Chapter 1

Theory of HPC

1.1 Introduction

High Performance Computing (HPC) takes his roots from the beginning of computer odyssey in the middle 20th century. A lot of rules, observations, theories and even Computer Science field itself emerged from it. In order to understand and characterize HPC and supercomputers, some knowledge on theory is required. This part describes the Von Neumann model, the generic model of computer on which every nowadays machine is built. It is presented along with the Flynn taxonomy that is a classification of the different execution models. Base on those elements we also present the differents shared memory models.

We then give more details on what is and how to reach performances though parallelism. And thus we need to define what performance implies in HPC.

The Amhdal's and Gustafson's laws are presented and detailed and thus the Strong and Weak scaling used in our study.

1.2 Von Neumann Model

First computers, in early 20th, were built using vacuum tubes making them high power consuming, hard to maintain and expansive to create. The most famous of first vacuum tubes supercomputers, The ENIAC, was based on decimal system. It might be the most known of first supercomputers but the real revolution came from its successor. In 1944 the first binary system based computer, called the Electric Discrete Variable Automatic Computer (EDVAC), was created. In the EDVAC team, a physists described the logical model of this computer and provides a model on which every nowadays computing device is based.

John Von Neumann published its *First Draft of a Report on the EDVAC* [VN93] in 1945. Extracted from this work, the model know as the Von Neumann model or more generally Von Neumann Machine appears. The model is presented on Fig. 1.1.

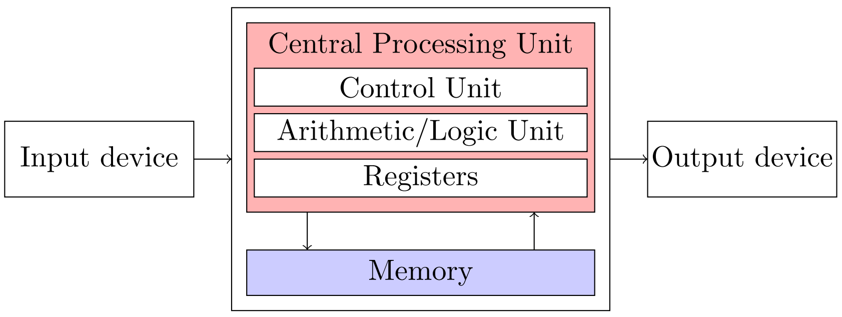


Figure 1.1: Von Neumann model

On that figure we can identify three parts, the input and output devices and in the middle the computational device itself.

Input/Output devices The input and output devices are used to store in a read/write way data. They can be represented as hard drives, solid state drives or even monitors or printers.

Inside the computational device we find the memory, for the most common nowadays architectures it can be considered as a Random Access Memory (RAM). Several kind of memory exists and will be discussed later.

Central Processing Unit The Central Processing Unit, CPU, is composed of several elements in this model. On one hand, the Arithmetic and Logic Unit, ALU, which takes as input one or two values and apply an operation on those data. They can be either logics with operations such as AND, OR, XOR, etc. or arithmetique with operations such as ADD, MUL, SUB, etc. Of course those operations are way more complex on modern CPUs. On the other hand, the Control Unit, CU, which control the data carriage to the ALU from the memory and the operation to be perform on data. It is also the part that takes care of the Program Counter (PC), the address of the next instruction in the program. We can also identify the Register section which represent data location used for both ALU and CU to store temporary results, the current instruction address, etc. Some representation may vary, the Registers can be represented directly inside the ALU or the CU.

Buses The links between those elements are important and called Buses and can be separated between data buses, control buses and adresses buses.

The usual processing flow on such an architecture can be summarized as a loop:
- Fetch next instruction from memory;
- Decode instruction using the Instruction Set Architecture (ISA). Known ISA are Reduce Instruction Set Architecture (RISC) and Complex Instruction Set Architecure (CISC);
- Evaluate operand(s) address(es);
- Fetch operand(s) from memory;
- Execute operation(s), with some instructions sets and new architectures several instructions can be processed in the same clock time;
- Store results, increase PC;

Every devices or machines we will describe in the next chapter will have the same architecture as a basis.

1.2.1 Terminology

Before characterizing the execution models, some terminology must be set to describe properly the machines.

Core: A core is a Von Neumann machine.

Socket/Host: A socket is mistakenly called a CPU in nowadays language. It is, for multi-cores sockets, composed of several cores. The name Host comes from the Host-Device architecture using accelerators.

Accelerators/Devices: Accelerators are devices that, in addition to the CPU, provide additional computation power. We can identify them as GPUs, FPGAs, ASICs, etc. A socket can have access to one or more accelerators and sockets can also share their usage.

Node: A node regroup one or more sockets that usually share memory and, linked to the sockets, one or more accelerators.

Cluster/Supercomputer The cluster group several nodes though an interconnect network.

Instruction(s) stream(s) / Data stream(s)	Single Data (SD)	Multiple Data (MD)
Single Instruction (SI)	SISD	SIMD
Multiple Instructions (MI)	MISD	MIMD

Figure 1.2: Flynn's taxonomy

1.3 Flynn taxonomy and executions models

The Von Neumann model gives us a generic idea of how a computational unit is fashioned. The constant demand in more powerful computers required the scientists to find more way to provide this computational capacity. In 2001, IBM proposed the first multi-core processor on the same die, the Power4 with its 2 cores. This evolution required new paradigms. A right characterization is then essential to be able to target the right architecture for the right purpose. The flynn taxonomy presents a hierarchical organization of computation machines and executions models.

In this classification [Fly72] from 1972, Michael J. Flynn presents the SIMD, SISD, MISD and MIMD models represented on Fig. 1.2. Every of those for execution model correspond to a specific machine and function.

1.3.1 Single Instruction, Single Data: SISD

This is the model corresponding to a single core CPU like in the Von Neumann model. This sequential model takes one instruction, operates on one data and the result is then store and the process continues over. SISD is important to consider as a reference for computational time and will be considered in the next part for Amdahl's and Gustafson's laws.

1.3.2 Single Instruction, Multiple Data: SIMD

This is the execution model corresponding to a many-core architecture like a GPU. SIMD can be extended from 2 to 16 elements for classical CPUs to hundreds and even thousands of core for GPGPUs. In the same clock, the same operation is executed on every process on different data. The best example stay the work on matrices like a stencil, same instruction executed on every element of the matrix.

1.3.3 Multiple Instructions, Multiple Data: MIMD

Every element executes its own instructions on its own data set. This can represent the behavior of a CPU using several cores, threads or even the differents nodes of a supercomputer cluster.

1.3.4 Multiple Instructions, Single Data: MISD

This last model can correspond to a pipelined computer but even in this case the data are modified after every operations. This is the least common exection model.

1.3.5 SIMT

We can also find another characterization to describe the new GPUs architecture: Single Instruction, Multiple Threads. This appears in one of NVIDIA's company paper [LNOM08]. This model describes a stack of SIMD architectures, every block of threads is working with the same

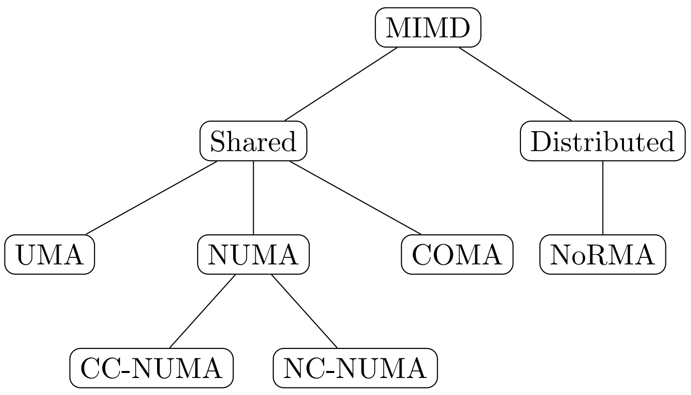


Figure 1.3: MIMD memory models

control processor on different data. This is the model we describe in next chapter used for the *warps* model in NVIDIA CUDA.

1.4 Memory

In addition of the execution model and parallelism the memory accesses parterns have a main role on performances especially in SIMD and MIMD.

Different memory technologies exists and the aim is always greater capacity, better speed and bandwidth while keeping the data integrity.

1.4.1 Memory technologies

We present here the volatile memory, represented in the memory part of the Von Neumann model.

SRAM

The Static Random Access Memory is built using so called ”flip-flop” circuits that can store the data as long as the machine is powered. This kind of memory is very expensive to produce due to the number of component needed and the size of the memory. Therefore it is usually limited for small amount of storage. The SRAM is mainly used for cache memory.

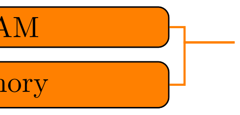
Cache is a memory mecanism that is useful to consider when targeting performance. This little memory is built over several levels. The closer to the CPU is L1, then L2 and generally no more than L3 except on specific architecture. When look for a data the CP will first check the L1 cache, otherwise L2 and L3 to get the data to higher level. This is based on the idea that if a data is used, it shall be use again in the near future.

DRAM

The Dynamic Random Access Memory, at the opposite to the SRAM, is based on transistors and capacitors to store the binary information. This memory is less expansive to produce but needs to be refresh at a determined time however the data are lost. There is several sub categories of DRAM used in different devices.

Depending on the way the bus are used we can find Single Data Rate, SDR, Double Data Rate, DDR and QDR, Quad Data Rates DRAM memories. The number of data carried can go from 1x to 4x but the limitation of those products is the price of memory constantly rising.

We can also find Error-Correcting Code, ECC, memory which implements a bunch of data correction algorithm be garanty the validity of them when error is not allowed.



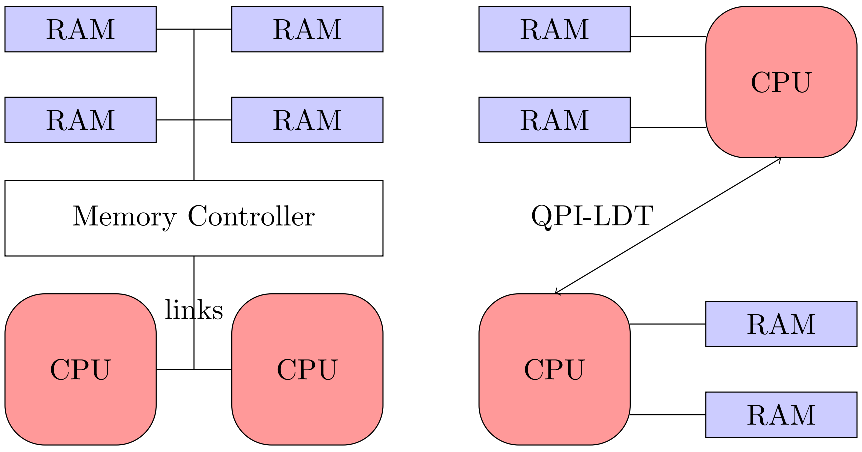


Figure 1.4: UMA vs NUMA

The different types of memory for MIMD model are summed up in Fig.1.3. Two main categories can be extract, share or distributed memories.

1.4.2 Shared memory

In case of the SISD the memory access is just serial and no really rules needs to be set for its usage. When it comes to multi-threaded and multi-cores like MIMD or SIMD execution models several kind of memory models are possible. We give a description of the most common shared memories architecures.

UMA

The Uniform Memory Access is a global memory shared by every threads or cores. In UMA every processors us its own cache as local memory. The addresses can be accessed directly by each processors which make the access time ideal. The downside is that more processors require more buses and thus UMA is hardly scalable. The cache consistancy problem also appears in this context and will be discussed in next part. Indeed, if a data is loaded in one processor cache and modifies, this information need to be spread to the memory and maybe other processes cache.

With the arising of accelerators like GPUs and their own memory, some constructors found ways to create UMA with heterogeneous memory. AMD creates the heterogeneous UMA, hUMA [RF13], in 2013 allowing CPU and GPU to target the same memory area.

NUMA

In Non Unified Memory Access every processor have access to its own memory but allows other processors to access those area though Lightning Data Transport, LDT or Quick Path Interconnect, QPI, for Intel architectures.

As we mention for the UMA memory, even if the processors does not directly access to the memory, a cache coherency is important. Two methods are possible: on one hand, the most used is Cache-Coherent NUMA (CC-NUMA) were protocols are used to keep data coherency through the memory. on the other hand No Cache NUMA (NC-NUMA) force the processes to avoid cache utilization and write results in main memory losing all the benefits of caching data.

COMA

In Cache-Only Memory Accesses, the whole memory is see as a cache from every processes. Attraction memory is setting up and will attract the data near the process that will use those

Name	FLOPS	Year	Name	FLOPS	Year
kiloFLOPS	10^3		petaFLOPS	10^{15}	2005
megaFLOPS	10^6		exaFLOPS	10^{18}	2020 ?
gigaFLOPS	10^9	$\approx$ 1980	zettaFLOPS	10^{21}	
teraFLOPS	10^{12}	1996	yottaFLOPS	10^{23}	

Figure 1.5: Floating-point Operation per Second and years in HPC.

data. This model is less commonly use and lead to, at best, same results as NUMA.

1.4.3 Distributed memory

The previous models are shared memory, in the case where the processes can access memory of their neighbors processes. In some cases, like supercomputer, it would be too heavy for processors to handle the requests of all the others through the network. Each process or node will then possess its own local memory, that can be share with local processes. Then, in order to access to other nodes memory, communications through the network have to be done and copyed in local memory. This distributed memory is called No Remote Memory Access (NoRMA).

1.5 FLOPS, Speedup, efficiency and scalability

In the previous parts we described the differents executions models, characterizations and memory models for HPC. Based on those tools we need to be able to emphasis the performances of a computer and a cluster.

1.5.1 FLOPS

The Floating point Operation Per Second consider the number of floating-point operation that the system will executes in a second. They are an unit of performance for computers, higher FLOPS is better. This is the scale also use to consider supercomputers computational power. For a cluster we can compute the theoretical FLOPS (peak) with:

$$FLOPS_{cluster} = \#nodes \times \frac{\#sockets}{\#nodes} \times \frac{\#cores}{\#sockets} \times \frac{\#GHz}{\#core} \times \frac{FLOPS}{cycle} \tag{1.1}$$

On Fig.1.5, the scale of FLOPS and the year of the first world machine is presented.

FLOPS is the main way to reprensent a computer's performance but other ways exists like Instructions Per Seconds (IPS) or Operations Per Second (OPS). Some benchmarks also provide their own metrics.

1.5.2 Scalability

The scalability express the way a program react to parallelism. When an algorithm is implemented on a serial machine and is ideal to solve a problem, one may consider to use it on more than one core, socket, node or even cluster. Indeed, one may expect less computation time, bigger problem or a combination of both while using more ressources. This completely depend on the algorithm parallelisation and is expressed through scalability. A scalable program will

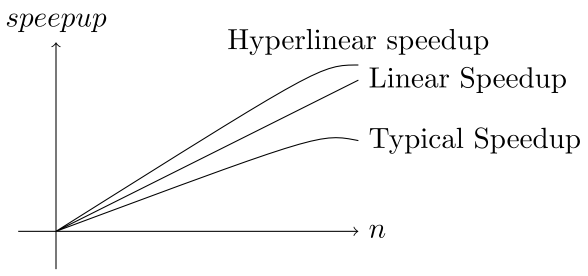


Figure 1.6: Observed speedup

scale on as many processors as we give, whereas a poorly scalable one will give same of even worst results as the serial code. Scalability can be approch using speedup and efficiency.

1.5.3 Speedup and efficiency

The latency is the time necessary to complete a task in a program. Lower latency is better.

The speedup compare the latency of both sequential and parallel algorithm. In order to get relevent results, one may consider the best serial program against the best parallel implementation.

Considering n the number of processes and $n = 1$ the sequential case. And T_n the execution time with n processes and T_1 with one process, the sequential execution time. The speedup can be defined using the latency by the formula:

$$\text{speedup} = S_n = \frac{T_1}{T_n} \tag{1.2}$$

As shown on figure 1.6 several kind of speedup can be observed.

Linear The linear speedup usually represents the target for every program in HPC. Indeed, having the speedup growing exactly as the number of processors grows is the ideal case. Codes fall typical into two cases, typical and hyperlinear speedup.

Typical speedup This represents the most common observed speedup. As the number of processors grows, the program face several of the HPC walls like communications wall or memory wall. The increasing number of computational power is reduiced to the sequential part or lose time in communications/exchanges.

Hyperlinear speedup In some cases we can observe an hyperlinear speedup, meaning that the results in parallel are even better than the ideal case. This can occur if the program can fit exactly in memory for less data on each processors or even fit perfectly for the cache utilization. The parallel algorithm can also be way more efficient than the sequential one.

The efficiency is defined by the speedup devided by the number of workers:

$$\text{efficiency} = E_n = \frac{S_n}{n} = \frac{T_1}{nT_n} \tag{1.3}$$

The efficiency, expressed in percent, represent the evolution of the code stability to growing number of processors. As the number of processes grows, a scalable application will keep an efficiency near 100%.

1.6 Amdhal's and Gustafson's law

The Amdhal's and Gustafson's laws are ways to evaluate the maximal possible speedup for an application taking in account different characteristics.

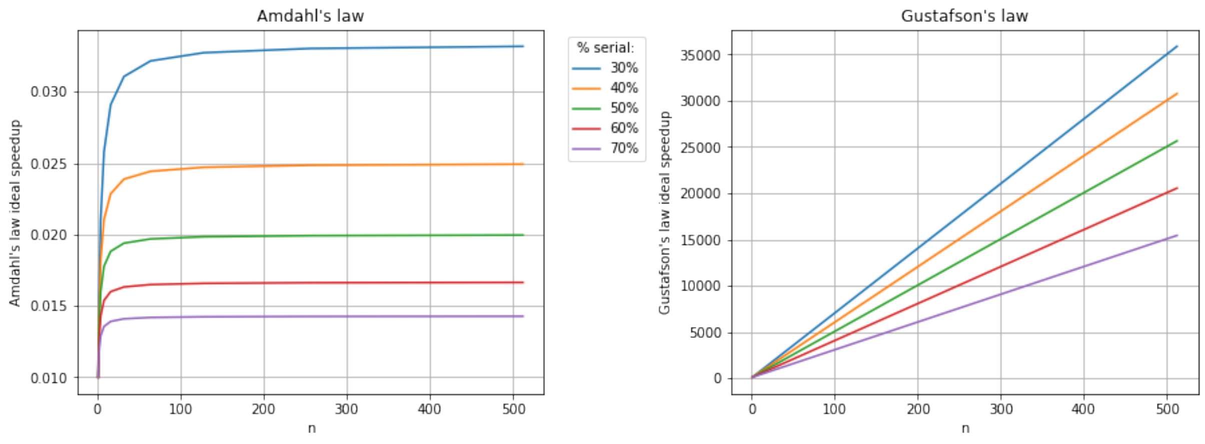


Figure 1.7: Theoretical speedup for Amdahl's (left) and Gustafson's (right) law

1.6.1 Amdahl's law

The Amdahl's law[Amd67] is use to find the theoretical speedup in latency of a program. We can separate a program into two parts, the one that can be execute in parallel and the one that is sequential. The law states that even if we reduce the parallel part using an infinity of processes the sequential part will reach 100% of the total computation time.

Extracted from the Amdahl paper the law can be writen as:

$$S_n = \frac{1}{Seq + \frac{Par}{n}} \tag{1.4}$$

Where $Seq + Par = 1$ and Seq and Par respectively the sequential and parallel ratio of a program. Here if we use up to $n = \inf$ processes, $S_n \leq \frac{1}{Seq}$ the sequential part of the code become the most time consumming.

And the efficiency become:

$$E_n = \frac{1}{n \times Seq + Par} \tag{1.5}$$

A representation of Amdahl's speedup is presented on Fig. 1.7 with varying percentage of serial part. The parallel part is like $Par = (100 - Ser)\%$.

1.6.2 Gustafson's law

The Amdhal's law is focused on time with problem of the same size. John L. Gustafson's idea is that using more computational units, the problem size can grow accordingly. He considered a constant computation time with evolving problem, growing the size accordingly to the number of processes. Indeed the parallel part grows as the problem size do, reducing the percentage of the serial part for the overall resolution.

The speedup can now be estimated by:

$$S_n = Seq + Par \times n \tag{1.6}$$

And the effiencity:

$$E_n = \frac{Seq}{n} + Par \tag{1.7}$$

Both Amdahl's and Gustafson's law are correct and they represent two solution to check the speedup of our applications. The strong scaling, looking at how the computation time vary evolving only the number of processes, not the problem size. The weak scaling, at opposite to strong scaling we look how the computation time evolute varying the problem size keeping the same amount of work per processes.

1.7 Conclusions

In this chapter we presented the different basic tools to be able to understand HPC. The Von Neumann model that represent every nowadays architecture. The Flynn taxonomy that is in constant evolution with new paradigms like recent SIMT from NVIDIA. We also presented the memory types that will be use at different layers in our clusters, from node memory, CPU-GPGPU shared memory space to global fast shared memory. We finished by presenting the most important laws with Amdahl's and Gustafson's laws. We introduice the concept of Strong and Weak scaling that will lead our tests through all the examples in Part II and Part III.

Bibliography

[Amd67] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[Fly72] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972.

[LNOM08] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, 28(2), 2008.

[RF13] Phil Rogers and CORPORATE FELLOW. Amd heterogeneous uniform memory access. *AMD Whitepaper*, 2013.

[VN93] John Von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.