Contents

Part I

HPC and Exascale

Introduction

This part of this thesis contains a state of the art presentation for theory and applications of High Performance Computing. It describes the tools we need for our study. High Performance Computing, HPC, does not have a strict definition. Its history starts with domain scientists in need of more complex and more reliable 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 scientific 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 on models, hardware and tools required for our study in 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 presents what is defined as performance for HPC and the main laws that define it.

The second chapter details the architecture base on these models. We present nowadays platforms with dominant constructors and architectures from multi-core to specific many-core machines. Representative members of today's supercomputers are described. We show that hybrid architectures seem to be the only plausible way to reach the exascale: they offer the best performance per watt ratio and nowadays API/tools allow to target them more easily and efficiently than ever.

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 main benchmarks used in the HPC world in order to rank the most powerful supercomputers. This chapter also shows that those benchmarks are not the best to give an accurate score or scale for "realistic" domain scientists' applications.

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. A lot of rules, observations and theories emerged from it and most of the Computer Science 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 parallelism is 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 century, were built using vacuum tubes making them high power consuming, hard to maintain and expensive to build. The most famous of first vacuum tubes supercomputers, the ENIAC, was based on a 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 was created, called the Electric Discrete Variable Automatic Computer (EDVAC). In the EDVAC team, a physicist described the logical model of this computer and provided 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 is known as the Von Neumann model or more appears as generally Von Neumann Machine. The model is presented on figure 1.1.

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

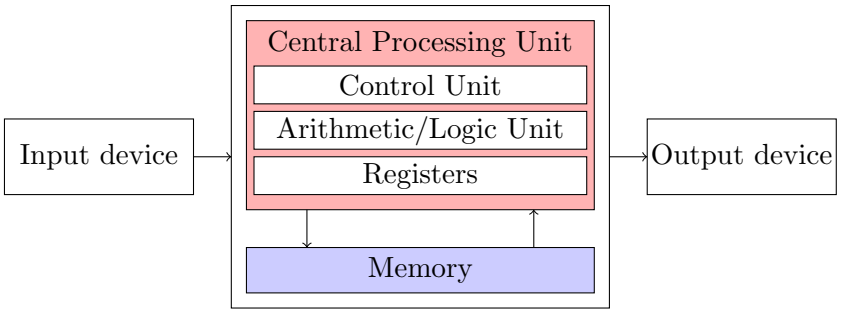

Figure 1.1: Von Neumann model

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 device we have to mention the memory for data storage. For the most common nowadays architectures it can be considered as a Random Access Memory (RAM). Several kinds 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, the data, and applies an operation on them. The operation can be either logic with operations such as AND, OR, XOR, etc. or arithmetic with operations such as ADD, MUL, SUB, etc. Those operations may be more complex on modern CPUs.
- On the other hand, 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 the part that takes care of the Program Counter (PC): the address of the next instruction in the program.
- We can also 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 some instructions 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 generic idea of how a computational unit is fashioned. The constant demand in more powerful computers required the scientists to find more 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

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

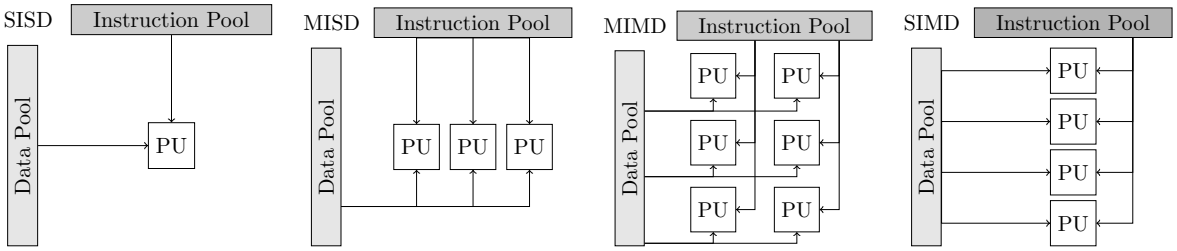

Figure 1.2: Flynn taxonomy schematic representation of execution models

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 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

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 stored and 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 operations flows 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

In MIMD every element 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, is the most famous parallelism way for HPC purpose: each process executes the same program. The programs are the same but does not share the same instruction counter. This model was proposed for the first time in [DGNP88] in

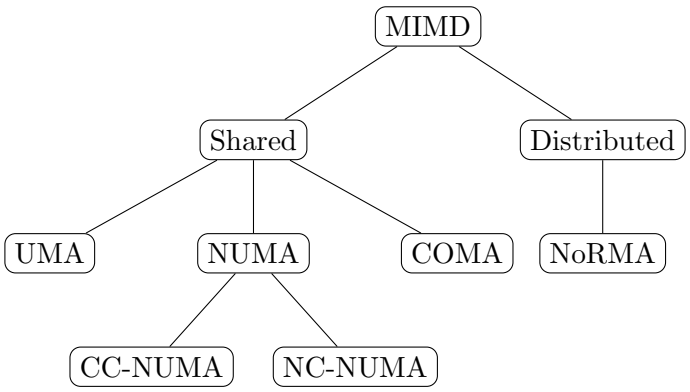

Figure 1.3: MIMD memory models

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.

MPMD

The Multiple Program Multiple Data model is also know for HPC: Some of the processes may execute programs for different purposes. 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 2 to 16 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: same instruction executed on every element of the matrix in a synchronous way: the processing elements share one instruction unit and program counter.

1.3.5 SIMT

We find another characterization 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, 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. This is the model we describe in part 3.3.3, used for the *warps* model in NVIDIA CUDA.

1.4 Memory

In addition to the execution model and parallelism, the memory access patterns have a main role in performances in 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.

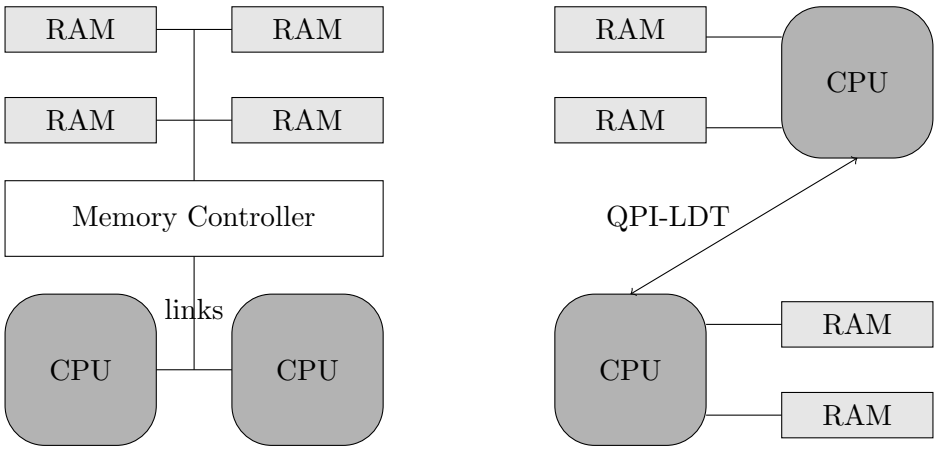

Figure 1.4: UMA vs NUMA memory models

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 key stone for the next parts of our study. We find them in multi-core but also many-core architectures and the performances are completely dependent of their usage.

UMA

The Uniform Memory Access architecture have a global memory shared by every threads or cores. In UMA every processor 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. 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 has 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 created the heterogeneous UMA, hUMA [RF13] in 2013, allowing CPU and GPU to target the same memory area, but the performances still need 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. Two methods are possible: on one hand, the most used is Cache-Coherent NUMA (CC-NUMA) where 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 to main memory, losing all the benefits of caching data.

COMA

In Cache-Only Memory Accesses, the whole memory is see as a cache shared by all the processes processes. An *Attraction memory* pattern is used to *attract* the data near the process that will

use them. This model is less commonly use and, in best cases, lead to same results as NUMA.

1.4.2 Distributed memory

The previous models are dedicated on shared memory, if 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 shared only between local processes. Then, in order to access to other nodes memory, communications through the network have 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

In the previous parts we described the different execution models and memory models for HPC. Based on those aspects we need to be able to emphasis the performances of a computer and a cluster.

The performance can be of several kinds. It can first 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 determine the speed of a processor. This is the amount of time between two pulses of the oscillator at the best frequency. Higher cycles per seconds is better. It can be used to estimate the highest computational power of a machine. This information is not perfect because the ALU is not busy all the time due to memory accesses, communications or side effects. Then we have to go through more accurate ways to characterize performance.

1.5.1 FLOPS

The FLoating point Operations Per Second value considers the number of floating-point operation that the system will execute in a second. Higher FLOPS is better and this 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:

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

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 of the first world machine reached this step. The next milestone, the exascale, is expected to be reach near 2020.

Even though, many ways exist to measure computer's performance: Instructions Per Seconds (IPS), Instructions per Cycle (IPC) or Operations Per Seconds (OPS) but none of these consider what an instruction or operation is. Thus, floating point can be considered as a basis, and a comparison can be provided.

1.5.2 Power consumption

Another way to consider machine performance is to estimate the number of operations regarding the power consumption. It considers 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 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 should just build supercomputer made of an addition of supercomputers. This is indeed not

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.

doable due to processors power consumption and cooling cost. This question have been studied in our laboratory with the development of tools like *Powmon* (Power Monitor). It 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 same of 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. Lower latency is 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:

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

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

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

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%.

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

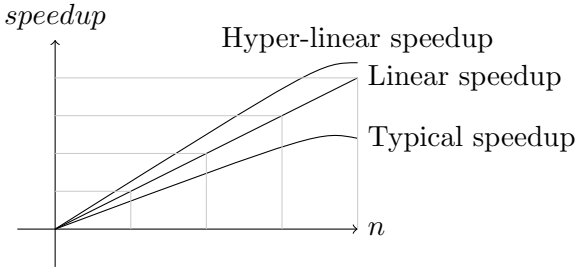

Figure 1.5: Observed speedup: linear, typical and hyper-linear speedups

Linear, reference speedup: The linear speedup usually represents the target for every program in HPC. Indeed, having a constant efficiency, which 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 like communications wall or memory wall. The increasing number of computational power may be lost in parallel overhead, and efficiency reduced.

Hyper-linear speedup: In some cases we observe an hyper-linear speedup, meaning that the results in parallel are even better than the ideal case. This increasing efficiency can occur if the program fits exactly in memory for 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 with optimizations application: if the search space 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:

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

Where Seq and Par respectively the sequential and parallel ratio of a program ($Seq + Par = 1$). Here if we use up to $n = \text{inf}$ 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.

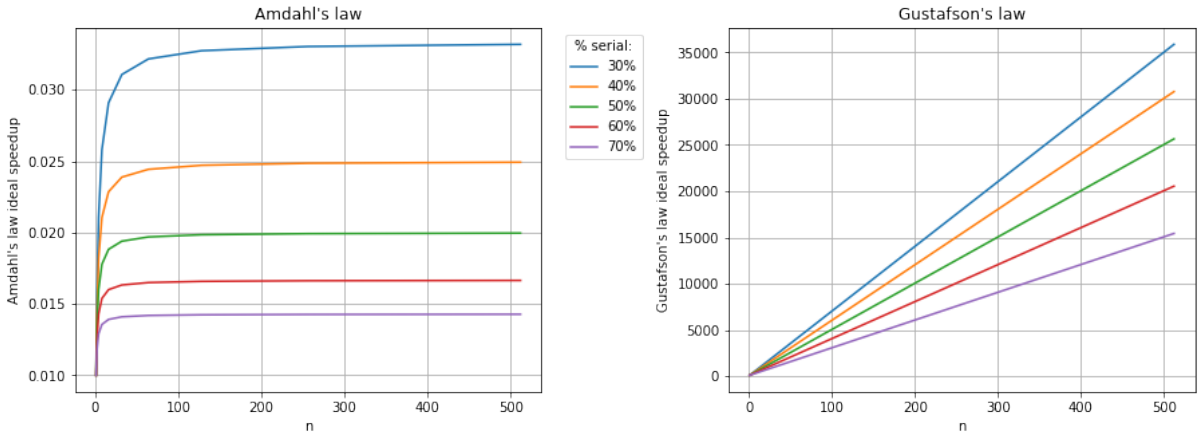

Figure 1.6: Theoretical speedup for Amdahl's (left) and Gustafson's (right) law

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 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:

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

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 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 evolutes varying 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 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 used 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 chapter introduces chronologically hardware enhancements and their optimization and always keeps a link with the models presented in this part.

As there is always a gap between models and implementation we will have to find ways to rank and characterize those architectures. This will be discussed in the last chapter.

Chapter 2

Hardware in HPC

2.1 Introduction

Although parallel models address most of the key points to discuss application performance it also depends on architectures hardware, which may influence the way to consider problems resolution. Thus 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 the optimizations, the last percents of gain are architecture's 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). This study keeps a focus on multi-core processors and GPUs as we based our tests on those devices.

This chapter also details the architecture of some remarkable supercomputers. This has to go with the description of interconnection network for the most used interconnection topologies.

We choose to present the architectures in a chronological order following the models presented in the previous chapter - SISD, MIMD and SIMD/SIMT - and presenting the last released technologies. We also present the optimizations of current technologies with the arising of parallelism and new types of memories.

2.2 Early improvements of Von Neumann machine

In this section we present the different hardware evolution from 1970s single core processors to nowadays multi-core and many-core architectures that are the milestones, the basic units, to build supercomputers. We can see the most important optimizations that are always implemented in the most recent machines: in/out of order processors, pre-fetching strategies, vectorization and the memory technologies breakthroughs.

2.2.1 Single core processors

The first processors, around the 1970s, were built using a single computation core as described in the Von Neumann model. Many evolutions were made on those single core processors for the memory, the order of the instructions and the frequency increase.

Transistor shrink and frequency

A lot of new ways to produce smaller transistors have been discovered. Their size was about $10\mu m$ in 1971 and reach $10nm$ in nowadays machines. This allowed the constructors to add more transistors on the same die and build more complex ISA and features for the CPUs.

In parallel of the shrink of transistors, the main feature for better performances with the single core architectures came from the frequency augmentation, the clock rate. Indeed, as the clock rate get higher, more operations can be performed on the core in the same amount of

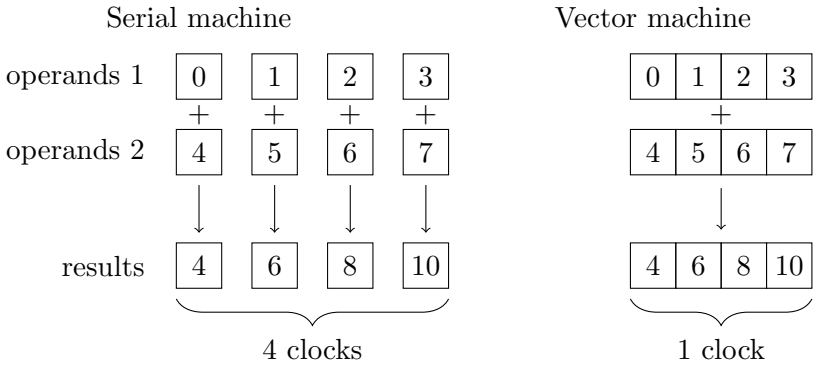

Figure 2.1: Vectorized processeur example on 4 integer addition: 128 bits wide bus

time. In 1970s the frequency was about 4 MHz allowing a maximum of 4 millions of cycles per seconds. Nowadays single cores can work at a frequency of 4GHz and even 5GHz performing billions of operations per cycles, but the next sections will show that due to power consumption and coding considerations, frequency is no more used to improve performances.

In/Out-Of-Order execution

In-order-process is the one described in previous chapter. The control unit fetches instructions and the operands from memory. Then the ALU computes the operation, and finally the result is stored to memory.

In this model the time to perform an instruction is the cumulation of: instruction fetching + operand(s) fetching + *computation* + result storage. This time may be high regarding the use of the ALU for *computation*, technically just one clock cycle. The idea of *out-of-order* execution is to compute the instructions without following the Program Counter order. Indeed, for independent tasks (pointed out with dependency graphs) while the process fetches the next instruction data, the ALU can perform another operation with already available operands. This leads to better usage of computational resources in the CPU and thus better overall performances.

Vectorization

Vector processors allow 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 the same clock cycle. As an example we can execute operations simultaneously on 4 to 8 floats with a bus size of 128 or 256 bits. This requires specific care for coding with *unrolling* and *loop tiling* that will be addressed later in this study, and to avoid bad behavior leading to bad performances. But latest architectures vectorization imposes to slightly lower the frequency of processors.

The Cray-1 supercomputer[Rus78], installed in 1975 in the Los Alamos National Laboratory, is a perfect example of vector processor supercomputer. This supercomputer was designed by Seymour Cray, the founder of Cray Research. Based on vector processor it was able to deliver up to 160 MFLOPS. It was the fastest supercomputer in 1978 and due to its shape and price it was humorously called *the world's most expansive love-seat*.

The behavior of vector machine is presented on figure 2.1 for a 16 bytes vector machine (4 integer of 4 bytes = 128 bits bus). We see on the left that performing the 4 operations requests 4 cycle and, at the opposite, 1 cycle on the right with the vectorized machine.

Linked with the CPU optimizations, the memory optimizations also have to be considered. Indeed, even if the ALU can perform billions of operations per second it needs to be fed with data by fast transfers.

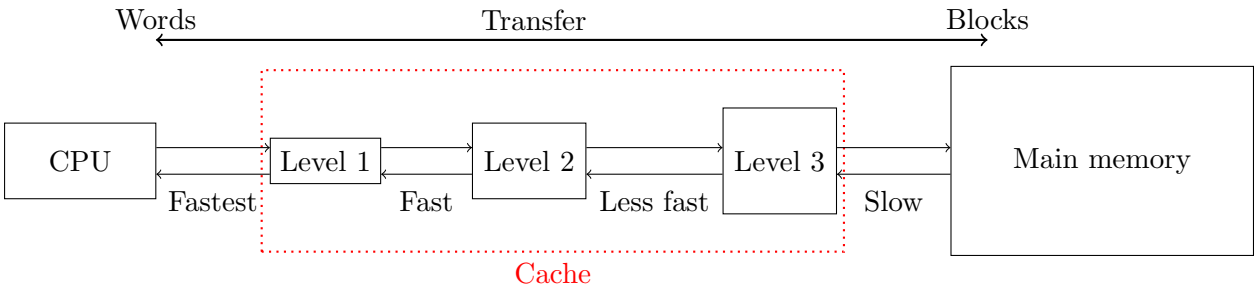

Figure 2.2: Cache memory technology on three levels L1, L2 and L3

Memory technology evolution

The memories technologies optimizations address several aspects. The early 1980s saw the augmentation of bus size from 4 bits to nowadays 32 bits for single precision and 64 bits for double precision. 128 bits or 256 bits buses can also be used to allow vectorization presented just before.

Different kind of technologies are considered: the SRAM and DRAM.

SRAM: The Static Random Access Memory is built using so called "flip-flop" circuits that can store data at anytime time with no time lost in "refresh operations". This kind of memory is very expensive to produce due to the number of transistors by memory cell. Therefore it is usually limited for small amounts of storage. The SRAM is mainly used for cache memory.

DRAM: The Dynamic Random Access Memory is based on transistors and capacitors to store binary information. This memory is less expensive to produce but needs to be refreshed at a determined frequency otherwise the data are lost. This refresh step is in fact a read-write operation on the whole memory at a specific frequency. There are several sub categories of DRAM used in different device depending on the way the bus are used with Single Data Rate (SDR), Double Data Rate (DDR) and Quad Data Rates (QDR). The number of data carried can vary from 1x to 4x but the limitation of those products is the price, constantly rising.

Cache memory:

Cache is a memory mechanism that is useful to consider when targeting performance. The main idea of cache technology is presented on figure 2.2. This memory is built hierarchically over several levels. The closer to the CPU is L1, then L2 and generally no more than L3 except on specific architectures. When looking for a data the CPU first checks if it is present in the L1 cache, otherwise L2 and L3 to get the data to higher level. From the main memory to the L3 cache *blocks* are exchanged, by chunks. With levels L2 and L1, lines of informations are exchanged usually called *words*. This is based on the idea that if a data is used, it shall be used again in the near future. Many cache architectures exist: direct, associative, fully associative, etc. Cache-hits occur when the data required is present in cache, and cache-miss when it has to be retrieved from lower levels or main memory. In a program the ratio of cache-miss has to be kept low in order to reach performance, and the impact may be very high.

Pre-fetching

Pre-fecthing was developed Based on memory optimization and especially the cache. 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 from 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 before the instruction is decoded.

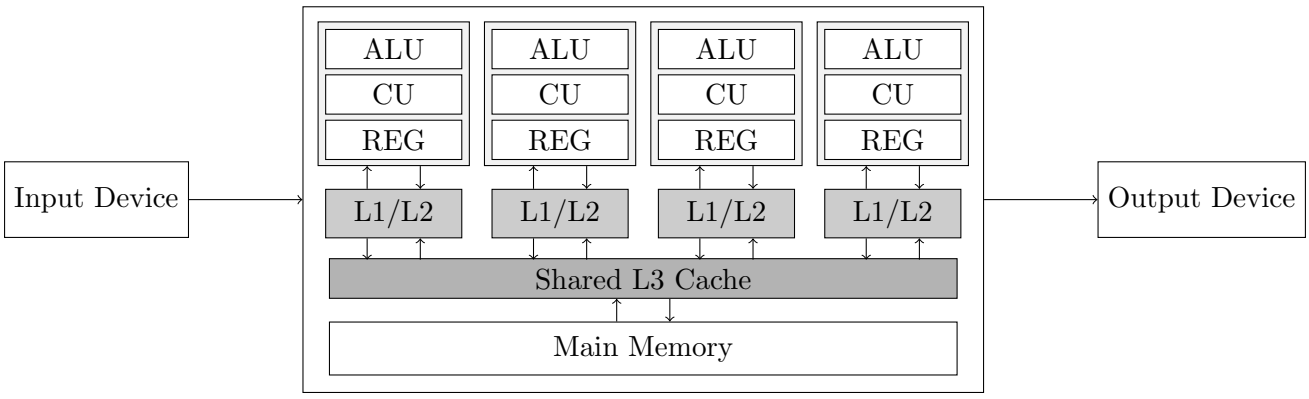

Figure 2.3: Multi-core CPU with 4 cores based on Von Neumann Model

The pre-fetch can either be hardware or software implemented and can concern data and even instructions.

2.2.2 Multi-CPU servers and multi-core processors

Around the beginning of 2000s the limitations of single core processors were too important. The frequency was already high and requested more power consumption and caused more heat dissipation. The first idea was to provide multi-CPU devices, embedding several CPU on the same motherboard and allowing them to share memory. The evolution of that is multi-core, having several processing units on the same die allowed more optimization inside the die and combining all the advantages of single core processors. But CPUs each embedding the function and units required consume n times more energy and cumulate heat effects. Thus, unable to answer the constant augmentation of computational power needed for research and HPC, IBM was the first company to create a multi-core CPU in 2001, the Power4.

Compared to the core inside multi-CPU, multi-core CPU shared one of the material (L3 caches, buses, ...) and are implemented on the same due; this allows to reach the same CPU performances with less transistors and less power consumption, avoiding most of the heat issues.

This architecture is presented on figure 2.3. The main memory is now shared between the cores. The registers and L1/L2 cache are the same but a L3 layer is added to the cache, and consistency has to be maintained over all the cores: if a process modifies a data in the memory this information has to be spread over all the other users of this data, even in their local cache.

We note here that in nowadays language the CPU, as describe in the Von Neumann model, is also the name of the die containing several cores. This is the architecture of most of nowadays processors. TThese multi-cores provide 2 to 32 cores in most cases. Thus the multi-core CPU are called "Host" and the attached accelerators are called "Devices".

2.3 21th century architectures

After years of development and research on hardware for Computer Science and specifically HPC, we present here the latest and best technologies to product efficient and general purpose supercomputers.

We present the latest architectures with multi-core, many-core and specific processors, and the most famous vendors.

2.3.1 Multi-core implementations

The most world spread architecture in public and hight performance computing is the multi-core processors. Most of nowadays accelerators require a classical processor to offload tasks and data

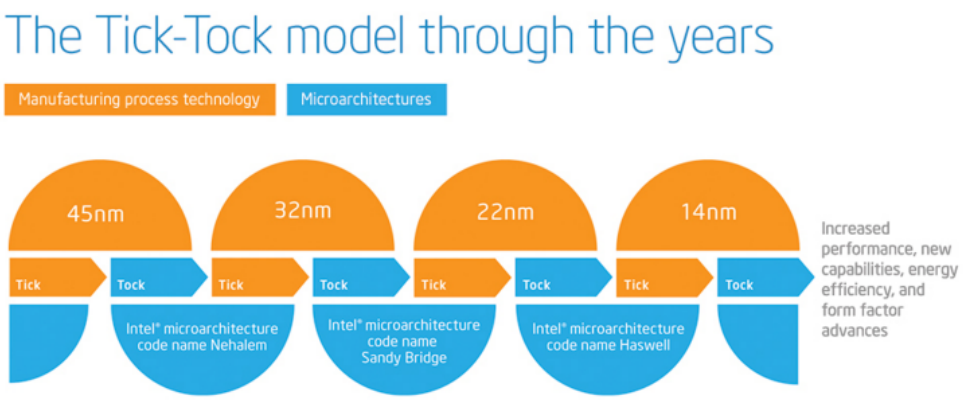

Figure 2.4: Intel Tick-Tock model

on it.

We start this presentation from the most popular processors in HPC world, the Intel company ones. We also present ARM which is another multi-core architecture based on RISC instructions set.

Intel

Intel was created in 1968 by a chemist and a physicist, 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 2.4. Since its creation this model followed the same fashion: a new manufacturing technology like shrink of the chip with better engraving on a "Tick" followed by a "Tock" by the delivery of a new micro-architecture. The Intel processors for HPC are called Xeon and feature ECC memory, higher number of cores, large RAM support, large cache-memory, Hyper-threading, etc. Compared to desktop processors, their performances are of a different magnitude. Every new processor has a code name. The last generations are chronologically called Westemere (2011), Sandy Bridge (2012), Ivy Bridge(2013), Haswell (2014), Broadwell (2015), Skylake (2015) and Kaby lake (2016).

Kaby Lake, the last architecture provided, does not exactly fit the usual "Tick-Tock" process because it is just based on optimizations of the Skylake architecture. It is produced like Skylake in 14nm. The tick-tock model seems to be hard to maintain due to the difficulties to engrave in less than 10nm with quantum tunneling. This leads to using larger many-cores architecture and base next supercomputer evolutions road-map on hybrid models.

Hyper-threading Another specificity of Intel Xeon processors is Hyper-threading (HT). This technology makes a single physical processing unit (core) appearing as two logical ones 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 allow 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 of applications and can be disabled for normal use of the processors in the context of general purpose HPC architectures.

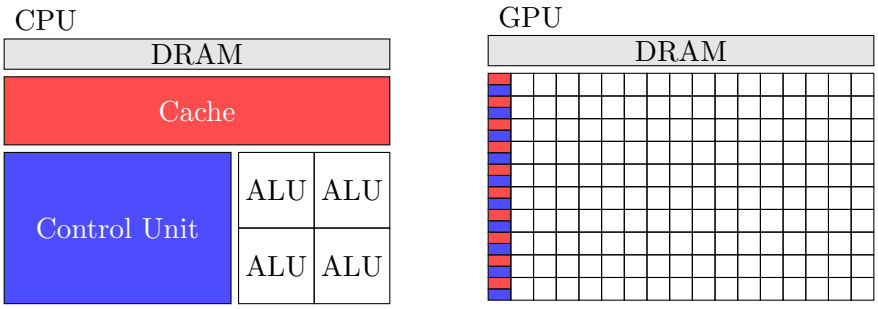

Figure 2.5: Multi-core versus Many-core architecture, case of GPUs

ARM

Back in 1980s, ARM stood for Acorn RISC Machine in reference to 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 processors 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 many transistors to operate and thus a smaller silicon area on the die. 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 imposes more work on the source code compilation to fit the simple architecture. That makes the instructions sources longer and therefore more single instructions to execute.

The ARM company provides several versions of ARM processors named Cortex-A7X (2015), Cortex-A5X (2014) and Cortex-A3X (2015), respectively featured for highest-performances, for balancing performances and efficiency or for less power consumption.

The new ARMv8 architecture starts to provide 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 focuses 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 into 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.

2.3.2 Many-core architecture, SIMT execution model

Several architectures can be defined as many-cores. Those devices integrate thousands of cores that are usually control by less control units. 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

When a CPU can usually have 2 to 32 computation cores that can operate on different instruction streams, the SIMT architecture of the GPU is slightly different. The cores are grouped and have to share the same instruction at the same clock time but all the groups can have their own instruction.

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

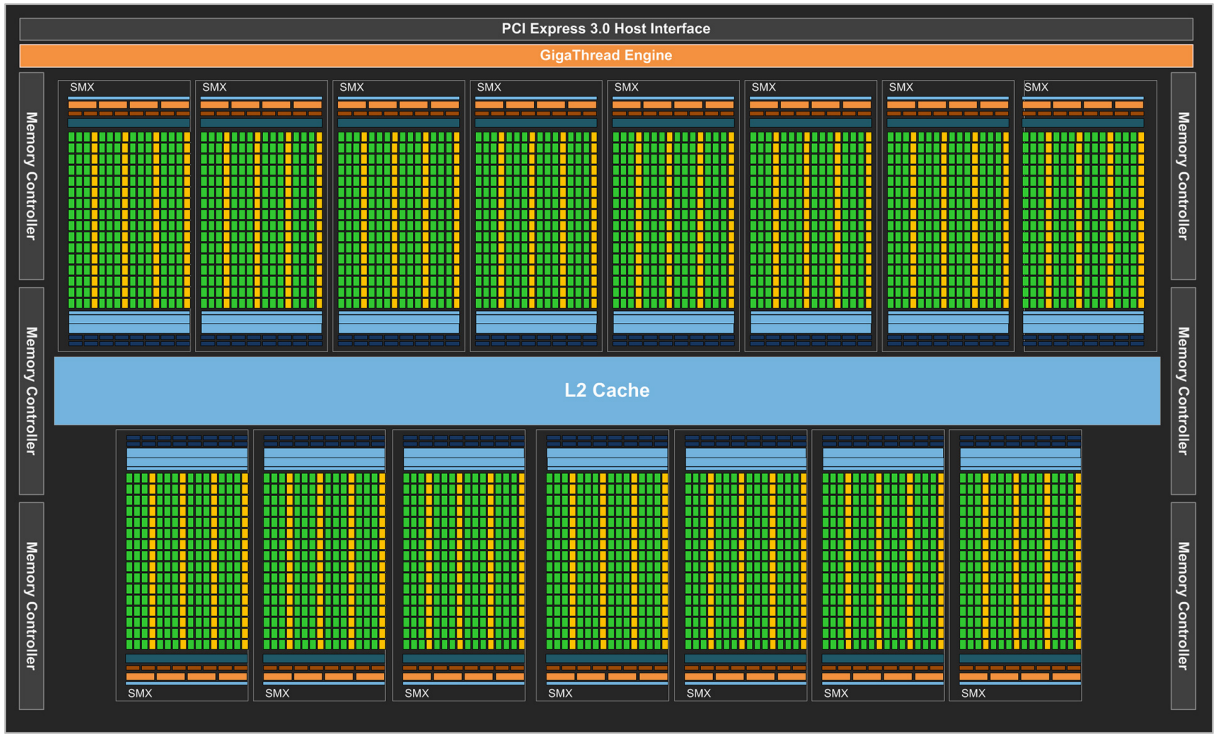

Figure 2.6: NVIDIA Tesla Kepler architecture. Single-precision in green and double-precision in yellow

Figure 2.5 present the vision between CPU and GPU processors. We see on that figure the usual topology with the ALU lined up in front of their control unit and shared cache memory. Every ALU also have its own memory and registers to operate local computations.

Those devices are called General Purpose Graphics Processing Units (GPGPUs). They are derivative from classical GPUs used for graphics purpose. Pioneer shows that they can be use efficiently for classical scientific computations. The vendor provides then specific GPU for general purpose computing. We present here the two main companies providing GPGPUs for HPC world: 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 graphics rendering.

Known as the pioneer in graphics, cryptocurrency, portable devices and now Artificial Intelligence (IA), 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 versions of the architecture are named following famous physicists, chronologically: Tesla, Fermi, Kepler, Maxwell, Pascal and Volta.

We describe here the Kepler brand GPU and more specifically the K20Xm GPU on which we based our study. This NVIDIA Tesla Kepler GPU is based on the GK110 graphics processor describes in the white-paper[Nvi12] on 28nm process. The figure 2.6 is a representation of the physical elements of this graphics processor. 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 low 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.

2.3.3 Other architectures

Numerous architecture have not been presented because out of scope in this study. We present here two technologies we have been confronted in our researches and that can be tomorrow solution for exascale in HPC.

FPGA

Field Programmable Gates Array are device that can be 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.

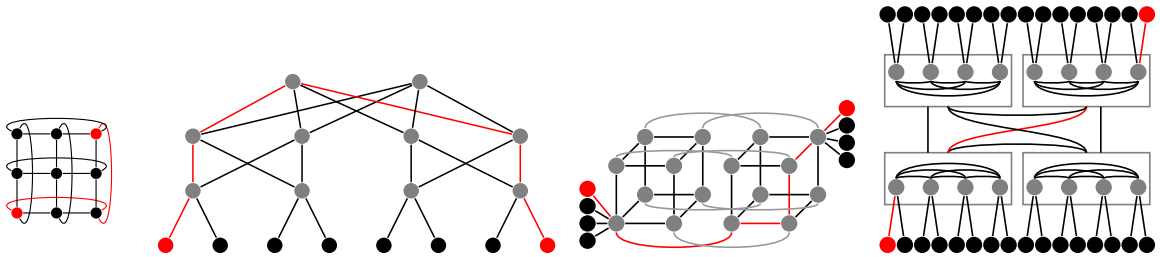

Figure 2.7: Torus, Fat-Tree, HyperX, DragonFly

ASIC

Application Specified Integrated Circuits 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.

2.4 Distributed architectures

The technologies presented in previous part is the milestone of supercomputers. They are used together in a whole system to create machine delivering incredible computational power.

2.4.1 Architecture of a supercomputer

From the hardware described before we can create the architecture of a cluster from the smallest unit, cores, nodes, to the whole system:

Core: A core is the smallest unit in our devices. It can refer to the Von Neumann model in case of core with ALU and CU. We can separate core from CPU to GPU, the first one able to be independent whereas the second ones working together and sharing the same program counter.

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

Accelerators/Devices: Accelerators are devices that, 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.

Computation node: The next layer of our HPC system if the computation node. Grouping together several socket and accelerators sharing memory;

Rack: A rack is a set of computation nodes, generally a vertical stack. It can also include specific nodes dedicated to the network or the Input/Output.

Interconnection: The nodes are grouped together with hard wire connection following a specific interconnection topology with very high bandwidth.

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

In order to connect node together and allow distributed programming an interconnect technology is required. Interconnection network is the way the nodes of a cluster are connected together.

2.4.2 Interconnection topologies

Several topologies exists from point to point to multi dimensional torus. The figure 2.7 is a rep-

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

resentation 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. This is the topology of the ROMEO supercomputer we used for our tests. 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.

Unfortunately this augmentation of clock rate is not sustainable due to the energy required and the heat generated by the running component. Another idea came in 19th century with the first multi-core processors.

2.4.3 Remarkable supercomputers

The TOP500[2] is the reference benchmarks for the world size supercomputers. This benchmark is based the LINPACK and aim to solve a dense system of linear equations. 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, November 2017. 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, completely done in China. The Sunway CPUs were designed 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 2.8. 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. For the latest TOP500 list, November 2017: the total memory is 1.31 PB and the number of cores available

[2] `https://www.top500.org`

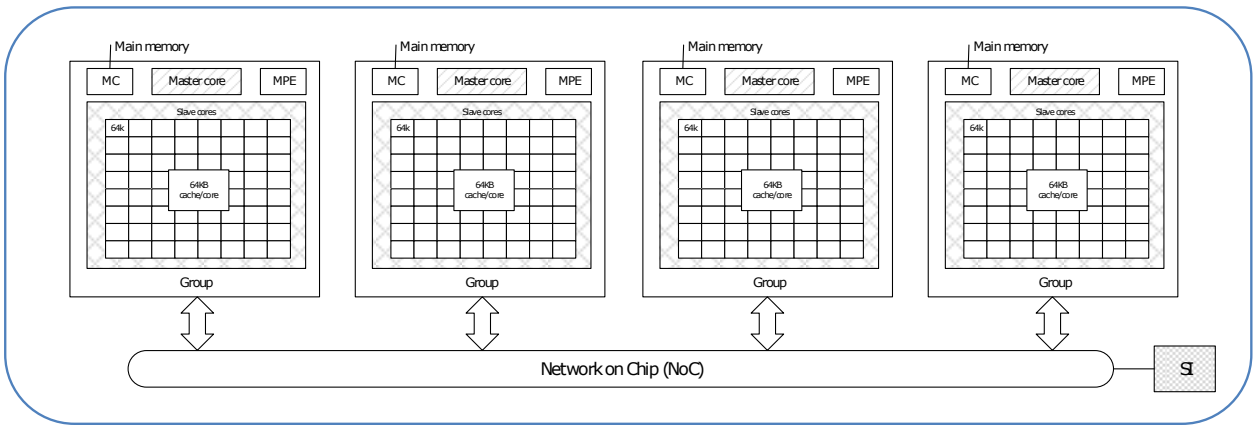

Figure 2.8: Sunway Taihulight node architecture from *Report on the Sunway TaihuLight System*, Jack Dongarra, June 24, 2016.

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 of November 2017.

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 2.9 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.

Sequoia/Mira

Sequoia supercomputer was top 1 of the TOP500 2012 list. It is based on BlueGene from IBM. The BlueGene project made up to three main architectures with BlueGene/L, BlueGene/P and BlueGene/Q. It is very interesting to notice the BlueGene architecture because even in the last GRAPH500 list, November 2017, there is 15 of these machines in the TOP20. The algorithm used on these supercomputers will be our basis in the part II regarding our implementation of the GRAPH500 benchmark.

2.5 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

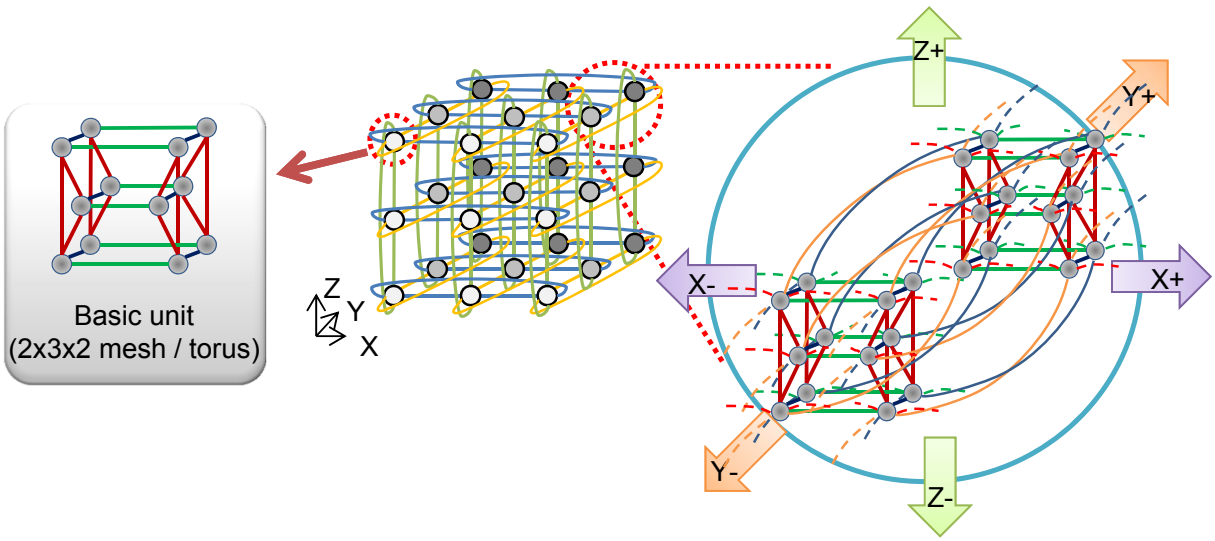

Figure 2.9: TOFU Interconnect schematic from *The K-Computer: System Overview*, Atsuya Uno, SC11

meso-center (Medium size HPC center) is used for HPC for theoretic research and domain science like applied mathematics, physics, biophysics and chemistry.

This project is supported by Europe, National fundings, Grand-Est region and Reims Metropole. It aims to host research and production codes of the region for industrial, research and academics purposes.

We are currently working on the fourth 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.

2.5.1 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.

ROMEO is based on the Slurm[3] workload manager for node distribution among the users. This manager allows different usage of the cluster with classical reservation-submission or more asynchronous computation with best-effort. We developed advantages of both submissions systems in part II.

2.5.2 New ROMEO supercomputer, June 2018

Avoir les info et decrire le nouveau ROMEO

[3] `https://slurm.schedmd.com/`

2.6 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 with programming models. Then, fitting those models, we present the possible options in the language, the API, the distribution and the accelerators code.

This chapter details the most important programming models and the 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, HPGC and GREEN500 to give their advantages and weaknesses.

3.2 Parallel and distributed programming Models

The Flynn taxonomy developed in chapter 1 was a characterization of the executions models. This model can be extended to programming models which are an extension of MIMD. We consider here a *Random Access Machine* (RAM). The memory of this machine consists of an unbounded sequence of registers each of which may hold an integer value. In this model the applications can access to every memory words directly in write or read manner. There is three main operations: load from memory to register; compute operation between data; store from register to memory. This model is use to estimate the complexity of sequential algorithms. If we consider the unit of time of each operation (like in cycle) we can have an idea of the overall time of the application. We identify two types of RAM, the Parallel-RAM using shared memory and the Distributed-RAM using distributed memory.

3.2.1 Parallel Random Access Machine

The Parallel Random Access Machine [FW78], PRAM, is a model in which the global memory is shared between the processes and each process have its own local memory/registers. The execution is synchronous, processes execute the same instructions at the same time. In this model each process is identify with its own index enabling to target different data. The problem in this model will be the concurrency in reading (R) and writing (W) data as the memory is shared between the processes. Indeed, mutual exclusion have to be set with exclusive (E) or concurrent (C) behaviors and we find 4 combinations: EREW, ERCW, CREW and CRCW. As the reading is not critical for data concurrency the standard model will be Concurrent Reading and Exclusive Writing: CREW.

An example of PRAM model applied on a reduction is given on figure 3.1. In this example the computation of minimum, maximum or a reduction can be perform on an array of n values in $log(n)$ steps and $n-1$ operations. This kind of reduction or scan are used to reach performances

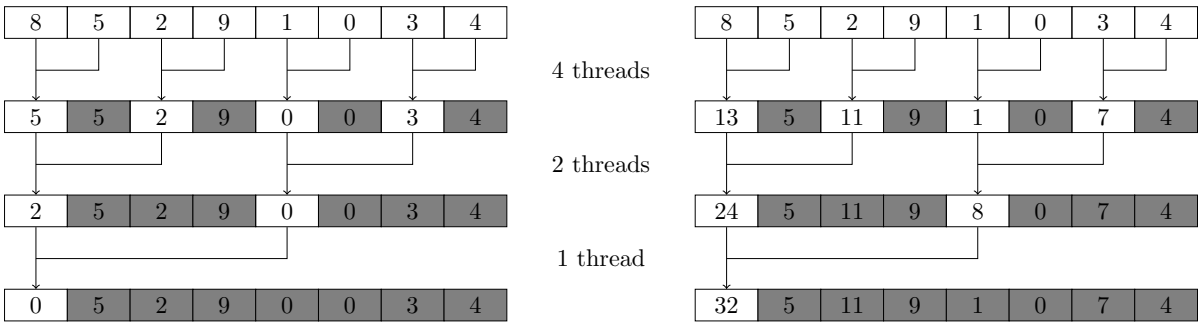

Figure 3.1: PRAM model example on minimum and reduction computation

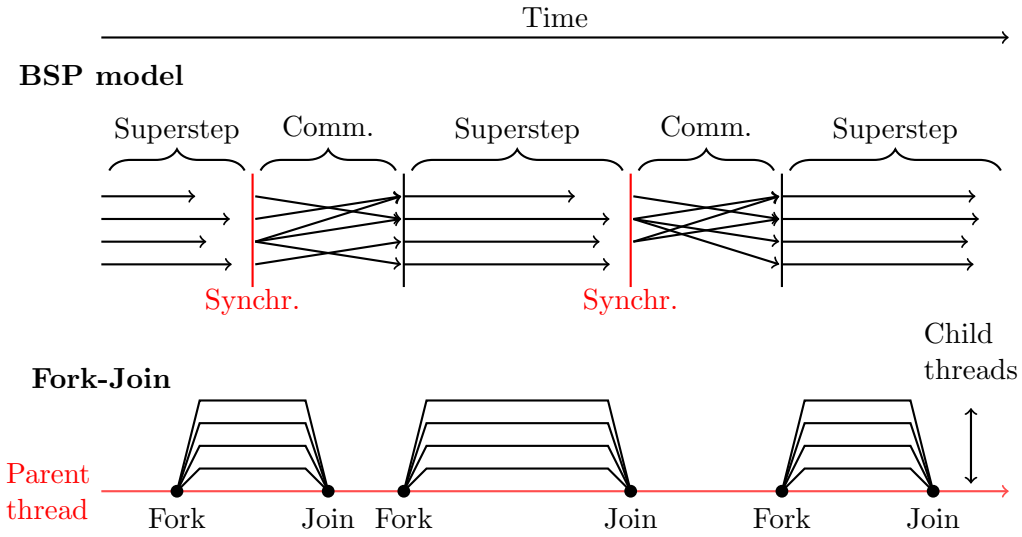

Figure 3.2: Bulk Synchronous Parallel model and Fork-Join model

on multi-core and many-core code in this study. We will give example of those reductions in different languages and API.

3.2.2 Distributed Random Access Machine

For machine that base their memory model on NoRMA the execution model can be qualify of Distributed Random Access Machine, DRAM. It is based on NoRMA memories detailed in part 1.4. This model is in opposition to PRAM because the synchronization between processes is made by communications and messages. Those communications can be of several kind and depend of physical architecture, interconnection network and software used.

3.2.3 H-PRAM

A DRAM can be composed of an ensemble of PRAM system interconnected. Each of them working on their own data and instructions. This is an intermediate model between PRAM and DRAM having a set of shared memory and synchronous execution, the overall execution being asynchronous and having distributed memory.

3.2.4 Bulk Synchronous Parallelism

This model was presented in 1990 in [Val90]. Being the link of HRAM and PRAM The Bulk Synchronous Parallelism model is based on three elements:

- a set of processor and their local memory;

- a network for point-to-point communications between processors;
- a unit allowing global synchronization and barriers.

This model is the most common on HPC clusters. It can be present event on node themselves: a process can be assign on a core or set of cores and the shared memory is separated between the processes. The synchronization can be hardware but in most cases it is handle by the runtime used. A perfect example of runtime, presented later, is MPI.

In this model the applications apply a succession of *supersteps* separated by *synchronizations* steps and data exchanges.

At opposite to H-PRAM which represent the execution as a succession of independent blocks working synchronously, BSP propose independent blocks of asynchronous applications synchronized by synchronization steps.

In a communication/synchronization step we can consider the number of received messages h_r and the number of send ones h_s.

The time lost in communication in one synchronization step is:

$$T_{comm} = hg + I \tag{3.1}$$

With $h = max(h_s, h_r)$, g the time to transfer data and I the start-up latency of the algorithm. Indeed, the entry points and exit points of communications super-step can be a bottleneck considered in I.

The time for computing a super-step is:

$$T_{comp} = \frac{w}{r} + I \tag{3.2}$$

With w the maximum number of flops in the computation of this super-step, r the speed of the CPU expressed in FLOPS and I the start-up latency of the algorithm. Indeed, the entry points and exit points of communications super-step can be a bottleneck considered in I.

The BSP model estimates the cost of one super-step with:

$$T_{comm} + T_{comp} = w + gh + 2l \tag{3.3}$$

With T a measure of time, a wall clock that measure elapsed time. We also note that usually g and I are function of the number of processes involved.

It can then be use to compute the overall cost in BSP model summing all super-steps s:

$$T = \sum_s \frac{max(w_s)}{r} + h_s g + I \tag{3.4}$$

The problem of performances in this model can come from unequal repartitions of work, the load balancing. The processes with less than w of work will be idle.

3.2.5 Fork-Join model

The Fork-Join model or pattern is presented in figure 3.2. A main thread pilot the overall execution. When requested by the application, typically following the idea of *divide-and-conquer* approach, the main thread will fork and then join other threads. The *Fork* operation, called by a logical thread parent, creates new logical threads children working in concurrency. There is no limitations in the model and we find nested fork-join where a child can also call fork to generate sub-child and so on. The *Join* can be called by both parents and child. Children call join when done and the parent join by wait until children completion. The Fork operation increase concurrency and join decrease concurrency.

3.3 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.

```
...
int min_val = MAX;
#pragma omp parallel for reduction(min:min_val)
for(int i = 0 ; i < X ; ++i){
        min_val = arr[i]<min_val?arr[i]:min_val;
}
...
```

Figure 3.3: OpenMP reduction code for minimum

3.3.1 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 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 4.0 also allow the user to target accelerators. During compilation the user specify on which processor or accelerator the code will be executed in parallel.

We use OpenMP as a basis for the implementation of our CPU algorithms. Perfect for loop parallelization and parallel sections, we show that we can have the best results for CPU algorithms in most of the case. In our case, OpenMP is always use on the node to target all the processors cores in the shared memory.

We note that the new versions of OpenMP also allows to target directly accelerators like NVIDIA ones.

The code corresponding to a reduction using OpenMP API is presented on figure 3.3. This is linked with the PRAM model reduction presented at the beginning of this part.

3.3.2 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.

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

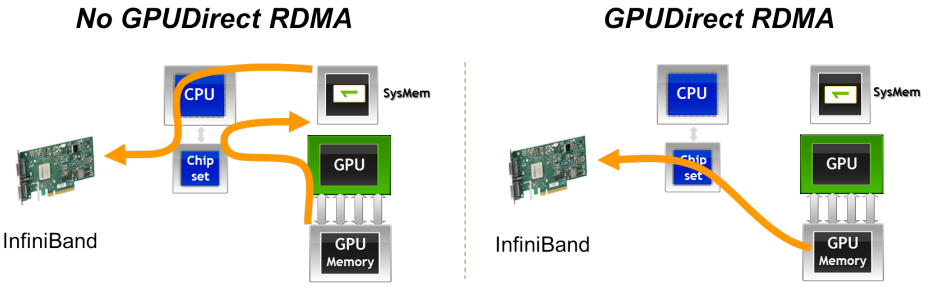

Figure 3.4: GPUDirect RDMA from NVIDIA Developer Blog, *An Introduction to CUDA-Aware MPI*

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.

Most of our code presented here are based on MPI for the distribution on the cluster. The advantage is its presence on all the cluster and the control over the data transfers.

For NVIDIA this technology is called GPUDirect RDMA and presented on figure 3.4.

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* 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 by Stanford University, Los Alamos National Laboratory (LANL) and NVIDIA. This runtime is data-centered targeting distributed heterogeneous

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

```
__inline__ __device__ void warpReduceMin(int& val, int& idx)
{
    for (int offset = warpSize / 2; offset > 0; offset /= 2) {
        int tmpVal = __shfl_down(val, offset);
        int tmpIdx = __shfl_down(idx, offset);
        if (tmpVal < val) {
            val = tmpVal;
            idx = tmpIdx;
        }
    }
}
```

Figure 3.5: CUDA kernel for reduction of minimum

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.

3.3.3 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.

As presented in figure 3.6, 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

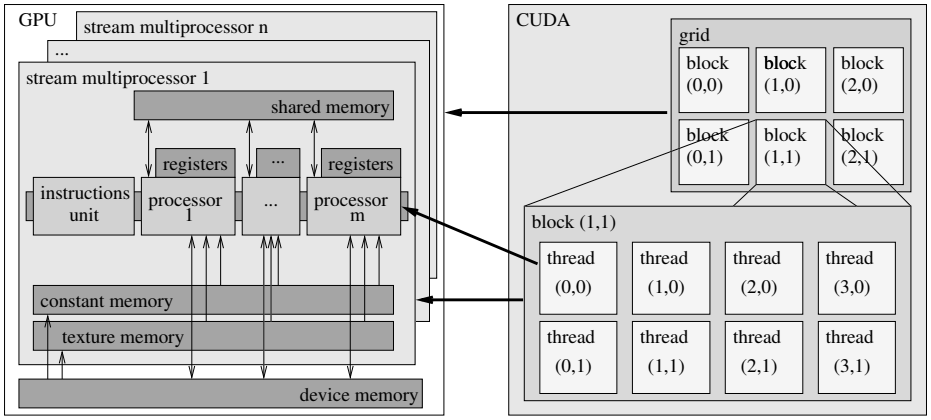

Figure 3.6: NVIDIA GPU and CUDA architecture overview

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.

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

Dynamic Parallelism This feature allow the GPU kernels to run other kernels themselves. When more sub-tasks have to be generated this can be done directly on the GPU using dynamic parallelism.

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.

Indeed, working with a very low level API like CUDA can lead to better performances. This can be very costly in development time. Regarding the time involved in coding compared to the performances gain, one may consider to use higher levels API like OpenACC.

The reduction using the CUDA API is presented on figure 3.5. It shows, compared to OpenMP, the complexity of CUDA code.

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 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 vendors wrapper. 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

```
...
int min_val = MAX;
#pragma acc parallel loop copyin(arr[0:n]) reduction(min:min_val)
for (int i=0; i<n; i++) {
    min_val = arr[i]<min_val?arr[i]:min_val;
}
...
```

Figure 3.7: OpenACC code for reduction of minimum

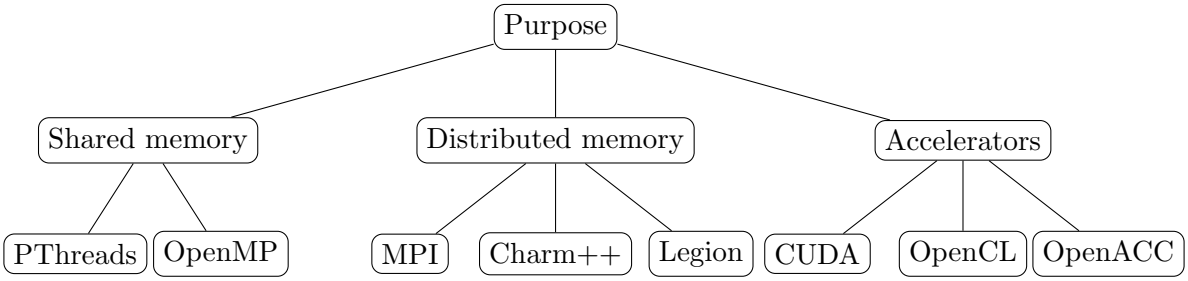

Figure 3.8: Runtimes, libraries, frameworks or APIs

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 reduction example is given on figure 3.7. This shows the simplicity of writing accelerator code using high level API like OpenACC compared to the complex CUDA code. Indeed, this approach works in most of the case but for specific ones CUDA might outperform OpenACC.

The runtime, libraries, frameworks and APIs are summarized in figure 3.8. They are used in combination. The usual one is MPI for distribution, OpenMP and CUDA to target processors and GPUs.

3.4 Benchmarks

All those models, theory, hardware and software leads to better understanding and characterization of machines to produce better algorithm and solve bigger and harder problems. The question that arise is: how to know if a machine is better than another? We answer that ques-

[7] https://www.openacc.org/

tion with FLOPS, IPC, OPS or just the frequency of the machine. The models like BSP or law's like Amdahl and Gustafson ones propose to find the best/worst case during the execution.

In real application the only way to really know what will be the behavior of a supercomputer is to try, test real code on it. This is call benchmarking. Several kind of benchmarks exists and target a specific application of supercomputers. We present here the most famous benchmarks of HPC and their specificities.

3.4.1 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.

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.

3.4.2 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 equipped 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.

3.4.3 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.

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

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

3.4.4 HPCG

The High Performance Conjugate Gradient benchmark[11] is a new benchmark created in 2015 and presented for the first time at SuperComputing 2015. The last list, November 2017 contains 115 supercomputers ranked. The list also offer to compare the results of Linpack compared to Conjugate Gradient. This benchmark is a first implementation of having both computation and communications aspects of HPC in the same test.

This benchmark is presented and features:
- Sparse matrix-vector multiplication;
- Vector updates;
- Global dot products;
- Local symmetric Gauss-Seidel smoother;
- Sparse triangular solve (as part of the Gauss-Seidel smoother);
- Driven by multigrid preconditioned conjugate gradient algorithm that exercises the key kernels on a nested set of coarse grids;
- Reference implementation is written in C++ with MPI and OpenMP support.

The benchmarks presented in this section are the most famous of HPC world. Indeed, they are not the perfect representative of the nowadays application. The upcoming of big data and artificial intelligence in addition to classical "real life" applications impose HPC to evolute and find new ways to target new architectures. The TOP500 target the computational problem but does not handle a lot of irregularity. Indeed, solving dense linear equation is straight forward and also use the memory in a regular way. The Graph500 is very interesting to focus on communication and does handle irregular behavior for communications and memory. The Green500 does target energy wall but can also be applied to any benchmark. The most interesting one may be the HPCG benchmark. It does create irregularity during computation and communication along to memory traversal.

3.5 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.

[11] http://www.hpcg-benchmark.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.

[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.

[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.

[FW78] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118. ACM, 1978.

[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.

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

[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.

[Rus78] Richard M Russell. The cray-1 computer system. *Communications of the ACM*, 21(1):63–72, 1978.

[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.

[Val90] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[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.