

Chapter 1

Models for HPC

1.1 Introduction

High Performance Computing (HPC) takes its roots from the beginning of computer odyssey in the middle of 20th century. From this emerged rules, observations, theories and most computer science fields. Knowledge of the theory is required to understand and characterize HPC and supercomputers. This part describes the Von Neumann model, the generic model of sequential computer on which every modern machine is built. The Von Neumann model, along with the Flynn taxonomy which is the classification of the different execution models, will be presented. We also review the different memory models based on those elements.

Parallelism will be discussed in detail and we present how it can be used to reach performances, and thus examine what performance implies in HPC. The Amdahl's and Gustafson's laws are presented in detail, along with the strong and weak scaling used in our study.

1.2 Von Neumann Model

The first early 20th century computers were built using vacuum tubes making them high power consuming, and therefore were hard to maintain and expensive to build. The most famous of first vacuum tubes supercomputers, the ENIAC, was based on a decimal system. As well-known as this supercomputer may be, the real revolution came from its successor. The first binary system based computer was created in 1944 and was called the Electric Discrete Variable Automatic Computer (EDVAC). A physicist from the EDVAC team a described the logical model of this computer and provided a model on which every modern computing device is based.

John Von Neumann published the *First Draft of a Report on the EDVAC* [VN93] in 1945. The model known as the Von Neumann model, more commonly referred as the Von Neumann Machine, came from this work. The model is presented on figure 1.1. The figure has three identified parts: the input and output devices and, in the middle, the computational device itself in the middle.

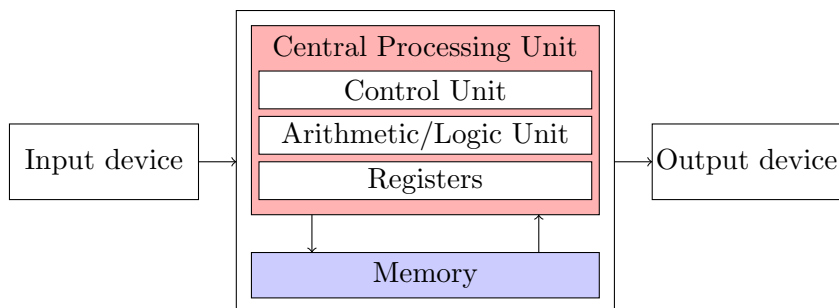


Figure 1.1: Von Neumann model

Input/Output devices The input and output devices are used to store data in a read/write way. 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.

Memory Inside the computational unit we find the general memory to store program and data. The most common architectures it can be considered as a Random Access Memory (RAM). There are several types of memory and will be discussed later in this thesis.

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) takes as input one or two values, the data and applies an operation on them. The operation can be either logic with operators such as AND, OR, XOR, etc. or arithmetic with operators such as ADD, MUL, SUB, etc. These operators may be more complex on modern CPUs.
- Conversely, we find the *Control Unit* (CU) which controls the data carriage to the ALU from the memory and the operation to be performed on data. It is also involved in the Program Counter (PC), which is the address of the next instruction in the program.
- We can identify the *Registers* section which represents data location used for both ALU and CU to store temporary results, the current instruction address, etc. Some representations may vary since 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 in data buses, control buses and addresses buses. These will have a huge importance for the first machines optimizations, growing the size of the buses from 2, 8, 16, 32, 64 and even more for vector machines with 128 and 256 bits.

The usual processing flow on such architectures can be summarized as a loop:

- Fetch instruction at current PC from memory;
- Decode instruction using the Instruction Set Architecture (ISA). The main 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).
- Store results, increase PC.

With the instruction sets and new architectures, several similar operations can be processed in the same clock time. Every device or machine we describe in the next chapter has 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 general idea of how a computational unit is fashioned. The constant demand for more powerful computers required scientists to find different ways 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 become 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.

		Data Stream(s) \rightarrow	
Instruction Stream(s) \downarrow		Single Data (SD)	Multiple Data (MD)
	Single Instruction (SI)	SISD	SIMD <i>SIMT</i>
	Multiple Instruction (MI)	MISD	MIMD <i>SPMD/MPMD</i>

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

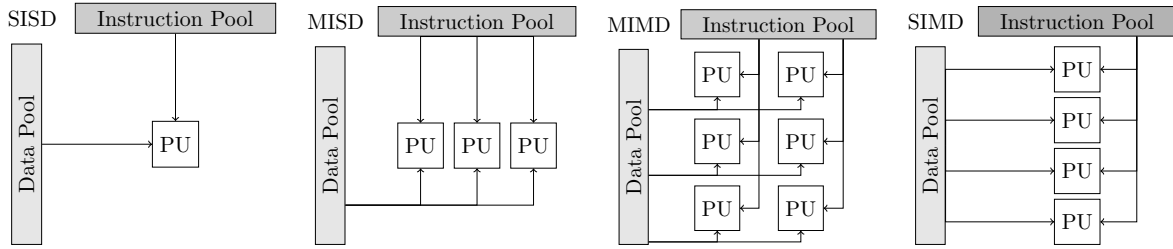


Figure 1.2: Flynn taxonomy schematic representation of execution models

In this classification [Fly72] from 1972, Michael J. Flynn presents the SISD, MISD, MIMD, and SIMD models. Each of those execution models, presented on table 1.1 and figure 1.2, corresponds to a specific machine and function.

1.3.1 Single Instruction, Single Data: SISD

The model corresponding to a single core CPU is similar to the Von Neumann model. This sequential model takes one instruction, operates on one data and the result is then stored before the process continues over. SISD is important to consider for a reference computational time and will be taken into account 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 operation flows are applied to the datum, which is transferred 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 in MIMD executes its own instructions flow 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 are identified in this model: SPMD and MPMD.

SPMD

The Single Program Multiple Data model (SPMD), where each process executes the same program, is the most famous parallelism way for HPC purpose. All programs are the same, but does not share the same instruction counter. SPMD was proposed for the first time in [DGNP88] in 1988 using Fortran. This is the common approach working with runtime like MPI. The programs are the same and the executions are independent but based on their ID the processes will target different data.



Figure 1.3: MIMD memory models

MPMD

The Multiple Program Multiple Data (MPMD) model is also known for HPC. MPMD involves some of the processes to execute different programs, 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 a master program generating tasks for the slaves CPUs/GPGPUs programs.

1.3.4 Single Instruction, Multiple Data: SIMD

This execution model corresponds to many-core architectures like a GPU. SIMD can be extended from two to sixteen elements for classical CPUs to hundreds and even thousands of core for GPGPUs. In the same clock cycle, the same operation is executed on every process on different data. A good example is the work on matrices with stencils: the same instruction is executed on every element of the matrix in a synchronous way and the processing elements share one instruction unit and program counter.

1.3.5 SIMT

Another characterization was determined to describe the new GPUs architecture: Single Instruction Multiple Threads. This first appears in one of NVIDIA's company paper [LNOM08]. This model describes a combination of MIMD and SIMD architectures where every block of threads is working with the same control processor on different data and in such a way that every block has its own instruction counter. The model we describe in Part ??, used for the *warps* model in NVIDIA CUDA.

1.4 Memory

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

Those different types of memory for SIMD/MIMD model are summarized in figure 1.3, and presented below.

1.4.1 Shared memory

Several kinds of memory models are possible when it comes to multi-threaded and multi-cores execution models like MIMD or SIMD models. We give a description of the most common shared memories architectures. Those shared memory models are the keystone for the next parts of

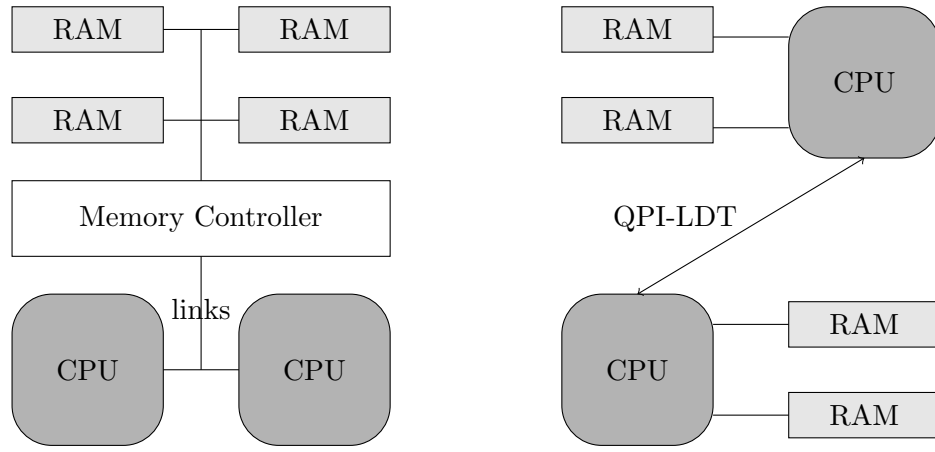


Figure 1.4: UMA vs NUMA memory models

this study. Not only are they found in multi-core but also in many-core architectures with the performances are completely dependent on their usage.

UMA

The Uniform Memory Access architecture have a global memory shared by every threads or cores. Every processor in UMA uses its own cache as local private memory and the accesses consume the same amount of time. 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. 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 has to be spread to the memory and possibly other processes cache.

With the rise of accelerators like GPUs and their own memory, some constructors found ways to create UMA with heterogeneous memory. AMD created the heterogeneous UMA, hUMA [RF13] in 2013, allowed CPU and GPU to target the same memory area, but the performances still needed to be checked in an HPC context.

NUMA

In Non Uniform Memory Access every processor has access to its own private memory but allows other processors to access it though Lightning Data Transport (LDT) or Quick Path Interconnect (QPI) for Intel architectures.

As mentioned for the UMA memory, even if the processors do not directly access to the memory cache, coherency is important. There are two methods possible to aid in coherency. First, the most used is Cache-Coherent NUMA (CC-NUMA) where protocols are used to keep data coherency through the memory. Second, No Cache NUMA (NC-NUMA) forces the processes to avoid cache utilization and write results to main memory, losing all the benefits of caching data.

COMA

In Cache-Only Memory Accesses, the whole memory is seen as a cache shared by all the processes processes. A *Attraction memory* pattern is used to *attract* the data near the process that will use them. This model is less commonly used and, in best cases, leads to same results as NUMA.

1.4.2 Distributed memory

The previous models are dedicated to shared memory, as in the processes can access memory of their neighboring processes. In some cases, like supercomputers, it would be too heavy for

Name	FLOPS	Year	Name	FLOPS	Year
kiloFLOPS	10^3		petaFLOPS	10^{15}	2005
megaFLOPS	10^6		exaFLOPS	10^{18}	2020 ?
gigaFLOPS	10^9	≈ 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.

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 shared only between local processes. In order to access to other nodes memory, communications through the network has to be done and copied into local memory. This distributed memory model is called No Remote Memory Access (NoRMA). This requires transfer schemes that have to be added to local read-write accesses.

1.5 Performances characterization in HPC

Previously we described the different execution models and memory models for HPC. We need to be able to emphasize the performances of a computer and a cluster based on those aspects.

There can be several types of performances. First, it can be defined by the speed of the processors themselves with the frequency defined in GHz. We define the notion of *cycle* to be the number that determines the speed of a processor. This is the amount of time between two pulses of the oscillator at the best frequency with higher cycles per seconds being better. This can be used to estimate the highest computational power of a machine. The information is not perfect, however, because the ALU is not constantly busy due to memory accesses, communications or side effects. Therefore, we need to utilize more accurate ways to characterize performance.

1.5.1 FLOPS

The Floating point Operations Per Second (FLOPS) value considers the number of floating-point operations that the system will execute in a second. Higher FLOPS is better and is the most common 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:

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

With $\#nodes$ the number of computational node of the system, $\frac{\#sockets}{\#node}$ the number of sockets (=processors) per node, $\frac{\#cores}{\#socket}$ the number of cores in the processor, $\frac{\#GHz}{\#core}$ the frequency of each core and finally $\frac{\#FLOP}{\#cycle}$ the number of floating-point operations per cycles for this architecture.

Table 1.2 presents the scale of FLOPS and the year that the first world machine reached this step. The next milestone, the exascale, is expected to be reach near 2020.

There still exists many ways to measure computer's performance such as: Instructions Per Seconds (IPS), Instructions per Cycle (IPC) or Operations Per Seconds (OPS). However, it is hard to consider what an operation or instruction can be. Thus, floating point can be considered as a basis, providing a comparison for performance.



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

1.5.2 Power consumption

An additional way to consider machine performance is to estimate the number of operations regarding the power consumption. This considers all the previous metrics like FLOPS, IPS, IPC or OPS. Benchmarks like the Green500 consider the FLOPS delivered over the watts consumed. For current architectures the many-cores accelerated architectures with GPUs seems to deliver the best FLOPS per watt ratio.

This subject is the underlying goal of our study. Without the power consumption limit we would already be able to build exascale or even more powerful machine by combining billions and billions of nodes. This is not doable due to processors power consumption and cooling cost. The question have been studied in our laboratory with the development of tools like *Powmon* (Power Monitor). This power monitor allows to instantaneously query the power consumption with CPU and GPU on a running node of a cluster.

1.5.3 Scalability

After considering architectures evaluation, one have to measure the applications performances. The scalability expresses the way a program reacts 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, while using more resources one may expect less computation time or access larger instances, or a combination of both. This is completely dependent of the algorithm's parallelization and is expressed through scalability. A scalable program will scale on as many processors as provided, whereas a poorly scalable one will give the same, or even worst results, than the serial code. Scalability can be approached using speedup and efficiency.

1.5.4 Speedup and efficiency

The latency is the time required to complete a task in a program with lower latency being better.

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

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

$$\text{speedup} = S_n = \frac{T_1}{T_n} \quad (1.2)$$

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

$$\text{efficiency} = E_n = \frac{S_n}{n} = \frac{T_1}{nT_n} \quad (1.3)$$

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%.

As shown on figure 1.5 several kinds of speedup can be observed.

Linear, reference speedup: The linear speedup usually represents the target for every program in HPC. To have a constant efficiency means that the speedup grows linearly 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 faces several of the HPC walls, such as communications wall or memory wall. The increasing number of computational power may be lost in parallel overhead resulting in efficiency being reduced.

Hyper-linear speedup: In some cases, we observe an hyper-linear speedup, meaning the results in parallel are even better than the ideal case. This increasing efficiency can occur if the program fits exactly in memory with less data on each processor or even fit perfectly for the cache utilization. The parallel algorithm can also be more efficient than the sequential one. For example, if the search space in an optimized application is divided over processing units, one can find good solutions more quickly.

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 into 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 executed in parallel with optimal speedup and the one that is intrinsically 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:

$$S_n = \frac{1}{Seq + \frac{Par}{n}} \quad (1.4)$$

Where Seq and Par respectively the sequential and parallel ratio of a program ($Seq + Par = 1$). 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 consuming.

And the efficiency become:

$$E_n = \frac{1}{n \times Seq + Par} \quad (1.5)$$

A representation of Amdahl's speedup is presented on Fig. 1.6 with varying percentage of serial part.

Gustafson's law

The Amdahl's law focuses on time with problem of the same size. John L. Gustafson's idea is that using more computational units and the same amount of time, the problem size can grow accordingly. He considered a constant computation time with evolving problem, growing the



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

size accordingly to the number of processes. Indeed the parallel part grows as the problem size does, reducing the percentage of the serial part for the overall resolution.

The speedup can now be estimated by:

$$S_n = Seq + Par \times n \quad (1.6)$$

And the efficiency:

$$E_n = \frac{Seq}{n} + Par \quad (1.7)$$

Both Amdahl's and Gustafson's law are applicable and they represent two solutions 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 the opposite to strong scaling, regards to how the computation time changes with the varying of the problem size, but keeping the same amount of work per processes.

1.6 Conclusions

In this chapter, we presented the different basic considerations to be able to understand HPC: the Von Neumann model that is implemented in every current architecture and the Flynn taxonomy that is in constantly in evolution with new paradigms like recent SIMT from NVIDIA. We discussed the memory types that will be used with the 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. The concept of strong and weak scaling was introduced and 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 chapter introduces chronologically hardware enhancements and their optimization, always keeping a link with the models presented in this part.

We will have to find ways to rank and characterize those architectures as there is always a gap between models and implementation. This will be discussed 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.
- [DGNP88] Frederica Darema, David A George, V Alan Norton, and Gregory F Pfister. A single-program-multiple-data computational model for epex/fortran. *Parallel Computing*, 7(1):11–24, 1988.
- [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.