Optimizing Applications for NUMA

<keywords: NUMA, memory management, data affinity>

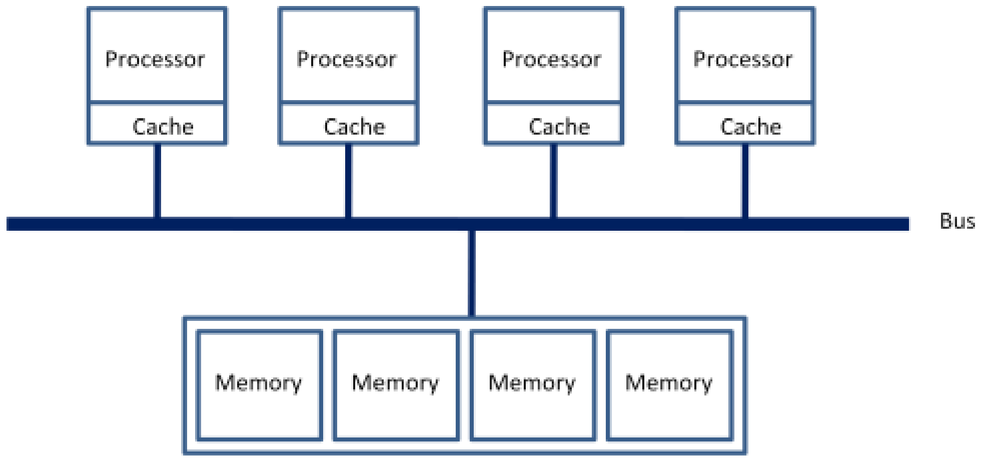
Abstract

NUMA, or **Non-Uniform Memory Access**, is a shared memory architecture that describes the placement of main memory modules with respect to processors in a multiprocessor system. Like most every other processor architectural feature, ignorance of NUMA can result in sub-par application memory performance. Fortunately, there are steps that can be taken to mitigate any NUMA-based performance issues or to even use NUMA architecture to the advantage of your parallel application. Such considerations include processor affinity, memory allocation using implicit operating system policies, and the use of the system APIs for assigning and migrating memory pages using explicit directives.

This article is part of the larger series, "Intel Guide for Developing Multithreaded Applications," which provides guidelines for developing efficient multithreaded applications for Intel® platforms.

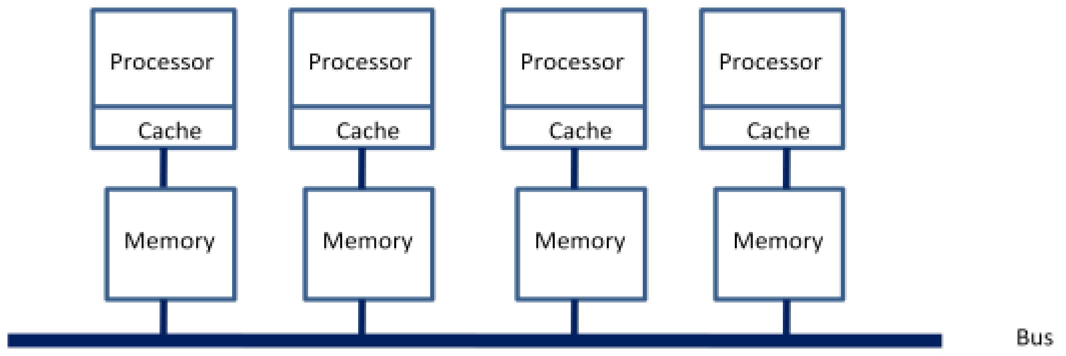
Background

Perhaps the best way to understand NUMA is to compare it with its cousin **UMA**, or **Uniform Memory Access.** In the UMA memory architecture, all processors access shared memory through a bus (or another type of interconnect) as seen in the following diagram:



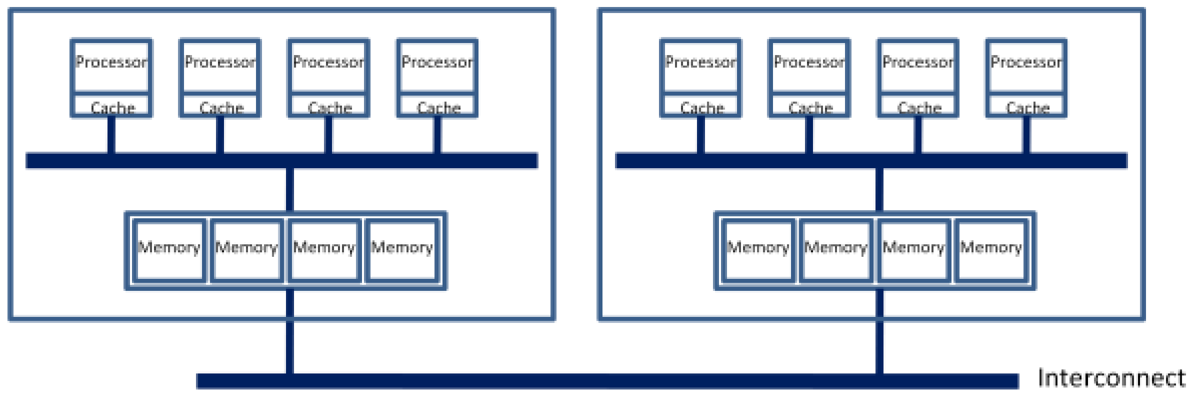
UMA gets its name from the fact that each processor must use the same shared bus to access memory, resulting in a memory access time that is uniform across all processors. Note that access time is also independent of data location within memory. That is, access time remains the same regardless of which shared memory module contains the data to be retrieved.

In the NUMA shared memory architecture, each processor has its own *local* memory module that it can access directly with a distinctive performance advantage. At the same time, it can also access any memory module belonging to another processor using a shared bus (or some other type of interconnect) as seen in the diagram below:



What gives NUMA its name is that memory access time varies with the location of the data to be accessed. If data resides in local memory, access is fast. If data resides in remote memory, access is slower. The advantage of the NUMA architecture as a *hierarchical* shared memory scheme is its potential to improve *average case access time* through the introduction of fast, local memory.

Modern multiprocessor systems mix these basic architectures as seen in the following diagram:



In this complex hierarchical scheme, processors are grouped by their physical location on one or the other multi-core CPU package or "node." Processors within a node share access to memory modules as per the UMA shared memory architecture. At the same time, they may also access memory from the remote node using a shared interconnect, but with slower performance as per the NUMA shared memory architecture.

Advice

Two key notions in managing performance within the NUMA shared memory architecture are *processor affinity* and *data placement*.

Processor Affinity

Today's complex operating systems assign application threads to processor cores using a scheduler. A scheduler will take into account system state and various policy objectives (e.g., "balance load across cores" or "aggregate threads on a few cores and put remaining cores to sleep"), then match application threads to physical cores accordingly. A given thread will execute on its assigned core for some period of time before being swapped out of the core to wait, as other threads are given the chance to execute. If another core becomes available, the scheduler may choose to migrate the thread to ensure timely execution and meet its policy objectives.

Thread migration from one core to another poses a problem for the NUMA shared memory architecture because of the way it disassociates a thread from its local memory allocations. That is, a thread may allocate memory on node 1 at startup as it runs on a core within the node 1 package. But when the thread is later migrated to a core on node 2, the data stored earlier becomes remote and memory access time significantly increases.

Enter processor affinity. Processor affinity refers to the persistence of associating a thread/process with a particular processor resource instance, despite the availability of other instances. Using a system API, or by modifying an OS data structure (e.g., affinity mask), a specific core or set of cores can be associated with an application thread. The scheduler will then observe this affinity in its scheduling decisions for the lifetime of the thread. For example, a thread may be configured to run only on cores 0 through 3, all of which belong to quad core CPU package 0. The scheduler will choose among cores 0 through 3 without even considering migrating the thread to another package.

Exercising processor affinity ensures that memory allocations remain local to the thread(s) that need them. Several downsides, however, should be noted. In general, processor affinity may significantly harm system performance by restricting scheduler options and creating resource contention when better resources management could have otherwise been used. Besides preventing the scheduler from assigning waiting threads to unutilized cores, processor affinity restrictions may hurt the application itself when additional execution time on another node would have more than compensated for a slower memory access time.

Programmers must think carefully about whether processor affinity solutions are right for a particular application and shared system context. Note, finally, that processor affinity APIs offered by some systems support priority "hints" and affinity "suggestions" to the scheduler in addition to explicit directives. Using such suggestions, rather than forcing unequivocal structure on thread placement, may ensure optimal performance in the common case yet avoid constraining scheduling options during periods of high resource contention.

Data Placement Using Implicit Memory Allocation Policies

In the simple case, many operating systems transparently provide support for NUMA-friendly data placement. When a single-threaded application allocates memory, the processor will simply assign memory pages to the physical memory associated with the requesting thread's node (CPU package), thus ensuring that it is local to the thread and access performance is optimal.

Alternatively, some operating systems will wait for the first memory access before committing on memory page assignment. To understand the advantage here, consider a multithreaded application with a start-up sequence that includes memory allocations by a main control thread, followed by the creation of various worker threads, followed by a long period of application processing or service. While it may seem reasonable to place memory pages local to the allocating thread, in fact, they are more effectively placed local to the worker threads that will access the data. As such, the operating system will observe the first access request and commit page assignments based on the requester's node location.

These two policies (local to first access and local to first request) together illustrate the importance of an application programmer being aware of the NUMA context of the program's deployment. If the page placement policy is based on first access, the programmer can exploit this fact by including a carefully designed data access sequence at startup that will generate “hints” to the operating system on optimal memory placement. If the page placement policy is based on requester location, the programmer should ensure that memory allocations are made by the thread that will subsequently access the data and not by an initialization or control thread designed to act as a provisioning agent.

Multiple threads accessing the same data are best co-located on the same node so that the memory allocations of one, placed local to the node, can benefit all. This may, for example, be used by prefetching schemes designed to improve application performance by generating data requests in advance of actual need. Such threads must generate data placement that is local to the actual consumer threads for the NUMA architecture to provide its characteristic performance speedup.

It should be noted that when an operating system has fully consumed the physical memory resources of one node, memory requests coming from threads on the same node will typically be fulfilled by sub-optimal allocations made on a remote node. The implication for memory-hungry applications is to correctly size the memory needs of a particular thread and to ensure local placement with respect to the accessing thread.

For situations where a large number of threads will randomly share the same pool of data from all nodes, the recommendation is to stripe the data evenly across all nodes. Doing so spreads the memory access load and avoids bottleneck access patterns on a single node within the system.

Data Placement Using Explicit Memory Allocation Directives

Another approach to data placement in NUMA-based systems is to make use of system APIs that explicitly configure the location of memory page allocations. An example of such APIs is the `libnuma` library for Linux.

Using the API, a programmer may be able to associate virtual memory address ranges with particular nodes, or simply to indicate the desired node within the memory allocation system call itself. With this capability, an application programmer can ensure the placement of a particular data set regardless of which thread allocates it or which thread accesses it first. This may be useful, for example, in schemes where complex applications make use of a memory management thread acting on behalf of worker threads. Or, it may prove useful for applications that create many short-lived threads, each of which have predictable data requirements. Pre-fetching schemes are another area that could benefit considerably from such control.

The downside of this scheme, of course, is the management burden placed on the application programmer in handling memory allocations and data placement. Misplaced data may cause performance that is significantly worse than default system behavior. Explicit memory management also presupposes fine-grained control over processor affinity throughout application use.

Another capability available to the application programmer through NUMA-based memory management APIs is memory page migration. In general, migration of memory pages from one node to another is an expensive operation and something to be avoided. Having said this, given an application that is both long-lived and memory intensive, migrating memory pages to re-establish a NUMA-friendly configuration may be worth the price.[3] Consider, for example, a long-lived application with various threads that have terminated and new threads that have been created but reside on another node. Data is now no longer local to the threads that need it and sub-optimal access requests now dominate. Application-specific knowledge of a thread’s lifetime and data needs can be used to determine whether an explicit migration is in order.

Usage Guidelines

The key issue in determining whether the performance benefits of the NUMA architecture can be realized is ***data placement***. The more often that data can effectively be placed in memory local to the processor that needs it, the more overall access time will benefit from the architecture. By providing each node with its own local memory, memory accesses can avoid throughput limitations and contention issues associated with a shared memory bus. In fact, memory constrained systems can theoretically improve their performance by up to the number of nodes on the system by virtue of accessing memory in a fully parallelized manner.

Conversely, the more data fails to be local to the node that will access it, the more memory performance will suffer from the architecture. In the NUMA model, the time required to retrieve data from an adjacent node within the NUMA model will be significantly higher than that required to access local memory. In general, as the *distance* from a processor increases, the cost of accessing memory increases.

Additional Resources

Intel® Software Network Parallel Programming Community

Drepper, Ulrich. "What Every Programmer Should Know About Memory". November 2007.

Intel® 64 and IA-32 Architectures Optimization Reference Manual. See Section 8.8 on "Affinities and Managing Shared Platform Resources". March 2009.

Lameter, Christoph. "Local and Remote Memory: Memory in a Linux/NUMA System". June 2006.