Non-Uniform Memory Access (NUMA)

Nakul Manchanda and Karan Anand
New York University
{nm1157, ka804} @cs.nyu.edu

ABSTRACT

NUMA **refers to the computer memory design choice available for multiprocessors. NUMA means that it will take longer to access some regions of memory than others. This work aims at explaining what NUMA is, the background developments, and how the memory access time depends on the memory location relative to a processor. First, we present a background of multiprocessor architectures, and some trends in hardware that exist along with NUMA. We, then briefly discuss the changes NUMA demands to be made in two key areas. One is in the policies the Operating System should implement for scheduling and run-time memory allocation scheme used for threads and the other is in the programming approach the programmers should take, in order to harness NUMA's full potential. In the end we also present some numbers for comparing UMA vs. NUMA's performance.**

Keywords: *NUMA, Intel i7, NUMA Awareness, NUMA Distance*

SECTIONS

In the following sections we first describe the background, hardware trends, Operating System's goals, changes in programming paradigms, and then we conclude after giving some numbers for comparison.

Background

Hardware Goals / Performance Criteria

There are 3 criteria on which performance of a multiprocessor system can be judged, viz. Scalability, Latency and Bandwidth. Scalability is the ability of a system to demonstrate a proportionate increase in parallel speedup with the addition of more processors.

Latency is the time taken in sending a message from node A to node B, while bandwidth is the amount of data that can be communicated per unit of time. So, the goal of a multiprocessor system is to achieve a highly scalable, low latency, high bandwidth system.

Parallel Architectures

Typically, there are 2 major types of Parallel Architectures that are prevalent in the industry: Shared Memory Architecture and Distributed Memory Architecture. Shared Memory Architecture, again, is of 2 types: Uniform Memory Access (UMA), and Non-Uniform Memory Access (NUMA).

Shared Memory Architecture

As seen from the figure 1 (more details shown in "Hardware Trends" section) all processors share the same memory, and treat it as a global address space. The major challenge to overcome in such architecture is the issue of Cache Coherency (i.e. every read must reflect the latest write). Such architecture is usually adapted in hardware model of general purpose CPU's in laptops and desktops.

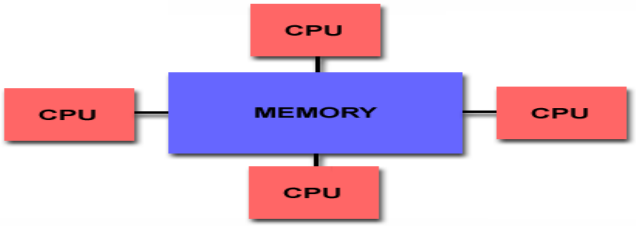

Figure 1 Shared Memory Architecture (from [1])

Distributed Memory Architecture

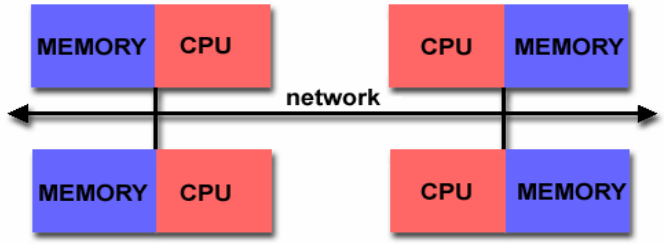

Figure 2 Distributed Memory (from [1])

In figure 2 (more details shown in "Hardware Trends" section) type of architecture, all the processors have their own local memory, and there is no mapping of memory addresses across processors. So, we don't have any concept of global address space or cache coherency. To access data in another processor, processors use explicit communication. One example where this architecture is used with clusters, with different nodes connected over the internet as network.

Shared Memory Architecture – UMA

Shared Memory Architecture, again, is of 2 distinct types, Uniform Memory Access (UMA), and Non-Uniform Memory Access (NUMA).

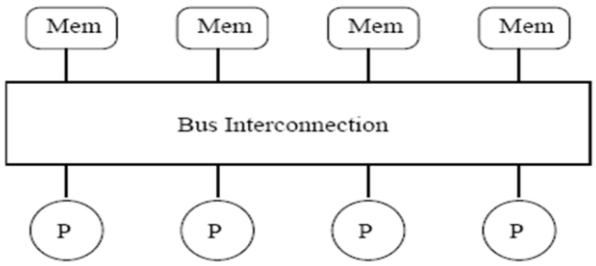

Figure 3 UMA Architecture Layout (from [3])

The Figure 3 shows a sample layout of processors and memory across a bus interconnection. All the processors are identical, and have equal access times to all memory regions. These are also sometimes known as Symmetric Multiprocessor (SMP) machines. The architectures that take care of cache coherency in hardware level, are knows as CC-UMA (cache coherent UMA).

Shared Memory Architecture – NUMA

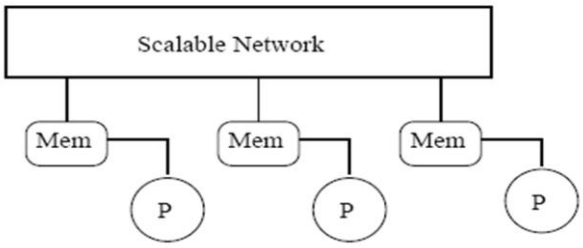

Figure 4 NUMA Architecture Layout (from [3])

Figure 4 shows type of shared memory architecture, we have identical processors connected to a scalable network, and each processor has a portion of memory attached directly to it. The primary difference between a NUMA and distributed memory architecture is that no processor can have mappings to memory connected to other processors in case of distributed memory architecture, however, in case of NUMA, a processor may have so. It also introduces classification of local memory and remote memory based on access latency to different memory region seen from each processor. Such systems are often made by physically linking SMP machines. UMA, however, has a major disadvantage of not being scalable after a number of processors [6].

Hardware Trends

We now discuss 2 practical implementations of the memory architectures that we just saw, one is the Front Side Bus and the other is Intel's Quick Path Interconnect based implementation.

Traditional FSB Architecture (used in UMA)

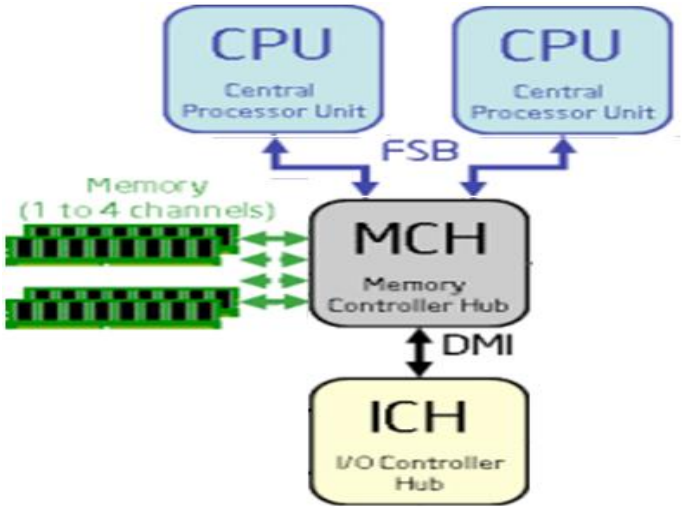

Figure 5 Intel's FSB based UMA Arch. (from [4])

As shown in Figure 5, FSB based UMA architecture has a Memory Controller Hub, which has all the memory connected to it. The CPUs interact with the MCH whenever they need to access the memory. The I/O controller hub is also connected to the MCH, hence the major bottleneck in this implementation is the bus, which has a finite speed, and has scalability issues. This is because, for any communication, the CPU's need to take control of the bus which leads to contention problems.

Quick Path Interconnect Architecture (used in NUMA)

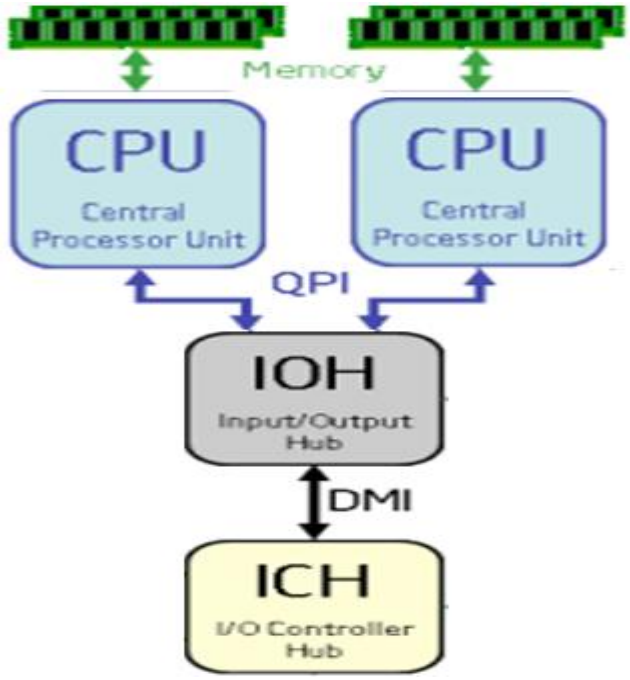

Figure 6 Intel's QPI based NUMA Arch. (from [4])

The key point to be observed in this implementation is that the memory is directly connected to the CPU's instead of a memory controller. Instead of accessing memory via a Memory Controller Hub, each CPU now has a memory controller embedded inside it. Also, the CPU's are connected to an I/O hub, and to each other. So, in effect, this implementation tries to address the common-channel contention problems.

New Cache Coherency Protocol

This new QPI based implementation also introduces a new cache coherency protocol, "MESIF" instead of "MESI". The new state "F" stands for forward, and is used to denote that a cache should act as a designated responder for any requests.

Operating System Policies

OS Design Goals

Operating Systems, basically, try to achieve 2 major goals, viz. Usability and Utilization. By usability, we mean that OS should be able to abstract the hardware for programmer's convenience. The other goal is to achieve optimal resource management, and the ability to multiplex the hardware amongst different applications.

Features of NUMA aware OS

The basic requirements of a NUMA aware OS are to be able to discover the underlying hardware topology, and to be able to calculate the NUMA distance accurately. NUMA distances tell the processors (and / or the programmer) how much time it would take to access that particular memory.

Besides these, the OS should provide a mechanism for processor affinity. This is basically done to make sure that some threads are scheduled on certain processor(s), to ensure data locality. This not only avoids remote access, but can also take the advantage of hot cache. Also, the operating system needs to exploit the first touch memory allocation policy.

Optimized Scheduling Decisions

The operating systems needs to make sure that load is balanced amongst the different processors (by making sure that data is distributed amongst CPU's for large jobs), and also to implement dynamic page migration (i.e. use latency topology to make page migration decisions).

Conflicting Goals

The goals that the Operating System is trying to achieve are conflicting in nature, in the sense, on one hand we are trying to optimize the memory placement (for load balancing), and on the other hand, we would like to minimize the migration of data (to overcome resource contention). Eventually, there is a trade off which is decided on the basis of the type of application.

Programming Paradigms

NUMA Aware Programming Approach

The main goals of NUMA aware programming approach are to reduce lock contention and maximize memory allocation on local node. Also, programmers need to manage their own memory for maximum portability. This is can prove to be quite a challenge, since most languages do not have an in-built memory manager.

Support for Programmers

Programmers rely on tools and libraries for application development. Hence the tools and libraries need to help the programmers in achieving maximum efficiency, also to implement implicit parallelism. The user or the system interface, in turn needs to have programming constructs for associating virtual memory addresses. They also need to provide certain functions for obtaining page residency.

Programming Approach

The programmers need to explore the various NUMA libraries that are available to help simplify the task. If the data allocation pattern is analyzed properly, "First Touch Access" can be exploited fully. There are several lock-free approaches available, which can be used.

Besides these approaches, the programmers can exploit various parallel programming paradigms, such as Threads, Message Passing, and Data Parallelism.

Performance Comparison

Scalability – UMA vs NUMA

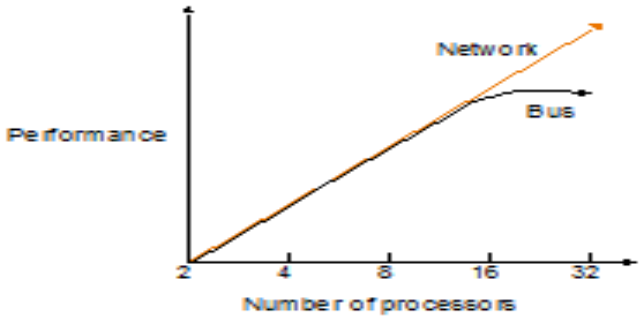

Figure 7 UMA vs. NUMA – Scalability (from [6])

We can see from the figure, that UMA based implementation have scalability issues. Initially both the architectures scale linearly, until the bus reaches a limit and stagnates. Since there is no concept of a "shared bus" in NUMA, it is more scalable.

Cache Latency

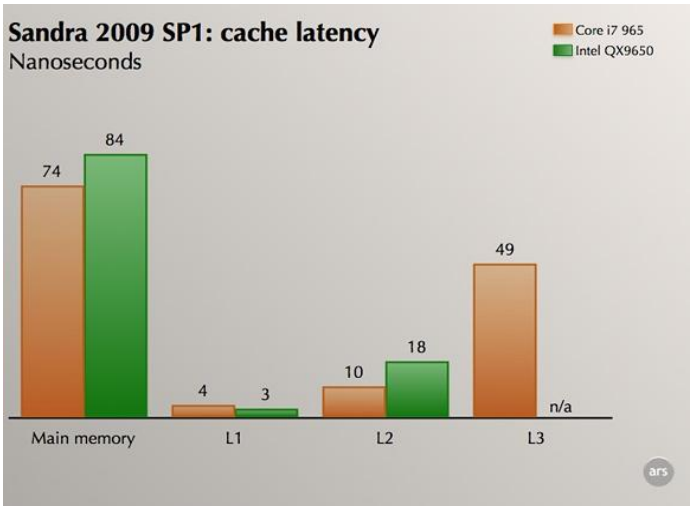

Figure 8 UMA vs NUMA - Cache Latency (from [4])

The figure shows a comparison of cache latency numbers of UMA and NUMA. There is no layer 3 cache in UMA. However, for Main Memory and Layer 2 cache, NUMA shows a considerable improvement. Only for Layer 1 cache, UMA marginally beats NUMA.

CONCLUSION

The hardware industry has adapted NUMA as a architecture design choice, primarily because of its characteristics like scalability and low latency. However, modern hardware changes also demand changes in the programming approaches (development libraries, data analysis) as well Operating System

policies (processor affinity, page migration). Without these changes, full potential of NUMA cannot be exploited.

REFERENCES

[1] "Introduction to Parallel Computing.": https://computing.llnl.gov/tutorials/parallel_comp/

[2] "Optimizing software applications for NUMA": http://software.intel.com/en-us/articles/optimizing-software-applications-for-numa/

[3] "Parallel Computer Architecture - Slides": http://www.eecs.berkeley.edu/~culler/cs258-s99/

[4] "Cache Latency Comparison": http://arstechnica.com/hardware/reviews/2008/11/nehalem-launch-review.ars/3

[5] "Intel – Processor Specifications": http://www.intel.com/products/processor/index.htm

[6] "UMA-NUMA Scalability" www.cs.drexel.edu/~wmm24/cs281/lectures/ppt/cs282_lec12.ppt