on the current system.

Most common usage is to set policy for a process: The policy is passed as first argument and afterwards the program name and its argument. The available policy switches are:

–**membind=nodemask** Only allocate memory on the nodes in *nodemask*.

–**interleave=nodemask** Interleave all memory allocations over nodes in *nodemask*

–**cpubind=nodemask** Execute process only on the CPUs of the nodes specified in *tnodemask*. Note that *–cpubind* can be specified in addition with other policies because it separately affects the scheduler.

–**preferred=node** Allocate memory preferable on node *node*.

numactl –show prints the current process NUMA state as inherited from the parent shell.

numactl –hardware gives an overview of the available NUMA resources on the system.

For more details see the *numactl(8)* man page.

7 numactl: shared memory

So far numactl has only been used to change the default policy of a process. In addition it is also able to change policies in shared memory segments. This is useful to change the policy of an application

An example would be a multi process program that uses a common shared memory segment. For the individual processes it is best to use the default policy of allocating memory on their current nodes. This way they get the best memory latency for their local data structure. But the shared memory segment is shared by multiple processes who run on different nodes. To avoid a hot spot on the node which allocated the memory originally it may be an advantage to set interleaved policy for the whole shared memory segment. This way all the accesses to it should be spread out evenly over all nodes.

More complex policies are possibly. When parts of the shared memory are mostly used by specific processes and are only accessed rarely by others they could be bound to specific nodes or only interleaved to a subset of nodes which are near to each other.

Shared memory here can be SYSV shared memory (from the *shmat* system call), mmaped files in tmpfs or shmfs (normally in /dev/shm on a SUSE system)

or a hugetlbfs file. hugetlbfs will do the policy in huge pages granularity (2MB on AMD64 systems). The shared memory policy can be set up before the application starts[4]. The policy will stay assigned to areas in the shared memory segment until it is deleted.

The set policy only applies to new pages allocated. Already existing pages in the shared memory segment will not be moved to conform to the policy.

Set up a 1GB tmpfs file to interleave its memory over all nodes.

```
numactl --length=1G --file=/dev/shm/interleaved --interleave=all
```

An hugetlbfs file can be set up in the same way, although all lengths must be multiplies of the huge page size of the system[5]

An offset into the shared memory segment or file can be specified with *–offset=number*. All numeric arguments can have unit prefixes: *G* for Gigabytes, *M* for Megabytes, *K* for KBytes. The mode of the new file can be specified with –mode=mode

Alternatively this can be enforced with the *–strict* option. When *–strict* is set and an already allocated page doesn't conform to the new policy numactl will report an error. numactl has several more options to control the type of the shared memory segment. For details see the *numactl(8)* man page.

8 libnuma: basics, checking for NUMA

So far we have described numactl which controls the policy of whole processes. The disadvantage of numactl is that the policy applies to the whole program, not to individual memory areas (except for shared memory) or threads. For some programs more fine grained policy control is needed.

This can be done with *libnuma*. libnuma is a shared library that can be linked to programs and offers a stable API for NUMA policy. It provides a higher level interface than using the NUMA API system calls directly and is the recommended interface for applications. *libnuma* is part of the *numactl* rpm.

Applications link with libnuma as follows:

```
cc ... -lnuma
```

The NUMA API functions and macros are declared in the *numa.h* include file.

Before any NUMA API functions can be used the program has to call *numa_available()*. When this function returns a negative value, there is no NUMA policy support on the system. In this case the behavior of all other NUMA API functions is undefined and should not be called.

```
#include <numa.h>
   ...
   if (numa_available() < 0) {
       printf(``Your system does not support NUMA API\n'');
       ...
   }
   ...
```

The next step is usually to call *numa_max_node()*. This function discovers and returns the number of nodes in the system. A word on thread safety: All libnuma state is kept local per thread. Changing a policy in one thread will not affect the other threads in the process.

[4]This assumes the application doesn't insist on creating the shared memory segment itself.

[5]grep Hugepagesize /proc/meminfo gives the huge page size of the current system.

The following sections give an overview of the various *libnuma* functions with some examples. For a more detailed reference please see the *numa(3)* man page. A few more obscure ones are not described here.

9 libnuma: nodemasks

libnuma manages sets of nodes in abstract data types called *nodemask_t* defined in *numa.h*. A *nodemask_t* is a simple fixed size bit set of node numbers. Each node in the system has a unique number. The highest number is the number returned by *numa_max_node()* The maximum size is implementation defined in the *NUMA_NUM_NODES* constant. It is passed by reference to many NUMA API functions.

A node mask is initialized to the empty with nodemask_zero()

```
nodemask_t mask;
nodemask_zero(&mask);
```

An single node can be set with *nodemask_set* and cleared with *nodemask_clr*. *nodemask_equal* compares two nodemasks. *nodemask_isset* tests if a bit is set in the node mask.

```
nodemask_set(&mask, maxnode);        /* set node highest */
if (nodemask_isset(&mask, 1)) {      /* is node 1 set? */
    ...
}
nodemask_clr(&mask, maxnode);   /* clear highest node again */
```

There are two predefined node masks: *numa_all_nodes* stands for all nodes in the system and *numa_no_nodes* is the empty set.

10 libnuma: simple allocation

libnuma offers functions to allocate memory with a specified policy.

These allocation functions round all allocations to pages (4KB on AMD64 systems) and are relatively slow. They should only be used for allocating large memory objects which exceed the cache sizes of the CPU and where NUMA policy is likely a win. When no memory can be allocated they return NULL. All memory allocated with the numa_alloc family of functions should be freed with *numa_free*.

numa_alloc_onnode allocates memory on a specific node:

```
void *mem = numa_alloc_onnode(MEMSIZE\_IN\_BYTES, 1);
if (mem == NULL)
/* report out of memory error */
... pass mem to a thread bound to node 1 ...
```

The thread must eventually *numa_free* the memory. By default *numa_alloc_onnode* will try to allocate memory on the specified node first, but fall back to other nodes when there is not enough memory. When *numa_set_strict(1)* was executed first it will not do this fall back and fail the allocation when there is not enough memory on the intended node. Before that the kernel will try to swap out memory on the node and clear other caches, which can lead to delays. To get a glimpse of how much memory can be available on a node see the *numa_node_size* function below.

numa_alloc_interleaved allocates memory interleaved on all nodes in the system.

```
void *mem = numa_alloc_interleaved(MEMSIZE\_IN\_BYTES);
if (mem == NULL)
/* report out of memory error */
... run memory bandwidth intensive algorithm on mem ...
numa\_free(mem, MEMSIZE_IN_BYTES);
```

Please note that using memory interleaved over all nodes is not always a performance win. Sometimes depending on the NUMA architecture of the machine the program runs on only a subset of neighboring nodes gives better bandwidth. The *numa_alloc_interleaved_subset* function can be used to interleave on a subset of nodes.

Another function is *numa_alloc_local* which allocates memory on the local node. This is normally the default for all allocations, but useful to specify explicitly when the process has a different process policy. *numa_alloc* allocates memory with the current process policy.

11 libnuma: process policy

Each thread has a default memory policy inherited from its parent. Unless changed with numactl this policy is normally to allocate memory preferably on the current node. When existing code in a program cannot be modified to use the *numa_alloc* functions described in the previous section directly it is sometimes useful to change the process policy in a program.

numa_set_interleave_mask enables interleaving for the current thread. All future memory allocations will allocate memory round robin interleaved over the nodemask specified. Passing *numa_all_nodes* will interleave memory to all nodes. Passing *numa_no_nodes* turns off interleaving again. numa_get_interleave_mask returns the current interleave mask. This can be useful to save restore interleaving masks in a library.

```
numamask_t oldmask = numa_get_interleave_mask();
numa_set_interleave_mask(&numa_all_nodes);
/* run memory bandwidth intensive legacy library that allocates memory */
numa_set_interleave_mask(&oldmask);
```

numa_set_prefered sets the preferred node of the current thread. The memory allocator tries to allocate memory on that node first, and if there isn't enough memory free falls back to other nodes.

numa_set_membind sets a strict memory binding mask to a nodemask. strict means that the memory must be allocated on the specified nodes, when there is not enough memory free after swapping the allocation will fail. *numa_get_membind* returns the current memory binding mask.

numa_set_localalloc sets the process policy to the standard local allocation policy.

12 libnuma: changing the policy of existing memory areas

When working with shared memory it is often not possible to use the *numa_alloc* family of functions to allocate memory. The memory has to be gotten from *shmat()* or from *mmap* instead. To allow *libnuma* programs to set policy on such areas there are additional functions to set memory policy for already existing areas.

numa_interleave_memory will set an interleaving policy with an interleaving mask. Passing *numa_all_nodes* will interleave to all nodes in the system.

```
void *mem = shmat( ... );   /* get shared memory */
numa_interleave_mask(mem, size, numa_all_nodes);
```

numa_tonode_memory will allocate the memory on a specific node, while *numa_tonodemask_memory* puts the memory onto a mask of nodes.

numa_setlocal_memory gives the memory area a policy to allocate on the current node. *numa_police_memory* uses the current policy to allocate memory. This can be useful when the memory policy is changed later.

When numa_set_strict(1) was executed previously to set strict policy these calls will call *numa_error* when any of the already existing pages in the memory area do not conform to the new policy. Otherwise existing pages are ignored.

13 libnuma: binding to CPUs

The functions discussed so far allocated memory on specific nodes. Another part of NUMA policy is to run the thread on the correct node. This is done by the *numa_run_on_node* function which binds the current thread to all CPUs in node. *numa_run_on_node_mask* binds the current thread to any of the CPUs included in a nodemask.

Run current thread to node 1 and allocate memory there:

```
numa_run_on_on_node(1);
numa_set_prefered(1);
```

A simple way to use *libnuma* is the *numa_bind* function. It binds both the CPU and the memory of the process allocated in the future to a specific nodemask. It is equivalent to the previous example.

Bind process CPU and memory allocation to node 1 using *numa_bind*:

```
nodemask_t mask;
nodemask_zero(&mask);
nodemask_set(&mask, 1);
numa_bind(&mask);
```

The thread can be reset to execute on all nodes again by binding it to *numa_all_nodes*:

```
numa_run_on_node_mask(&numa_all_nodes);
```

The *numa_get_run_node_mask* function returns the nodemask of nodes the current thread is allowed to run on.

14 libnuma: inquiring about the environment

numa_node_size returns the memory size of a node. The return argument is the total size of its memory, which is not necessarily all available to the program. The second argument is a pointer that can be filled with the free memory on the node. The program can allocate node memory somewhere between the free memory (which is normally low because Linux uses free memory for caches) and the maximum memory size. This function gives a hint for how much memory is available for allocation on each node, but it should be only taken as a hint, preferably with some way for the administrator to overwrite. There is also a *numa_node_size64* function which uses an long long argument for the free memory instead of long.

numa_node_to_cpus returns the CPU numbers of all CPUs in a node. This can be used to find out how many CPUs there are in a node. It gets as argument the node number and a pointer to an array. The last argument is the byte length of

the array. The array is filled with a bit mask of CPU numbers. When the array is not long enough to contain all CPUs the function returns -1 and set *errno* to *ERANGE*. It is recommended that applications always handle this error or pass a very big buffer (e.g. 512 bytes). Otherwise there may be failures on very big machines. Linux already runs on 1024 CPU machines and is expected to be moved to even bigger machines.

15 libnuma: error handling

Error handling in *libnuma* is relatively simple. The main reason for this is that errors in setting NUMA policy can be usually ignored. The worst result of a wrong NUMA policy is that the program runs slower than it could be.

When an error occurs while setting a policy the *numa_error* function is called. By default it prints an error to *stderr*. When the *numa_exit_on_error* global variable is set it will exit the program. The function is declared weak and can be overwritten by defining a replacement function in the main program. For example a C++ program could throw a C++ exception in there.

Memory allocation functions always return *NULL* when no memory is available.

16 NUMA allocation statistics with numastat

For each node in the system, the kernel maintains some statistics pertaining to NUMA allocation status as each page is allocated. This information may be useful for testing the effectiveness of a NUMA policy.

The statistics are retrieved with the *numastat* command. The statistics are collected on a per-node basis. On systems with multiple CPU cores per node, *numstat* aggregates the results from all cores on a node to form a single result for the entire node. The *numastat* command reports the following statistics for each node:

numa_hit is incremented when a process requests a page from a particular node, and it receives a page from the requested node, then this counter is incremented for that particular node. The process may be running on any node in the system.

numa_miss is incremented when a process requests a page from a particular node, and it instead receives a page from some other node, then this counter is incremented at the node where the page was actually allocated. The process may be running on any node in the system.

numa_foreign is incremented when a process requests a page from a particular node, and it instead receives a page from some other node, this counter is incremented at the original node from which the page was requested. The process may be running on any node in the system. Each *numa_foreign* event has a corresponding *numa_miss* event on another node.

interleave_hit is incremented on the node on which a page is allocated when the allocation obeys the interleave policy for the address range. In addition, *numa_hit* and either *local_node* or *other_node* are incremented as well on the node on which the page is allocated. Note, there is no count of the number of pages allocated as interleaved, but not on the requested node due to the requested node having no free pages.

local_node - When a process requests a page, and the resulting page is located on the same node where the process is running, then this counter is incremented on that particular node.

other_node - When a process requests a page, and the resulting page is located on a different node than where the process is running, then this counter is incremented

for the node on which the page is actually allocated.

The difference between *numa_miss*, *numa_hit* and *local_node*, *foreign_node* is that the first two count hit or miss for the NUMA policy, while the later count if the allocation was on the same node as the requesting thread was running on.

To help further clarify how the *numastat* values consider the following examples.

1) The following example shows which counters are incremented when a process running on node 0 requests a page on node 0 and it is allocated on node 0.

```
                          node3       node2       node1       node0
numa_hit                                                        +1
numa_miss
numa_foreign
interleave_hit
local_node                                                      +1
other_node
```

2) The following example shows which counters are incremented when a process running on node 0 requests a page on node 0 yet it is allocated on node 1 due to a shortage of free pages on node 0.

```
                          node3       node2       node1       node0
numa_hit
numa_miss                                             +1
numa_foreign                                                    +1
interleave_hit
local_node
other_node                                            +1
```

3) The following example shows which counters are incremented when a process running on node 0 requests and receives a page on node 1. Note the difference between this and example 1).

```
                          node3       node2       node1       node0
numa_hit                                              +1
numa_miss
numa_foreign
interleave_hit
local_node
other_node                                            +1
```

4) The following example shows which counters are incremented when a process running on node 0 requests a page on node 1 yet is allocated on node 0 due to a shortage of free pages on node 1.

```
                          node3       node2       node1       node0
numa_hit
numa_miss                                                       +1
numa_foreign                                          +1
interleave_hit
local_node                                                      +1
other_node
```

As a further example, consider a four node machine, with 4 GB of RAM per node. Initially, *numastat* reports the following statistics for this machine:

```
> numastat
                          node3       node2       node1       node0
```

```
numa_hit                58956          142758          424386          319127
numa_miss                   0               0               0               0
numa_foreign                0               0               0               0
interleave_hit          19204           20238           19675           20576
local_node              43013          126715          409434          305254
other_node              15943           16043           14952           13873
```

Now suppose that a program called *memhog* runs on node1, allocating 8 GB of RAM during execution. After *memhog* completes, *numastat* reports the following statistics:

```
> numastat
                        node3           node2           node1           node0
numa_hit                58956          142758          424386          320893
numa_miss               48365         1026046               0               0
numa_foreign                0               0         1074411               0
interleave_hit          19204           20238           19675           20577
local_node              43013          126856         1436403          307019
other_node              64308         1042089           14952           13874
```

Each column represents a node where an allocation event took place. Notice that the *memhog* program tried to allocate 1,074,411 pages from node1, but was unable to do so. Instead, the process wound up with 1,026,046 pages from node2, and 48,365 pages from node3.

17 System call overview

NUMA API adds three new system calls: *mbind*, *set_mempolicy*, *get_mempolicy*. Normally user applications should use the higher level *libnuma* interface and not call the system calls directly. The system call interface is declared in *numaif.h*. The system calls are currently not defined in glibc, applications use them should link to *libnuma*. They may return -1 and *ENOSYS* in *errno* when the kernel does not support NUMA policy.

All of these system calls get masks of nodes as argument, similar to the *nodemask_t* type of *libnuma*. In the system call interface they are defined as arrays of longs, each long containing a string of bits, together with an additional argument that gives the highest node number in the bitmap.

set_mempolicy sets the memory policy of the current thread. The first argument is the policy. Valid policies are *MPOL_BIND*, *MPOL_INTERLEAVE*, *MPOL_DEFAULT*, *MPOL_PREFERED*. These act the same as the numactl policies described earlier. How the node mask argument is used depends on the policy. For *MPOL_INTERLEAVE* it specifies the interleave mask, for *MPOL_BIND* and *MPOL_PREFERED* it contains the membind mask.

get_mempolicy retrieves the memory policy of the current thread. In addition to output arguments for policy and nodemask and the nodemask size it has an address and flags argument. When the *MPOL_MF_ADDR* bit is set in flags the vma policy of address is returned in the other arguments. When *MPOL_F_NODE* is set in addition the current node of the page at address is returned.

mbind sets the memory policy for a memory area. The first argument is a memory address and a length, the rest is a policy with mask and length similar to *set_mempolicy*. In addition it has a flags argument. When *MPOL_MF_STRICT* is passed for flags the call will fail when any existing pages in the mapping violate the specified policy.

For more details see the *get_mempolicy(2)*, *set_mempolicy(2)*, *mbind(2)* man pages.

18 Limitations

On bigger system there is normally a hierarchy in the interconnect. This means that neighboring nodes are faster to access than more remote nodes. NUMA API currently represents the system as a flat set of nodes. Future versions will allow the application to query node distances.

Future versions of NUMA API may allow to set policy on the file cache.