Multi-core Programming
Assignment 1

Ali Gholami

Department of Computer Engineering & Information Technology
Amirkabir University of Technology

https:// aligholamee.github.io
aligholami7596@gmail.com

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.

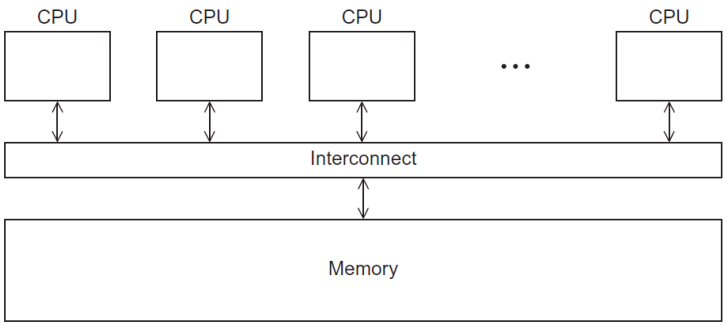

Figure 1.1: A simple shared memory architecture with multiple processors.

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.

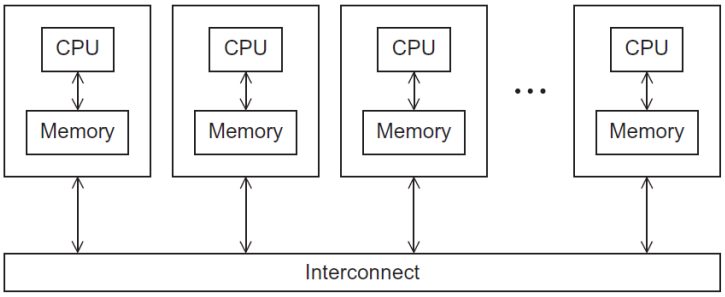

Figure 1.2: A distributed shared memory architecture with multiple processors.

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

UMA

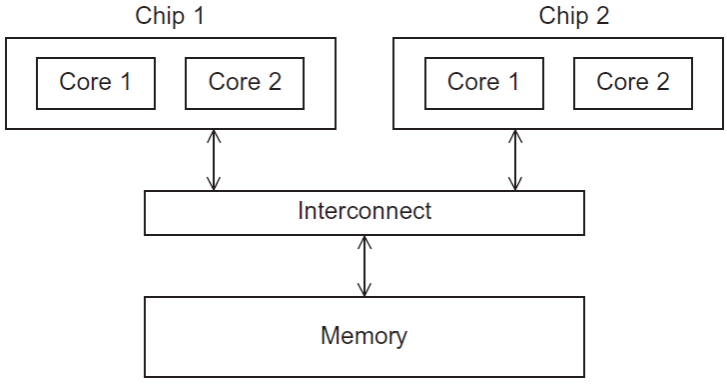

Figure 1.3: Demonstration of Uniform Memory Access in Shared Memory Architecture.

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

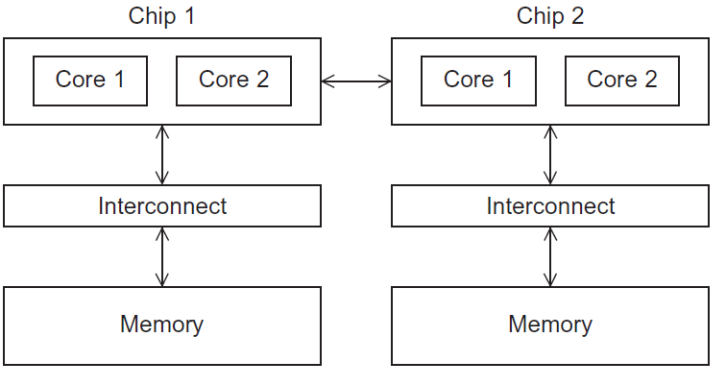

Figure 1.4: Demonstration of Non-uniform Memory Access in Shared Memory Architecture.

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.

2 Speedup Metrics Analysis for Parallel Computing

Amdahl's Law

Simply speaking, it depicts that the speed up is limited by the serial part of a program. For this reason, parallel computing with many processors is useful only for highly parallelizable programs.

$$S = \frac{1}{(1-\alpha) + \frac{\alpha}{p}} \tag{2.1}$$

where

- α is the proportion of the program that can be implemented in parallel;
- p is the number of cores, thread, parallel units;
- resulting speed-up.

Amdahl's Law Assumption

Theoretical speedup in latency of the execution of a task at fixed problem size. [4]

Gustafson's law

Gustafson's law proposes that programmers tend to set the size of problems to fully exploit the computing power that becomes available as the resources improve. Therefore, if faster equipment is available, larger problems can be solved within the same time.

$$S = p + (1-p)(1-\alpha) \tag{2.2}$$

where

- α is the proportion of the program that can be implemented in parallel;
- p is the number of cores, thread, parallel units;
- resulting speed-up.

Gustafson's Law Assumption

Theoretical speedup in latency of the execution of a task at fixed execution time. [5]

Sun-Ni's Law

Analogous to Amdahl's law, which says that the problem size remains constant as system sizes grow, and Gustafson's law, which proposes that the problem size should scale but be bound by a fixed amount of time, Sun-Ni's Law states the problem size should scale but be bound by the memory capacity of the system. [6] [7]

$$S = \frac{(1-\alpha) + \alpha * P(m)}{(1-\alpha) + \frac{\alpha * P(m)}{m}} \tag{2.3}$$

where

- α is the proportion of the program that can be implemented in parallel;
- m is the fraction that memory capacity has increased;
- P(m) parallel workload increase as memory increases in m.

Sun-Ni's Law Assumption

As computing power increases, the corresponding increase in problem size is constrained by the systems memory capacity.

3 Performance Analysis of SCMT Processors

In this section, we'll analyze the possibility of performance decrease, while running a multi-threaded application on a single core processor($SCMT$). Let's analyze the issue with a simple example. Assume we want to train a multi-layer convolutional neural network on a set of matrices. A single core processor can do this by loading the matrix in its cache and start performing the convolution computation for each pixel(index of matrix). In the time the convolution is being calculated, there might be a need for another matrix to perform another computation, let's say a max pooling. Since we have chosen a *large capacity cache*, the matrix is loaded into the cache(let's say the matrix shape is [800, 600]). The process of copying the data to the cache and more formally, *context switching* between threads(We have assumed that thread 1 uses matrix A and thread 2 needs matrix B) can be a huge bottleneck. When the number of threads increase, the context switching happens more frequently and the cost of that is exhaustively expensive. Another phenomenon can be possibly the *False Sharing* which is happened while a particular section of an array is updated in the cache of one core. The other core therefore invalidates the same contents in its cache although there wasn't the need for that.

4 Computing Analysis of Asus N56JK

We'll provide the following information below. These information was extracted using *lscpu* command on an *Ubuntu 16.04*.

Overall cpu architecture — Frequencies — Cache Layers & Line Size

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 60
Model name:            Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz
Stepping:              3
CPU MHz:               2494.217
CPU max MHz:           3500.0000
CPU min MHz:           800.0000
BogoMIPS:              4988.43
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              6144K
NUMA node0 CPU(s):     0-7
```

Now let's see the memory details. Using the command *sudo lshw -short -C memory* we can extract memory details on Linux terminal.

Overall memory status

```
H/W path          Class          Description
=====================================================
/0/0              memory         64KiB BIOS
/0/8/9            memory         1MiB L2 cache
/0/8/a            memory         256KiB L1 cache
/0/8/b            memory         6MiB L3 cache
/0/c/2            memory         4GiB SODIMM DDR3 Synchronous 1600 MHz
```

Finally, we'll show the *Intel(R) Core(TM) i7-4710HQ* architecture and core connection topology. According to page 6 of the 2012 IDF presentation "Technology Insight: Intel Next

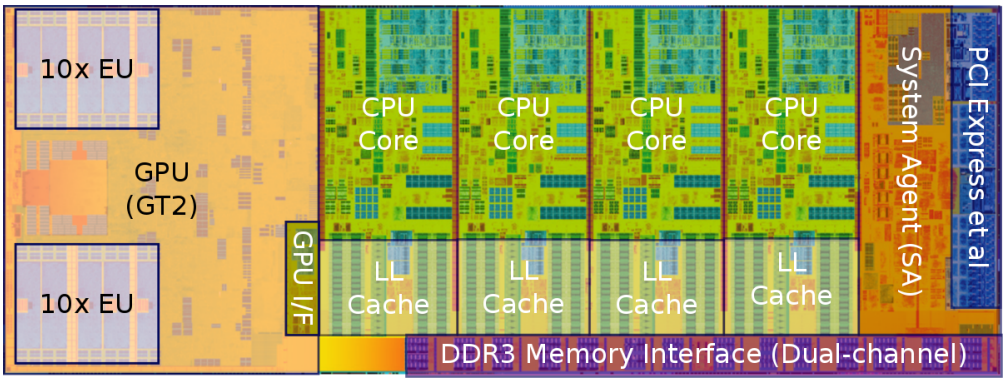

Figure 4.1: Intel(R) Haswell Quad-core Floorplan.

Generation Microarchitecture Code Name Haswell", Haswell retains the *ring* interconnect of Sandy Bridge and Ivy Bridge.

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.

[4] Rodgers, David P. *Improvements in multiprocessor system design*. ACM SIGARCH Computer Architecture News. New York, NY, USA: ACM, June 1985.

[5] Gustafson, John L. *Reevaluating Amdahl's Law*. Communications of the ACM, May 1988.

[6] X.H. Sun, and L. Ni. *Scalable Problems and Memory-Bounded Speedup*. Journal of Parallel and Distributed Computing, September 1993.

[7] Xian-He Sun and Lionel Ni. *Another View on Parallel Speedup*. IEEE Supercomputing Conference, 1990.