Started Nov 1999 by Kanoj Sarcar <kanoj@sgi.com>

What is NUMA?

This question can be answered from a couple of perspectives: the hardware view and the Linux software view.

From the hardware perspective, a NUMA system is a computer platform that comprises multiple components or assemblies each of which may contain 0 or more CPUs, local memory, and/or IO buses. For brevity and to disambiguate the hardware view of these physical components/assemblies from the software abstraction thereof, we'll call the components/assemblies 'cells' in this document.

Each of the 'cells' may be viewed as an SMP [symmetric multi-processor] subset of the system--although some components necessary for a stand-alone SMP system may not be populated on any given cell. The cells of the NUMA system are connected together with some sort of system interconnect--e.g., a crossbar or point-to-point link are common types of NUMA system interconnects. Both of these types of interconnects can be aggregated to create NUMA platforms with cells at multiple distances from other cells.

For Linux, the NUMA platforms of interest are primarily what is known as Cache Coherent NUMA or ccNUMA systems. With ccNUMA systems, all memory is visible to and accessible from any CPU attached to any cell and cache coherency is handled in hardware by the processor caches and/or the system interconnect.

Memory access time and effective memory bandwidth varies depending on how far away the cell containing the CPU or IO bus making the memory access is from the cell containing the target memory. For example, access to memory by CPUs attached to the same cell will experience faster access times and higher bandwidths than accesses to memory on other, remote cells. NUMA platforms can have cells at multiple remote distances from any given cell.

Platform vendors don't build NUMA systems just to make software developers' lives interesting. Rather, this architecture is a means to provide scalable memory bandwidth. However, to achieve scalable memory bandwidth, system and application software must arrange for a large majority of the memory references [cache misses] to be to "local" memory--memory on the same cell, if any--or to the closest cell with memory.

This leads to the Linux software view of a NUMA system:

Linux divides the system's hardware resources into multiple software abstractions called "nodes". Linux maps the nodes onto the physical cells of the hardware platform, abstracting away some of the details for some architectures. As with physical cells, software nodes may contain 0 or more CPUs, memory and/or IO buses. And, again, memory accesses to memory on "closer" nodes--nodes that map to closer cells--will generally experience faster access times and higher effective bandwidth than accesses to more remote cells.

For some architectures, such as x86, Linux will "hide" any node representing a physical cell that has no memory attached, and reassign any CPUs attached to that cell to a node representing a cell that does have memory. Thus, on these architectures, one cannot assume that all CPUs that Linux associates with a given node will see the same local memory access times and bandwidth.

In addition, for some architectures, again x86 is an example, Linux supports the emulation of additional nodes. For NUMA emulation, linux will carve up the existing nodes--or the system memory for non-NUMA platforms--into multiple nodes. Each emulated node will manage a fraction of the underlying cells' physical memory. NUMA emluation is useful for testing NUMA kernel and application features on non-NUMA platforms, and as a sort of memory resource management mechanism when used together with cpusets.
[see Documentation/cgroups/cpusets.txt]

For each node with memory, Linux constructs an independent memory management subsystem, complete with its own free page lists, in-use page lists, usage statistics and locks to mediate access. In addition, Linux constructs for each memory zone [one or more of DMA, DMA32, NORMAL, HIGH_MEMORY, MOVABLE], an ordered "zonelist". A zonelist specifies the zones/nodes to visit when a selected zone/node cannot satisfy the allocation request. This situation, when a zone has no available memory to satisfy a request, is called "overflow" or "fallback".

Because some nodes contain multiple zones containing different types of memory, Linux must decide whether to order the zonelists such that allocations fall back to the same zone type on a different node, or to a different zone type on the same node. This is an important consideration because some zones, such as DMA or DMA32, represent relatively scarce resources. Linux chooses a default zonelist order based on the sizes of the various zone types relative to the total memory of the node and the total memory of the system. The default zonelist order may be overridden using the numa_zonelist_order kernel boot parameter or sysctl. [see Documentation/kernel-parameters.txt and Documentation/sysctl/vm.txt]

By default, Linux will attempt to satisfy memory allocation requests from the node to which the CPU that executes the request is assigned. Specifically, Linux will attempt to allocate from the first node in the appropriate zonelist for the node where the request originates. This is called "local allocation." If the "local" node cannot satisfy the request, the kernel will examine other nodes' zones in the selected zonelist looking for the first zone in the list that can satisfy the request.

Local allocation will tend to keep subsequent access to the allocated memory "local" to the underlying physical resources and off the system interconnect-- as long as the task on whose behalf the kernel allocated some memory does not later migrate away from that memory. The Linux scheduler is aware of the NUMA topology of the platform--embodied in the "scheduling domains" data structures [see Documentation/scheduler/sched-domains.txt]--and the scheduler attempts to minimize task migration to distant scheduling domains. However, the scheduler does not take a task's NUMA footprint into account directly. Thus, under sufficient imbalance, tasks can migrate between nodes, remote from their initial node and kernel data structures.

System administrators and application designers can restrict a task's migration to improve NUMA locality using various CPU affinity command line interfaces, such as taskset(1) and numactl(1), and program interfaces such as sched_setaffinity(2). Further, one can modify the kernel's default local allocation behavior using Linux NUMA memory policy.
[see Documentation/vm/numa_memory_policy.]

System administrators can restrict the CPUs and nodes' memories that a non-privileged user can specify in the scheduling or NUMA commands and functions using control groups and CPUsets. [see Documentation/cgroups/CPUsets.txt]

On architectures that do not hide memoryless nodes, Linux will include only zones [nodes] with memory in the zonelists. This means that for a memoryless node the "local memory node"--the node of the first zone in CPU's node's zonelist--will not be the node itself. Rather, it will be the node that the kernel selected as the nearest node with memory when it built the zonelists. So, default, local allocations will succeed with the kernel supplying the closest available memory. This is a consequence of the same mechanism that allows such allocations to fallback to other nearby nodes when a node that does contain memory overflows.

Some kernel allocations do not want or cannot tolerate this allocation fallback behavior. Rather they want to be sure they get memory from the specified node or get notified that the node has no free memory. This is usually the case when a subsystem allocates per CPU memory resources, for example.

A typical model for making such an allocation is to obtain the node id of the node to which the "current CPU" is attached using one of the kernel's numa_node_id() or CPU_to_node() functions and then request memory from only the node id returned. When such an allocation fails, the requesting subsystem may revert to its own fallback path. The slab kernel memory allocator is an example of this. Or, the subsystem may choose to disable or not to enable itself on allocation failure. The kernel profiling subsystem is an example of this.

If the architecture supports--does not hide--memoryless nodes, then CPUs attached to memoryless nodes would always incur the fallback path overhead or some subsystems would fail to initialize if they attempted to allocated memory exclusively from a node without memory. To support such architectures transparently, kernel subsystems can use the numa_mem_id() or cpu_to_mem() function to locate the "local memory node" for the calling or specified CPU. Again, this is the same node from which default, local page allocations will be attempted.