Multi-core Programming
Assignment 1

Ali Gholami

Department of Computer Engineering & Information Technology
Amirkabir University of Technology

http://ceit.aut.ac.ir/~aligholamee
aligholamee@aut.ac.ir

Abstract

This assignment reviews principal architectures for a parallel computer. Two main architectures as *Shared Memory* and *Message Passing* and their subtypes like *UMA, NUMA, hUMA* and *COMA* are introduced. We'll also review the substantial *Amdahl's law*, *Gustafson Barsis's law* and *Sun-Nis law* to understand the *performance* and *scalability* of parallel architectures. Finally, we'll be measuring various metrics for parallelism in an *Asus N56JK* laptop.

Keywords. *Parallel Architecture, Multi-core, Multi-thread, Uniform Memory Access, NUMA, COMA, Amdahl, Gustafson Barsis, Sun-Nis.*

1 Communication Models in Parallel Computers

There are different communication models in parallel computers. The two main architectures are *Shared Memory* and *Message Passing*.

Shared Memory

In a computer with *shared memory* architecture among its processors, each of its processors can access the memory simultaneously with other processors. The intuition can be represented as figure 1.1.

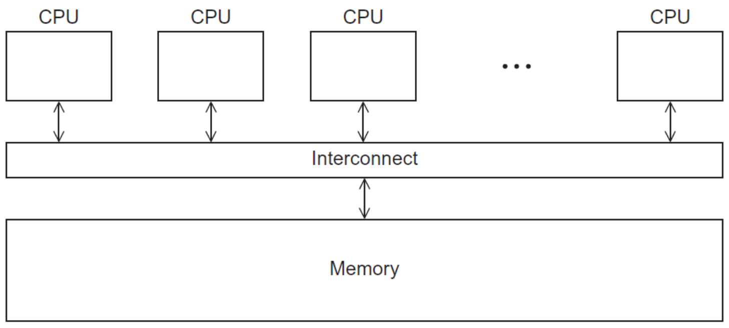

Figure 1.1: A simple shared memory architecture with multiple processors.

A shared memory architecture conducts two main physical architectures:

- Non-distributed Shared Memory
- Distributed Shared Memory

A *Non-distributed Shared Memory* was introduced in figure 1.1. As a physical point of view, the *Non-distributed Shared Memory* is *centralized* memory that can be accessed by the processors through a *communication bus*. Memories can be physically independent of each other. They can be implemented with processors. In this case, each processors has its own memory which is called a *Distributed Shared Memory*. An intuition of this architecture can be grasped by looking at figure 1.2.

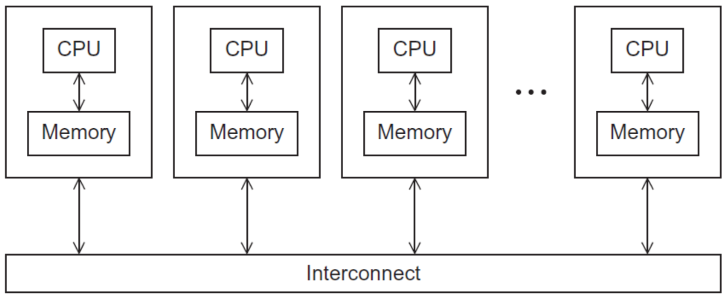

Figure 1.2: A distributed shared memory architecture with multiple processors.

Accessing the memory can be either *uniform* or *non-uniform*. We'll describe each further.

UMA

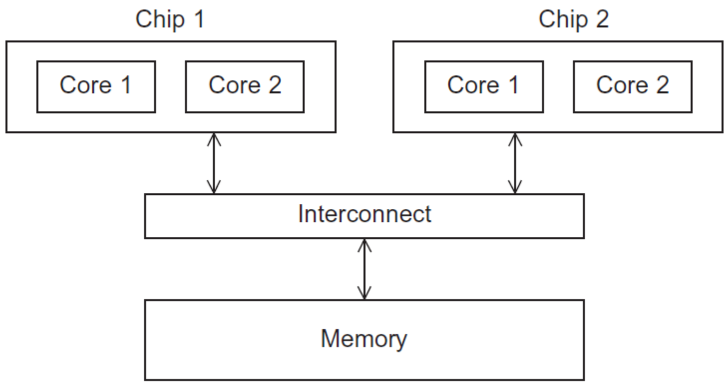

Figure 1.3: Demonstration of Uniform Memory Access in Shared Memory Architecture.

UMA stands for *Uniform Memory Access*. This is called *uniform* because, each of the processors have the same *Access Time* to the memory. Some of the features for *UMA* is given below:

1. Suitable for *Time Sharing* applications by multiple users.

2. It can be used to speed up a single large program in a *Time Critical* application.

NUMA

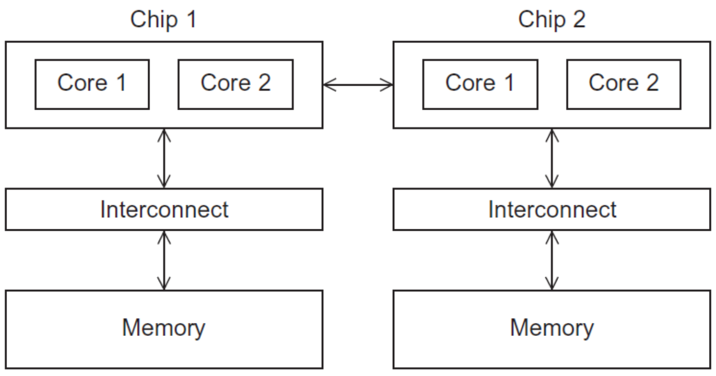

Figure 1.4: Demonstration of Non-uniform Memory Access in Shared Memory Architecture.

NUMA Stands for *Non-uniform Memory Access*. The *Memory Access Time* could be different for each processor. To be more accurate, *NUMA* is the phenomenon that memory at various points in the address space of a processor have different performance characteristics. A *NUMA* system classifies memory into *NUMA* nodes. All memory available in one node has the same access characteristics for a particular processor. Nodes have an affinity to processors and to devices. These are the devices that can use memory on a *NUMA* node with the best performance since they are locally attached. [1]

hUMA

In 2013, *AMD* introduced *hUMA*; *heterogeneous Unifrom Memory Access*. Before that, *AMD* had talked about the *HSA* concept which stands for *Heterogeneous Systems Architecture*. This concepts depicts that systems could have multiple different kinds of processors. More specifically, *versatile CPUs* and *specialized GPUs* can be integrated on a system. Let's dive into the history and find the root of this invention.

1. Modern *GPUs* have enormous parallel arithmetic power, especially floating point arithmetic, but are poorly-suited to *single-threaded* code with lots of branches.

2. Modern *CPUs* are well-suited to *single-threaded* code with lots of branches, but less well-suited to massively parallel number crunching.

Thus, Splitting workloads between a *CPU* and a *GPU*, using each for the workloads it's good at, has driven the development of general purpose *GPU (GPGPU)* software and development.

But there still exists some problems:

1. The *CPU* and *GPU* have their own pools of memory. Physically, these might use the same chips on the motherboard (as most integrated *GPUs* carve off a portion of system memory for their own purposes). From a software perspective, however, these are completely separate.

2. This means that whenever a CPU program wants to do some computation on the *GPU*, it has to copy all the data from the *CPU's* memory into the *GPU's* memory. When the *GPU* computation is finished, all the data has to be copied back. This need to copy back and forth wastes time and makes it difficult to mix and match code that runs on the *CPU* and code that runs on the GPU.

AMD's Magic

With *hUMA*, the *CPU* and *GPU* share a single memory space. The *GPU* can directly access *CPU* memory addresses, allowing it to both read and write data that the *CPU* is also reading and writing.

Cache-coherent vs Non-cache-coherent Systems

hUMA is a cache coherent system.

- In *non-cache-coherent* systems, programs have to explicitly signal that they have changed data that other processors might have cached, so that those other processors can discard their stale cached copy. This makes the hardware simpler, but introduces great scope for software errors that result in bugs that are hard to detect, diagnose, and fix.

- In a *cache-coherent* the *CPU* and *GPU* will always see a consistent view of data in memory. If one processor makes a change then the other processor will see that changed data, even if the old value was being cached. [2]

COMA

The shared memory concept makes it easier to write parallel programs, but tuning the application to reduce the impact of frequent long latency memory accesses still requires substantial programmer effort. Researchers have proposed using compilers, operating systems, or architectures to improve performance by allocating data close to the processors that use it. The Cache-Only Memory Architecture (*COMA*) increases the chances of data being available locally because the hardware transparently replicates the data and migrates it to the memory module of the node that is currently accessing it. Each memory module acts as a huge cache memory in which each block has a tag with the address and the state. [3] In NUMA, each address in the global address space is typically assigned a fixed home node. When processors access some data, a copy is made in their local cache, but space remains allocated in the home node. Instead, with COMA, there is no home. An access from a remote node may cause that data to migrate.

- It reduces the number of redundant copies and may allow more efficient use of the memory resources.
- It raises problems of how to find a particular data (there is no longer a home node) and what to do if a local memory fills up.

References

[1] Christoph Lameter. *NUMA (Non-Uniform Memory Access): An Overview.* ACM Queue, Vol. 11, no. 7. Retrieved on September 1st, 2015.

[2] Peter Bright. *AMD's "heterogeneous Uniform Memory Access" coming this year in Kaveri.* Ars Technica, April 30, 2013.

[3] F. Dahlgren and J. Torrellas. *Cache-only memory architectures.* IEEE Computer, June 1999.