Chapter 1

Models of HPC

1.1 Introduction

High Performance Computing (HPC) takes his roots from the beginning of computer odyssey in the middle of 20th century. A lot of rules, observations, theories emerged from it and even Computer Sciences fields. 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 sequential 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. We also present the different memory models based on those elements.

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

The Amdahl's and Gustafson's laws are presented and detailed along with 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 build. 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 physicists 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 figure 1.1.

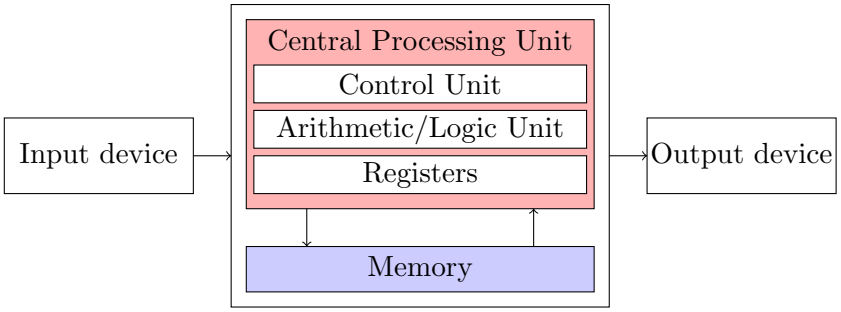

Figure 1.1: Von Neumann model

On that figure we 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, monitors, printers or even mouse and keyboard. The input and output devices can also be the same, reading and writing in the same area.

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 arithmetics with operations such as ADD, MUL, SUB, etc. Of course those operations are way more complex on modern CPUs. On the other hand, we find 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 called Buses and can be separated between data buses, control buses and addresses buses. They will have a huge importance for the first machine optimization, growing the size of the buses from 2, 8, 16 to nowadays 32 and 64.

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

Every devices or machines we describe in the next chapter have this architecture as a basis. One will consider execution models and architecture models to characterize HPC architectures.

1.3 Flynn taxonomy and execution 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 power. 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.

Instruction Stream(s) \ Data Stream(s)	Single Data (SD)	Multiple Data (MD)
Single Instruction (SI)	SISD	SIMD *SIMT*
Multiple Instruction (MI)	MISD	MIMD *SPMD/MPMD*

Table 1.1: Flynn taxonomy for execution models completed with SPMD and SIMT models

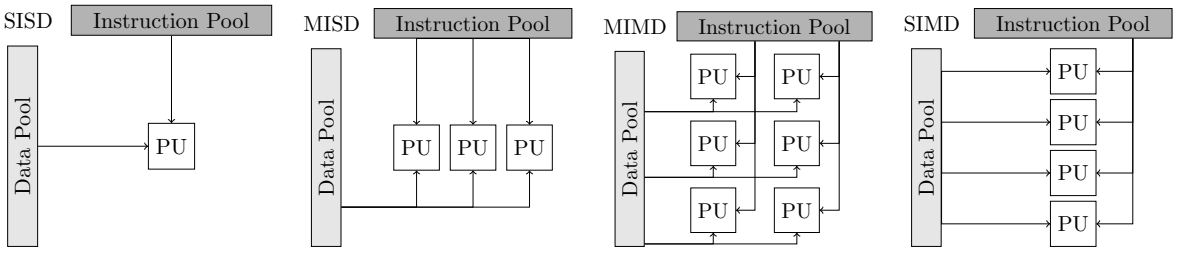

Figure 1.2: Flynn taxonomy schematic representation of execution models

In this classification [Fly72] from 1972, Michael J. Flynn presents the SISD, MISD, MIMD, and SIMD models represented on in table 1.1 and figure 1.2. Every of those 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 computational time and will be considered in the next part for Amdahl's and Gustafson's laws.

1.3.2 Multiple Instructions, Single Data: MISD

This model can correspond to a pipelined computer. Different operations are applied to the datum, which is transfered to the next computational unit and so on. This is the least common execution model.

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 processor using several cores, threads or even the different nodes of a supercomputer cluster. Two subcategories can be identified in MIMD.

SPMD

The Single Program Multiple Data model, SPMD, is the most famous parallelism way for HPC purpose: each process execute the same program. At opposite to SIMD the programs are the same but does not share the same instruction counter. This model was proposed for the first time in [**?**] in 1988 using Fortran. This is the common approach working with runtime like MPI. The programs are the same and the execution similar but based on their ID the processes will target different data.

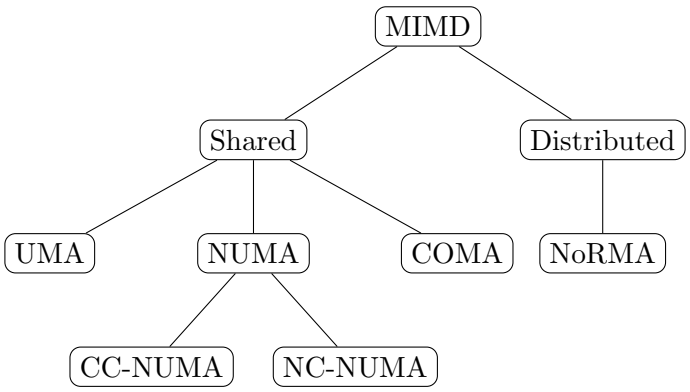

Figure 1.3: MIMD memory models

MPMD

The Multiple Program Multiple Data model is also know for HPC. Generally with a separation between a main program generating data for sub-programs. This is the model on which we work in part II regarding the Langford problem resolution using split of the resolution tree.

1.3.4 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.

1.3.5 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 combination of MIMD and SIMD architectures, every block of threads is working with the same control processor on different data and every block can have its own instruction counter. This is the model we describe in part **??** used for the *warps* model in NVIDIA CUDA.

1.4 Memory

In addition of the execution model and parallelism the memory access patterns have a main role on performances especially in SIMD and MIMD. In this classification we identify three categories: UMA, NUMA and NoRMA for shared and distributed cases. This model have been pointed out in the Johnson's taxonomy[Joh88].

Those different types of memory for SIMD/MIMD model are summed up in figure 1.3.

1.4.1 Shared memory

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 architectures.

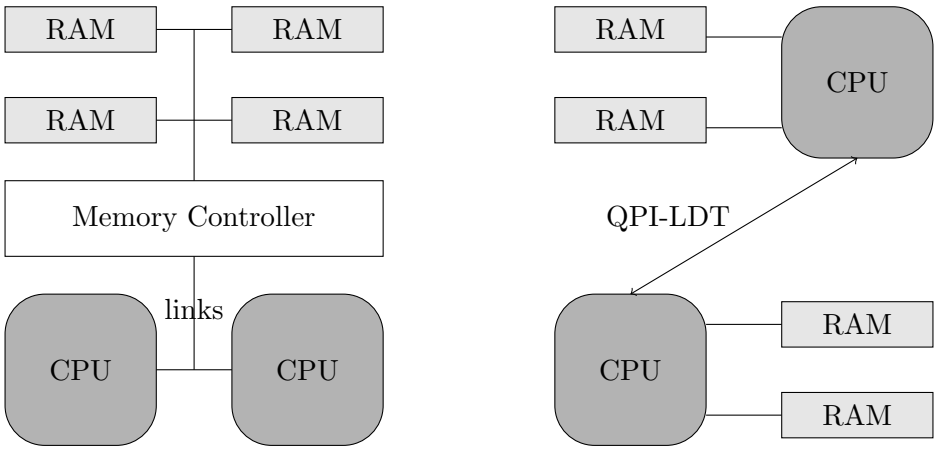

Figure 1.4: UMA vs NUMA memory models

UMA

The Uniform Memory Access is a global memory shared by every threads or cores. In UMA every processor uses its own cache as local private memory. The addresses can be accessed directly by each processor which makes the access time ideal. The downside is that more processors require more buses and thus UMA is hardly scalable. The cache consistency problem also appears in this context and will be discussed in next part. Indeed, if a data is loaded in one processor cache and modified, 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 private 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 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) forces 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 data. This model is less commonly use and lead to, in best cases, same results as NUMA.

1.4.2 Distributed memory

The previous models are based on shared memory, in the case where the processes can access memory of their neighbors processes. In some cases, like supercomputers, 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 copied in local memory. This distributed memory is called No Remote Memory Access (NoRMA).

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^{24}	

Table 1.2: Floating-point Operation per Second and years of reach in HPC.

1.5 Performances characterization in HPC

In the previous parts we described the different 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.

The performance can be of several kind. It can first be define by the speed of the processor itself with the frequency defined in GHz. This information is not perfect because the ALU is not busy all the time due to memory accesses, communications or side effects. It can be used to estimate the highest computational power of a machine.

1.5.1 FLOPS

The Floating point Operation Per Second considers 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 also the scale used to consider supercomputers computational power. For a cluster we can compute the theoretical FLOPS (peak) based on the processor frequency in GHz with:

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

On figure 1.2, the scale of FLOPS and the year of the first world machine is presented. The next milestone, the exascale, is expected to be reach near 2020.

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

1.5.2 Power consumption

Another way to consider machine performance is to estimate the number of operations regarding the power consumption. It can consider all the previous metrics like FLOPS, IPS, IPC or OPS. Benchmarks, like the GREEN500, consider the FLOPS delivered over the watts consumed. For nowadays architectures the many-cores architectures like GPUs seems to deliver the best FLOPS per watt ratio.

1.5.3 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,

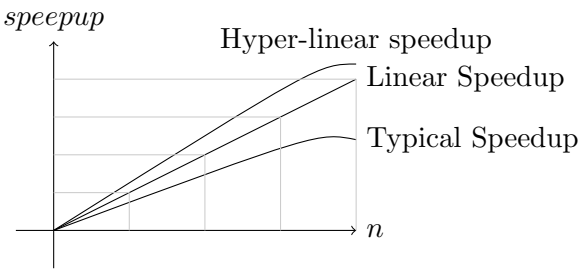

Figure 1.5: Observed speedup: linear, typical and hyper-linear speedups

bigger problem or a combination of both while using more resources. This completely depend on the algorithm parallelization and is expressed through scalability. A scalable program will 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 approach using speedup and efficiency.

1.5.4 Speedup and efficiency

The latency is the time required 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 relevant 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 with T_n the execution time working on n processes and T_1 working on 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 1.5 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 hyper-linear 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 reduced to the sequential part or lose time in communications/exchanges.

Hyper-linear speedup In some cases we can observe an hyper-linear 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 processor or even fit perfectly for the cache utilization. The parallel algorithm can also be way more efficient than the sequential one.

In addition to speedup, the efficiency is defined by the speedup divided by the number of workers:

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

The efficiency, usually expressed in percent, represents 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%.

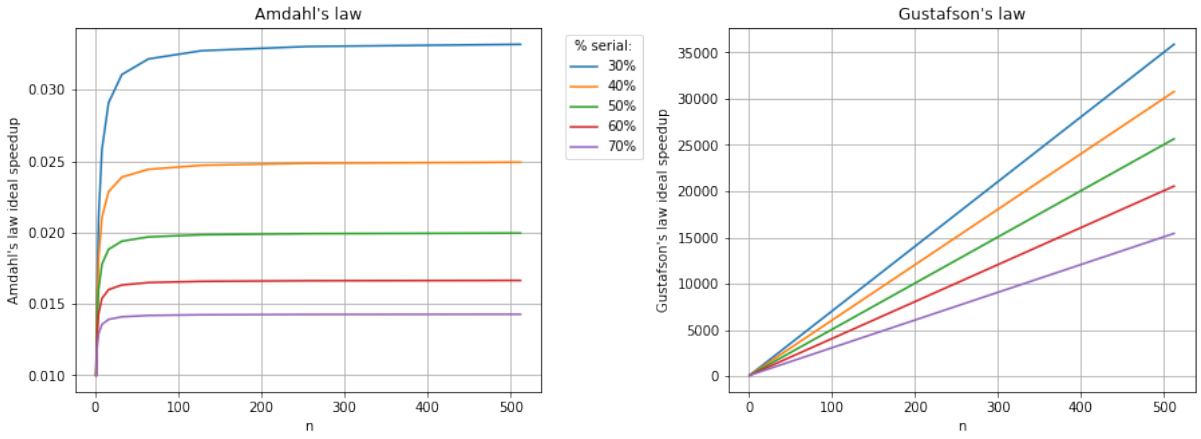

Figure 1.6: Theoretical speedup for Amdahl's (left) and Gustafson's (right) law

1.5.5 Amdahl's and Gustafson's law

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

Amdahl's law

The Amdahl's law[Amd67] is used 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 written 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 $n = \text{inf}$ processes, $S_n \leq \frac{1}{Seq}$ the sequential part of the code become the most time consuming.

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. 1.6 with varying percentage of serial part. The parallel part is like $Par = (100 - Ser)\%$.

Gustafson's law

The Amdahl'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 efficiency:

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

Both Amdahl's and Gustafson's law are applicable 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.

1.6 Conclusions

In this chapter we presented the different basic tools to be able to understand HPC: the Von Neumann model that is implemented in 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 introduced the concept of strong and weak scaling that will lead our tests through all the examples in Part II and Part III.

Those models have now to be confronted to the reality with hardware implementation and market reality, the vendors. The next part will introduce chronologically hardware and their optimization but always keeping a link with the models presented in this part. As there is always a gap between models and implementation we will have to find way to rank and characterize those architecture. This will be discuss in the last chapter.

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.

[Joh88] Eric E Johnson. Completing an mimd multiprocessor taxonomy. *ACM SIGARCH Computer Architecture News*, 16(3):44–47, 1988.

[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.