Part I

HPC and Exascale

Introduction

This part is a state of the art of High Performance Computing and describes the tools we need for our study. High Performance Computing, HPC, does not have a strict definition. The history starts with domain scientists in need of more complex and longer computation for models checking or simulations. They developed their own tools beginning with vacuum tubes computers which can be consider as a milestone in HPC history. Since this first machine the technology became more and more complex at every layer. The hardware conception, the software to handle it and even the models and topologies. HPC is now a field on its own but always dedicated to the end purpose, domain scientist computations. HPC experts are interested in supercomputer construction, architecture and code optimization, interconnection behaviors and creating more software, framework or tools to facilitate access to these very complex machines.

In this part we give a non-exhaustive definition of HPC focusing our needs for this study through three chapters.

We first focus on what are the theoretic models for the machines and the memory we base our work on. This first chapter also present what we define performance for HPC and the main laws that concern it.

The second chapter details the architecture base on these models. We present nowadays dominant constructors and architectures from multi-cores to specific many-cores machines. Representative member of today supercomputers are described. We show that hybrid architectures seems to be the only plausible way to reach the exascale.

In the third chapter we detail the main software to target those complex architectures. We present tools, frameworks and API for shared and distributed memory. We also introduce the mains benchmarks use in HPC world to rank the most powerful supercomputers. This chapter also shows that those benchmarks are not the bests to give an accurate score or scale for "realistic" domain scientists applications.

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 different shared memory models.

We then give more details on what is and how to reach performances though parallelism. 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 use 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 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.

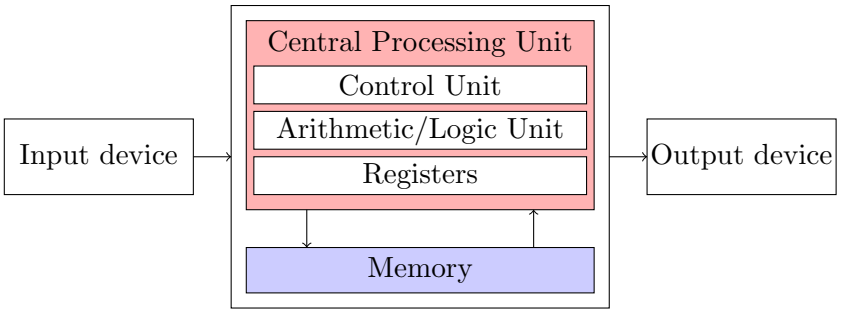

Figure 1.1: Von Neumann model

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.

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 next instruction from memory;
- Decode instruction using the Instruction Set Architecture (ISA). Known ISA are Reduce Instruction Set Architecture (RISC) and Complex Instruction Set Architecture (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 describe in the next chapter have this 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. It can implement complex ISA, huge buses, vectorization operations, etc.

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

Accelerators/Devices: Accelerators are devices that, when attached to the Host, provide additional computational power. We can identify them as GPUs, FPGAs, ASICs, etc. A socket can have access to one or more accelerators. They can also share the accelerator usage.

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

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

Table 1.1: Flynn's taxonomy

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

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 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.

In this classification [Fly72] from 1972, Michael J. Flynn presents the SIMD, SISD, MISD and MIMD models represented on figure 1.1. 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 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 processor using several cores, threads or even the different 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 execution 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

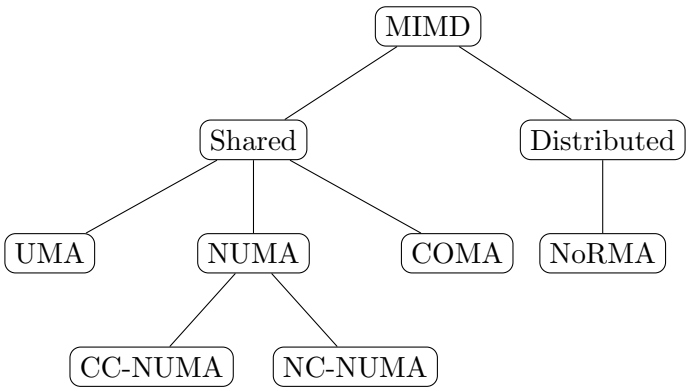

Figure 1.2: MIMD memory models

model describes a stack of SIMD architectures, every block of threads is working with the same 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 patterns 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 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 mechanism 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. Many cache architectures exist like direct, associative, fully associative, etc.

DRAM

The Dynamic Random Access Memory 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.

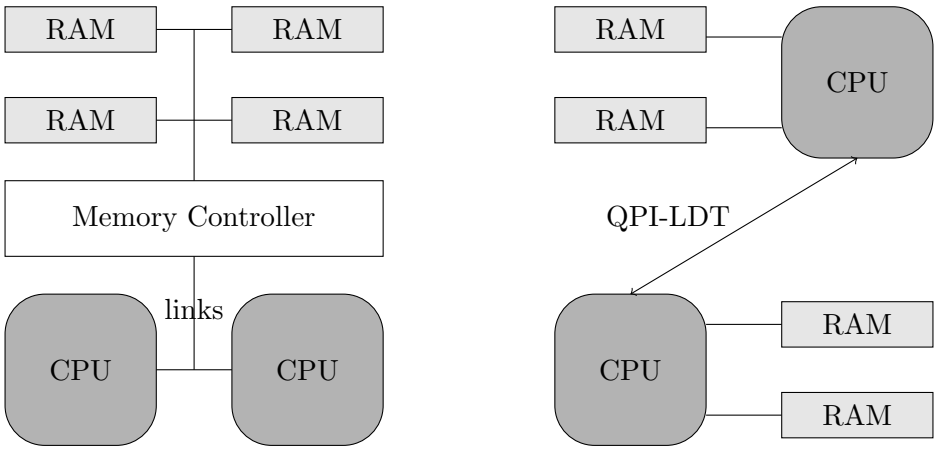

Figure 1.3: UMA vs NUMA memory models

The different types of memory for MIMD model are summed up in figure 1.2. 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 architectures.

UMA

The Uniform Memory Access is a global memory shared by every threads or cores. In UMA every processors use its own cache as local private 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 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) 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^{24}	

Table 1.2: Floating-point Operation per Second and years of reach 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 possesses 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).

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.

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:

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

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) 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,

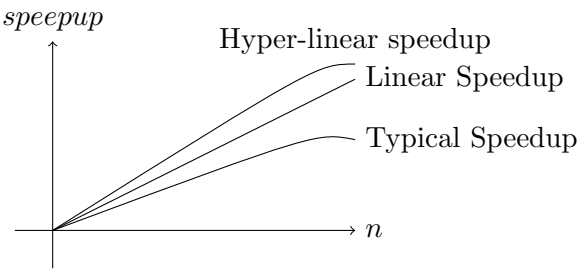

Figure 1.4: Observed speedup

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.

1.5.3 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 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. 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 1.4 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 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 divided 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%.

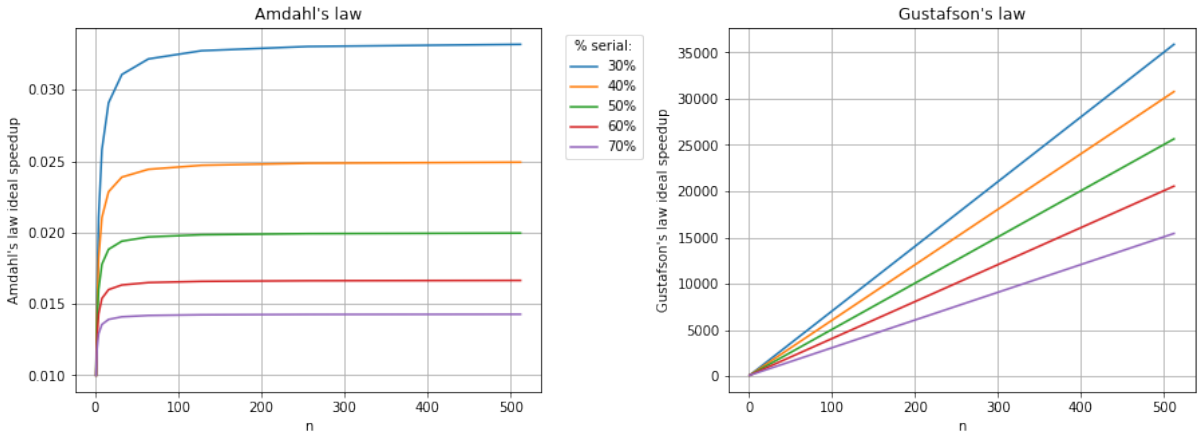

Figure 1.5: Theoretical speedup for Amdahl's (left) and Gustafson's (right) law

1.5.4 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 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 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 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. 1.5 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 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.

1.6 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 introduced the concept of strong and weak scaling that will lead our tests through all the examples in Part II and Part III.

Chapter 2

Hardware in HPC

2.1 Introduction

The knowledge of hardware architecture is essential to reach performances through optimizations. Even if the nowadays software, API, framework or runtime already handle most of optimizations, the last percents of gain always need to be architecture dependent. In this chapter we describe the most important devices architectures from classical processors, General Purpose Graphics Processing Units (GPGPUs), Field Programmable Gate Arrays (FPGAs) and Application-specific integrated circuits (ASICs). in this study we will focus on multi-core processors and GPUs. We based our tests on those devices.

This chapter also details the architecture of supercomputers themselves. This has to go with the description of interconnection network with the most famous interconnection topologies.

2.2 Architectures

The processors, as we know them today, begins theirs history around the 1970s. It is the reflection of the Von Neumann Machine we presented in Chapter I. Historically more features were added to this simple core machine, going from 4 bits 1971 (Intel 4004), 8 bits in 1972 (Intel 8008), 12 bits, 16 bits, 32 bits and 64 bits bus size for recent CPUs. The cores/CPUs also get a huge performance gain based on the frequency acceleration, from the 100kHz to GHz nowadays. Plenty of other optimizations were added:

Multiple CPU cores: Multiple CPU cores on the same die. They can have independent or share part of the cache and access to the same main memory. The first machine were the IBM power4 with dual core.

In/Out-Of-Order: In-order-process is the one describes in previous chapter, the control unit fetches instruction in memory, then the operands and the ALU computes, and finally the results is stored in memory. In this model the time to perform an instruction, cumulation of instruction fetching + operand fetching + computation + store the result, can be high and the ALU itself is busy only one step for computation itself. The idea of Out-of-order is to compute the instructions without following the Program Counter order. Indeed, for independent tasks (this is know based on dependency graphs) while the process fetch the next instructions data, the ALU can perform another operation with already available data.

Pre-fetching: When a data is not available in L1 cache, it has to be moved from either L2 to L1 or L3 to L2 to L1 or in the worst case RAM to L3 to L2 to L1. Pre-fecthing technology is a way to, knowing the next instructions operands, pre-fetch the data in closer cache. The pre-fetch can either be hardware or software implemented and can concern data and even instructions.

Vectorization: Processors allows the instructions to be executed at the same time in a SIMD manner. If the same instruction is executed on coalescent data they can be executed in

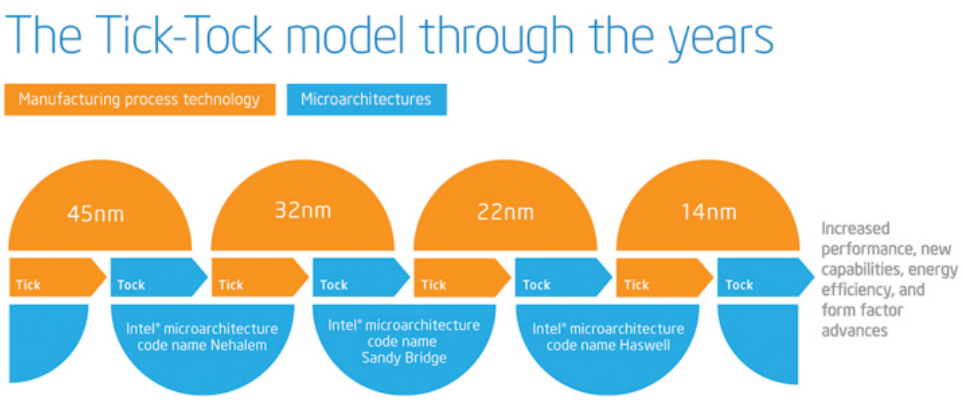

Figure 2.1: Intel Tick-Tock model

the same clock cycle. Of course this tool require specific care during coding.
Those optimizations can be found either in the classical processor model or accelerators. They are of first importance when we detail optimizations on our codes in part 2 and 3.

2.2.1 Multi-core

Nowadays processors share mostly the same architecture. They are called multi-cores and provide up to 2 to 16 cores and each constructor have its own specificities. Those processors are called "Host" because they are usually bootable and most of the accelerators need to be attached to them in order to work.

Intel

Intel was created in 1968 by a chemist and a physicists, Gordon E. Moore and Robert Noyce, in Mountain View, California. Nowadays processors are mostly Intel ones, this world leader equips around 90% of the supercomputers (November 2017 TOP500 list). In 2007 Intel adopted a production model called the "Tick Tock", presented on figure 2.1.

Since its creation this model followed the same fashion, a new manufacturing technology like shrink of the chip with better engraving on a "Tick" and a new micro-architecture delivered on a "Tock". The Intel processors for HPC are called Xeon and features ECC memory, higher number of cores, large RAM support, large cache-memory, Hyper-threading, etc. compared to desktop processors. Every new processor have a code name. The last generations are chronologically called Westemere, Sandy Bridge, Ivy Bridge, Haswell, Broadwell, Skylake and Kaby lake. Kaby Lake, the last architecture of processor, does not exactly fit the usual "Tick-Tock" process because it is just based on optimizations of the Skylake architecture. It is produce like Skylake in 14nm. This model seems to be hard to maintain due to the difficulties to engrave in less than 10nm with quantum tunneling. This leads to using more many-cores architecture and base next supercomputer generations on hybrid models.

Hyper-threading

Another specificity of Intel processor is Hyper-threading (HT). This technology makes a single physical processor appearing as two logical processors for user's level. In fact a processor embedding 8 cores appears as a 16 cores for user. Adding more computation per node can

technically allows the cores to switch context when data are fetched from the memory using the processor 100% during all the computation. A lot of studies have been released on HT from Intel itself [Mar02] to other studies [BBDD06, LAH+02]. This optimization does not fit to all the cases and can be disable for normal use of the processors.

ARM

Back in 1980s, ARM stood for Acorn RISC Machine in reference of the first company implementing this kind of architecture, Acorn Computers. This company later changed the name to Advanced RISC Machine (ARM). ARM is a specific kind of processor based on RISC architecture as its ISA despite usual processors using CISC. The downside of CISC machines makes them hard to create and they require way more transistor and thus energy to work. The ISA from the RISC is simpler and requires less transistors to operate. Therefore, the energy required and the heat dissipated is less important. It would then be easier to create massively parallel processors based on ARM. On the other hand, simple ISA impose more work on the source compilation to fit the simple architecture. That makes the instructions sources longer and therefore more single instructions to execute.

The ARM company provide several version of ARM processors named Cortex-A7X, Cortex-A5X and Cortex-A3X respectively balancing highest-performances, performances and efficiency and less power consumption. We find here the same kind of naming as Intel processors.

The new ARMv8 architecture starts to have the tools to target HPC context [RJAJVH17]. The European approach towards energy efficient HPC, Mont-Blanc project[1], already constructs ARM based supercomputers. For the exascale project in Horizon 2020 this project focus on using ARM-based systems for HPC with many famous contributors with Atos/Bull as a project coordinator, ARM, French Alternative Energies and Atomic Energy Commission (CEA), Barcelona Supercomputing Center (BSC), etc. The project is decomposed in several steps to finally reach exascale near 2020. The third step, Mont-Blanc 3, is about to work on a pre-exascale prototype powered by Cavium's ThunderX2 ARM chip based on 64-bits ARMv8.

2.2.2 Many-cores

Several architectures ban be defined as many-cores. Those devices integrate thousands of cores that are usually control by a control unit. We can consider those cores as "simpler" since they have to work synchronously and under the coordination of a control unit. They are based on SIMD Flynn taxonomy. Some devices are specific like the Xeon Phi of Intel integrating a hundred of regular processor cores which can work independently.

GPU

GPUs are based on the SIMD model of the Flynn taxonomy presented previously, *Single Instruction, Multiple Data*. The specific execution model is called SIMT (*Single Instruction, Multiple Thread*). It enables the execution of millions of coordinated threads in a data-parallel mode. Two main companies provide GPGPUs for HPC: NVIDIA and AMD.

NVIDIA GPU architecture The NVIDIA company was fonded in April 1993 in Santa Clara, Carolina, by three persons in which Jensen Huang, the actual CEO. The company name seems to come from *invidia* the Latin word for Envy and vision for the graphics generation.

Known as the pioneer in graphics, cryptocurrency, portable devices and now AI, it seems to be even the creator of the name "GPU". NVIDIA's GPUs, inspired from visualization and gaming at a first glance, are available as a dedicated device for HPC purpose since the company released the brand named *Tesla*. The public GPUs can also be use for dedicated computation but does not feature ECC memory, double precision or special functions/FFT cores. The different

[1] http://montblanc-project.eu/

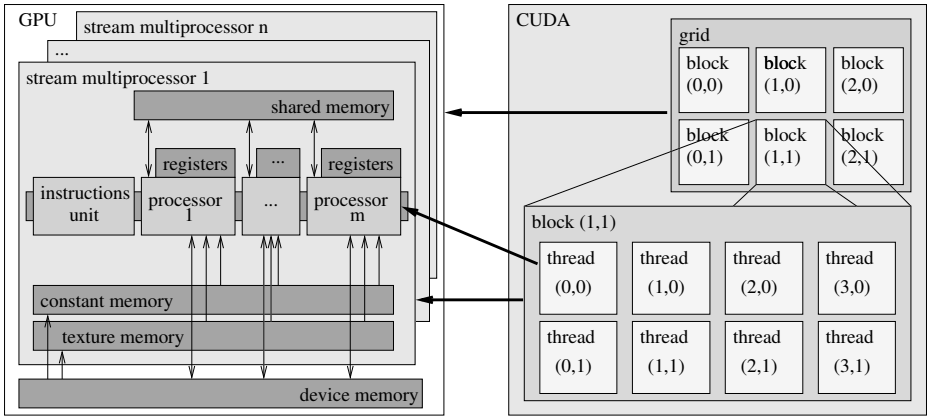

Figure 2.2: NVIDIA GPU and CUDA architecture overview

versions of the architecture are named following famous physicists, chronologically: Tesla, Fermi, Kepler, Pascal and Volta. We describe here the Kepler brand GPU on which we based our study.

As presented in figure 2.2, NVIDIA GPUs include many *Streaming Multiprocessors* (SM), each of which is composed of many *Streaming Processors* (SP). In the Kepler architecture, the SM new generation is called SMX. Grouped into *blocks*, *threads* execute *kernels* functions synchronously. Threads within a block can cooperate by sharing data on an SMX and synchronizing their execution to coordinate memory accesses; inside a block, the scheduler organizes *warps* of 32 threads which execute the instructions simultaneously. The blocks are distributed over the GPU SMXs to be executed independently.

In order to use data in a device kernel, it has to be first created on the CPU, allocated on the GPU and then transferred from the CPU to the GPU; after the kernel execution, the results have to be transferred back from the GPU to the CPU. GPUs consist of several memory categories, organized hierarchically and differing by size, bandwidth and latency. On the one hand, the device's main memory is relatively large but has a slow access time due to a huge latency. On the other hand, each SMX has a small amount of shared memory and L1 cache, accessible by its SPs, with faster access, and registers organized as an SP-local memory. SMXs also have a constant memory cache and a texture memory cache. Reaching optimal computing efficiency requires considerable effort while programming. Most of the global memory latency can then be hidden by the threads scheduler if there is enough computational effort to be executed while waiting for the global memory access to complete. Another way to hide this latency is to use streams to overlap kernel computation and memory load.

It is also important to note that branching instructions may break the threads synchronous execution inside a warp and thus affect the program efficiency. This is the reason why test-based applications, like combinatorial problems that are inherently irregular, are considered as bad candidates for GPU implementation.

We give details on the K20X GPU we mainly used in this study in the ROMEO supercomputer center. This NVIDIA Tesla Kepler GPU is based on the GK110 graphics processor describes in the white-paper[Nvi12] on 28nm process. The K20X comes in active and passive cooling mode with respectively K20Xc and K20Xm. This GPU embeds 2688 CUDA cores distributed in 14 SMX (we note that GK110 normally provides 15 SMX but only 14 are present on the K20X). In this model each SMX contains 192 single precisions cores, 64 double precision cores, 32 special function units and 32 load/store units. In a SMX the memory provides 65536 32-bits registers, 64KB of shared memory L1 cache, 48KB of read-only cache The L2 cache is 1546KB shared by the SMX for a total of 6GB of memory adding the DRAM. The whole memory is protected using Single-Error Correct Double-Error Detect (SECDED) ECC code. The power consumption is estimated to 225W. This GPGPU is expected to produce 1.31 TFLOPS for double-precision and 3.95 TFLOPS of single-precision.

AMD Another company is providing GPUs for HPC, Advanced Micro Devices (AMD). In front of the huge success of NVIDIA GPU that leads from far the HPC market, it is hard for AMD to find a place for its GPGPUs in HPC. Their HPC GPUs are called FirePro. They are targeted using a language near CUDA but not hold by a single company called OpenCL. An interesting creation of AMD is the Accelerated Processing Units (APUs) which embedded the processor and the GPU on the same die since 2011. This solution allows them to target the same memory.

In the race to market and performances, AMD found a accord with Intel to provide dies featuring Intel processor, AMD GPU and common HBM memory. The project is call Kaby Lake-G and announce for first semester of 2018 but for public, not HPC itself.

Intel Xeon Phi

Another specific HPC product from Intel is the Xeon Phi. This device can be considered as a Host or Device/Accelerator machine. Intel describes it as "a bootable host processor that delivers massive parallelism and vectorization". This architecture embedded multiple multi-cores processors interconnected. This is call Intel's Many Integrated Core (MIC). The architectures names are Knights Ferry, Knights Corner and Knight Landing [SGC+16]. The last architecture, Knight Hill, was recently canceled by Intel due to performances and to focus the Xeon Phi for Exascale. The main advantage of this architecture compared to GPGPUs is the x86 compatibility of the embedded cores and the fact this device can boot and use to drive other accelerators. They also feature more complex operations and handle double precision natively. We considered the Xeon Phi in the many-cores architecture despite the fact that it is composed of completely independent processors. This is due to the number of cores that is very high and the fact it can be use as an accelerator instead of the host.

PEZY

Another many-core architecture just appears in the last benchmarks. The PEZY Super Computer 2, PEZY-SC2, is the third many-core microprocessor developed by the company PEZY. The three first machine ranked in the GREEN500 list are accelerator using this many-core die. We also note that in the November 2017 list the 4th supercomputer, Gyoukou, is also powered by PEZY-SC2 cards.

2.2.3 FPGA

Field Programmable Gates Array are device that can de reprogram to fit the needs of the user after their construction. The leader was historically Altera with the Stratix, Arria and Cyclone FPGAs and is now part of Intel. With the FPGAs the user have access to the hardware itself and can design its own circuit. Nowadays FPGA can be targeted with OpenCL programming language. The arrival of Intel in this market promises the best hopes for HPC version of FPGAs. The main gap for users is the circuit building itself, perfect to respond to specific needs but hard to setup.

2.2.4 ASIC

ASICs are dedicated device construct for on purpose. An example of ASIC can be the Gravity Pipe (GRAPE) which is dedicated to compute gravitation given mass/positions. Google leads the way for ASIC and just created its dedicated devices to boost AI bots. We also find ASIC in some optimized communication devices like in fast interconnection network in HPC.

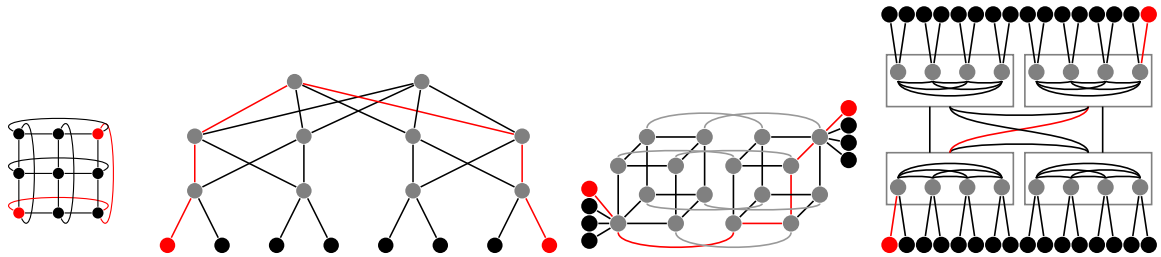

Figure 2.3: Torus, Fat-Tree, HyperX, DragonFly

Name	Gbs	Year	Name	Gbs	Year
Single DR	2.5	2003	Enhanced DR	25	2014
Double DR	5	2005	Highg DR	50	2017
Quad DR	10	2007	Next DR	100	2020
Fourth DR	14	2011			

Table 2.1: InfiniBand technologies name, year and bandwidth

2.3 Interconnection and clusters

2.3.1 Interconnection network

Interconnection network is the way the nodes of a cluster are connected together. Several topologies exists from point to point to multi dimensional torus.

The figure 2.3 is a representation of famous topologies. Each interconnect technology has its own specificity. These networks takes in account the number of nodes to interconnect and the targeted bandwidth/budget. Several declination of each network are not detailed here. the Mesh and the Torus are use as a basis in lower layers of others more complex interconnection networks. A perfect example is the supercomputer called K-Computer describe in the next section. The Fat Tree presented here is a k-ary Fat Tree, higher the position in the tree more connection are found and the bandwidth is important. The nodes are available as the leafs, on the middle level we find the switches and on top the routers. Another topology, HyperX[ABD+09], is base on Hyper-Cube. The DragonFly[KDSA08] interconnect is recent, 2008, and use in nowadays supercomputers.

InfiniBand (IB) is the most spread technology used for interconnect with different kind of bandwidth presented in figure 2.1. It provides high bandwidth and small latency and companies like Intel, Mellanox, etc provide directly adapters and switches specifically for IB.

2.3.2 Remarkable supercomputers

The TOP500 is the reference benchmarks for the world size supercomputers. Most of the TOP10 machines have specific architectures and, of course, the most efficient ones. In this section we give details on several supercomputers about their interconnect, processors and specific accelerators.

Sunway Taihulight

Sunway Taihulight is the third Chinese supercomputer to be ranked in the first position of the TOP500 list. A recent report from Jack J. Dongarra, a figure in HPC, decrypt the architecture of this supercomputer[Don16]. The most interesting point is the conception of this machine,

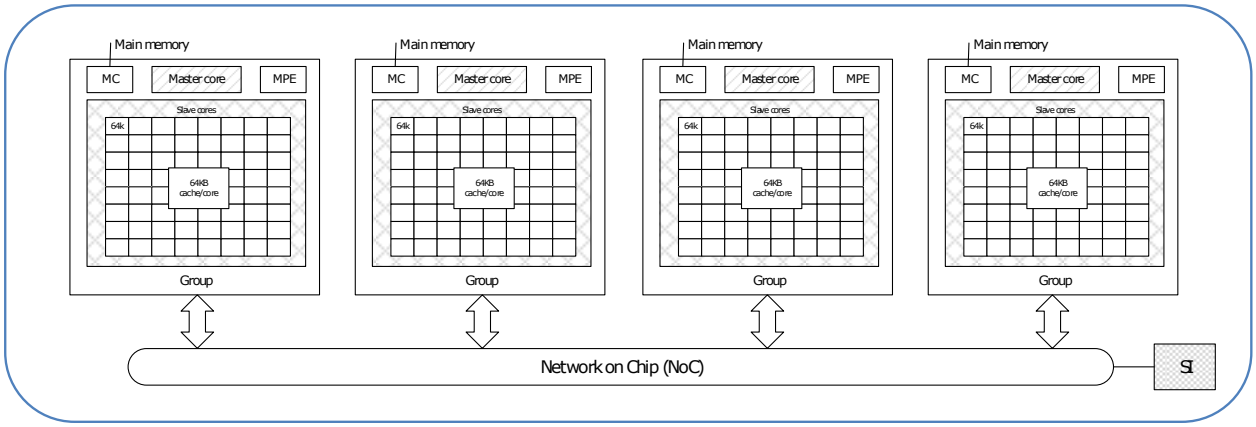

Figure 2.4: Sunway Taihulight node architecture from *Report on the Sunway TaihuLight System*, Jack Dongarra, June 24, 2016.

completely done in China. The Sunway CPUs were invented and built in China, the Vendor is the Shanghai High Performance IC Design Center.

The SW26010, a many core architecture processor, features 260 cores based on RISC architecture and a specific conception depicted on figure 2.4. The processor is composed of the master core, a Memory Controller (MC), a Management Processing Element (MPE) that manages the Computing Processing Elements (CPE) which are the slaves cores.

The interconnect network is called Sunway Network and connected using Mellanox Host Channel Adapter (HCA) and switches. This is a five level interconnect going through computing nodes, computing board, super-nodes and cabinets to the complete system. The total memory is 1.31 PB and the number of cores available is 10,649,600. The peak performance is 125.4 PFLOPS and the Linpack is 93 PFLOPS which induce 74.16% of efficiency.

Piz Daint

The supercomputer of the CSCS, Swiss National Supercomputing Center, is currently ranked 2nd of the November 2017 TOP500 list. This GPUs accelerated supercomputer is a most powerful representative of GPU hybrid acceleration. This is also the most powerful European supercomputer. He is composed of 4761 hybrids and 1210 multi-core nodes. The hybrids nodes embedded an Intel Xeon E5-2690v3 and an NVIDIA Tesla Pascal P100 GPGPU. The interconnect is based on a Dragonfly network topology and Cray Aries routing and communications ASICs. The peak performance is 25.326 TFLOPS using only the hybrid nodes and the Linpack gives 19.590 TFLOPS. The low power consumption rank Piz Daint as 10th in the GREEN500 list.

K-Computer

K-Computer was the top 1 supercomputer of TOP500 2011 list. The TOFU interconnect network makes the K-Computer unique [ASS09] and stands for TOrus FUsion. This interconnect presented in figure 2.5 mixes a 6D Mesh/Torus interconnect. The basic units are based on a mesh and are interconnected together in a 3 dimensional torus. In this configuration each node can access to its 12 neighbors directly. It also provide a fault tolerant network with many routes to reach distant node.

2.4 ROMEO Supercomputer

The ROMEO supercomputer center is the computation center of the Champagne-Ardenne region in France. Hosted since 2002 by the University of Reims Champagne-Ardenne, this so called

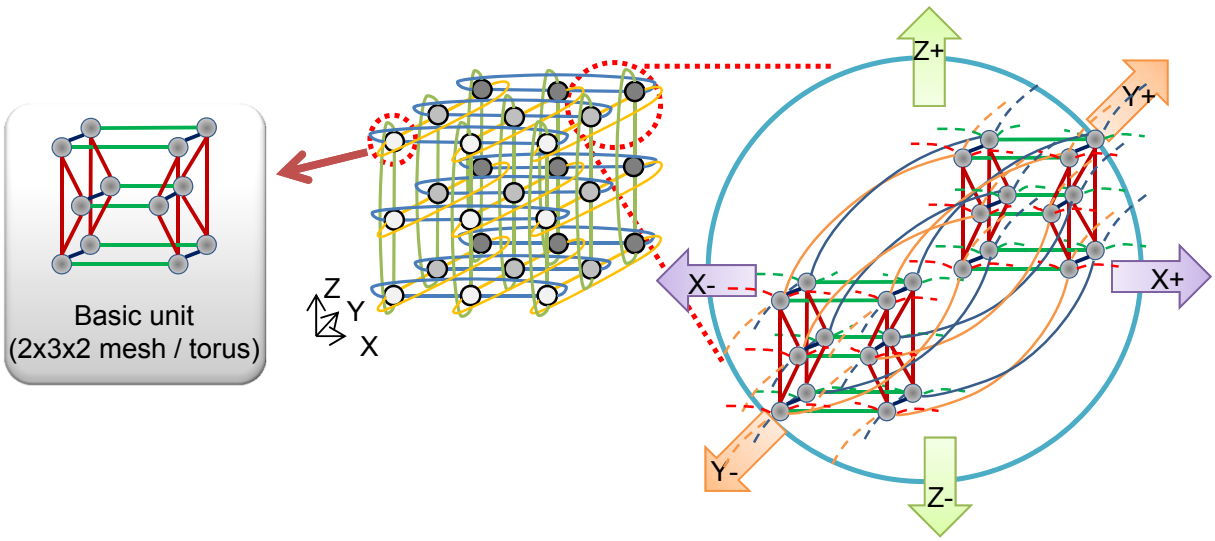

Figure 2.5: TOFU Interconnect schematic from *The K-Computer: System Overview*, Atsuya Uno, SC11

meso-center (French name for software and hardware architectures) is used for HPC for theoretic research and domain science like applied mathematics, physics, biophysics and chemistry.

This project is support by the Champagne-Ardenne region and the CEA (French Alternative Energies and Atomic Energy Commission), aim to host research and production codes of the region for industrial, research and academics purposes.

We are currently working on the third version of ROMEO, installed in 2013. As many of our tests in this study have been done on this machine, we will carefully describe its architecture.

This supercomputer was ranked 151st in the TOP500 and 5th in the GREEN500 list.

2.4.1 ROMEO hardware architecture

ROMEO is a Bull/Atos supercomputer composed of 130 BullX R421 computing nodes.

Each node is composed of two processors Intel Ivy Bridge 8 cores @ 2,6 GHz. Each processor have access to 16GB of memory for a total of 32GB per node, the total memory if 4.160TB. Each processor if linked, using PCIe-v3, to an NVIDIA Tesla K20Xm GPGPU. This cluster provide then 260 processors for a total of 2080 CPU cores and 260 GPGPU providing 698880 GPU cores. The computation nodes are interconnected with an Infiniband QDR non-blocking network structured as a FatTree. The Infiniband is a QDR providing 10GB/s.

The storage for users is 57 TB and the cluster also provide 195 GB of Lustre and 88TB of parallel scratch file-system.

In addition to the 130 computations nodes, the cluster provides a visualization node NVIDIA GRID with two K2 cards and 250GB of DDR3 RAM. The old machine, renamed Clovis, is always available but does not features GPUs.

The supercomputer supports MPI with GPU Aware and GPUDirect.

2.4.2 New ROMEO supercomputer, June 2018

Avoir les info et decrire le nouveau ROMEO

2.5 Conclusion

In this chapter we reviewed the most important nowadays hardware architectures and technologies. In order to use the driver or API in the most efficient way we need to keep in mind the

way the data and instructions are proceed by the machine.

As efficiency is based on computation power but also communications we showed different interconnection topologies and their specificities. We presented perfect use cases of the technologies in nowadays top ranked systems. They also show that every architecture is unique in its construction and justify the optimization work dedicated to reach performance.

We can see through the new technologies presented here that every one is moving toward hybrids architectures featuring multi-core processors accelerated by one or more devices, many-core architectures. The exascale supercomputer of 2020 will be shape with hybrid architectures and they represent the best of nowadays technology for purpose of HPC. Combining CPU and GPUs or FPGA on the same die, sharing the same memory space can also be the solution.

Chapter 3

Software in HPC

3.1 Introduction

After presenting the rules of HPC and the hardware that compose the cluster, we introduce the most famous ways to target those architectures and supercomputers. Several options are present in the language, the multi-processing API, the distribution and the accelerators code. This chapter details the most important software options for HPC programming and include the choices we made for our applications.

Then it presents the software used to benchmark the supercomputers. We present here the most famous, the TOP500, GRAPH500 and GREEN500 to give their advantages and weaknesses.

3.2 Software/API

In this section we present the main runtime, API and frameworks use in HPC and in this study in particular. The considered language will be C/C++, the most present in HPC world along with Fortran.

3.2.1 Shared memory programming

On the supercomputers nodes we find one or several processors that access to UMA or NUMA memory. Several API and language provide tools to target and handle concurrency and data sharing in this context. The two main ones are PThreads and OpenMP for multi-core processors. We can also cite Cilk++ or TBB from Intel.

PThreads

The Portable Operating System Interface (POSIX) threads API is an execution model based on threading interfaces. It is developed by the IEEE Computer Society. It allows the user to define threads that will execute concurrently on the processor resources using shared/private memory. PThreads is the low level handling of threads and the user need to handle concurrency with semaphores, conditions variables and synchronization "by hand". This makes the PThreads hard to use in complex applications and used only for very fine-grained control over the threads management.

OpenMP

Open Multi-Processing, OpenMP[1] [Cha08, Sup17], is an API for multi-processing shared memory like UMA and CC-NUMA. It is available in C/C++ and Fortran. The user is provided with

[1] http://www.openmp.org

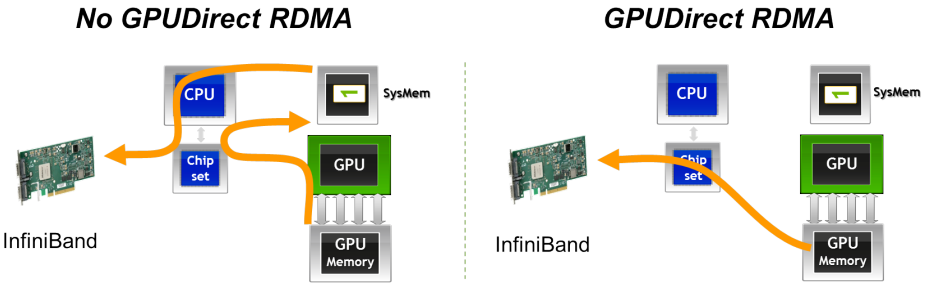

Figure 3.1: GPUDirect RDMA from NVIDIA Developer Blog, *An Introduction to CUDA-Aware MPI*

pragmas and functions to declare parallel loop and regions in the code. In this model the main thread, the first one before forks, command the fork-join operations.

The last versions of OpenMP also allow the user to target accelerators. During compilation the user specify on which processor or accelerator the code will be executed in parallel.

3.2.2 Distributed programming

In the cluster once the code have been developed locally and using the multiple cores available, the new step is to distribute it all over the nodes of the cluster. This step requires the processes to access NoRMA memory from a node to another. Several runtime are possible for this purpose and concerning our study. We should also cite HPX, the c++ standard distribution library, or AMPI for Adaptive MPI, Multi-Processor Computing (MPC) from CEA, etc.

MPI

The Message Passing Interface, MPI, is the most famous runtime for distributed computing [Gro14, Gro15]. Several implementations exists from Intel MPI[2] (IMPI), MVAPICH[3] by the Ohio State University and OpenMP[4] combining several MPI work like Los Alamos MPI (LA-MPI). Those implementation follow the MPI standards 1.0, 2.0 or the latest, 3.0.

This runtime provides directs, collectives and asynchronous functions for process(es) to process(es) communication. A process can be a whole node or one or several cores on a processor.

Some MPI implementations offer a support for accelerators targeting directly their memory through the network without multiple copies on host memory. The data go through one GPU to the other through network and PCIe. This feature is used in our code in part 2 and 3.

For NVIDIA this technology is called GPUDirect RDMA and presented on figure 3.1.

In term of development MPI can be very efficient if use carefully. Indeed, the collectives communications such as *MPI_Alltoall*, *MPI_Allgather*, etc. can be a bottleneck when scaling up to thousands of processes. A specific care have to be taken in those implementation with privilege to asynchronous communications to hide computation than synchronous idle CPU time.

Charm++

Charm++[5] is an API for distributed programming developed by the University of Illinois Urbana-Champaign. It is asynchronous messages paradigm driven. In contrary of runtime like MPI that are synchronous but can handle asynchronous, charm++ is natively asynchronous. It is based on *chare object* that can be activated in response to messages from other *chare objects*

[2]https://software.intel.com/en-us/intel-mpi-library
[3]http://mvapich.cse.ohio-state.edu/
[4]http://www.open-mpi.org
[5]http://charmplusplus.org/

with triggered actions and callbacks. The repartition of data to processors is completely done by the API, the user just have to define correctly the partition and functions of the program. Charm++ also provides a GPU manager implementing data movement, asynchronous kernel launch, callbacks, etc.

A perfect example can be the hydrodynamics N-body simulation code Charm++ N-body Gravity Solver, ChaNGa [JWG+10], implemented with charm++ and GPU support.

Legion

Legion[6] is a distributed runtime support but Stanford University, Los Alamos National Laboratory (LANL) and NVIDIA. This runtime is data-centered targeting distributed heterogeneous architectures. Data-centered runtime focuses to keep the data dependency and locality moving the tasks to the data and moving data only if requested. In this runtime the user defines data organization, partitions, privileges and coherency. Many aspect of the distribution and parallelization are then handle by the runtime itself.

The FleCSI runtime develops at LANL provide a template framework for multi-physics applications and is built on top of Legion. We give more details on this project and Legion on part 3.

3.2.3 Accelerators

In order to target accelerators like GPU, several specific API have been developed. At first they were targeted for matrix computation with OpenGL or DirectX through specific devices languages to change the first purpose of the graphic pipeline. The GPGPUs arriving forced an evolution and new dedicated language to appear.

CUDA

The Compute Device Unified Architecture is the API develop in C/C++ Fortran by NVIDIA to target its GPGPUs. The API provide high and low level functions. The driver API allows a fine grain control over the executions.

The CUDA compiler is called NVidia C Compiler, NVCC. It converts the device code into Parallel Thread eXecution, PTX, and rely to the C++ host compiler for host code. PTX is a pseudo assembly language translated by the GPU in binary code that is then execute. As the ISA is simpler than CPU ones and able the user to work directly in assembly for very fine grain optimizations.

Specific tools have been made for HPC in the NVIDIA GPGPUs.

Dynamic Parallelism This feature allow the GPU kernels to run other kernels themselves. This feature

Hyper-Q This technology enable several CPU threads to execute kernels on the same GPU simultaneously. This can help to reduce the synchronization time and idle time of CPU cores for specific applications.

NVIDIA GPU-Direct GPUs' memory and CPU ones are different and the Host much push the data on GPU before allowing it to compute. GPU-Direct allows direct transfers from GPU devices through the network. Usually implemented using MPI.

OpenCL

OpenCL is a multi-platform framework targeting a large part of nowadays architectures from processors to GPUs, FPGAs, etc. A large group of company already provided conform version of the OpenCL standard: IBM, Intel, NVIDIA, AMD, ARM, etc. This framework allows to produce a single code that can run in all the host or device architectures. It is quite similar

[6] http://legion.stanford.edu/

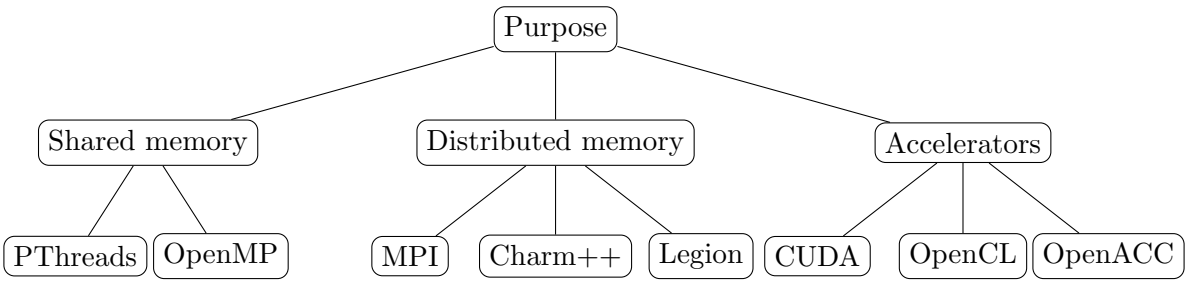

Figure 3.2: Runtimes, libraries, frameworks or APIs

to NVIDIA CUDA Driver API and based on kernels that are written and can be used in On-line/Off-line compilation meaning Just In Time (JIT) or not. The idea of OpenCL is great by rely on the Indeed, one may wonder, what is the level of work done by NVIDIA on its own CUDA framework compare to the one done to implement OpenCL standards? What is the advantage for NVIDIA GPU to be able to be replace by another component and compare on the same level? Those questions are still empty but many tests prove that OpenCL can be as comparable as CUDA but rarely better[KDH10, FVS11].

In this study most of the code had been developed using CUDA to have the best benefit of the NVIDIA GPUs present in the ROMEO Supercomputer. Also the long time partnership of the University of Reims Champagne-Ardenne and NVIDIA since 2003 allows us to exchange directly with the support and NVIDIA developers.

OpenACC

Open ACCelerators is a "user-driven directive-based performance-portable parallel programming model"[7] developed with Cray, AMD, NVIDIA, etc. This programming model propose, in a similar way to OpenMP, pragmas to define the loop parallelism and the device behavior. As the device memory is separated specific pragmas are use to define the memory movements. Research works[WSTaM12] tend to show that OpenACC performances are good regarding the time spend in the implementation itself compare to fine grain CUDA or OpenCL approaches. The little lack of performances can also be explain by the current contribution to companies in the wrapper for their architectures and devices.

The runtime, libraries, frameworks and APIs are summarized in figure 3.2 They are used in combination. The usual one is MPI for distribution, OpenMP and CUDA to target processors and GPUs.

3.3 Benchmarks

This section regroup a bunch of the most famous nowadays benchmarks for HPC.

3.3.1 TOP500

The most famous benchmark is certainly the TOP500[8]. It gives the ranking of the 500 most powerful, known, supercomputers of the world as its name indicates. Since 1993 the organization assembles and maintains this list updated twice a year in June and November.

This benchmark is based on the LINPACK[DMS+94] a benchmark introduced by Jack J. Dongarra. This benchmark rely on solving dense system of linear equations. As specified in this document this benchmark is just one of the tools to define the performance of a supercomputer. It reflects "the performance of a dedicated system for solving a dense system of linear equations". This kind of benchmark is very regular in computation giving high results for FLOPS.

[7] https://www.openacc.org/
[8] http://www.top500.org

In 1965 the Intel co-fonder Gordon Moore made an observation[Pre00] on the evolution of devices. He pointed the fact that the number of transistors in a dense integrated circuit doubles approximately every eighteen months. This is know as the Moore's law. Looking at the last TOP500 figure presented on figure **??**, in the introduction of this document, we saw that nowadays machines does not fit in the law anymore. This is due to the size of transistor and the energy needed to reach more powerful machines. The Moore's law have been sustains by the arrival of many-cores architectures such as GPU or Xeon Phi. Tomorrow machines architectures will have to be based on hybrid with more paradigms and tools to take part of massive parallelism.

3.3.2 GREEN500

In conjunction of the TOP500, the GREEN500[9] focus on the energy consumption of supercomputers. The scale is based on FLOPS per watts [FC07]. Indeed the energy wall is the main limitation for next generation and exascale supercomputers. In the last list, November 2017, the TOP3 machines are accelerated with PEZY-SC many-core devices. The TOP20 supercomputers are all equiped with many-cores architectures: 5 with PEZY-SC, 14 with NVIDIA P100 and 1 with the Sunway many-core devices. This show clearly that the nowadays energy efficient solutions resides in many-core architecture and more than that, hybrid supercomputers.

3.3.3 GRAPH500

The GRAPH500[10] benchmark[MWBA10] focus on irregular memory accesses, and communications. The authors try to find ways to face the futures large-scale large-data problems and data-driven analysis. This can be see as a complement of the TOP500 for data intensive applications. The aim is to generate a huge graph to fill all the maximum memory on the machine and then operate either:

BFS: A Breadth-First Search which is an algorithm starting from a root and exploring recursively all the neighbors. This requires a lot of irregular communications and memory accesses.

SSSP: A Single Source Shortest Path which is an algorithm searching the shortest path from one node to the others. Like the BFS it has an irregular behavior but also requires to keep more data during the computation.

This benchmark will be detailed in Part II Chapter II in our benchmark suite.

3.4 Conclusion

In this chapter we presented the most used software tools for HPC. From inside node with shared memory paradigms, accelerators and distributed memory using message passing runtime with asynchronous or synchronous behavior.

The tools to target accelerators architectures tend to be less architecture dependent with API like OpenMP, OpenCL or OpenACC targeting all the machines architectures. Unfortunately the vendor themselves have to be involve to provide the best wrapper for their architecture. In the mean time vendor dependent API like CUDA for NVIDIA seems to deliver the best performances.

We show through the different benchmark that hybrid architecture start to have their place even in computation heavy and communication heavy context. They are the opportunity to reach exascale supercomputers in horizon 2020.

[9] https://www.top500.org/green500/
[10] https://www.graph500.org/

Conclusion

This part detailed the state of the art theory, hardware and software in High Performance Computing and the tools we need to detail our experiences.

In the first chapter we introduced the models for computation and memory. We also detailed the main laws of HPC.

The second chapter was an overview of hardware architectures in HPC. The one that seems to be the most promising regarding computational power and energy consumption seems to be hybrid architectures. Supercomputers equipped with classical processors accelerated by devices like GPGPUs, Xeon Phi or, for tomorrow supercomputers, FPGAs.

In the third section we showed that the tools to target such complex architecture are ready. They provide the developer a two or three layer development model with MPI for distribution over processes, OpenMP/PThreads for tasks between the processor's cores and CUDA/OpenCL/OpenMP/OpenACC to target the accelerator.

We also showed in the last part that the benchmarks proposed to rank those architectures are based on regular computation. They are node facing realistic domain scientists code behavior. The question that arise is: How the hybrid architecture will handle irregularity in term of computation and communication? This question will be developed in the next part through one example for irregular computation and another for irregular communication using accelerators.

Bibliography

[ABD+09] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren, and Robert S Schreiber. Hyperx: topology, routing, and packaging of efficient large-scale networks. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 41. ACM, 2009.

[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.

[ASS09] Yuichiro Ajima, Shinji Sumimoto, and Toshiyuki Shimizu. Tofu: A 6d mesh/torus interconnect for exascale computers. *Computer*, 42(11), 2009.

[BBDD06] Luciano Bononi, Michele Bracuto, Gabriele D'Angelo, and Lorenzo Donatiello. Exploring the effects of hyper-threading on parallel simulation. In *Distributed Simulation and Real-Time Applications, 2006. DS-RT'06. Tenth IEEE International Symposium on*, pages 257–260. IEEE, 2006.

[Cha08] Barbara Chapman. *Using OpenMP : portable shared memory parallel programming*. MIT Press, Cambridge, Mass, 2008.

[DMS+94] Jack J Dongarra, Hans W Meuer, Erich Strohmaier, et al. Top500 supercomputer sites, 1994.

[Don16] Jack Dongarra. Report on the sunway taihulight system. *PDF). www. netlib. org. Retrieved June*, 20, 2016.

[FC07] Wu-chun Feng and Kirk Cameron. The green500 list: Encouraging sustainable supercomputing. *Computer*, 40(12), 2007.

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

[FVS11] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225. IEEE, 2011.

[Gro14] William Gropp. *Using MPI : portable parallel programming with the Message-Passing-Interface*. The MIT Press, Cambridge, MA, 2014.

[Gro15] William Gropp. *Using advanced MPI : modern features of the Message-Passing-Interface*. The MIT Press, Cambridge, MA, 2015.

[JWG+10] Pritish Jetley, Lukasz Wesolowski, Filippo Gioachin, Laxmikant V Kalé, and Thomas R Quinn. Scaling hierarchical n-body simulations on gpu clusters. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[KDH10] Kamran Karimi, Neil G Dickson, and Firas Hamze. A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581*, 2010.

[KDSA08] John Kim, Wiliam J Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*, pages 77–88. IEEE, 2008.

[LAH+02] Tau Leng, Rizwan Ali, Jenwei Hsieh, Victor Mashayekhi, and Reza Rooholamini. An empirical study of hyper-threading in high performance computing clusters. *Linux HPC Revolution*, 45, 2002.

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

[Mar02] Deborah T Marr. Hyperthreading technology architecture and microarchitecture: a hyperhtext history. *Intel Technology J*, 6:1, 2002.

[MWBA10] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 19:45–74, 2010.

[Nvi12] C Nvidia. Nvidias next generation cuda compute architecture: Kepler gk110. *Technical report, Technical report, Technical report, 2012.[28]¡*, 2012.

[Pre00] I Present. Cramming more components onto integrated circuits. *Readings in computer architecture*, 56, 2000.

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

[RJAJVH17] Alejandro Rico, José A Joao, Chris Adeniyi-Jones, and Eric Van Hensbergen. Arm hpc ecosystem and the reemergence of vectors. In *Proceedings of the Computing Frontiers Conference*, pages 329–334. ACM, 2017.

[SGC+16] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights landing: Second-generation intel xeon phi product. *Ieee micro*, 36(2):34–46, 2016.

[Sup17] Bronis Supinski. *Scaling OpenMP for Exascale Performance and Portability : 13th International Workshop on OpenMP, IWOMP 2017, Stony Brook, NY, USA, September 20-22, 2017, Proceedings*. Springer International Publishing, Cham, 2017.

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

[WSTaM12] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc—first experiences with real-world applications. In *European Conference on Parallel Processing*, pages 859–870. Springer, 2012.