UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

École Doctorale Sciences Technologie Santé

THÈSE

Pour obtenir le grade de:

Docteur de l'Université de Reims Champagne-Ardenne

Discipline : **Informatique**

Spécialité : **Calcul Haute Performance**

Présentée et soutenue par:

Julien LOISEAU

le 18 Mars 2018

Hybrids Architectures to Reach Exascale

Les Architectures Hybrides pour Atteindre l'Exascale

Sous la direction de :
Michaël KRAJECKI, Professeur des Universités

JURY

Pr. Françoise Baude	I3S, Université de Nice Sophia Antipolis	Présidente
Pr. Michaël Krajecki	CReSTIC, Université de Reims Champagne-Ardenne	Directeur
Dr. François Alin	CReSTIC, Université de Reims Champagne-Ardenne	Co-directeur
Dr. Christophe Jaillet	CReSTIC, Université de Reims Champagne-Ardenne	Co-directeur
Pr. William Jalby	Université de Versailles Saint-Quentin	Rapporteur
Dr. Benjamin Bergen	Ingénieur au Los Alamos National Laboratory, USA	Rapporteur
Pr. Zineb Habbas	LCOMS, Université de Lorraine	Rapporteur
Guillaume Colin de Verdiere	Ingénieur au CEA	Invité

Centre de Recherche en Sciences et Technologies de l'Information et de la Communication de l'URCA EA3804
Équipe Calcul et Autonomie dans les Systèmes Hétérogènes

Acknoledgments

Croyez ceux qui cherchent la vérité, doutez de ceux qui la trouvent; doutez de tout, mais ne doutez pas de vous-même.

Resume

French abstract (1700 caracteres)
ENGLISH TITLE
English abstract (1700 caracteres)
English keywords
Intitule et adresse de l'unite ou labo de la these

Contents

List of Figures

List of Tables

Introduction

In the aurora 2020-2021 for United States of America and maybe before, like 2019 for China, the world of High Performance Computing (HPC) will reach another milestone in the power of machines: the Exascale. These supercomputers will be 100 times faster than the estimated overfall operations performed by the human brain and its 10^{16} **FL**oating point **O**perations **P**er **S**econd (FLOPS) and achieve a computational power of a trillion (10^{18}) FLOPS. This odyssey started long time ago with the first vacuum tubes computers and the need of balistic computation in war. Nowdays the aim is very nearby and the power of a nation is represented by its army and money but also by the computational power of its supercomputers. HPC's applications also spread into all the area of science and technology.

Since 1962, considering the Cray CDC 6600 as the first supercomputer, the power of those machines have increase following an observation of the co-fonder of the Intel company, Gordon Moore. Better known under the name of "Moore's Law", it speculates in 1965 that: considering the constant evoluation of technology the number of transistors on a dense integrated circuit will double approximately every two years. Thus, the computational power, that depend intrinsectly of the number of transistors on the chip, will increase and more important, as money is the sinews of war, the cost of the die for the same performances will decrease. This observation can be related to the supercomputers results through the years in the TOP500 list. As shown on figure 1, even if estimated in early 1965, the Moore's law seems to be accurate and sustainable.

This linear evolution is not just gave by the shrink in the semiconductor with smaller transistors. In fact the first one-core Central Processing Units (CPUs) were made using more transistors but also faster frequency. But they faced limitations in reaching high frenquencies with the power consumption and the inherant cooling of the heat generated by the chip. This is why, at some point in early twentieth century, IBM proposed the first multi-core processor on the same die, the Power4. The constructors started to create chips with more than one core to increase the computational power in conjonction with the shrink of semiconductors, answering the constant demand of more powerful devices and allowing the Moore's law to thirve. This increase of the overall power of the chip comes with some downside costs in synchronizations steps between the cores for memory access, work sharing and complexity. The general purpose CPU usually features from 2 to less than a hundred of cores in a single chip.

In order to reach even more computational power some researchers started to use many-core approches. Using thousand of cores these devices are using very "simple" computing units, with slow frequency and low power consumption but add more complexity and requirement for their efficient programming with even more synchronizations needed between the cores. Usually those many-core architectures are used coupled with a CPU that send the data and drive them, even if some can be Host or Device as well like the Intel Xeon Phi. Usually called accelerators, those Devices are used in addition to the Host to provide their efficient computatinal power in the key part of execution. The most famous accelerators are the Xeon Phi, the General Purpose Graphics Processing Unit (GPGPU) initialy used in graphics, Field Programmable Gates Array (FPGA) or even dedicated Application-Specific Integrated Circuit (ASIC). The model using a Host with in addition Device(s) appears and we will refer at it as "Hybrid Architecture". In fact a cluster can be composed of CPUs, CPUs with accelerator(s) of the same kind, CPUs with

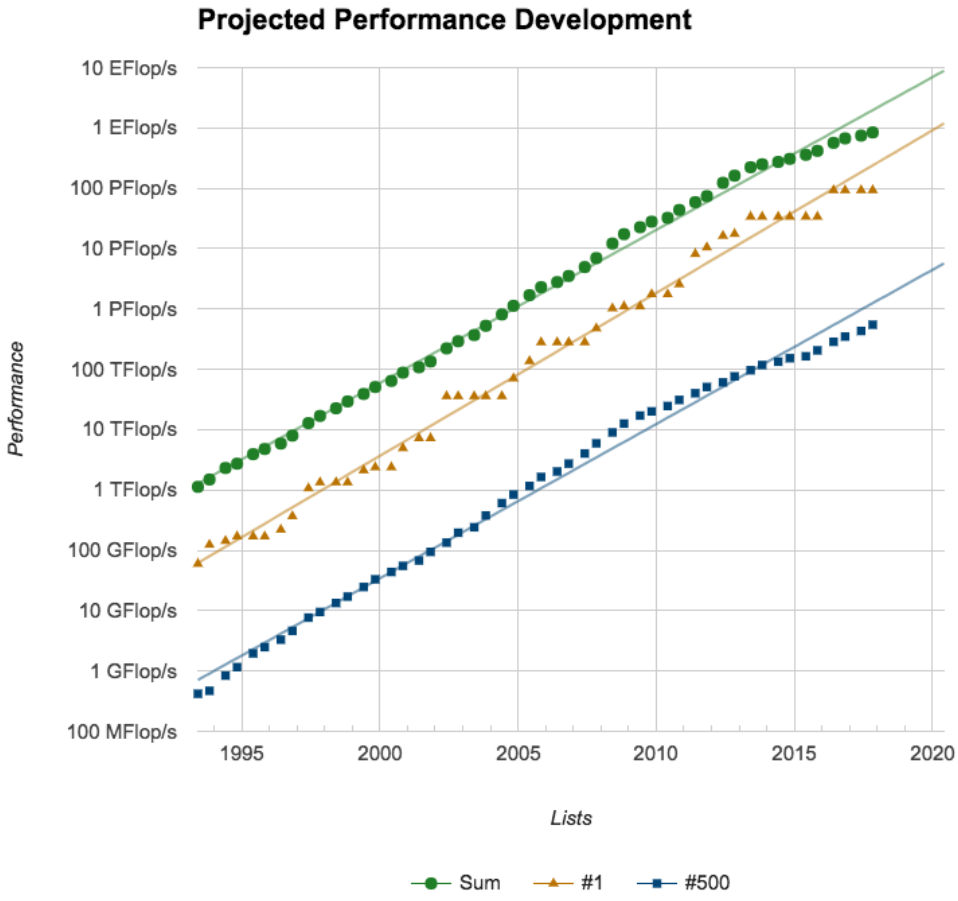


Figure 1: Computational power evolution in the TOP500 list

heterogeneous accelerators or even accelerators like Xeon Phi driving different kind of accelerators.

Since 2013-2014 many companies, like the Gordon Moore's company Intel itself, stated that that the Moore's law is over. This can be see on figure 1, in the right part of the graph, the evolution is not linear anymore and tend to dicrease slowly in time. This can be imputed to two main factors: on one hand, we slowly reach the maximal shrink size of the transistors implying hard to handle side effects and on another hand the power wall implyed by the power consumption required by so many transistors and frequency speed on the chip.

Even with all these devices, nowaday supercomputers are facing several problems in their conception and utilization. The three mains are the power consumption wall, the communication wall and the memory wall bounding the overall computional power of the machines. Some subproblems like the the interconnect wall, resilience wall or even the complexity wall also arise and make the task even harder.

In this context of doubts and questions about the future of HPC, this study propose several points of views. We think that the future of HPC is made with those hybrid acrchitectures or acceleration devices adapted to the need using well suited API, Framework and code. We consider that the classical benchmarks like the TOP500 are not enough to target all the main wall problems of those architectures and especially accelerators. Domain scientists application like Physics/Astrophysics/Chemistry/Biology require benchmarks based on more irregular cases with heavy computation, communications adn memory accesses.

We propose a metric that extracts the three main issues of HPC and apply them on accelerators architectures to figure out how to take advantage of those architectures and what are the limitations. The first step of this metrics target 2 problems and then a third one combining all our knowledge. The first two are targeting computation and communication wall over very irregular cases with high memory accesses, using an academic combinatorial problem and the Graph500 benchmark. The last is a computational scientific problem that cover both of the problems and appears to be hard to implement on supercomputers and even more on accelerated ones.

This thesis is composed of 3 parts and an overall conclusion.

The first will develope the state of the art in HPC from the main law to the hardware. We go through the basic laws from Amhdal to Gustafson and the specification of speedup and efficiency. Classical CPUs, GPGPUs and other accelerators are describe and discussed regarding the state of the art. The main methods of ranking and the issues regarding them are presented.

In the second part we propose our metric to characterize supercomputers architectures. The Langford problem is described as an irregular and computationally heavy problem. This shows how the accelerators, in this case GPU, are able to support the memory and computation wall. This allowed us to beat a world record on the last instances of this academic problem. The Graph500 problem is then proposed as an irregular and communications heavy problem. We present our implementation and the logic to take advantage of the GPU computational power in an

Then, in the last part, we consider a problem that is heavy and irregular regarding to computation and communications. This problem is the milestone of our metric and show how nowadays supercomputers can overcome those issues. This computational science problem is based on the SPH method and we intend to provide a tool for Physisists and Astrophysists and is called FleCSPH. Developed on top of the FleCSI framework from the Los Alamos National Laboratory it allowed us to exchange directly with the LANL domain scientists on their need.

The last part will conclude on this work and results and show some of the main prospects of this study and my future researches.

Part I

HPC and Exascale

Introduction

This part of my PhD thesis presents a state of the art of High Performance Computing. It describes the tools we need for our study. High Performance Computing, HPC, does not have a strict definition. The history starts with domain scientists in need of more complex and longer computation for models checking or simulations. They developed their own tools beginning with vacuum tubes computers which can be consider as a milestone in HPC history. Since this first machine the technology became more and more complex at every layer: the hardware conception, the software to handle it and even the models and topologies. HPC is now a 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 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 concern it.

The second chapter details the architecture base on these models. We present nowadays platforms with dominant constructors and architectures from multi-cores to specific many-cores 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 allows to target them more easily and efficiently.

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 bests to give an accurate score or scale for "realistic" domain scientists applications.

Chapter 1

Models of HPC

1.1 Introduction

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

Then we give more details on what is parallelism and how to reach performances though it. And thus we define what performance implies in HPC. The Amdahl's and Gustafson's laws are presented and detailed along with the strong and weak scaling used in our study.

1.2 Von Neumann Model

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

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

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

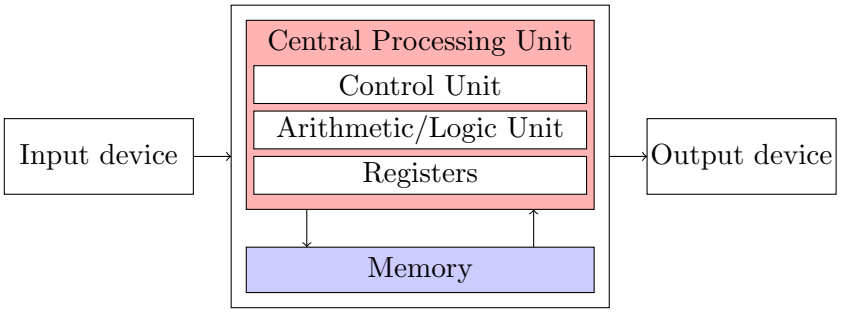


Figure 1.1: Von Neumann model

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

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

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

Buses The links between those elements are called Buses and can be separated between data buses, control buses and addresses buses. They will have a huge importance for the first machine optimization, growing the size of the buses from 2, 8, 16, 32, 64 and even more for vector machine with 128 and 256 bits.

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

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

1.3 Flynn taxonomy and execution models

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

In this classification [Fly72b] from 1972, Michael J. Flynn presents the SISD, MISD, MIMD, and SIMD models represented on in table 1.1 and figure 1.2. Every of those execution model correspond to a specific machine and function.

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

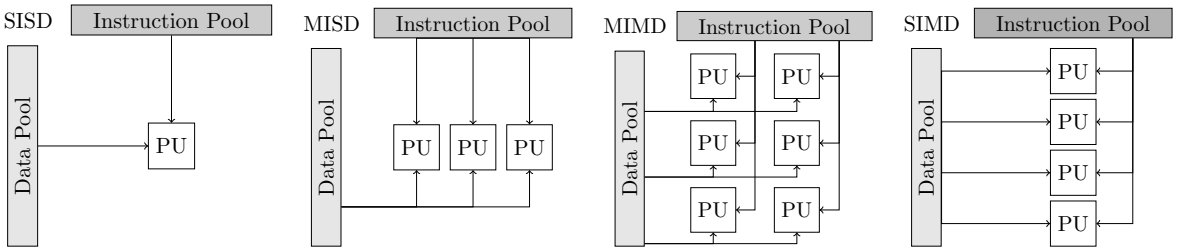


Figure 1.2: Flynn taxonomy schematic representation of execution models

1.3.1 Single Instruction, Single Data: SISD

This is the model corresponding to a single core CPU like in the Von Neumann model. This sequential model takes one instruction, operates on one data and the result is then store and the process continues over. SISD is important to consider as a reference computational time and will be taken in 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 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 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

The Single Program Multiple Data model, SPMD, is the most famous parallelism way for HPC purpose: each process execute the same program. At opposite to SIMD the programs are the same but does not share the same instruction counter. This model was proposed for the first time in [DGNP88] in 1988 using Fortran. This is the common approach working with runtime like MPI. The programs are the same and the execution similar but based on their ID the processes will target different data.

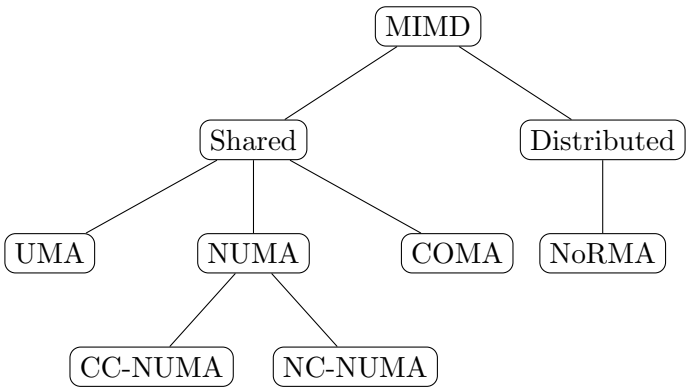


Figure 1.3: MIMD memory models

MPMD

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

1.3.4 Single Instruction, Multiple Data: SIMD

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

1.3.5 SIMT

We find another characterization to describe the new GPUs architecture: Single Instruction, Multiple Threads. This appears in one of NVIDIA's company paper [LNOM08]. This model describes a combination of MIMD and SIMD architectures, every block of threads is working with the same control processor on different data and every block 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 of the execution model and parallelism the memory access patterns have a main role on performances especially in SIMD and MIMD. In this classification we identify three categories: UMA, NUMA and NoRMA for shared and distributed cases. This model have been pointed out in the Johnson's taxonomy[Joh88].

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

1.4.1 Shared memory

When it comes to multi-threaded and multi-cores like MIMD or SIMD execution models, several kind of memory models are possible. We give a description of the most common shared memories architectures.

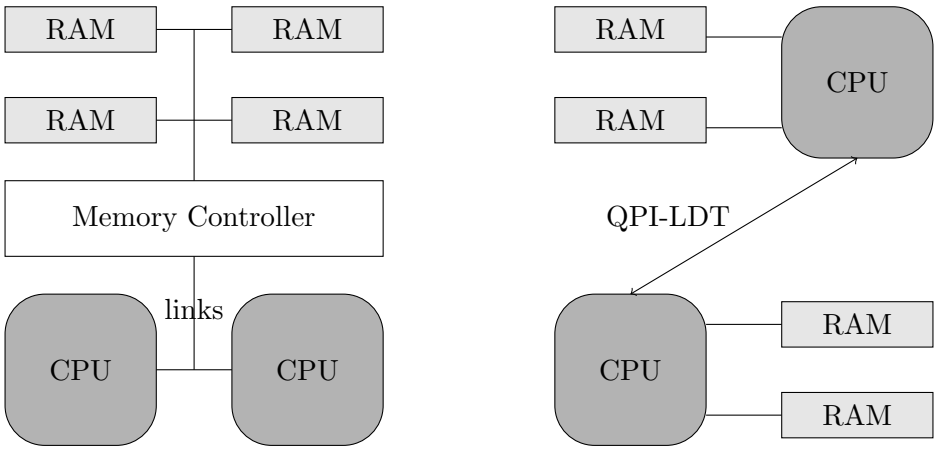


Figure 1.4: UMA vs NUMA memory models

UMA

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

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

NUMA

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

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

COMA

In Cache-Only Memory Accesses, the whole memory is see as a cache from every processes. Attraction memory is setting up and will attract the data near the process that will use those data. This model is less commonly use and lead to, in best cases, same results as NUMA.

1.4.2 Distributed memory

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

Name	**FLOPS**	**Year**	**Name**	**FLOPS**	**Year**
kiloFLOPS	10^3		petaFLOPS	10^{15}	2005
megaFLOPS	10^6		exaFLOPS	10^{18}	2020 ?
gigaFLOPS	10^9	$\approx$ 1980	zettaFLOPS	10^{21}	
teraFLOPS	10^{12}	1996	yottaFLOPS	10^{24}	

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

1.5 Performances characterization in HPC

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

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

1.5.1 FLOPS

The Floating point Operations Per Second considers the number of floating-point operation that the system will executes in a second. They are an unit of performance for computers. Higher FLOPS is better. This is also the scale used to consider supercomputers computational power. For a cluster we can compute the theoretical FLOPS (peak) based on the processor frequency in GHz with:

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

With $\#nodes$ the number of computational node of the system, $\frac{\#sockets}{\#node}$ the number of sockets (= processor) per node, $\frac{\#cores}{\#socket}$ the number of core 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.

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

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

1.5.2 Power consumption

Another way to consider machine performance is to estimate the number of operations regarding the power consumption. It can consider all the previous metrics like FLOPS, IPS, IPC or OPS. Benchmarks, like the Green500, consider the FLOPS delivered over the watts consumed. For nowadays architectures the many-cores architectures like GPUs seems to deliver the best FLOPS per watt ratio.

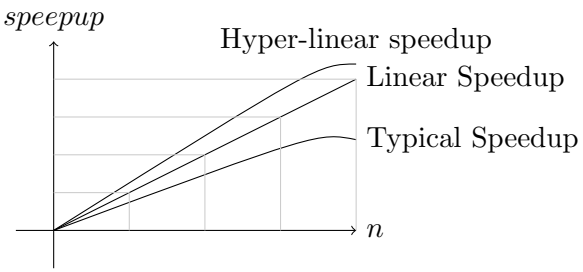


Figure 1.5: Observed speedup: linear, typical and hyper-linear speedups

1.5.3 Scalability

The scalability express the way a program react to parallelism. When an algorithm is implemented on a serial machine and is ideal to solve a problem, one may consider to use it on more than one core, socket, node or even cluster. Indeed, one may expect less computation time, bigger problem or a combination of both while using more resources. This completely depend on the algorithm parallelization and is expressed through scalability. A scalable program will scale on as many processors as we give, whereas a poorly scalable one will give same of even worst results as the serial code. Scalability can be approach using speedup and efficiency.

1.5.4 Speedup and efficiency

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

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

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

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

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

Linear: reference The linear speedup usually represents the target for every program in HPC. Indeed, having the speedup growing 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 face several of the HPC walls like communications wall or memory wall. The increasing number of computational power is reduced to the sequential part or lose time in communications/exchanges.

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

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

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

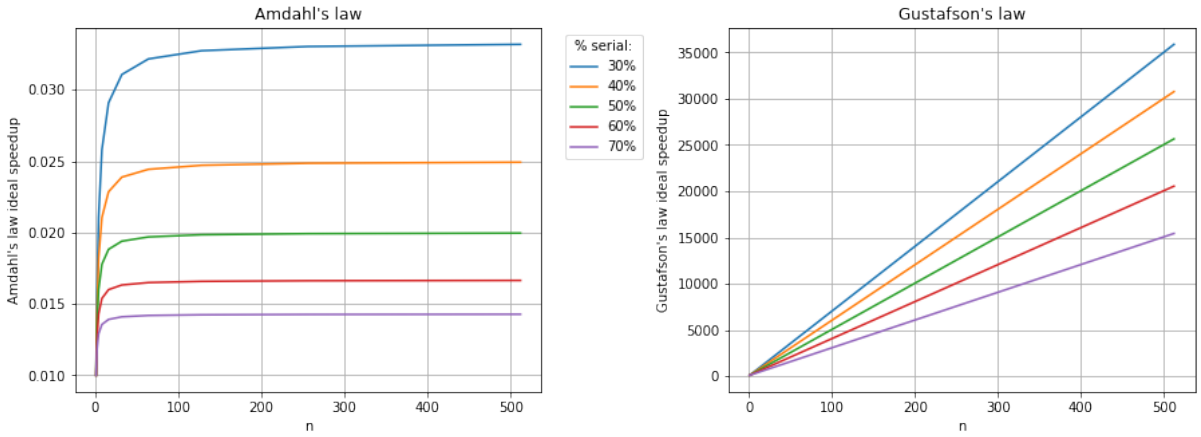


Figure 1.6: Theoretical speedup for Amdahl's (left) and Gustafson's (right) law

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%.

1.5.5 Amdahl's and Gustafson's law

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

Amdahl's law

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

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

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

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

And the efficiency become:

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

A representation of Amdahl's speedup is presented on Fig. 1.6 with varying percentage of serial part. The parallel part is like $Par = (100 - Ser)\%$.

Gustafson's law

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

The speedup can now be estimated by:

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

And the efficiency:

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

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

1.6 Conclusions

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

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

Chapter 2

Hardware in HPC

2.1 Introduction

The knowledge of hardware architecture is essential to reach performances through optimizations. Even if the nowadays software, API, framework or runtime already handle most of optimizations, the last percents of gain 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 with the most famous interconnection topologies.

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

2.2 Early improvements to Von Neumann machine

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

2.2.1 Single core processors

The first processors, around the 1970s, were built using a single computation core like described in the Von Neumann model. Many evolutions were made on those single cores processors from the memory, the order of instruction and the frequency.

Transistor shrink and frequency

A lot of new ways to produce smaller transistors had been discovered from the $10\mu m$ of 1971 to nowadays $10nm$. This allows the vendor to add more transistors on the same die, allowing more complex ISA and features for the CPU.

In parallel of the shrink of transistors, the main feature for better performances with the single core architecture came from the frequency augmentation, the clock rate. Indeed, as fast as the clock rate get, more operations can be computed in a second on the core. In 1970s the frequency was about 4 MHz allowing a maximum of 4 millions of cycles per seconds. Nowadays

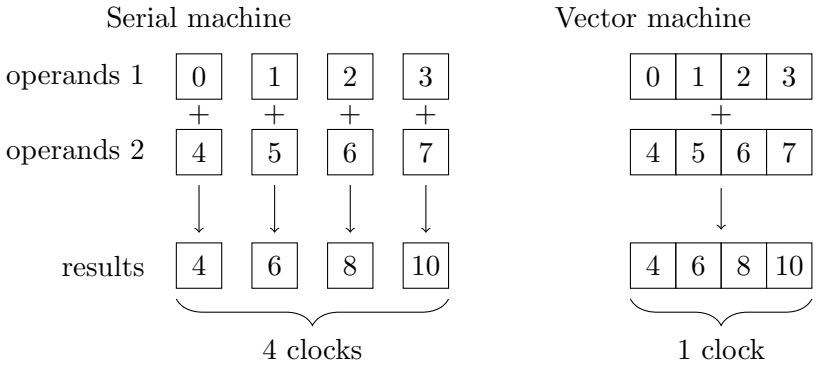


Figure 2.1: Vectorized processeur example on 4 integer addition: 128 bits wide bus

core can work at a frequency of 4GHz and even 5GHz performing billions of operations per cycles.

In/Out-Of-Order

In-order-process is the one describes in previous chapter. The control unit fetches instruction in memory and the operands. The ALU computes the operation, and finally the result is stored in memory.

In this model the time to perform an instruction is the cumulation of: instruction fetching + operand(s) fetching + *computation* + store the result. This time can be high regarding the time when the ALU itself is busy for *computation*, technically just one clock cycle. The idea of Out-of-order 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 instructions 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 allows the instructions to be executed at the same time in a SIMD manner. If the same instruction is executed on coalescent data they can be executed in the same clock cycle. We can, as an example, execute operations simultaneously on 4 to 8 floats with a bus size of 128 or 256 bits in the same cycle. This tool requires specific care during coding with *unrolling* and *loop tiling* and will be address later in this study. The main downside on latest architectures is that vectorization slightly lower the frequency of processors to operate. This requires even more care during programming to avoid bad behavior leading to bad performances.

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 of 1978 and due to its shape and price he was humorously called *the world's most expansive love-seat*.

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

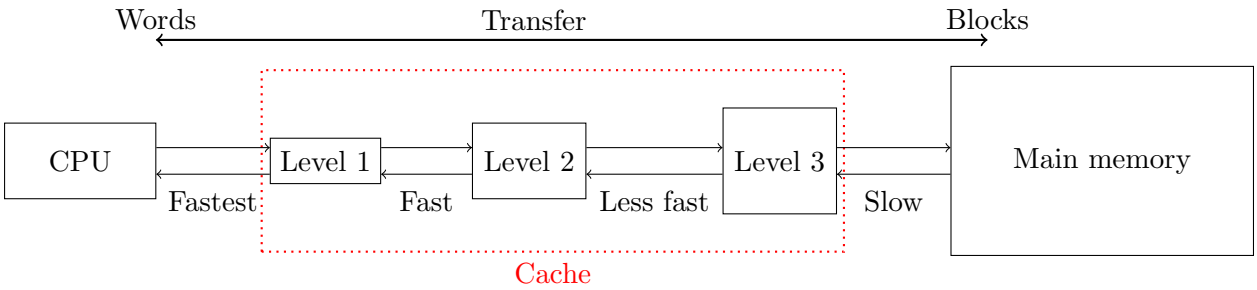


Figure 2.2: Cache memory technology on three levels L1, L2 and L3

Memory technology evolution

The memories technologies optimizations contain several aspect. In early 1980s the augmentation of bus size from 4 bits, 8 bits, 16 bits for nowadays 32 bits for single precision and 64 bits for double precision. 128 bits, 256 bits, etc. buses can also be find allowing technologies we just presented, vectorization. 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 the data as long as the machine is powered. This kind of memory is very expensive to produce due to the number of transistor by memory cell needed and the size of the memory. Therefore it is usually limited for small amount of storage. The SRAM is mainly used for cache memory. Cache is a memory mechanism that is useful to consider when targeting performance.

Cache memory: The main idea of cache technology is presented on figure 2.2. This little memory is built over several levels. The closer to the CPU is L1, then L2 and generally no more than L3 except on specific architecture. When looking for a data the CU will first check 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, they are chunks of memory. With the smaller level L2 and L1 line of informations usually called *words* are exchanged. This is based on the idea that if a data is used, it shall be use again in the near future. Many cache architectures exist like direct, associative, fully associative, etc. In a program the ratio of cache accesses *cache-hits* and *cache-miss* respectively when a data is present in cache and when a data have to be retrieved from lower level or main memory can be very important to reach performances.

DRAM: The Dynamic Random Access Memory is based on transistors and capacitors to store the binary information. This memory is less expansive to produce but needs to be refresh at a determined frequency however the data are lost. This refresh step is in fact a reading-writing operation on the whole memory at a specific frequency. There is several sub categories of DRAM used in different devices.

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

Pre-fetching

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

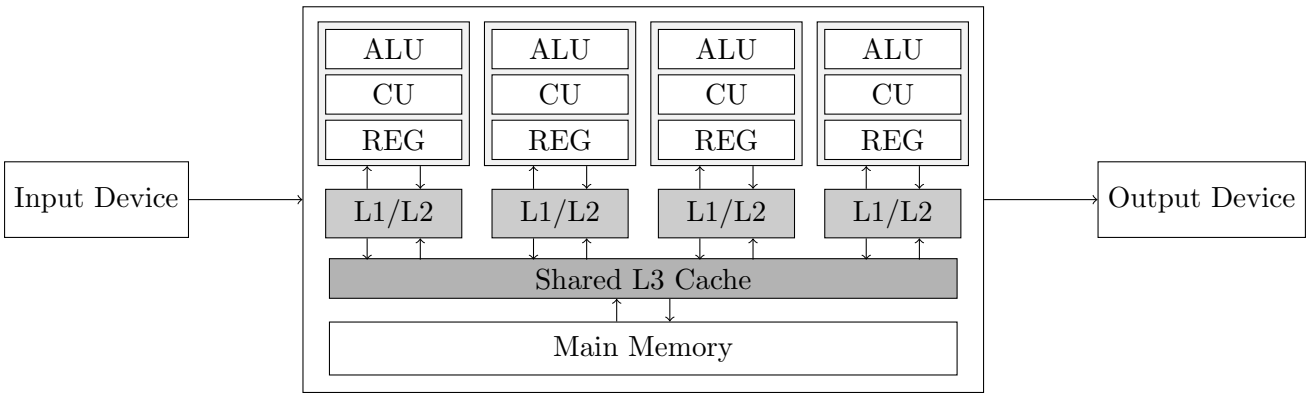


Figure 2.3: Multi-core CPU with 4 cores based on Von Neumann Model presented on figure **??**

2.2.2 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. 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.

The first idea was to provide multi-CPU devies, embedding several CPU on the same motherboard and allowing them to share memory. The evolution of that is multi-core, having several CPUs on the same die directly allowing more optimization inside the die combining all the advantages of single core processors. 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 CPUs. This is the architecture of most of nowadays processors. They are called multi-cores and provide up to 2 to 32 cores. Those processors are called "Host" because they are usually bootable and most of the accelerators need to be attached to them in order to work.

This architecture is presented on figure 2.3. The memory, like presented in the previous chapter, is now shared between the cores. The registers and cache are different: another layer is added to the cache and consistency have to be maintain over all the cores. If a process modify a data in the memory this information have to be spread over all the other users of this data, even in their local cache.

2.3 21th century architectures

After years of developpement 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

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

Intel

Intel was created in 1968 by a chemist and a physicists, Gordon E. Moore and Robert Noyce, in Mountain View, California. Nowadays processors are mostly Intel ones, this world leader equips around 90% of the supercomputers (November 2017 TOP500 list).

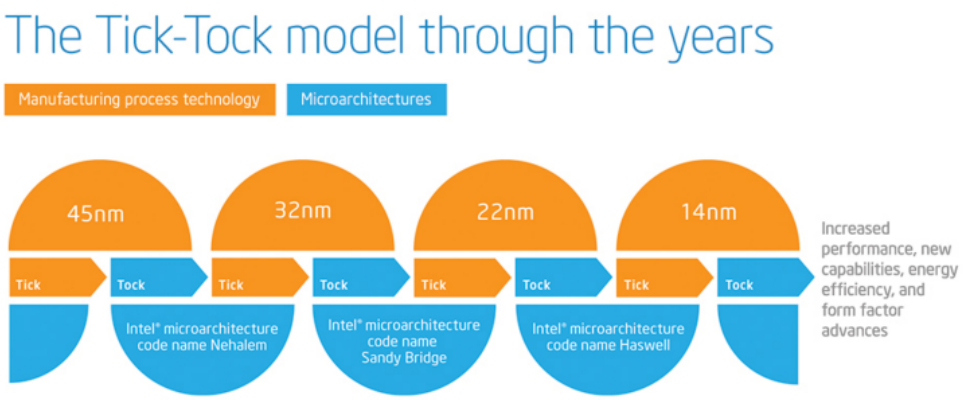


Figure 2.4: Intel Tick-Tock model

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

Hyper-threading Another specificity of Intel processor is Hyper-threading (HT). This technology makes a single physical processor appearing as two logical processors for user's level. In fact a processor embedding 8 cores appears as a 16 cores for user. Adding more computation per node can technically allows the cores to switch context when data are fetched from the memory using the processor 100% during all the computation. A lot of studies have been released on HT from Intel itself [Mar02] to other studies [BBDD06, LAH+02]. This optimization does not fit to all the cases and can be disable for normal use of the processors.

ARM

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

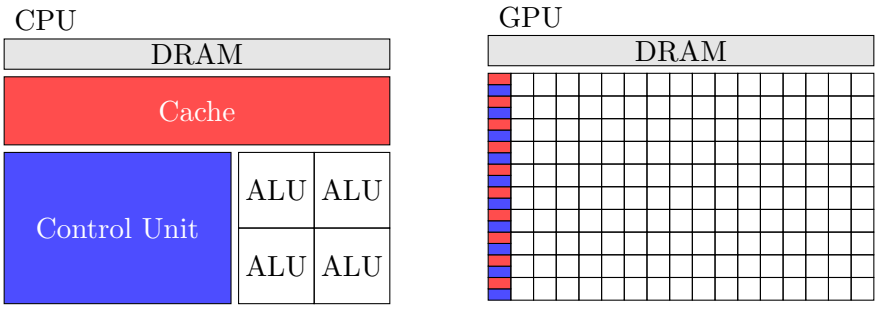


Figure 2.5: Multi-core versus Many-core architecture, case of GPUs

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

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

2.3.2 Many-cores, SIMT

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.

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.

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

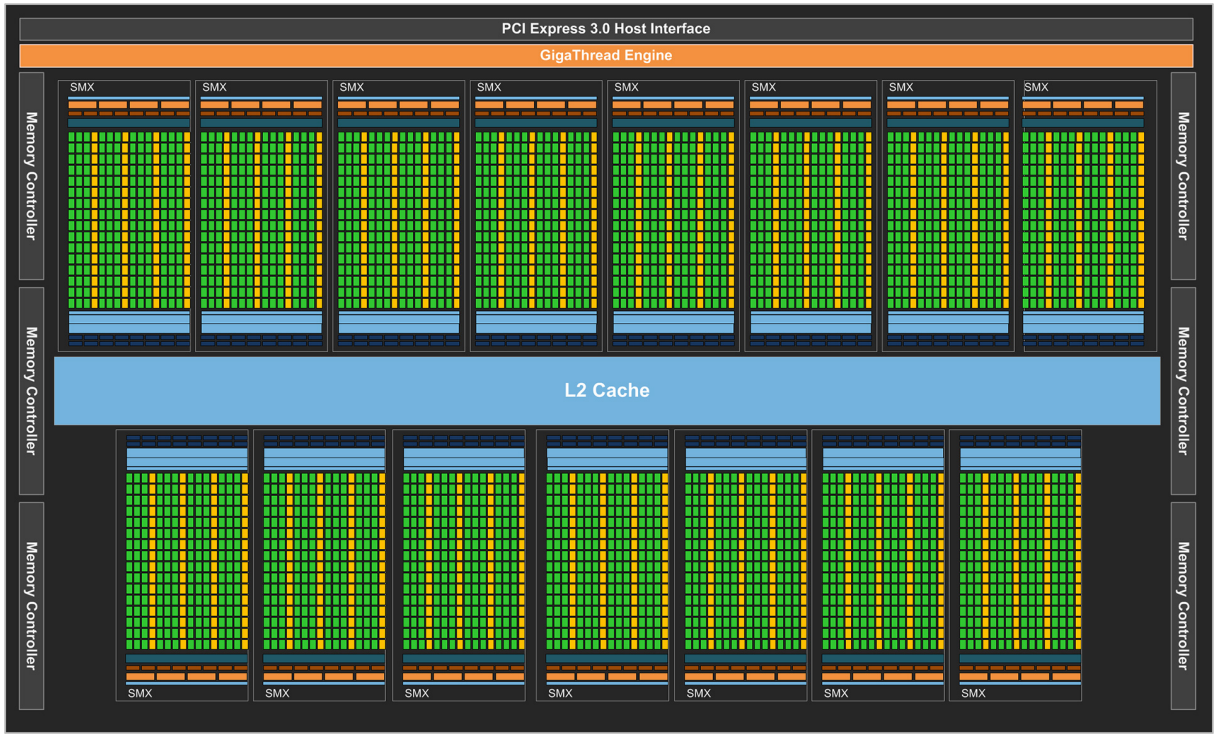


Figure 2.6: NVIDIA Tesla Kepler architecture. Single-precision in green and double-precision in yellow

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.

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.

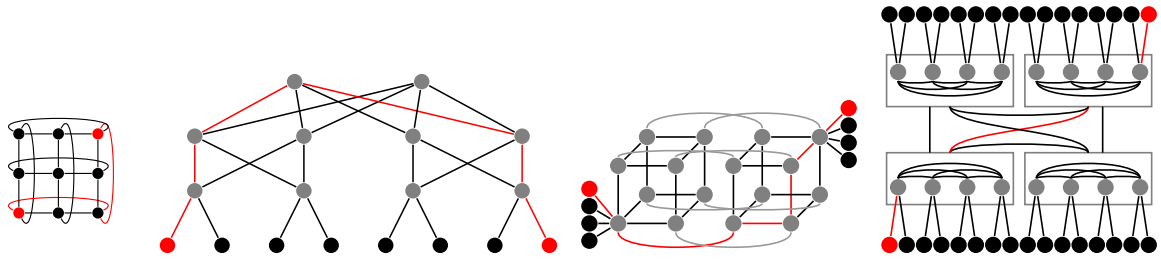


Figure 2.7: Torus, Fat-Tree, HyperX, DragonFly

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

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

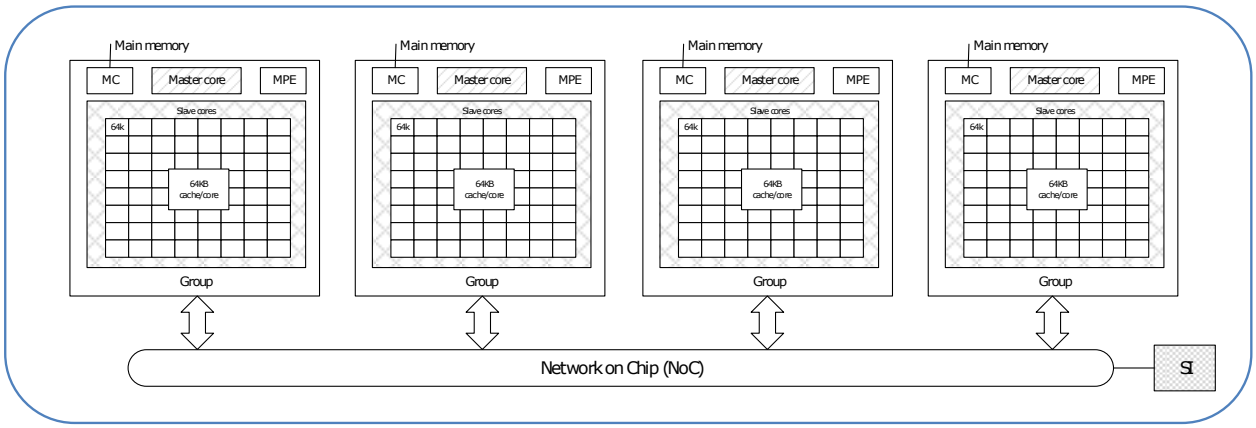


Figure 2.8: Sunway Taihulight node architecture from *Report on the Sunway TaihuLight System*, Jack Dongarra, June 24, 2016.

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

Sunway Taihulight

Sunway Taihulight is the third Chinese supercomputer to be ranked in the first position of the TOP500 list. A recent report from Jack J. Dongarra, a figure in HPC, decrypt the architecture of this supercomputer[Don16]. The most interesting point is the conception of this machine, completely done in China. The Sunway CPUs were invented and built in China. The Vendor is the Shanghai High Performance IC Design Center.

The SW26010, a many core architecture processor, features 260 cores based on RISC architecture and a specific conception depicted on figure 2.8. 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.

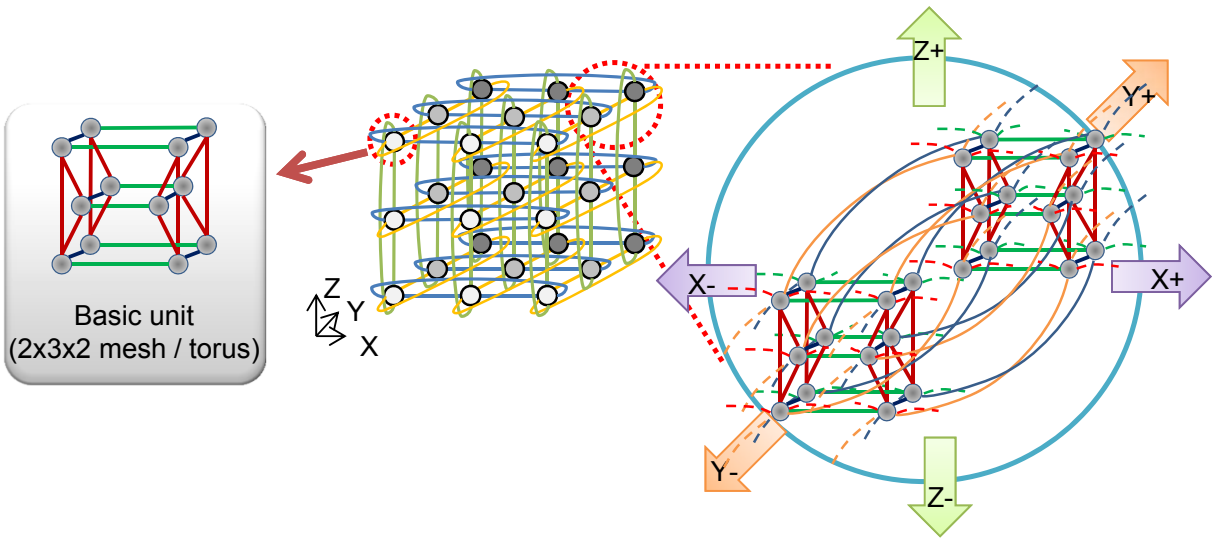


Figure 2.9: TOFU Interconnect schematic from *The K-Computer: System Overview*, Atsuya Uno, SC11

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

Piz Daint

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

K-Computer

K-Computer was the top 1 supercomputer of TOP500 2011 list. The TOFU interconnect network makes the K-Computer unique [ASS09] and stands for TOrus FUsion. This interconnect presented in figure 2.9 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 meso-center (French name for software and hardware architectures) is used for HPC for theoretic research and domain science like applied mathematics, physics, biophysics and chemistry.

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

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

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

2.5.1 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.

2.5.2 New ROMEO supercomputer, June 2018

Avoir les info et decrire le nouveau ROMEO

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.

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

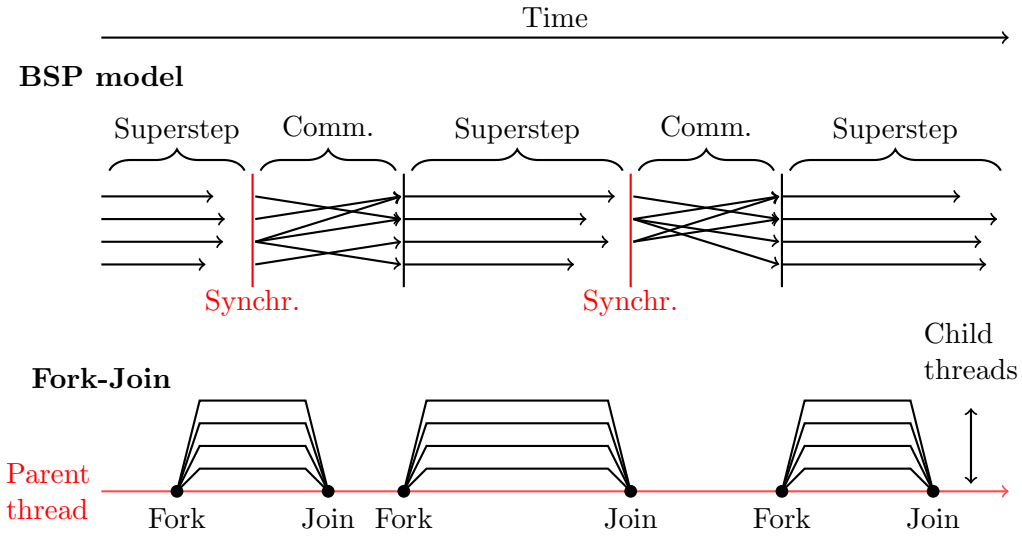


Figure 3.1: Bulk Synchronous Parallel model and Fork-Join model

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 3.1. A main thread pilot the overall execution. When requested by the application, typically following the idea of *divided-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.

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.

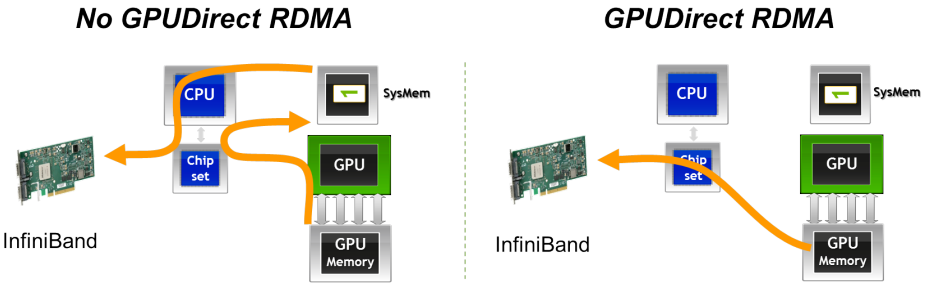


Figure 3.2: GPUDirect RDMA from NVIDIA Developer Blog, *An Introduction to CUDA-Aware MPI*

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.

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.

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 3.2.

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

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 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 3.3, 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

[5] http://charmplusplus.org/
[6] http://legion.stanford.edu/

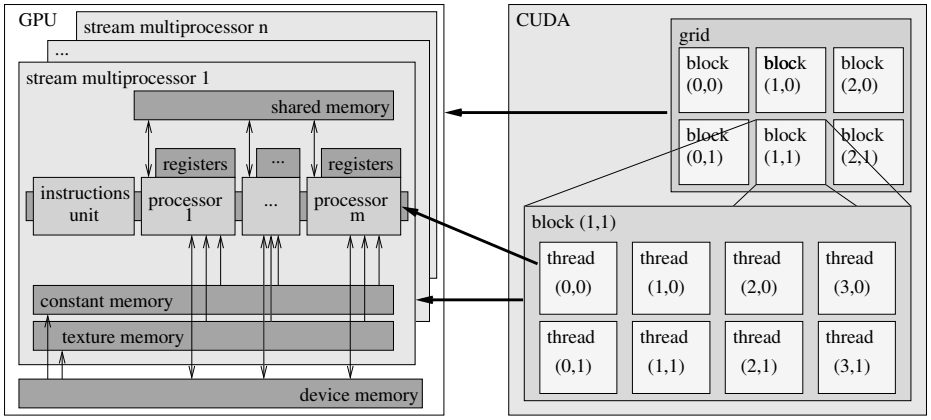


Figure 3.3: NVIDIA GPU and CUDA architecture overview

32 threads which execute the instructions simultaneously. The blocks are distributed over the GPU SMXs to be executed independently.

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

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

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.

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

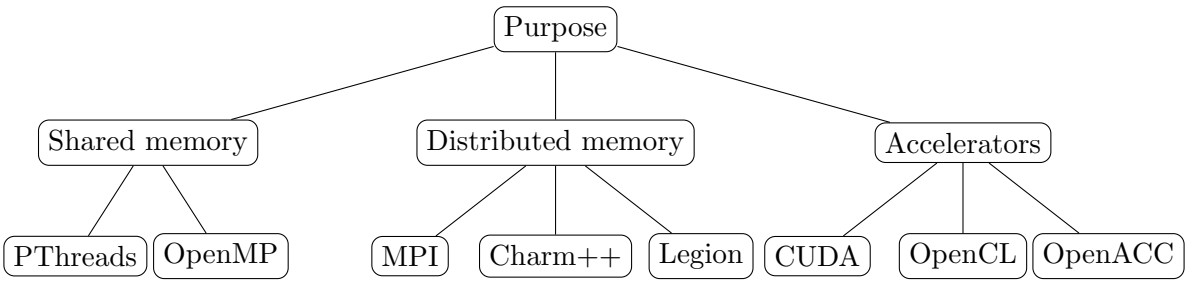


Figure 3.4: Runtimes, libraries, frameworks or APIs

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 on the same level? Those questions are still empty but many tests prove that OpenCL can be as comparable as CUDA but rarely better[KDH10, FVS11].

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

OpenACC

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

The runtime, libraries, frameworks and APIs are summarized in figure 3.4. 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 question 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

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

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 1, 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.

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;

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

[10] https://www.graph500.org/

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

- 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.

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.

Part II

Complex systems

Introduction

The tools for comprehension of High Performance Computing from theory, hardware and software give us the basis elements to go toward optimizations and benchmarking. We showed through examples that hybrid architectures seems to be the way to reach exascale in few years. In the same time many optimizations need to be done to fit the energy envelope and the ability to target all kind of applications. The need of regular memory accesses, synchronization and the host-device memory separation put some constraint on their usage. As many-cores architectures presented nowadays come with several downsides we need to confront them to classical processor in a specific set of problems, a metric, a dedicated benchmark suite.

In benchmarks like the TOP500 the target is to solve a problem with regular computation and communication behavior. We think that this behavior does not fit realistic (production) applications. In many domains like meteorology, oceanography, astrophysics, big data, ... the underlying issue are the irregular input and behavior. The irregularity in an application can have several definitions. This is defined by [JTB] as a problem which: can not be characterize a priori, is input data dependent and evolves with the computation itself. In [SL06] the author specifies that the work involve subcomputations which cannot be determined before and implies work distribution during runtime. The irregularity can then spread on all the layers of the resolution: the communications, the computation and the memory searches.

In order to target interesting applications, we identify several bottlenecks and limitations in HPC. Those limitations are called *walls* against which nowadays architectures are confronted to reach exascale.

Memory Wall: This problem is targeted for the first time in [WM95]. The author explains that:

> We all know that the rate of improvement in microprocessor speed exceeds the rate of improvement in DRAM memory speed, each is improving exponentially, but the exponent for microprocessors is substantially larger than that for DRAMs.

In the case of accelerator another layer of memory is added. The memory of the host processors and devices accelerators cannot be accessed directly and copies from one to the other are requested. The problems of coalescent accesses are also addressed in this study. Some companies try to find ways using shared memory between host and device but this technology is always under developments and tests for HPC purpose.

The two problems we propose for our metric implement heavy memory utilization. The first one dynamically use the memory in an irregular way. The tree traversal of the first method generates temporary data and binary tests. The algebraic method uses a dedicated big integers library and carries propagation are applied just when necessary. In the second benchmark the memory is saturated and prepared at the startup. It is then accessed in an irregular way during all the computation.

Communication wall: We showed that the supercomputer architecture is based on a set of racks, composed of nodes composed of computation units. The network topology can never be

perfect for all the kind of problems and even the fastest technologies are limited to the software handling. Limiting the big synchronizations steps, like in the BSP model, allows the system to be asynchronous and hide computation by communications. Unfortunately this is not applicable to all the applications and a huge care have to be taken to approach perfect scaling.

We decided to target this problem in two main ways. In the first benchmark the model corresponds to FIIT: Finite number of Independent and Irregular Tasks, introduced in [FKF03]. The tasks can be solved independently and then merged at the end. We show that the accelerators can also take advantage of specific distribution methods like Best-Effort. In the second problem communications are central because the data are too big to be represented on a single machine and cannot be computed independently. The irregularity in communication is very high. In this benchmark the number of data shared is never know before reaching the end of a superstep.

Power wall: The energy consumption of nowadays and future supercomputers is the main wall in HPC. Indeed, an exascale supercomputer could be construct using several petascale supercomputers but, with todays architectures, will require a full nuclear plant to operate. In this objective low energy consumption and innovative architectures need to be find. The energy considered is required to power the machine itself but also handle the heat generated.

This wall is the underlying goal of this study. The hybrid architectures seems to deliver better performances for less watts. This thesis shows through different benchmarks what is the real benefit of accelerators on realistic problems and that the performances can be even better for less watts of consumption.

Computational wall: The computational wall is a combination of the wall presented before. By increasing the memory wall, the energy consumption and the communications we can increase the overall computation power of the supercomputer. The limitation in computational power also come from the fact that the Moore's law seems to be over. Vendors have more difficulties to shrink transistors due to physicals side effects. The frequency itself seems to reach its highest values due to the energy require to operate, the heat dissipated and synchronization issues.

This is targeted in the first benchmark we propose. The algebraic method is heavy in computation with irregular memory accesses. The second one does focus on irregular communication and memory usage with test base operations.

This is why we present in this part a new metric for the main HPC walls to confront many-core architectures to multi-core architectures. The first academic problem we choose for our benchmark characterizes the behavior of accelerators and more specifically GPUs focusing on irregular computations and memory accesses. It is the problem of Langford and it shows how we can take advantage of accelerators for this kind of problem.

The second problem we choose is focused on irregular memory accesses and communications. It is based on a benchmark we presented in the previous part, the Graph500 benchmark.

Chapter 4

Computational Wall: Langford Problem

4.1 Introduction

Our goal is to determine the behavior of accelerators compared to classical processor in case of irregular-computationally heavy problems. For this purpose we choose the Langford problem which is an academic problem of combinatorial counting.

We present the problem and expose the two possible methods to solve it:

- The tree traversal, call Miller's method, providing us benchmark targeting irregular memory and computation.
- The algebraic version, calls Godfrey's method, showing the performances of many-core vs multi-core architectures in respect to computationally heavy and memory irregular behaviors.

We show the optimizations made to the regular processor algorithm to efficiently implement this application on GPU. We compare the two approaches with classical processor and GPU implementation. The results are then presented to show the acceleration using the whole ROMEO supercomputer.

4.1.1 The Langford problem

C. Dudley Langford gave his name to a classic permutation problem [Gar56, Sim83]. While observing his son manipulating blocks of different colors, he noticed that it was possible to arrange three pairs of different colored blocks (yellow, red and blue) in such a way that only one block separates the red pair - noted as pair 1 - , two blocks separate the blue pair - noted as pair 2 - and finally three blocks separate the yellow one - noted as pair 3 - , see figure 4.1.

This problem has been generalized to any number n of colors and any number s of blocks having the same color. $L(s, n)$ consists in searching for the number of solutions to the Langford problem, up to a symmetry. In November 1967, Martin Gardner presented $L(2, 4)$ (two cubes and four colors) as being part of a collection of small mathematical games and he stated that

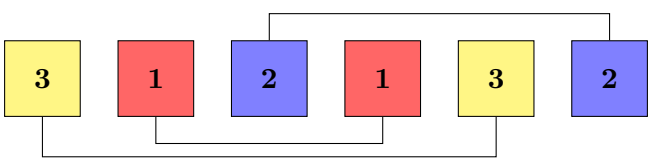


Figure 4.1: L(2,3) arrangement

Instance	Solutions	Method	Computation time
L(2,3)	1	Miller algorithm	-
L(2,4)	1		-
...	...		...
L(2,16)	326,721,800		120 hours
L(2,19)	256,814,891,280		2.5 years (1999) DEC Alpha
L(2,20)	2,636,337,861,200	Godfrey algorithm	1 week
L(2,23)	3,799,455,942,515,488		4 days with CONFIIT
L(2,24)	46,845,158,056,515,936		3 months with CONFIIT
L(2,27)	111,683,611,098,764,903,232		2 days on ROMEO
L(2,28)	1,607,383,260,609,382,393,152		23 days on ROMEO

Table 4.1: Solutions and time for Langford problem using different methods

$L(2,n)$ has solutions for all n such that:

$$\text{solutions for: } \begin{cases} n = 4k \\ n = 4k-1 \end{cases} \quad k \in \mathbb{N}^+ \tag{4.1}$$

The central resolution method consists in placing the pairs of cubes, one after the other, on the free places and backtracking if no place is available (see figure 4.3 for detailed algorithm).

The Langford problem has been approached in different ways: discrete mathematics results, specific algorithms, specific encoding, constraint satisfaction problem (CSP), inclusion-exclusion ... [Mil99, Wal01, Smi00, Lar09]. In 2004, the last solved instance, $L(2,24)$, was computed by our team [JK04a] using a specific algorithm. Table 4.1 presents the latest results and number of solutions. $L(2,27)$ and $L(2,28)$ have just been computed but no details were given.

The most efficient known algorithms are: the Miller backtrack method, the Godfrey algebraic method and the Larsen inclusion-exclusion method. The Miller one is based on backtracking and can be modeled as a CSP; it allowed us to move the limit of explicits solutions building up to $L(2,21)$ but combinatorial explosion did not allow us to go further. Then, we use the Godfrey method to achieve $L(2,24)$ more quickly and then recompute $L(2,27)$ and $L(2,28)$, presently known as the last instances. The Larsen method is based on inclusion-exclusion [Lar09]; although this method is effective, practically the Godfrey one is better. The latest known work on the Langford Problem is a GPU implementation proposed in [ABL15] in 2015. Unfortunately this study does not provide any performance considerations but just gives the number of solution of $L(2,27)$ and $L(2,28)$.

4.2 Miller algorithm

The Miller's method is based on a tree traversal to be able to check all the cubes positions. This method finds its limits because even using branch cutting algorithm to traverse the tree, the number of branches to explore stay very high. We implemented a multi-core and many-core version of it for the purpose of our benchmark. Indeed, this huge tree traversal requires irregular computation and memory accesses: two elements fitting the wall we want to confront to our architectures.

In this section we present our multi-GPU cluster implementation of the Miller's algorithm. First, we introduce the backtrack method itself and the elements allowing us to consider it as a good candidate for our metric. Then we present our implementation in order to fit the GPUs architecture. The last section presents our results.

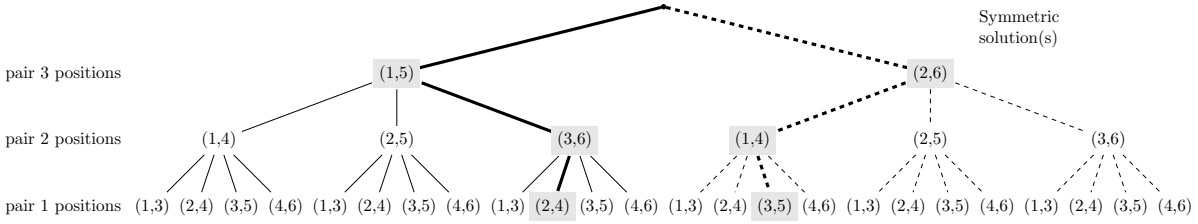


Figure 4.2: Search tree for $L(2,3)$

4.2.1 CSP

Combinatorial problems are NP-complete [GJ79] and can be described as satisfiability problems (SAT) using a polynomial transformation. They can be transformed into CSP formalism. A *Constraint Satisfaction Problem* (CSP), first introduced by Montanari [Mon74], is defined as a triple $< X, D, C >$ where:

$$\begin{cases} X = \{X_1, ..., X_n\}\text{: a finite set of variables} \\ D = \{D_1, ..., D_n\}\text{: their finite domains of values} \\ C = \{C_1, ..., C_p\}\text{: a finite set of constraints} \end{cases} \tag{4.2}$$

The goal in this formalism is to assign values in D to n-uple X respecting all the C p-uple constraints. This approach is a large field of research. [AC14] developed *local search* and compares GPU to CPU. This first work brings to light that GPU is a real contributor to the global computation speed. [CDPD$^+$14] proposes a solver using *propagator* on a GPU architecture to solve CSP problems. [JAO$^+$11] cares about GPU weak points, loading bandwidth and global memory latency.

Considering a basic approach, combinatorial problems formed into CSP can be represented as a tree search. Each level corresponds to a given variable, with values in its domain. Leaves of the tree correspond to a complete assignment (all variables are set). If it meets all the constraints this assignment is called an acceptor state. Depending on the constraints set, the satisfiability evaluation can be made either on complete or partial assignment.

4.2.2 Backtrack resolution

As presented above the Langford problem is known to be a highly irregular combinatorial problem. We first present here the general tree representation and the ways we regularize the computation for GPUs. Then we show how to parallelize the resolution over a multi-GPU cluster.

Langford's problem tree representation

As explained, CSP formalized problems can be transformed into tree evaluations. In order to solve $L(2,n)$, we consider a tree of height n: see example of $L(2,3)$ in figure 7.7.

- Every level of the tree corresponds to a cube color.
- Each node of the tree corresponds to the placement of a pair of cubes without worrying about the other colors. Color p is represented at depth $n-p+1$, where the first node corresponds to the first possible placement (positions 1 and $p+2$) and i^{th} node corresponds to the placement of the first cube of color p in position i, $i \in [1,\ 2n-1-p]$.
- Solutions are leaves generated without any placement conflict.
- As we consider the solution up to a symmetry, the left part is represented dashed and is in fact not traversed.

```
while not done do
 test pair            <- test
 if successful then
   if max depth then
     count solution
     higher pair
   else
     lower pair       <- remove
 else
   higher pair        <- add
```

Figure 4.3: Backtrack algorithm

```
for pair 1 positions
  assignment                  <- add
  for pair 2 positions
    assignment                <- add
    for ...
      for pair n positions
        assignment            <- add
        if final test ok then
          count solution
```

Figure 4.4: Regularized algorithm

	pair 1	pair 2	pair 3
1	0 0 0 1 0 1	0 0 1 0 0 1	0 1 0 0 0 1
2	0 0 1 0 1 0	0 1 0 0 1 0	1 0 0 0 1 0
3	0 1 0 1 0 0	1 0 0 1 0 0	
4	1 0 1 0 0 0		

Figure 4.5: Bitwise representation of pairs positions in $L(2,3)$

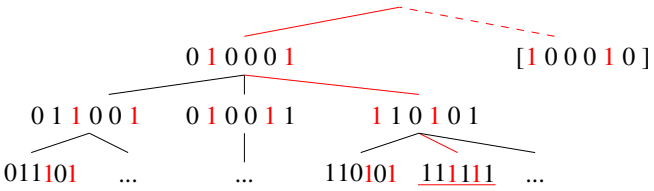


Figure 4.6: Bitwise representation of the Langford $L(2,3)$ placement tree

There are many ways to browse the tree and find the solutions: *backtracking*, *forward-checking*, *backjumping*, etc [Pro93]. We limit our study to the naive *backtrack* resolution and choose to evaluate the variables and their values in a static order; in a depth-first manner, the solutions are built incrementally and if a partial assignment can be aborted, the branch is cut. A solution is found each time a leaf is reached.

The recommendation for performance on GPU accelerators is to use non test-based programs. Due to its irregularity, the basic *backtracking* algorithm, presented on figure 4.3, is not supposed to suit the GPU architecture. Thus a vectorized version is given when evaluating the assignments at the leaves' level, with one of the two following ways: assignments can be prepared on each tree node or totally set on final leaves before testing the satisfiability of the built solution (figure 4.4).

Data representation

In order to count every Langford problem solution, we first identify all possible combinations for one color without worrying about the other ones. Each possible combination is coded within an integer, a bit to 1 corresponding to a cube presence, a 0 to its absence. This is what we called a *mask*. This way figure 4.5 presents the possible combinations to place the one, two and three weight cubes for the $L(2,3)$ Langford instance.

Furthermore the masks can be used to evaluate the partial placements of a chosen set of colors: all the 1 correspond to occupied positions; the assignment is consistent *iff* there are as many 1 as the number of cubes set for the assignment.

With the aim to find solutions, we just have to go all over the tree and *sum* one combination of each of the colors: a solution is found *iff* all the bits of the sum are set to 1.

Each route on the tree can be evaluated individually and independently; then it can be evaluated as a thread on the GPU. This way the problem is massively parallel and can be, indeed, computed on GPU. figure 4.6 represents the tree masks' representation.

Specific operations and algorithms

Three main operations are required in order to perform the tree search. The first one, used for both backtrack and regularized methods, aims to add a pair to a given assignment. The second one, allowing to check if a pair can be added to a given partial assignment, is only necessary

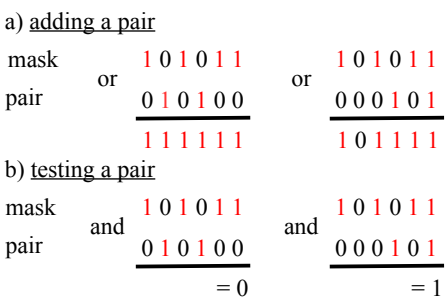


Figure 4.7: Testing and adding position

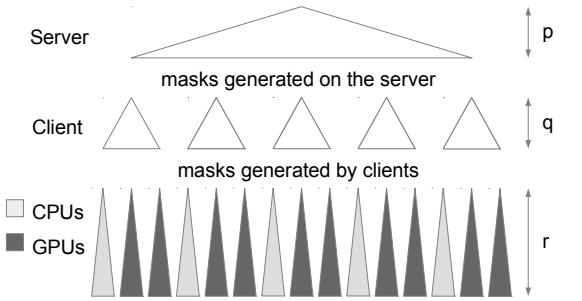


Figure 4.8: Server client distribution

for the original backtrack scheme. The last one is used for testing if a global assignment is an available solution: it is involved in the regularized version of the Miller algorithm.

Add a pair: Top of figure 4.7 presents the way to add a pair to a given assignment. With a *binary OR*, the new mask contains the combination of the original mask and of the added pair. This operation can be performed even if the position is not available for the pair (however the resulting mask is inconsistent).

Test a pair position: On the bottom part of the same figure, we test the positioning of a pair on a given mask. For this, it is necessary to perform a *binary AND* between the mask and the pair.

$= 0$: *success*, the pair can be placed here

$\neq 0$: *error*, try another position

Final validity test: The last operation is for *a posteriori* checking. For example the mask 101111, corresponding to a leaf of the tree, is inconsistent and should not be counted among the solutions. The final placement mask corresponds to a solution *iff* all the places are occupied, which can be tested as $\neg mask = 0$.

Using this data representation, we implemented both *backtrack* and *regularized* versions of the Miller algorithm, as presented in figure 4.3 and 4.4.
The next section presents the way we hybridize these two schemes in order to get an efficient parallel implementation of the Miller algorithm.

4.2.3 Hybrid parallel implementation

Section 4.2.2 presents the Miller algorithm and the tools needed for the resolution. The irregularity is present in the tree traversal and specific data representation has been chosen for efficient use of the memory. This section presents our methodology to implement Miller's method on a multi-GPU cluster.

Tasks generation: In order to parallelize the resolution we have to generate tasks. Considering the tree representation, we construct tasks by fixing the different values of a first set of variables [pairs] up to a given level. Choosing the development level allows to generate as many tasks as necessary. This leads to a *Finite number of Irregular and Independent Tasks* (*FIIT* applications [Kra99]).

Cluster parallelization: The generated tasks are independent and we spread them in a client-server manner: a server generates them and makes them available for clients. As we consider the cluster as a set of CPU-GPU(s) machines, the clients are these machines. At the machines level, the role of the CPU is, first, to generate work for the GPU(s): it has to generate sub-tasks, by continuing the tree development as if it were a second-level server, and the GPU(s) can be considered as second-level client(s).
The sub-tasks generation, at the CPU level, can be made in parallel by the CPU cores. Depending on the GPUs number and their computation power the sub-tasks generation rhythm may be adapted, to maintain a regular workload both for the CPU cores and GPU threads: some CPU cores, not involved in the sub-tasks generation, could be made available for sub-tasks computing.
This leads to the 3-level parallelism scheme presented in figure 4.8, where p, q and r respectively correspond to: (p) the server-level tasks generation depth, (q) the client-level sub-tasks generation one, (r) the remaining depth in the tree evaluation, *i.e.* the number of remaining variables to be set before reaching the leaves.

***Backtrack* and *regularized* methods hybridization:** The Backtrack version of the Miller algorithm suits CPU execution and allows to cut branches during the tree evaluation, reducing the search space and limiting the combinatorial explosion effects. A regularized version had to be developed, since GPUs execution requires synchronous execution of the threads, with as few branching divergence as possible; however this method imposes to browse the entire search space and is too time-consuming.
We propose to hybridize the two methods in order to take advantage of both of them for the multiGPU parallel execution: for tasks and sub-tasks generated at sever and client levels, the tree development by the CPU cores is made using the backtrack method, cutting branches as soon as possible [and generating only possible tasks]; when computing the sub-tasks generated at client-level, the CPU cores involved in the sub-tasks resolution use the backtrack method and the GPU threads the regularized one.

4.2.4 Experiments tuning

In order to take advantage of all the computing power of the GPU we have to refine the way we use them: this section presents the experimental study required to choose optimal settings. This tuning allowed us to prove our proposal on significant instances of the Langford problem.

Registers, blocks and grid: In order to use all GPUs capabilities, the first way was to fill the blocks and grid. To maximize occupancy (ratio between active warps and the total number of warps) NVIDIA suggests to use 1024 threads per block to improve GPU performances and proposes a CUDA occupancy calculator[1]. But, confirmed by the Volkov's results[Vol10], we experimented that better performances may be obtained using lower occupancy. Indeed, another critical criterion is the inner GPU registers occupation. The optimal number of registers (57 registers) is obtained by setting 9 pairs placed on the client for $L(2, 15)$, thus 6 pairs are remaining for GPU computation.

In order to tune the blocks and grid sizes, we performed tests on the ROMEO architecture. Figure 4.9 represents the time in relation with the number of blocks per grid and the number of threads per block. The most relevant result, observed as a local minimum on the 3D surface, is obtained near 64 or 96 threads per block; for the grid size, the limitation is relative to the GPU global memory size. It can be noted that we do not need shared memory because their are no data exchanges between threads. This allows us to use the total available memory for the L1 cache for each thread.

[1] `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`

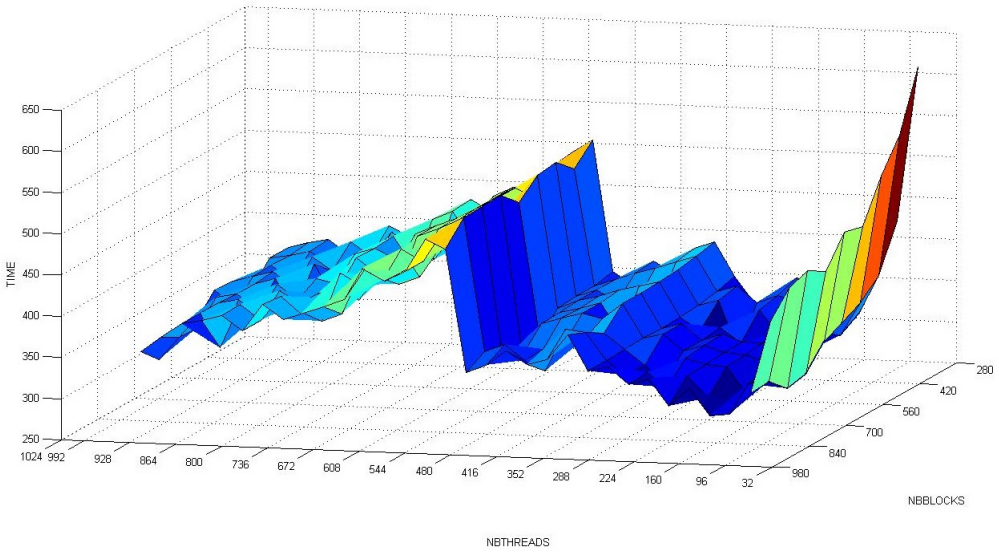


Figure 4.9: Time depending on grid and block size on $n = 15$

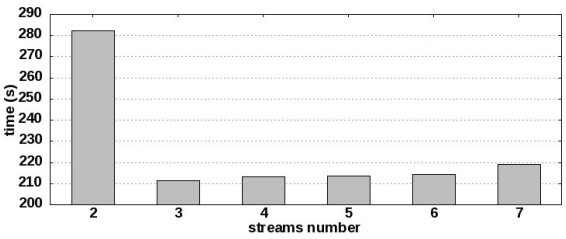


Figure 4.10: Computing time depending on streams number

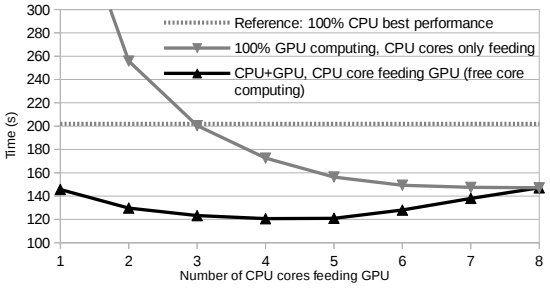


Figure 4.11: CPU cores optimal distribution for GPU feeding

Streams: A client has to prepare work for GPU. There are four main steps: generate the tasks, load them into the device memory, process the task on the GPU and then get the results.

CPU-GPU memory transfers cause huge time penalties (about 400 cycles latency for transfers between CPU memory and GPU *device memory*). At first, we had no overlapping between memory transfer and kernel computation because the tasks generation on CPU was too long compared to the kernel computation. To reduce the tasks generation time we used OpenMP in order to use the eight available CPU cores. Thus CPU computation was totally hidden by memory transfers and GPU kernel computation. We tried using up to 7 streams; as shown by figure 4.10, using only two simultaneous streams did not improve efficiency because the four steps did not overlap completely; the best performances were obtained with three streams; the slow increase in the next values is caused by synchronization overhead and CUDA streams management.

Setting up the server, client and GPU depths: We now have to set the depths of each actor, server (p), client (q) and GPU (r) (see figure 4.8).

First we set the $r = 5$ for large instances because of the GPU limitation in terms of registers by threads, exacerbated by the use of numerous $64bits$ integers. For $r \geq 6$, we get too many registers (64) and for $r \leq 4$ the GPU computation is too fast compared to the memory load overhead.

Clients are the buffers between the server and the GPUs: $q = n - p - r$. So we have conducted tests by varying the server depth, p. The best result is obtained for $p = 3$ and

n	CPU (8c)	GPU (4c) + CPU (4c)
15	2.5	1.5
16	21.2	14.3
17	200.3	120.5
18	1971.0	1178.2
19	22594.2	13960.8

(a) Regularized method (seconds)

n	CPU (8c)	GPU (4c) + CPU (4c)
17	29.8	7.3
18	290.0	73.6
19	3197.5	803.5
20	–	9436.9
21	–	118512.4

(b) Backtrack (seconds)

Table 4.2: Comparison between multi-core processors and GPUs for regularized and backtrack method

performance decreases quickly for higher values. This can be explained since more levels on the server generates smaller tasks; thus GPU use is not long enough to overlap memory exchanges.

CPU: Feed the GPUs and compute The first work of CPU cores is to prepare tasks for GPU so that we can generate overlapping between memory load and kernel computation. In this configuration using eight cores to generate GPU tasks under-uses CPU computation power. It is the reason why we propose to use some of the CPU cores to take part of the sub-problems treatment. Figure 4.11 represents computation time in relation with different task distributions between CPU and GPU. We experimentally demonstrated that only 4 or 5 CPU cores are enough to feed GPU, the other ones can be used to perform backtrack resolution in competition with GPUs.

4.2.5 Results

Regularized method results We now show the results obtained for our massively parallel scheme using the previous optimizations, comparing the computation times of successive instances of the Langford problem. These tests were performed on 20 nodes of the ROMEO supercomputer, hence 40 CPU/GPU machines.

The previous limit with Miller's algorithm was $L(2, 19)$, obtained in 1999 after 2.5 years of sequential effort and at the same time after 2 months with a distributed approach[Mil99]. Our computation scheme allowed us to obtain it in less than 4 hours (Table 4.2a), this being not only due to Moore law progress.
Note that the computation is 1.6 faster with CPU+GPU together than using 8 CPU cores. In addition, the GPUs compute $200,000\times$ more nodes of the search tree than the CPUs, with a faster time.

The computation time between two different consecutive instances being multiplied by 10 approximately, this could allow us to obtain $L(2, 20)$ in a reasonable time.

Backtracking on GPUs It appears at first sight that using backtracking on GPUs without any regularization is a bad idea due to threads synchronization issues. But in order to compare CPU and GPU computation power in the same conditions we decided to implement the original backtrack method on GPU (see figure 4.3) with only minor modifications. In these conditions we observe very efficient work of the NVIDIA scheduler, which perfectly handles threads desynchronization. Thus we use the same server-client distribution as in 4.2.3, each client generates masks for both CPU and GPU cores. The workload is then statically distributed on GPU and CPU cores. Executing the backtrack algorithm on a randomly chosen set of sub-problems allowed us to set the GPU/CPU distribution ratio experimentally to 80/20%.

The experiments were performed on 129 nodes of the ROMEO supercomputer, hence 258 CPU/GPU machines and one node for the server. Table 4.2b shows the results with this configuration. This method first allowed us to perform the computation of $L(2, 19)$ in less than 15 minutes, $15\times$ faster than with the regularized method; then, we pushed the limitations of the

Miller algorithm up to $L(2,20)$ in less than 3 hours and even $L(2,21)$ in about 33 hours[2].

This first benchmark application provides a perfect example for the behavior of many-core accelerators confronted to irregular application. Even if the application does not seemed to fit with the direct tree traversal, the results prove that if used in the right way performance can be achieve in this case.

4.3 Godfrey's algebraic method

The previous part presents the Miller algorithm for the Langford problem, this method cannot achieve bigger instances than the $L(2,21)$. It allows us the target specific walls with memory and computation irregularities.
An algebraic representation of the Langford problem has been proposed by M. Godfrey in 2002. This example is perfect in our study to target computationally heavy context with irregular memory behavior. In this part we describe this algorithm and optimizations, and then our implementation on multiGPU clusters. As a side note, this implementation also allowed us to break a new world record on the computation time needed to solve the last instances, $L(2,27)$ and $L(2,28)$.

4.3.1 Method description

Consider $L(2,3)$ and $X = (X_1, X_2, X_3, X_4, X_5, X_6)$. It proposes to modelize $L(2,3)$ by:

$$\begin{aligned} F(X,3) =& (X_1X_3 + X_2X_4 + X_3X_5 + X_4X_6)\times \\ & (X_1X_4 + X_2X_5 + X_3X_6)\times \\ & (X_1X_5 + X_2X_6) \end{aligned} \tag{4.3}$$

In this approach each term represents a position of both cubes of a given color and a solution to the problem corresponds to a term developed as $(X_1X_2X_3X_4X_5X_6)$; thus the number of solutions is equal to the coefficient of this monomial in the development. More generally, the solutions to $L(2,n)$ can be deduced from $(X_1X_2X_3X_4X_5...X_{2n})$ terms in the development of $F(X,n)$.

If $G(X,n) = X_1...X_{2n}F(X,n)$ then it has been shown that:

$$\sum_{(x_1,...,x_{2n})\in\{-1,1\}^{2n}} G(X,n)_{(x_1,...,x_{2n})} = 2^{2n+1}L(2,n) \tag{4.4}$$

So

$$\sum_{(x_1,...,x_{2n})\in\{-1,1\}^{2n}} (\prod_{i=1}^{2n} x_i) \prod_{i=1}^{n} \sum_{k=1}^{2n-i-1} x_k x_{k+i+1} = 2^{2n+1}L(2,n) \tag{4.5}$$

That allows to get $L(2,n)$ from polynomial evaluations. The computational complexity of $L(2,n)$ is of $O(4^n \times n^2)$ and an efficient big integer arithmetic is necessary. This principle can be optimized by taking into account the symmetries of the problem and using the Gray code: these optimizations are described below.

4.3.2 Method optimizations

Some works focused on finding optimizations for this arithmetic method[Jai05]. Here we explain the symmetric and computation optimizations used in our algorithm.

[2]Even if this instance has no interest since it is known to have no solution

Evaluation parity:

As $[F(-X,n) = F(X,n)]$, G is not affected by a global sign change. In the same way the global sign does not change if we change the sign of each pair or impair variable.

Using these optimizations we can set the value of two variables and accordingly divide the computation time and result size by four.

Symmetry summing:

In this problem we have to count each solution up to a symmetry; thus for the first pair of cubes we can stop the computation at half of the available positions considering
$S'_1(x) = \sum_{k=1}^{n-1} x_k x_{k+2}$ instead of $S_1(x) = \sum_{k=1}^{2n-2} x_k x_{k+2}$. The result is divided by 2.

Sums order:

Each evaluation of $S_i(x) = \sum_{k=1}^{2n-i-1} x_k x_{k+i+1}$, before multiplying might be very important regarding to the computation time for this sum. Changing only one value of x_i at a time, we can recompute the sum using the previous one without global re-computation. Indeed, we order the evaluations of the outer sum using Gray code sequence. Then the computation time is considerably reduced.

Based on all these improvements and optimizations we can use the Godfrey method in order to solve huge instances of the Langford problem. The next section develops the main issues of our multiGPU architecture implementation.

4.3.3 Implementation details

In this part we present the specific adaptations required to implement the Godfrey method on a multiGPU architecture.

Optimized big integer arithmetic:

In each step of computation, the value of each S_i can reach $2n - i - 1$ in absolute value, and their product can reach $\frac{(2n-2)!}{(n-2)!}$. As we have to sum the S_i product on 2^{2n} values, in the worst case we have to store a value up to $2^{2n}\frac{(2n-2)!}{(n-2)!}$, which corresponds to 10^{61} for $n = 28$, with about 200 bits.

So we need few big integer arithmetic functions. After testing existing libraries like GMP for CPU or CUMP for GPU, we came to the conclusion that they implement a huge number of functionalities and are not really optimized for our specific problem implementation: product of "small" values and sum of "huge" values.

Finally, we developed a light CPU and GPU library adapted to our needs. In the summation, for example, as maintaining carries has an important time penalty, we have chosen to delay the spread of carries by using buffers: carries are accumulated and spread only when useful (for example when the buffer is full). Figure 4.12 represents this big integer handling.

This big integer library imposes many constraint on the accelerator memory. We conduct tests using simple and double precision for the basic unit. The carries propagation is also triggered when necessary only. This strategy implies random memory accesses to the words and irregular behavior between the threads.

Gray sequence in memory:

The Gray sequence cannot be stored in an array because it would be too large (it would contain 2^{2n} byte values). This is the reason why only one part of the Gray code sequence is stored in memory and the missing terms are directly computed from the known ones using arithmetic

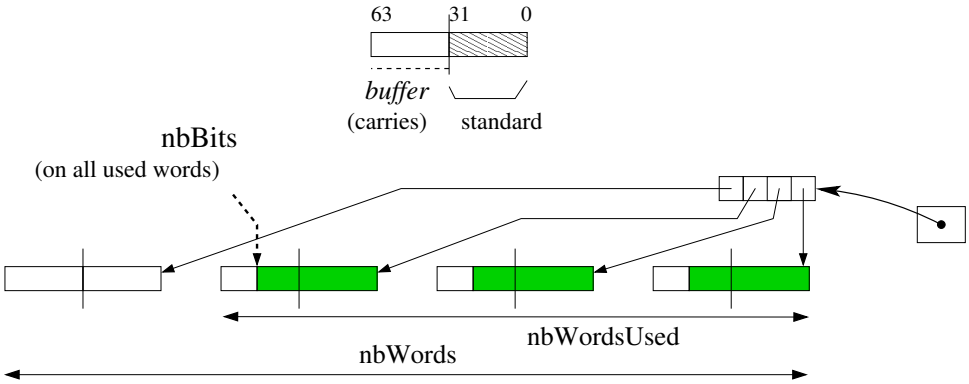


Figure 4.12: Big integer representation, 64 bits words

considerations. The size of the stored part of the Gray code sequence is chosen to be as large as possible to be contained in the processor's cache memory, the L1 cache for the GPUs threads: so the accesses are fastened and the computation of the Gray code is optimized. For an efficient use of the E5-2650 v2 ROMEO's CPUs, which disposes of 20 MB of level-3 cache, the CPU Gray code sequence is developed recursively up to depth 25. For the K20Xm ROMEO's GPUs, which dispose of 8 KB of constant memory, the sequence is developed up to depth 15. The rest of the memory is used for the computation itself.

Tasks generation and computation:

In order to perform the computation of the polynomial, two variables can be set among the $2n$ available. For the tasks generation we choose a number p of variables to generate 2^p tasks by developing the evaluation tree to depth p.

The tasks are spread over the cluster, either synchronously or asynchronously.

Synchronous computation: A first experiment was carried out with an MPI distribution of the tasks of the previous model. Each MPI process finds its tasks list based on its process id; then converting each task number into binary gives the task's initialization. The processes work independently; finally the root process ($id = 0$) gathers all the computed numbers of solutions and sums them.

Asynchronous computation: In this case the tasks can be computed independently. As with the synchronous computation, the tasks' initializations are retrieved from their number. Each machine can get a task, compute it, and then store its result; then when all the tasks have been computed, the partial sums are added together and the total result is provided.

4.3.4 Experimental settings

This part presents the experimental context and methodology, and the way the experiments were carried out. This study has similar goals as for the Miller's resolution experiments.

Methodology:

We present here the way the experimental settings were chosen. Firstly we define the tasks distribution, secondly we set the number of threads per GPU block; finally, we set the CPU/GPU distribution.

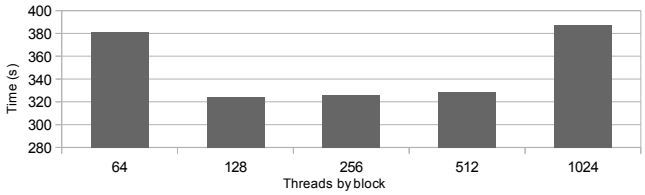
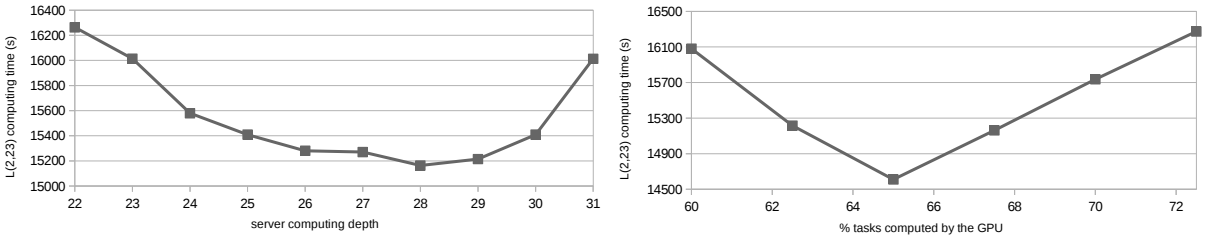


Figure 4.13: $L(2, 20)$, number of threads per block



(a) Influence on server generation depth

(b) Influence of tasks repartition

Figure 4.14: Influences of repartitions of depths and CPU-GPU tasks

Tasks distribution depth: This value being set it is important to get a high number of blocks to maintain sufficient GPU load. Thus we have to determine the best number of tasks for the distribution. As presented in part 4.3.3 the number p of bits determines 2^p tasks. On the one hand, too many tasks are a limitation for the GPU that cannot store all the tasks in its 6GB memory. On the other hand, not enough tasks means longer tasks and too few blocks to fill the GPU grid. figure 4.14a shows that for the $L(2, 23)$ instance the best task number is with generation depth 28.

Number of threads per block: In order to take advantage of the GPU computation power, we have to determine the threads/block distribution. Inspired by our experiments with Miller's algorithm we know that the best value may appear at lower occupancy. We perform tests on a given tasks set varying the threads/block number and grid size associated. Figure 4.13 presents the tests performed on the $n = 20$ problem: the best distribution is around 128 threads per block.

CPU vs GPU distribution:

The GPU and CPU computation algorithm will approximately be the same. In order to take advantage of all the computational power of both components we have to balance tasks between CPU and GPU. We performed tests by changing the CPU/GPU distribution based on simulations on a chosen set of tasks. Figure 4.14b shows that the best distribution is obtained when the GPU handles 65% of the tasks. This optimal load repartition directly results from the intrinsics computational power of each component; this repartition should be adapted if using a more powerful GPU like Tesla K40 or K80.

Computing context:

As presented in part 2.5.1, we used the ROMEO supercomputer to perform our tests and computations. On this supercomputer SLURM[JG03] is used as a reservation and job queue manager. This software allows two reservation modes: a static one-job limited reservation or the opportunity to dynamically submit several jobs in a Best-Effort manner.

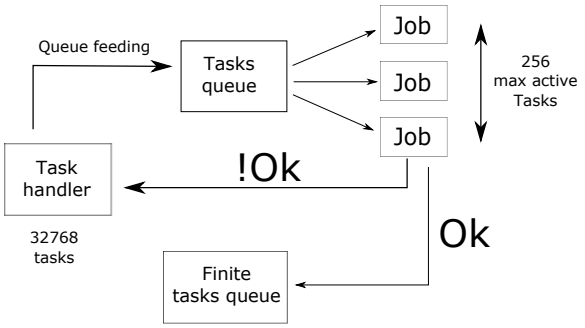


Figure 4.15: Best-effort distribution

Static distribution: In this case we used the synchronous distribution presented in 4.3.3. We submited a reservation with the number of MPI processes and the number of cores per process. This method is useful to get the results quickly if we can get at once a large amount of computation resources. It was used to perform the computation of small problems, and even for $L(2, 23)$ and $L(2, 24)$.
As an issue, it has to be noted that it is difficult to quickly obtain a very large reservation on such a shared cluster, since many projects are currently running.

Best effort: SLURM allows to submit tasks in the specific Best-Effort queue, which does not count in the user *fair-share*. In this queue, if a node is free and nobody is using it, the reservation is set for a job in the best effort queue for a minimum time reservation. If another user asks for a reservation and requests this node, the best effort job is killed (with, for example, a SIGTERM signal). This method, based on asynchronous computation, enables a maximal use of the computational resources without blocking for a long time the entire cluster.

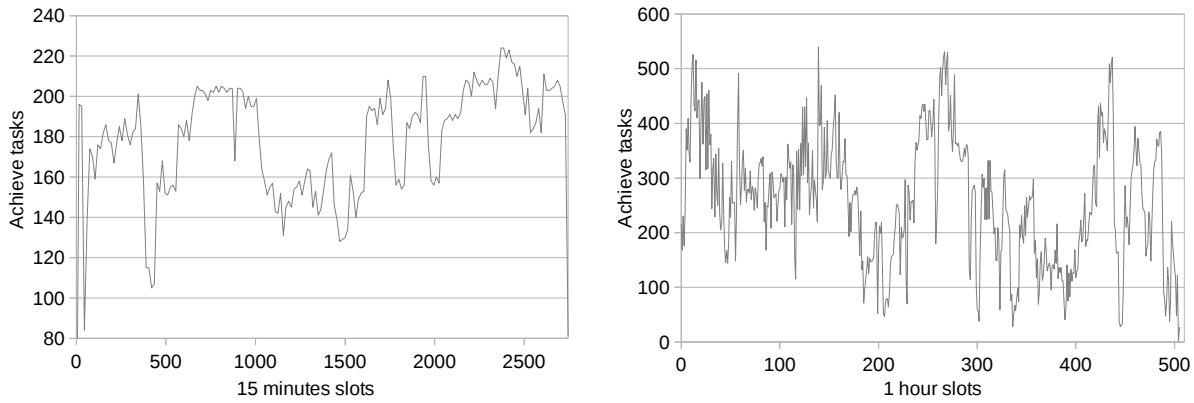
For $L(2, 27)$ and even more for $L(2, 28)$ the total time required is too important to use the whole machine off a challenge period, thus we chose to compute in a Best-Effort manner. In order to fit with this submission method we chose a reasonable time-per-task, sufficient to optimize the treatments with low loading overhead, but not too long so that killed tasks are not too penalizing for the global computation time. We empirically chose to run 15-20 minute tasks and thus we considered $p = 15$ for $n = 27$ and $p = 17$ for $n = 28$.

The best effort based algorithm is presented on figure 4.15. The task handler maintains a maximum of 256 tasks in the queue; in addition the entire process is designed to be fault-tolerant since killed tasks have to be launched again. When finished, the tasks generate an output containing the number of solutions and computation time, that is stored as a file or database entry. At the end the outputs of the different tasks are merged and the global result can be provided.

4.3.5 Results

After these optimizations and implementation tuning steps, we conducted tests on the ROMEO supercomputer using best-effort queue to solve $L(2, 27)$ and $L(2, 28)$. We started the experiment after an update of the supercomputer, that implied a cluster shutdown. Then the machine was restarted and was about 50% idle for the duration of our challenge. The computation lasted less than 2 days for $L(2, 27)$ and 23 days for $L(2, 28)$. The following describes performances considerations.

Computing effort - For $L(2, 27)$, the effective computation time of the 32,768 tasks was about 30 million seconds (345.4 days), and 165,000" elapsed time (1.9 days); the average time of the tasks was 911", with a standard deviation of 20%. For the $L(2, 28)$ 131,072 tasks the total computation time was about 1365 days (117 million seconds), as 23 day elapsed time; the tasks lasted 1321" on average with a 12% standard deviation.



(a) $L(2, 27)$ tasks grouped by 15" slots

(b) $L(2, 28)$ tasks grouped by 1 hour slots

Figure 4.16: Task repartition for $L(2, 27)$ and $L(2, 28)$

Best-effort overhead - With $L(2, 27)$ we used a specific database to maintain information concerning the tasks: 617 tasks were aborted [by regular user jobs] before finishing (1.9%), with an average computing time of 766" (43% of the maximum requested time for a task). This consumed 472873", which overhead represents 1.6% of the effective computing effort.

Cluster occupancy - Figure 4.16a presents the tasks resolution over the two computation days for $L(2, 27)$. The experiment elapse time was 164700" (1.9 days). Compared to the effective computation time, we used an average of 181.2 machines (CPU-GPU couples): this represents 69.7% of the entire cluster.

Figure 4.16b presents the tasks resolution flow during the 23 days computation for $L(2, 28)$. We used about 99 machines, which represents 38% of the 230 available nodes.

For $L(2, 27)$, these results confirm that the computation took great advantage of the low occupancy of the cluster during the experiment. This allowed us to obtain a weak best-effort overhead, and an important cluster occupancy. Unfortunately for $L(2, 28)$ on such a long period we got a lower part of the supercomputer dedicated to our computational project. Thus we are confident in good perspectives for the $L(2, 31)$ instance if computed on an even larger cluster or several distributed clusters.

4.4 Conclusion

This first benchmark presents two methods to solve the Langford pairing problem on multi-GPU clusters. Those methods are presented as the Miller's and Godfrey's algorithms.

4.4.1 Miller's method: irregular memory + computation

In this first implementation we showed that the accelerator can target irregularity in memory and computation. The tree traversal, even with branch cutting, gives us excellent results on many-core architecture. They GPUs handle 80% of the computation effort. Also as any combinatorial problem can be represented as a CSP, the Miller algorithm can be seen as general resolution scheme based on the backtrack tree browsing. A three-level tasks generation allows to fit the multiGPU architecture. MPI or Best-Effort are used to spread tasks over the cluster, OpenMP for the CPU cores distribution and then CUDA to take advantage of the GPU computation power. We were able to compute $L(2, 20)$ with this regularized method and to get an even better time with the basic backtrack. This proves the proposed approach and also exhibits that the GPU scheduler is very efficient at managing highly divergent threads.

4.4.2 Godfrey's method: irregular memory + heavy computation

We also presented the Godfrey's method which target irregular memory accesses but also heavy computational behavior. The results shows that the many-core version completely exceed the classical processor best performances. The GPU handle 65% of the computation effort. In addition we beat the Langford limits using multi-GPUs. In order to use the supercomputer ROMEO, which is shared by a large scientific community, we have implemented a distribution that does not affect the machine load, using a best-effort queue. The computation is fault-tolerant and totally asynchronous. The utilization of this technology also show the reliability of nowadays accelerators technologies. Indeed, they can be use at demand on specific tasks without specific warm-up.

Langford problem results: This work also enabled us to compute $L(2, 27)$ and $L(2, 28)$ in respectively less than 2 days and 23 days on the University of Reims ROMEO supercomputer. This is the fastest time on those problems nowadays. The total number of solutions is:

L(2,27) = 111,683,611,098,764,903,232
L(2,28) = 1,607,383,260,609,382,393,152

This first benchmark implementation is clearly in favor of hybrid architecture. The many-core architecture shows a perfect handling of irregularity and heavy computations.

Chapter 5

Communication Wall: Graph500

5.1 Introduction

In order to face the communication wall in our metric for accelerators, we choose the Graph500 problem. This benchmark is based on huge tree traversal through a randomly generated graph. We already work on this kind of example with the first implementation of the Langford algorithm. The Graph500 push this example to its limits using terabyte to petabytes of data to represent the graph on which the traversal is executed. In this case the computation is absent or just reduce to tests on datum. This metric targets directly the memory and communication wall in irregular case.

In this part we present the benchmark itself, from the graph generation to the traversal. We focus on the BFS part, we do not work on the latest Graph500 using SSSP. We explain why this benchmark is perfect to target communication wall and memory wall. We present our implementation on GPUs and the results compared to the multi-core processor version.

5.1.1 Breadth First Search

The most commonly used search algorithms for graphs are Breadth First Search (BFS) and Depth First Search (DFS). Many graph analysis methods, such as the finding of shortest path for unweighted graphs and centrality, are based on BFS.

As it is a standard approach method in graph theory, its implementation and optimization require extensive work. This algorithm can be seen as frontier expansion and exploration. At each step the frontier is expanded with the unvisited neighbors.

The sequential and basic algorithm is well known and is presented on Algorithm 1.

This algorithm is presented on figure 5.1. At each step from the current queue we search through the neighbors of nodes. All the new nodes, not yet traversed, are added in the queue

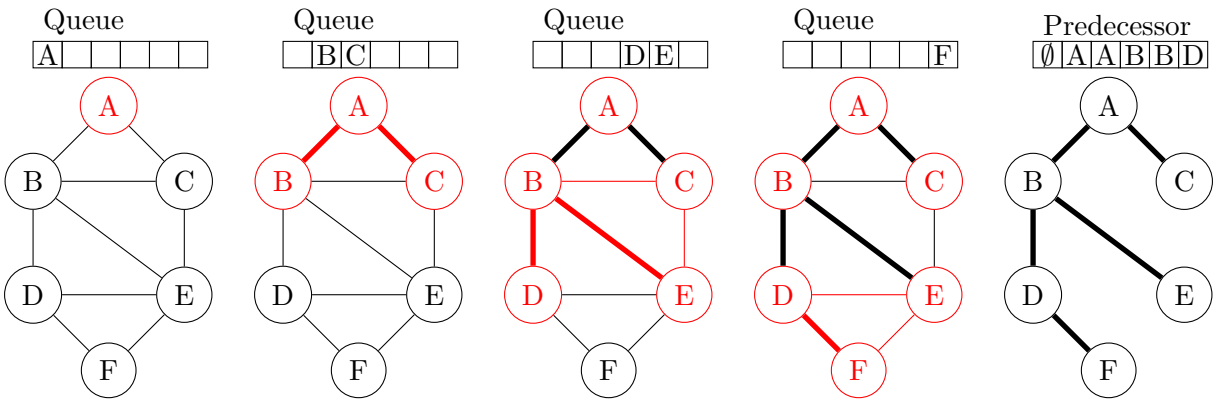


Figure 5.1: Example of Breadth First Search in an undirected graph

Algorithm 1 Sequential Breadth First Search algorithm

```
1: function COMPUTE_BFS(G = (V, E): graph representation, v_s: source vertex, In: current
   level input, Out: current level output, Vis: already visited vertices)
2:     In ← {v_s};
3:     Vis ← {v_s};
4:     P(v) ←⊥ ∀v ∈ V;
5:     while In ≠ ∅ do
6:         Out ← ∅
7:         for u ∈ In do
8:             for v|(u, v) ∈ E do
9:                 if v ∉ Vis then
10:                    Out ← Out ∪ {v};
11:                    Vis ← Vis ∪ {v};
12:                    P(v) ← u;
13:                end if
14:            end for
15:        end for
16:        In ← Out
17:    end while
18: end function
```

for the next step. We keep the information of neighbors exploration in order to recreate the predecessor array.

This algorithm is very famous thanks to its use in many applications but also thanks to the world supercomputer ranking called Graph500[1]. This benchmark is designed to measure the performance on very irregular problems like BFS on a large scale randomized generated graph. The first Graph500 list was released in November 2010. The last list, issued in November 2017, is composed of 235 machines ranked using a specific metric: Traversed Edges Per Second, denoted as TEPS. The aim is to perform a succession of 64 BFS on a large scale graph in the fastest possible way. Then the ratio of edges traversed per the time of computation is used to rank the machines.

This benchmark is more representative of communication and memory accesses than computation itself. Other benchmarks can be used to rank computational power such as LINPACK for the TOP500 list. Indeed the best supercomputers (K-Computer, Sequoia, Mira, ...) on the ladder have a very specific communication topology and sufficient memory, and are large enough to quickly visit all the nodes of the graph.

This benchmark is conduct using GPUs as accelerators. There are many CPU algorithms available, which are listed on the Graph500 website. In order to rank the ROMEO supercomputer we had to create a dedicated version of the Graph500 benchmark in order to fit the supercomputer architecture. As this supercomputer is accelerated by GPUs, three successive approaches had to be applied: first create an optimized CPU algorithm; second provide a GPU specific version and third take advantage of both CPU and GPU computation power.

The first section performs a survey of graph representation and analysis; it also describes some specific implementations. The second section describes the Graph500 protocol and focuses on the Kronecker graph generation method and the BFS validation. The third section presents the chosen methods to implement graph representation and work distribution over the supercomputer nodes. It particularly focuses on the interest of a hybrid CSR and CSC representation. We compare the results for different graph scales and load distributions.

[1] `http://www.graph500.org`

5.2 Existing methods

The most efficient algorithm to compute BFS traversal is used and detailed in [CPW+12]. It uses a 2D partition of the graph which will be detailed later. This algorithm is used on the BlueGene/P and BlueGene/Q architectures but can be easily adapted to any parallel cluster.

We use another key study in order to build our Graph500 CPU/GPU implementation. [MGG15] proposes various effective methods on GPU for BFS. Merrill & al. explaine and teste a few efficient methods to optimize memory access and work sharing between threads on a large set of graphs. It focuses on Kronecker graphs in particular. First they propose several methods for neighbor-gathering with a serial code versus a warp-based and a CTA-based approach. They also use hybridization of these methods to reach the performance level. In a second part they describe the way to perform label-lookup, to check if a vertex is already visited or not. They propose to use a bitmap representation of the graph with texture memory on the GPU for fast random accesses. In the last phase, they propose methods to suppress duplicated vertices generated during the neighbor exploration phase. Then based on these operations they propose *expand-contract*, *contract-expand*, *two-phase* and finally *hybrid* algorithms to adapt the method with all the studied graph classes. The last part they propose a multi-GPU implementation. They use a 1D partition of the graph and each GPU works on its subset of vertices and edges.

In [FDB+14], a first work is proposed to implement a multi-GPU cluster version of the Graph500 benchmark. The scheme used in their approach is quite similar to the one in our study but with a more powerful communication network, namely FDR InfiniBand.

In our work we focus on the GPUDirect usage on the ROMEO supercomputer.

5.3 Environment

As previously mentioned, a CPU implementation is available on the official Graph500 website. A large range of software technology is covered with MPI, OpenMP, etc. All these versions use the same generator and the same validation pattern which is described in this part below.

The Graph500 benchmark is based on the following stages:

- *Graph generation.* The first step is to generate the Kronecker graph and mix the edges and vertices. The graph size is chosen by the user (represented as a based-2 number of vertices). The $EDGEFACTOR$, average ratio of edges by vertex, is always 16. Self-loop and multiple edges are possible with Kronecker graphs. Then 64 vertices for the BFS are randomly chosen. The only rule is that a chosen vertex must have at least one link with another vertex in the graph. *This stage is not timed*;

- *Structure generation.* The specific code part begins here. Based on the edge list and its structure the user is free to distribute the graph over the machines. In a following section we describe our choices for the graph representation. *This stage is timed*;

- *BFS iterations.* This is the key part of the ranking. Based on the graph representation, the user implements a specific optimized BFS. Starting with a root vertex the aim is to build the correct BFS tree (up to a race condition at every level), storing the result in a predecessor list for each vertex;

- *BFS verification.* The user-computed BFS is validated. The number of traversed edges is determined during this stage.

The process is fairly simple and sources can be found at `http://www.graph500.org`. The real problem is to find an optimized way to use parallelism at several levels: node distribution, CPU and GPU distribution and then massive parallelism on accelerators.

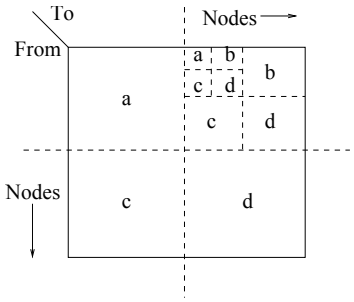


Figure 5.2: Kronecker generation scheme based on edge probability

5.3.1 Generator

The *Kronecker graphs*, based on Kronecker products, represent a specific graph class imposed by the Graph500 benchmark. These graphs represent realistic networks and are very useful in our case due to their irregular aspect [LCK+10]. The main generation method uses the Kronecker matrix product. Based on an initiator adjacency matrix K_1, we can generate a Kronecker graph of order $K_1^{[k]}$ by multiplying K_1 by itself k times. The Graph500 generator uses Stochastic Kronecker graphs, avoiding large scale matrix multiplying, to generate an edge list which is utterly mixed (vertex number and edge position) to avoid locality.

As presented on figure 5.2, the generation is based on edge presence probability on a part of the adjacency matrix. For the Graph500 the probabilities are $a = 0.57$, $b = c = 0.19$ and $d = 0.05$. The generator handle can be stored in a file or directly split in the RAM memory of each process. The first option is not very efficient and imposes a lot of I/O for the generation and verification stage but can be very useful for large scale problems. The second option is faster but uses a part of the RAM thus less ressources are available for the current BFS execution.

5.3.2 Validation

The validation stage is completed after the end of each BFS. The aim is to check if the tree is valid and if the edges are in the original graph. This is why we must keep a copy of the original graph in memory, file or RAM. This validation is based on the following stages, presented on the official Graph500 website. First, the BFS tree is a tree and does not contain cycles. Second, each tree edge connects vertices whose BFS levels differ by exactly one. Third, every edge in the input list has vertices with levels that differ by at most one or that both are not in the BFS tree. Finally, he BFS tree spans an entire connected component's vertices, and a node and its parent are joined by an edge of the original graph.

In order to meet the Graph500 requirements we use the proposed verification function provided in the official code.

5.4 BFS traversal

In this section we present the actual algorithm we used to perform the BFS on a multi-GPU cluster. In a first part we introduce the data structure; then we present the algorithm and the optimizations used.

5.4.1 Data structure

We performed tests of several data structures. In a first work we tried to work with bitmap. Indeed the regularity of computation can fit very well with the GPU architecture. But this representation imposes a significant limitation on the graph size. This representation is used on the BlueGene/Q architecture. Indeed they have some specific hardware bit-wise operations

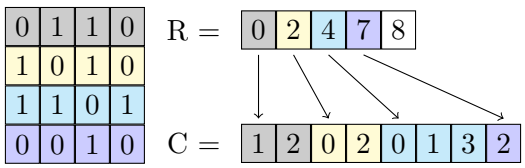


Figure 5.3: Compressed Sparse Row example

implemented in their processors and have a large amount of memory, allowing them to perform very large scale graph analysis.

In a second time we used common graph representations, Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) representation, which fit very well with sparse graphs such as the Graph500 ones.

Figure 5.3 is an example illustratinf the CSR compression. The M adjacency matrix represents the graph. R vector contains the cumulative number of neighbors for each vertex, of size $(\#vertices + 1)$. C, of size $(\#edges)$, is, for each index of R, the edges of a vertex. This representation is very compact and very efficient to work with sparse graphs.

5.4.2 General algorithm

When looking at the latest Graph500 list we see that the best machines are the BlueGene ones. We count about 26 BlueGene/Q and BlueGene/P machines in the first 50 machines. This is due to a quite specific version of the BFS algorithm proposed in [CPW+12]. It proposes a very specific 2D distribution for parallelism and massive use of the 5D torus interconnect.

In the BFS algorithm, like other graph algorithms, parallelism can take several shapes. We can split the vertices into partitions using 1D partition. Each thread/machine can then work on a subset of vertices. The main issue with this method is that the partitions are not equal since the number of edges per vertex can be very different; moreover in graphs like Kronecker ones where some vertices have a very high degree compared to other ones. Thus we are confronted with a major load balancing problem.

In [CPW+12] they propose a new vision of graph traversal, here BFS, on distributed-memory machines. Instead of using standard 1D distribution their BFS is based on a 2D distribution. The adjacency matrix is split into blocks of same number of vertices. If we consider $l \times l$ blocks $A_{i,j}$ we can split the matrix as follows:

$$M = \begin{bmatrix} A_{0,0} & A_{0,1} & \cdots & A_{0,l-1} \\ A_{1,0} & A_{1,1} & \cdots & A_{1,l-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{l-1,0} & A_{l-1,1} & \cdots & A_{l-1,l-1} \end{bmatrix}$$

Each bloc $A_{x,y}$ is a subset of edges. We notice that blocks $A_{0,l-1}$ and $A_{l-1,0}$ have the same edges but in a reverse direction for undirected graphs. Based on this distribution they use *virtual processors*, which are either machines or nodes, each associated with a block. This has several advantages. First we reduce the load balancing overhead and a communication pattern can be set up. Indeed each column shares the same *in_queue* and each row will generate an *out_queue* in the same range. Thus for all the exploration stages, communications are only on line and we just need a column communication phase to exchange the queues for the next BFS iteration. Algorithm 2 presents the BlueGene/Q and BlueGene/P parallel BFS.

This algorithm is presented in algorithm 3 and is based on the exploration phase, denoted by *ExploreFrontier()*. It performs the exploration phase independently on all the machines. Then several communication phases follow. The first two phases are performed on the same processes line. The last one is performed on a processes column.

Algorithm 3 Algorithm for tree traversal presented for BlueGene

1: $Vis_{i,j} \leftarrow In_{i,j}$
2: $P(N_{i,j}, v) \leftarrow \perp$ **for all** $v \in R^{1D}_{i,j}$
3: **if** $v_s \in R^{1D}_{i,j}$ **then**
4: $\quad P(N_{i,j}, v_s) \leftarrow v_s$
5: **end if**
6: **while** *true* **do**
7: $\quad (Out_{i,j}, Marks_{i,j} \leftarrow$ ExploreFrontier();
8: $\quad done \leftarrow \bigwedge_{0 \le k,l \le n} (Out_{k,l} = \emptyset)$
9: $\quad$ **if** *done* **then**
10: $\quad\quad$ **exit loop**
11: $\quad$ **end if**
12: $\quad$ **if** $j = 0$ **then**
13: $\quad\quad prefix_{i,j} = \emptyset$
14: $\quad$ **else**
15: $\quad\quad$ **receive** $prefix_{i,j}$ from $N_{i,j-1}$
16: $\quad$ **end if**
17: $\quad assigned_{i,j} \leftarrow Out_{i,j} \setminus prefix_{i,j}$
18: $\quad$ **if** $j \neq n - 1$ **then**
19: $\quad\quad$ **send** $prefix_{i,j} \cup Out_{i,j}$ **to** $N_{i,j+1}$
20: $\quad$ **end if**
21: $\quad Out_{i,j} \leftarrow \bigcup_{0 \le k \le n} Out_{i,k}$
22: $\quad$ WritePred()
23: $\quad Vis_{i,j} \leftarrow Vis_{i,j} \cup Out_{i,j}$
24: $\quad In_{i,j} \leftarrow Out_{j,i}$
25: **end while**

Iteration	Top-down	Bottom-up	Hybridization
0	27	22 090 111	27
1	8 156	1 568 798	8 156
2	3 695 684	587 893	587 893
3	19 565 465	12 586	12 586
4	214 578	8 256	8 256
5	5 865	1 201	1 201
6	12	156	12
Total	23,489,787	24,269,001	618,131

Table 5.1: BFS directions change from top-down to bottom-up

- On line 15, an exclusive scan is performed for each process on the same line, all the $A_{i,x}$ with $i \in [0, l-1]$. This operation allows us to know which vertices have been discovered in this iteration.
- On line 19, a broadcast of the current *out_queue* is sent to the processes on the same line. With this information they would be able to update the predecessor list only if they are the first parent of a vertex.
- On line 24, a global communication on each column is needed to prepare the next iteration. The aim is to replace the previous *in_queue* by the newly computed *out_queue*.

Two functions are not specified: *ExploreFrontier()* converts the *in_queue* into *out_queue* taking account of the previously visited vertices; *WritePred()* aims to generate the BFS tree and therefore store the predecessor list. In this algorithm the predecessor distribution is still in 1D to avoid vertex duplication. This part can be done using RDMA communication to update predecessor value or with traditional MPI all-to-all exchanges. It can be done during each iteration stage or at the end of the BFS but this requires using a part of the memory to store this data.

This algorithm, which is the basis of many implementations, is the main structure of our distribution.

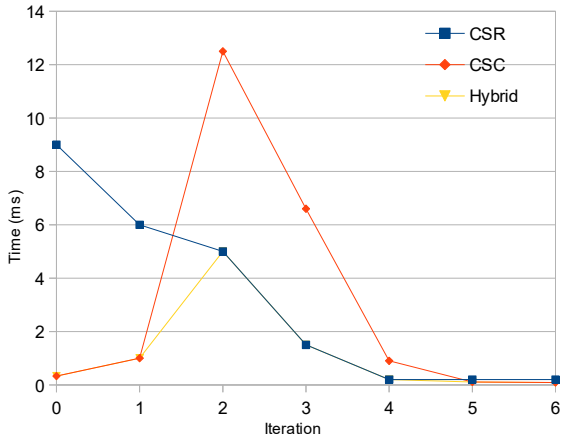
5.4.3 Direction optimization

In order to get an optimized computation in terms of TEPS we decided to sacrifice a small part of the memory for storing both the CSC and CSR representations. Indeed during the different BFS iterations the *in_queue* size varies a lot and, taking this into account, it is wiser to perform exploration from *top-down* or *bottom-up*. So, as proposed in [BAP13], we perform a direction-optimized BFS.

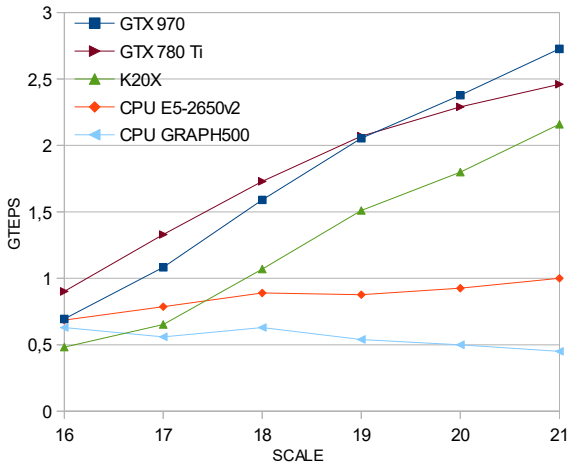
In the first case, *top-down*, we start from the vertices in the *in_queue* and check all the neighbors verifying each time if this neighbor has ever been visited. Then if not, it is added to the *out_queue*. When the *in_queue* is sparse, like for the first and latest iterations, this method is very efficient. In the second case, *bottom-up*, we start the exploration by the not-yet-visited vertices and verify if there is a link between those vertices and the *in_queue* ones. If yes, the not-yet-visited vertex is added to the *out_queue*. Table 5.1 shows the advantages of swithing from top-dowm to bottom-up regarding the number of edges travered. Figure 5.4a presents the two approches, whith the time visiting all the edges, and the benefits of their hybridization.

5.4.4 GPU optimization

In algorithm 2, two parts are not developed. namely *ExploreFrontier()* and *WritePred()*. Indeed these phases are optimized using the GPU. Based on the Merill et al. implementation, the algorithm is optimized to use the shared memory and the texture memory of the GPU. For our



(a) CSR and CSC approach comparison. On a 6 iterations BFS, the time with the two method is compared. The hybridization just takes the best time of each method



(b) Single CPU and accelerators comparison. CPU Graph500 represent the best implementation proposed by the Graph500 website.

version we decided to keep the bitmap implementation for the communications and the queues. So we have to fit the CSR and CSC implementations. On algorithm 4 we present the CSR algorithm; CSC is based on the same approach but starting from the *visited* queue.

In the CSR version each warp is attached to a 32 bit word of the *in_queue* bitmap. Then if this word is empty the whole warp is released; if it contains some vertices, the threads collaborate to load the entire neighbor list. Then they access the coalescent area in the main memory to load the neighbor list. A texture memory is used to accelerate the verification concerning this vertex. Indeed this memory is optimized to be randomly accessed. Then the vertex is added in the bitmap *out_queue*.

5.4.5 Communications

Based on the algorithm 2 communications pattern, we first used MPI with the CPU transferring the data. But the host-device transfer time between the CPU and the GPU was too time-consuming. In order to accelerate the transfers between the GPUs, we used a specific GPU MPI-aware library. This library allows direct MPI operations from the memory of one GPU to another and also implements direct GPU collective operations. GPUDirect can be used coupled with this library. In the last version we used this optimization with GDRCopy.

5.5 Results

5.5.1 CPU and GPU comparison

On figure 5.4b we present the single node implementation. Here we compare the best CPU implementation proposed by the Graph500 benchmark with our GPU implementation. On our cluster we worked with K20Xm GPUs. The GPU result is twice times better than the CPU one. We also carried out tests on some "general public" GPUs like GTX980 and GTX780Ti. The result is better on these GPUs because they do not implement the ECC memory and do not provide double precision CUDA cores. Indeed all the cores can be used for the Exploration phase.

5.5.2 Strong and weak scaling

On figure 5.5b and figure 5.5a we see the result of strong and weak scaling. In the strong scaling we used a $SCALE$ of 21 for different numbers of GPUs. The application scales up to

Algorithm 4 Exploration kernel based on CSR

Constants:
$NWARP$: number of WARPS per block

Variables:
pos_word: position of the word in in_queue
$word$: value of the word in in_queue
$lane\_id$: thread ID in the WARP
$warp\_id$: WARP number if this block
$comm[NWARP][3]$: shared memory array
$shared\_vertex[NWARP]$: vertex in shared memory

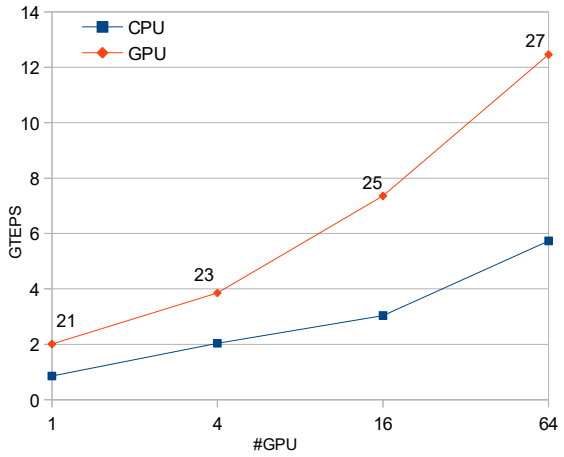
Begin
if $word = 0$ **then**
    free this WARP
end if
if $word$ &1 $<<$ $lane\_id$ **then**
    $id\_sommet \leftarrow= pos\_word * 32 + lane\_id$
    $range[0] \leftarrow C[id\_sommet]$
    $range[1] \leftarrow C[id\_sommet + 1]$
    $range[2] \leftarrow range[1] - range[0]$
end if
while $\_any(range[2])$ **do**
    if $range[2]$ **then**
        $comm[warp\_id][0] \leftarrow lane\_id$
    end if
    if $comm[warp\_id][0] \leftarrow lane\_id$ **then**
        $comm[warp\_id][0] \leftarrow range[0]$
        $comm[warp\_id][0] \leftarrow range[1]$
        $range[2] \leftarrow 0$
        $share\_vertex[warp\_id] = id\_sommet$
    end if
    $r\_gather \leftarrow comm[warp\_id][0] + lane\_id$
    $r\_gather\_end \leftarrow comm[warp\_id][2]$
    while $r\_gather < r\_gather\_end$ **do**
        $voisin \leftarrow R[r\_gather]$
        if $not \in tex\_visited$ **then**
            Adding in $tex\_visited$
            AtomicOr(out_queue,$voisin$)
        end if
        $r\_gather \leftarrow r\_gather + 32$
    end while
end while

16 GPUs but then the data exchanges are too penalizing; performance for 64 GPUs is lower. Indeed as the problem scale does not change, the computational part is reduced compared to the communication one. Using 16 GPUs we were able to perform up to 4.80 GTEPS.

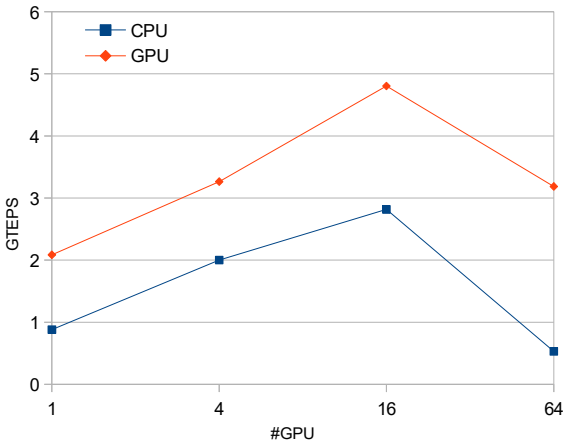
For the weak scaling, the *SCALE* evolves with the number of GPUs. So the computation part grows and the limitation of communications is reduced. On figure 5.5a, the problem SCALE is presented on each point. With our method we were able to reach up to 12 GTEPS using this scaling.

5.5.3 Communications and GPUDirect

Each node of the ROMEO supercomputer is composed of two CPU sockets and two GPUs, named GPU 0 and GPU 1. Yet the node just has one HCA (Host Channel Adapters), linked with CPU 0 and GPU 0. In order to use this link GPU 1 has to pass through a Quick Path Interconnect link (QPI) between the two CPU sockets. This link considerably reduces the bandwidth available for node-to-node communication. Another problem is that the two GPUs have to share the same HCA for their communication.



(a) CPU *vs* GPU weak scaling. The number of CPUs is the same as the number of GPUs.



(b) CPU *vs* GPU strong scaling. The *SCALE* is showed on the GPU line. The number of CPUs is the same as the number of GPUs.

Figure 5.5: Weak and Strong scaling between CPU and GPU

On figure 5.6, the tests are based on the GPU-only implementation. First we worked with the two GPUs of the nodes. We were able to perform up to to *SCALE* 29 with 12 GTEPS. The GPUDirect implementation does not allow the communication with a QPI link. So in order to compare the results, we used only the GPU 0 of each node of the supercomputer. Based on our algorithm implementation we need to use a number 2^{2n} of GPUs. Then the tests on figure 5.6 are for 256 GPUs (with GPU 0 and GPU 1) and with 64 GPUs (using just GPU 0 only). Thus we were able to reach a better value of GTEPS. As the major limitation is the communications stage, using only GPU 0 allowed us to obtain about 13.70 GTEPS on the ROMEO supercomputer.

5.6 Conclusions

The Graph500 is already a benchmark and focus on memory and communication wall in an irregular behavior. We presented an optimized implementation of the Graph500 benchmark for the ROMEO multi-GPU cluster for the purpose of our metric to emphasis behavior of hybrid architectures. It is based on the BlueGene/Q algorithm and GPU optimization for BFS traversal by Merrill et al. This work highlights different key points. First, we have chosen a hybrid memory

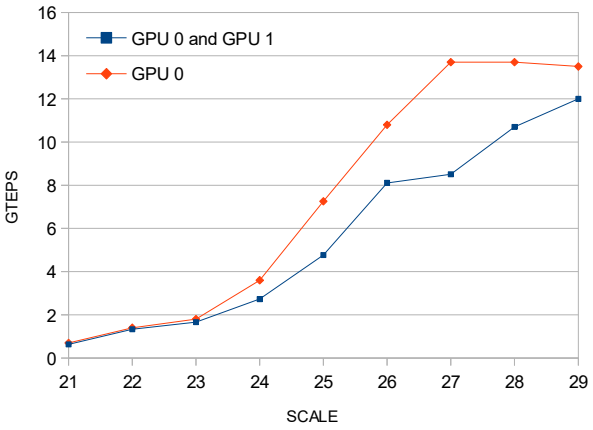


Figure 5.6: Full node GPUs *vs* GPU 0. The GPU 0 implementation not use the QPI link of the two CPU socket.

representation of graphs using both CSR and CSC. Although this representation requires more memory, it significantly reduces the computation workload and allows us to achieve outstanding performance. Second, the inter-node and intra-node communication is a critical bottleneck. Each compute node has two GPUs, however only one shares the same PCIe bridge with the Infiniband HCA that allows to take advantage of the GPUDirect technology. Third, due to the low compute power needed for BFS traversal, we get better performance by fully loading GPUs. Otherwise communication time cannot be overlapped with computation time. Thus to achieve the best performance we had to use only half of each node. Finally, using all these optimizations, we achieved satisfactory results. Indeed, by using GPUDirect on 64 GPUs, we are able to achieve 13,70 GTEPS. In this configuration CPUs are only used to synchronize GPUs kernels. All the communications are directly GPU to GPU using a CUDA-aware MPI library and GPUDirect.

These results have been published in the Graph500 list of November 2016 list. The ROMEO supercomputer was ranked 105th.

In addition to the previous benchmarks proposed in our thesis, the Graph500 complete the communication and memory irregular behavior. It shows the benefit of using many-core architectures in this context allowing twice better results using accelerators than CPUs.

Conclusion

In this part we show the real advantage of accelerators toward classical processor utilization. In the two exemples presented in this chapter we confronted the GPU to worst behaviors for many-core architectures: irregular applications. In the first one, with the problem of Langford, we worked on an irregular and computationally heavy application that does not requires much communications. The second one, the Graph500 BFS, was based on irregular communication over irregular memory usage. IN both cases we showed better results using GPU compared to classical multi-core processors.

5.6.1 Computational wall

We studies the Langford problem under two methods of resolution. In the first one, the Miller algorithm, the resolution was based on a tree traversal. We showed that even using the irregular algorithm directly on the GPU gave us way better results than on multi-core processors. We showed an acceleration of quad times using the GPUs with the backtrack method.

For the second method, the Godfrey algorithm, we presented it to show how we used the GPU in order to beat a new speed record for the last instances $L(2, 27)$ and $L(2, 28)$. We used the whole ROMEO supercomputer and were able to recompute them in 23 using best-effort on the cluster, a mean of 38% of the machine. This result can be theoretically reduce to 10 days by using the whole cluster in the same time: using a linear scaling which is not accurate because augmenting the number of node does not impact communications and will just allow the code to have less registers used.

5.6.2 Communication wall

Several aspect of the Graph500 BFS makes it a good benchmark. The graph generated is completely random and we cannot know in advance the exact number of edges and thus the perfect behavior for distribute the data. During the search of the BFS algorithm the memory is completely traversed with an irregular behavior due to the random generation of the graph.

In order to get performances we imposed regularization over our data. The Compressed Sparse Row and Compressed Sparse Column compression methods were used and the communications were based on bitmap transfers.

We showed that despite of the irregularity downside we were able to provide an efficient GPU algorithm, faster that the CPU algorithm and the reference code from Graph500 itself. We provided an acceleration of two times using our CPU algorithm and quad times using our GPU algorithm.

From this first study, on both applications, we shows that the The question that arise is now: what will be the behavior of GPUs confronted to both of those aspect ? In order to answer this question we present in the next part the Smoothed Particle Hydrodynamics problem on which we base the last part of this study. We show that GPUs can also be use in this context, targeting domain scientists codes.

Part III

Application

Introduction

The first part presented the tools needed to understand and target performance in HPC. The second part presented a metric showing the benefit of accelerators, in this case GPUs, over classical GPU in two contexts: irregular computation and irregular communication/memory behaviors. We showed that the accelerators gave a real advantage on those two problems and allows us to push the limits.

In order to conclude this study and our metric we decided to target another irregular behavior problem embedding both computation/communication/memory wall behavior. In order to show how accelerators handle real world problems, we searched for an application fulfilling our needs. Our choose fell on the Smoothed Particle Hydrodynamics problem applied to fluid and astrophysics. The first version of this application was developed during a 2 months internship at the Los Alamos National Laboratory from the FleCSI framework.

In this part we first present the Smoothed Particle Hydrodynamics method from a physical point of view and making a parallel with the computer science problems involved. The second chapter present our application, FleCSPH. Starting from the current FleCSI version we introduce the algorithm and methods to solve efficiently this problem on classical processor and the acceleration generated adding GPUs.

Chapter 6

General problem

6.1 Introduction

In this section we give details on our choices for the generic application confronted to both computation and communication walls in irregular context. This problem, the Smoothed Particle Hydrodynamics method implementation, is described on a physics aspect. We point out the difficulties involved in the resolution on supercomputers and more especially accelerators. A lot of work have been done on the comprehension of the physical aspect to produce a code that meet the behavior of real physics problems.

The first section is a presentation of the SPH method itself and the overall limitation we have to face. Then we describe different kind of problems we use as a benchmark of the application itself.

6.2 Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics (SPH) is an explicit numerical mesh-free Lagrangian method. It is used to solve hydrodynamical partial differential equations (PDEs) by discretized them into a set of fluid elements called particles. This computational method was invented for the purpose of astrophysics simulations by Monaghan, Gingold and Lucy in 1977 [Luc77, GM77]. This first SPH work conserved mass and they later proposed a method which also conserves linear and angular moment [GM82]. The method was extended for general fluid simulation and many more fields from ballistics to oceanography. The development of new reliable, parallel and distributed tools for this method is a challenge for future HPC architectures with the upcoming Exascale systems.

6.2.1 General description

The method, as illustrated in figure 6.1, computes the evolution of physical quantities for every particle regarding its neighbors in the radius of its smoothing length h. The particles in this radius are then valued according to their distance using a smoothing function W, also called a kernel. The fundamental SPH formulation for any physical quantity A is then to compute with all the neighbors of b of a a particle by:

$$A(\vec{r})_a \simeq \sum_b \frac{m_b}{\rho_b} A(\vec{r_b}) W(|\vec{r} - \vec{r_b}|, h) \tag{6.1}$$

On a physics aspect, this method has several advantages: it can handle deformations, low densities, vacuum, and makes particle tracking easier. It also conserves mass, linear and angular momenta, and energy by its construction that implies independence of the numerical resolution. Another strong benefit of using SPH is its exact advection of fluid properties. Furthermore, the particle structure of SPH easily combines with tree methods for solving Newtonian gravity

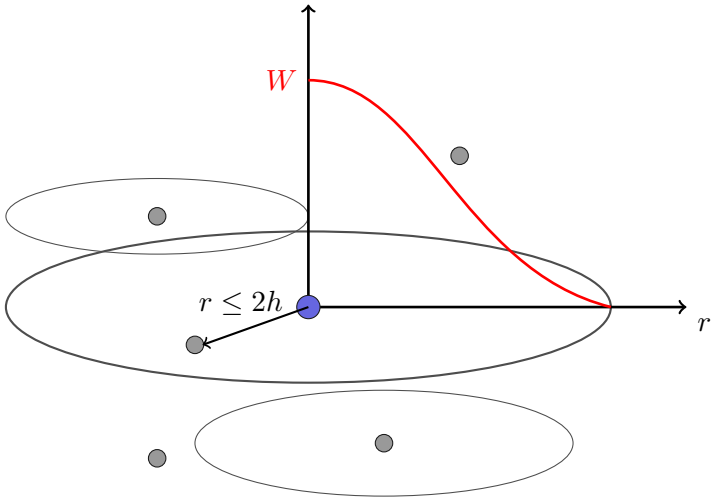


Figure 6.1: SPH kernel W and smoothing length h representation

through N-body simulations. As a mesh-free method, it avoids the need of grid to calculate the spatial derivatives.

However, there are cons to consider using SPH: it cannot be extend to all PDE formulations; it requires careful setup of initial distribution of particles; further, it can be struggle to resolve turbulence-dominated flows and special care must be taken when handling high gradients such as shocks and surface structure of neutron stars. Many works are leading to handle more cases and to push the limitations of this method [DRZR17, LSR16, RDZR16].

In this work, we are solving Lagrangian conservation equations (Euler equations) for density, energy and momentum of an ideal fluid [LL59] such that:

$$\frac{d\rho}{dt} = -\rho(\nabla \cdot \vec{v}), \quad \frac{du}{dt} = -\frac{P}{\rho}(\nabla \cdot \vec{v}), \quad \frac{d\vec{v}}{dt} = -\frac{1}{\rho}(\nabla P) \tag{6.2}$$

with ρ the density, P the pressure, u the internal energy and v the velocity, where $d/dt = \partial/\partial_t + \vec{v} \cdot \nabla$ which is convective derivative.

By using the volume element $V_b = m_b/\rho_b$, we can formulate the Newtonian SPH scheme [Ros09] such that

$$\rho_a = \sum_b m_b W_{ab}(h_a) \tag{6.3}$$

$$\frac{du_a}{dt} = \frac{P_a}{\rho_a^2} \sum_b m_b \vec{v}_{ab} \cdot \nabla_a W_{ab} \tag{6.4}$$

$$\frac{d\vec{v}_a}{dt} = -\sum_b m_b \left(\frac{P_a}{\rho_a^2} + \frac{P_b}{\rho_b^2} \right) \nabla_a W_{ab} \tag{6.5}$$

where $W_{ab} = W(|\vec{r}_a - \vec{r}_b|, h)$ is the smoothing kernel. The equations we would like to solve allow for emergence of discontinuities from smooth initial data. At discontinuities, the entropy increases in shocks. That dissipation occurs inside the shock-front. The SPH formulation here is inviscid so we need to handle this dissipation near shocks. There are a number of way to handle this problem, but the most widespread approach is to add artificial viscosity (or artificial dissipation) terms in SPH formulation such that:

$$\left(\frac{du_a}{dt} \right)_{art} = \frac{1}{2} \sum_b m_b \Pi_{ab} \vec{v}_{ab} \cdot \nabla_a W_{ab} \tag{6.6}$$

$$\left(\frac{d\vec{v}_a}{dt} \right)_{art} = -\sum_b m_b \Pi_{ab} \nabla_a W_{ab} \tag{6.7}$$

In general, we can express the equations for internal energy and acceleration with artificial viscosity

$$\frac{du_a}{dt} = \sum_b m_b \left(\frac{P_a}{\rho_a^2} + \frac{\Pi_{ab}}{2} \right) \vec{v}_{ab} \cdot \nabla_a W_{ab} \tag{6.8}$$

$$\frac{d\vec{v}_a}{dt} = -\sum_b m_b \left(\frac{P_a}{\rho_a^2} + \frac{P_b}{\rho_b^2} + \Pi_{ab} \right) \nabla_a W_{ab} \tag{6.9}$$

Π_{ab} is the artificial viscosity tensor. As long as Π_{ab} is symmetric, the conservation of energy, linear and angular momentum is assured by the form of the equation and antisymmetry of the gradient of kernel with respect to the exchange of indices a and b. Π_{ab} may define in different ways and here we use [MG83] such as:

$$\Pi_{ab} = \begin{cases} \frac{-\alpha \bar{c}_{ab} \mu_{ab} + \beta \mu_{ab}^2}{\bar{\rho}_{ab}} & \text{for } \vec{r}_{ab} \cdot \vec{v}_{ab} < 0 \\ 0 & \text{otherwise} \end{cases} \tag{6.10}$$

$$\mu_{ab} = \frac{\bar{h}_{ab} \vec{r}_{ab} \cdot \vec{v}_{ab}}{r_{ab}^2 + \epsilon \bar{h}_{ab}^2} \tag{6.11}$$

Using the usual form c_s as $c_s = \sqrt{\frac{\partial p}{\partial \rho}}$. The values of ϵ, α, and β have to be set regarding the problem targeted. As an example we used for the Sod shock tube problem: $\epsilon = 0.01h^2$, $\alpha = 1.0$, and $\beta = 2.0$.

There are many possibilities for the smoothing function, called the kernel. As an example the Monaghan's cubic spline kernel is given by:

$$W(\vec{r_{ij}}, h) = \frac{\sigma}{h^D} \begin{cases} 1 - \frac{3}{2}q^2 + \frac{3}{4}q^3 & \text{if} \quad 0 \leq q \leq 1 \\ \frac{1}{4}(2-q)^3 & \text{if} \quad 1 \leq q \leq 2 \\ 0 & \text{otherwise} \end{cases} \tag{6.12}$$

where $q = r/h$, r the distance between the two particles, D is the number of dimensions and σ is a normalization constant with the values:

$$\sigma = \begin{cases} \frac{2}{3} & \text{for 1D} \\ \frac{10}{7\pi} & \text{for 2D} \\ \frac{1}{\pi} & \text{for 3D} \end{cases} \tag{6.13}$$

In the computation of forces we also need to apply the gradient of the smoothing kernel. This is, in our example, for the cubic spline kernel:

$$\vec{\nabla}_i W(\vec{r_{ij}}, h) = \frac{\sigma}{h^{D+1}} \times \begin{cases} (-\frac{3}{h} + \frac{9}{4h}q)\vec{r_{ij}}, & \text{si } 0 \leq \frac{r}{h} < 1 \\ (\frac{-3}{r} + \frac{3}{h} - \frac{3}{4h}q)\vec{r_{ij}}, & \text{si } 1 \leq \frac{r}{h} < 2 \\ 0, & \text{si } \frac{r}{h} \geq 2 \end{cases} \tag{6.14}$$

Figure 6.2 is a representation of the cubic spline kernel with $\sigma = 1$ and $h = 1$. The abscissa axis represent r, distance between the particles and ordinate the value of the smoothing kernel. When the support of the function, 2 is reached the particles are ignored $W = 0$, represented in red on the figure.

To sum up, the SPH resolution scheme and its routines are presented on algorithm 5. The Equation of State (EOS) and the integration are problem dependent and will be define for each test case in section 6.3.

The main downside for the implementation of this method is the requirement for local computation on every particle. The particles have to be grouped locally to perform the computation of (6.3), (6.8) and (6.9). A communication step is needed before and after (6.3) to get the local physical data to be able to compute (6.8) and (6.9). The tree data structure allows us to perform $O(Nlog(N))$ neighbor search but also add a domain decomposition and distribution layer.

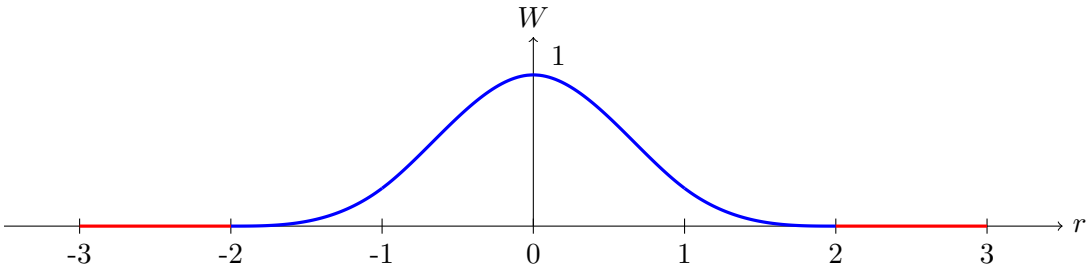


Figure 6.2: Cubic spline kernel example with $\sigma = 1$ and $h = 1$

Algorithm 5 SPH loop algorithm

1: **while** not last step **do**
2: Compute density for each particle (6.3)
3: Compute pressure using EOS
4: Compute acceleration from pressure forces (6.9)
5: Compute change of internal energy for acceleration (6.8)
6: Advance particles after integration
7: **end while**

6.2.2 Gravitation

For classical problems like fluid flow the gravitation can directly be applied on the particles with the force:

$$\vec{a_g} = m\vec{g} \tag{6.15}$$

In order to consider astrophysics problems we need to introduce self-gravitation and gravitation. Each particle imply an action on the others base on its distance and mass. The equation of gravitation for a particle i with j other particles is:

$$\vec{f}_{ai} = \sum_j -G\frac{m_i m_j}{|\vec{r_i} - \vec{r_j}|^3}\vec{r_{ij}} \tag{6.16}$$

This computation involve an $O(N^2)$ complexity and thus is not applicable directly. We applied the method called Fast Multipole Method, FMM and discussed in [BG97]. This method is perfectly adapted to our tree representation of the domain and particles.

This method aim to compute the gravitation up to an approximation determined by the user.

This method is based on Taylor series. The gravitation function of equation 6.16 can be approximate on a particle at position $\vec{r}$ by the gravitation computed at the centroid at position $\vec{r_c}$:

$$\vec{f}(\vec{r}) = \vec{f}(\vec{r_c}) + ||\frac{\partial \vec{f}}{\partial \vec{r}}|| \cdot (\vec{r} - \vec{r_c}) + \frac{1}{2}(\vec{r} - \vec{r_c})^\intercal \cdot ||\frac{\partial \vec{f}}{\partial \vec{r}\partial \vec{r}}|| \cdot (\vec{r} - \vec{r_c}) \tag{6.17}$$

From equation 6.16 we compute the term $||\frac{\partial \vec{f}}{\partial \vec{r}}||$:s

$$\frac{\partial \vec{f}}{\partial \vec{r}} = -\sum_p \frac{m_p}{|\vec{r_c} - \vec{r_p}|^3}\begin{bmatrix} 1 - \frac{3(x_c - x_p)(x_c - x_p)}{|\vec{r_c} - \vec{r_p}|^2} & -\frac{3(y_c - y_p)(x_c - x_p)}{|\vec{r_c} - \vec{r_p}|^2} & -\frac{3(z_c - z_p)(x_c - x_p)}{|\vec{r_c} - \vec{r_p}|^2} \\ -\frac{3(x_c - x_p)(y_c - y_p)}{|\vec{r_c} - \vec{r_p}|^2} & 1 - \frac{3(y_c - y_p)(y_c - y_p)}{|\vec{r_c} - \vec{r_p}|^2} & -\frac{3(z_c - z_p)(y_c - y_p)}{|\vec{r_c} - \vec{r_p}|^2} \\ -\frac{3(x_c - x_p)(z_c - z_p)}{|\vec{r_c} - \vec{r_p}|^2} & -\frac{3(y_c - y_p)(z_c - z_p)}{|\vec{r_c} - \vec{r_p}|^2} & 1 - \frac{3(z_c - z_p)(z_c - z_p)}{|\vec{r_c} - \vec{r_p}|^2} \end{bmatrix} \tag{6.18}$$

And we propose a compact version of the matrix with:

$$||\frac{\partial f^a}{\partial r^b}|| = -\sum_c \frac{m_c}{|\vec{r} - \vec{r_c}|^3}\left[\delta_{ab} - \frac{3.(r^a - r_c^a)(r^b - r_c^b)}{|\vec{r} - \vec{r_c}|^2}\right] \tag{6.19}$$

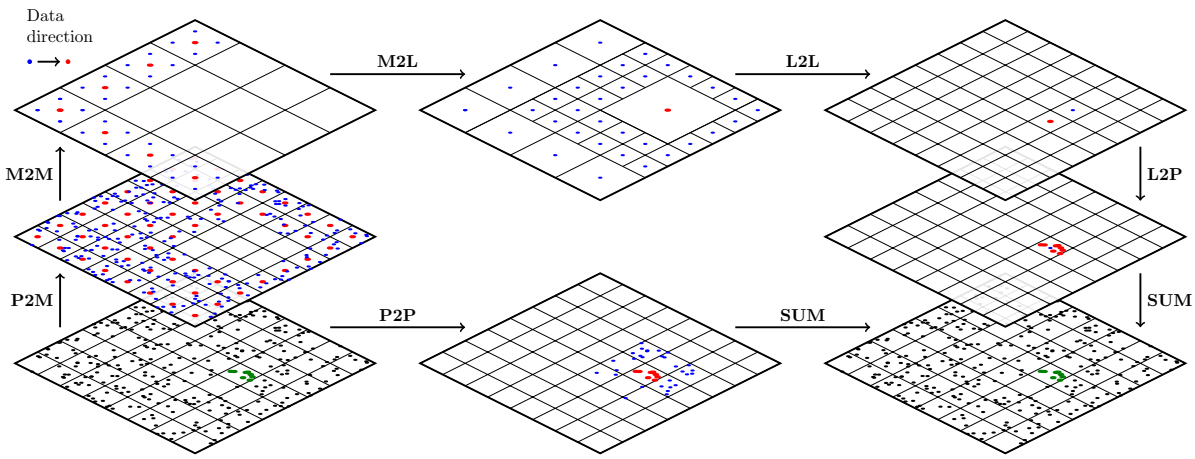


Figure 6.3: Fast Multipole Method schematics. Particles to Multipole (P2M), Multipole to Multipole (M2M), Multipole to Particles (M2P), Multipole to Local (M2L), Local to Local (L2L) and Particles to Particles (P2P). Schematic inspired from [YB11]

With δ_{ab} the Kronecker delta:

$$\delta_{ab} = \begin{cases} 1, & \text{if } a = b. \\ 0, & \text{if } a \neq b. \end{cases} \tag{6.20}$$

We note that a and b variate from 0 to 2 and $r^0 = x$, $r^1 = y$, and $r^2 = z$ as usual sense. For the term $||\frac{\partial \vec{f}}{\partial \vec{r} \partial \vec{r}}||$ we give the compact version by:

$$\begin{aligned} ||\frac{\partial^2 f^a}{\partial r^b \partial r^c}|| = -\sum_c \frac{3m_c}{|\vec{r} - \vec{r_c}|^5} \Big[\frac{5(r^a - r_c^a)(r^b - r_c^b)(r^c - r_c^c)}{|\vec{r} - \vec{r_c}|^2} - \\ \Big(\delta_{ab}(r^c - r_c^c) + \delta_{bc}(r^a - r_c^a) + \delta_{ac}(r^b - r_c^b) \Big) \Big] \end{aligned} \tag{6.21}$$

The method is summed up in figure with the different equations. We consider Centers Of Mass, COM, to be the centroid of particles based on their position. In several steps the information is first transmitted to the COMs, computing their position and mass.

6.3 Applications of SPH

The previous equations are generic and describe the behavior of SPH method. In order to check our

6.3.1 Sod shock tube

The Sod shock tube is the test consisting of a one-dimensional Riemann problem with the following initial parameters [Sod78].

$$(\rho, v, p)_{t=0} = \begin{cases} (1.0, 0.0, 1.0) & \text{if} \quad 0 < x \leq 0.5 \\ (0.125, 0.0, 0.1) & \text{if} \quad 0.5 < x < 1.0 \end{cases} \tag{6.22}$$

In our code, we use the same initial data as in section **??** with ideal gas EOS such as:

$$P(\rho, u) = (\Gamma - 1)\rho u \tag{6.23}$$

where Γ is the adiabatic index of the gas, we set $\Gamma = 5/3$.

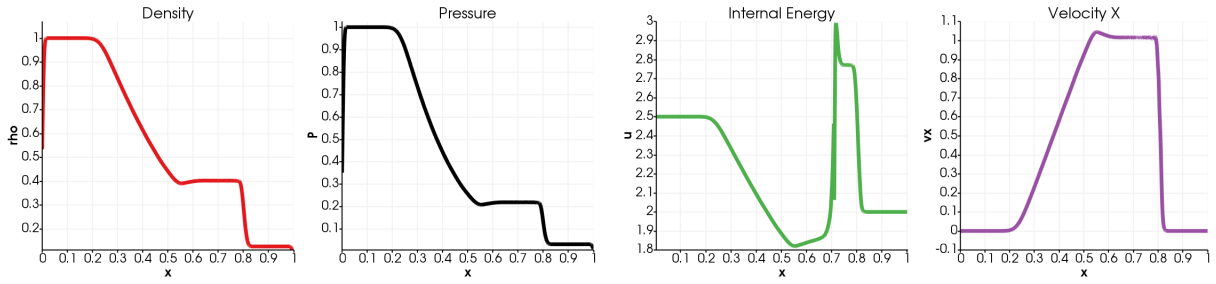


Figure 6.4: Sod shock tube with FleCSPH

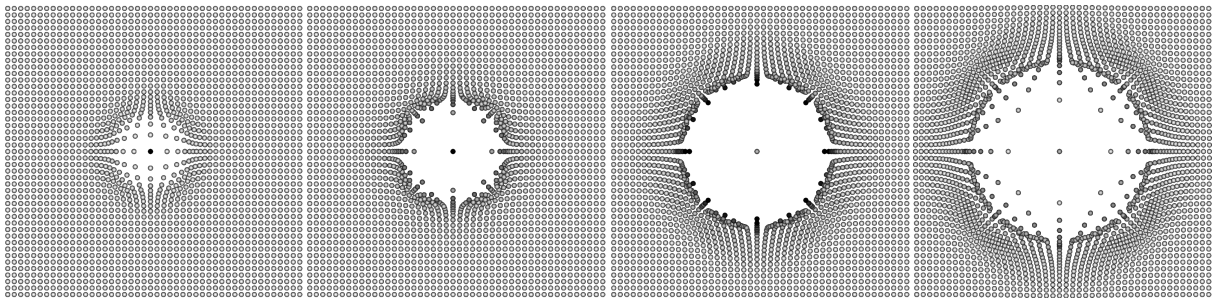


Figure 6.5: Sedov Blast Wave with FleCSPH at respectively $t = 0.01$, $t = 0.03$, $t = 0.06$ and $t = 0.1$

This test is used to check the physical accuracy of the code and thus the tree search itself. A simulation of our Sod shock experimentation is presented on Fig. 6.4 and shows physically correct results.

6.3.2 Sedov blast wave

A blast wave is the pressure and flow resulting from the deposition of a large amount of energy in a small very localized volume. There are different versions of blast wave test and we consider comparing it with the analytic solution for a point explosion as given by Sedov [Sed46], making the assumption that the atmospheric pressure relative to the pressure insider the explosion negligible. Here, we test 2D blast wave. In this simulation, we use ideal gas EOS with $\Gamma = 5/3$ and we are assuming that the undistributed area is at rest with a pressure $P_0 = 1.0^{-5}$. The density is constant ρ_0, also in the pressurized region.

An example of our Sedov Blast wave experimentation is presented on Fig. 6.5 and shows physically correct results.

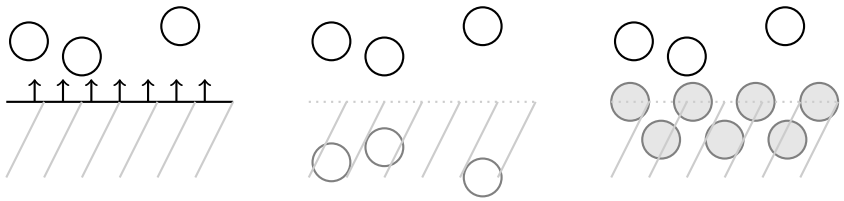
6.3.3 Fluid flow

After performing the tests regarding the physics reliability, we worked on fluid flow problem in 2D and 3D to reach high number of particles. The details can be found in [GGRC+12]. This test is based on an ideal EOS given by:

$$P = B\left[\left(\frac{\rho}{\rho_0}\right)^{\gamma} - 1\right] \tag{6.24}$$

with $\gamma = 7$ and $B = c_0\rho_0/\gamma$ being $\rho_0 = 1000\, kg.m^{-3}$ the reference density.

For this experiment, realistic boundaries conditions were needed. Several methods are possible with SPH we focused on the main ones, the repulsive wall, the mirror particles [LP91] and the dummies particles implementation [AHA12]. Those boundaries conditions implementation are presented in figure 6.6.



Repulsive force Mirror particles Dummy particles

Figure 6.6: Different boundaries condition methods

For the current implementation, we used the dummies particles method. The wall particles are just considered as normal particles, with specific equations, and their quantities are evolved during the run. The main difference is that their position does not evolve at the end of the step. They are identified in the code with a specific type, provided during the data generation.

6.3.4 Astrophysics: neutron stars coalescence

The final aim of our tests is to simulate astrophysical events. We are interested in one of the most important event recently discovered. Last year the Laser Interferometer Gravitational Wave, LIGO, detected the first gravitational wave generated by binary neutron stars merging [AAA+17b] and also more complexes event with Binary Black Holes coalescence in [AAA+17a].

Solving Lane-Emden Equation

We need to determine the density function based on the radius.

As we consider the star as a polytropic fluid, we use the equation of Lane-Emden which is a form of the Poisson equation:

$$\frac{d^2\theta}{d\xi^2} + \frac{2}{\xi}\frac{d\theta}{d\xi} + \theta^n = 0 \tag{6.25}$$

With ξ and θ two dimensionless variables. There is only exact solutions for a polytropic index $n = 0.5$, 1 and 2. In our work we use a polytropic index of 1 which can correspond to a NS simulation.

For $n = 1$ the solution of equation 6.25 is:

$$\theta(\xi) = \frac{sin(\xi)}{\xi} \tag{6.26}$$

We note $\xi_1 = \pi$, the first value of ξ as $\theta(\xi) = 0$. $\theta(\xi)$ is also defined as:

$$\theta(\xi) = \left(\frac{\rho(\xi)}{\rho_c}\right)^{\frac{1}{n}} = \frac{\rho(\xi)}{\rho_c} \tag{6.27}$$

With ρ_c the internal density of the star and ρ the density at a determined radius. ξ is defined as:

$$\xi = Ar = \sqrt{\frac{4\pi G}{K(n+1)}\rho_c^{(n-1)/n}} \times r = \sqrt{\frac{2\pi G}{K}} \times r \text{ (for } n = 1)$$

With K a proportionality constant.

From the previous equations we can write the stellar radius R as:

$$R = \sqrt{\frac{K(n+1)}{4\pi G}}\rho_c^{(1-n)/2}\xi_1 = \sqrt{\frac{K}{2\pi G}} \times \xi_1 \tag{6.28}$$

(We note that for $n = 1$ the radius does not depend of the central density.)

If, for example, we use dimensionless units as $G = R = M = 1$ (for the other results we use CGS with $G = 6.674 \times 10^{-8} cm^3 g^{-1} s^{-2}$) We can compute K as:

$$K = \frac{R^2 2\pi G}{\xi_1^2} \tag{6.29}$$

	NS_1	NS_2	NS_3	NS_4
Radius (cm)	$R = G = M = 1$	1500000	1400000	960000
K	0.636619	95598.00	83576.48	39156.94

Then we deduce the density function of r as :

$$\rho(\xi) = \frac{sin(A \times r)}{A \times r} \times \rho_c \text{ with } A = \sqrt{\frac{2\pi G}{K}}$$

As we know the total Mass M, the radius R and the gravitational constant G we can compute the central density as:

$$\rho_c = \frac{MA^3}{4\pi(sin(AR) - ARcos(AR))}$$

Then we normalize the results to fit $R = M = G = 1$: $K' = K/(R^2 G)$, $m_i' = m_i/M$, $h_i' = h_i/R$, $\vec{x_i}' = \vec{x_i}/R$

6.4 Conclusion

As the SPH method is used in a large panel of fields from astrophysics to fluid mechanic, there are numerous related works. We can cite a code developed in the LANL, 2HOT [War13] that introduced the Hashed Oct Tree structure used in our implementation. There is also GADGET-2 [Spr05], GIZMO [Hop14] and the most recent publication is GASOLINE [WKQ17] based on PKDGRAV, a specific tree+gravity implementation. Several implementations already implement GPU code and tree construction and traversal, one can cite GOTHIC [MU17], presenting gravitational tree code accelerated using the latest Fermi, Kepler and Maxwell architectures. But a lot of GPU accelerated work still focused on fluid problems and not on astrophysical problems [HKK07, CDB+11]. We also note that these implementations focus on SPH problems and does not provide a general purpose and multi-physics framework like we intent to provide through FleCSPH and FleCSI.

Chapter 7

FleCSPH

7.1 Introduction

The previous part describes the method we want to implement and presents the wall in the smoothed particle hydrodynamics method. We showed that this application meet the communication and computation wall in an irregular behavior context. FleCSPH is a complex application that, beside of being interesting for our purpose, need to be accurate on the physics aspect to be use by the domain scientists from LANL. I also note that this is my first work on developing a production code from scratch.

This section gives details about the FleCSPH framework. We first present FleCSI, the base project in the Los Alamos National Laboratory on which FleCSPH is based. For FleCSPH we give details on the domain decomposition strategy used with Morton Ordering. The tree traversal algorithm choices are then explained. We show the first results on the multi-CPU version of FleCSPH. In the last part we present our multi-GPU implementation of FleCSPH and the advantage using accelerators.

7.2 LANL and FleCSI

FleCSI[1] [BMC16] is a compile-time configurable framework designed to support multi-physics application development. It is developed at the Los Alamos National Laboratory as part of the Los Alamos Ristra project. As such, FleCSI provides a very general set of infrastructure design patterns that can be specialized and extended to suit the needs of a broad variety of solver and data requirements. FleCSI currently supports multi-dimensional mesh topology, geometry, and adjacency information, as well as n-dimensional hashed-tree data structures, graph partitioning interfaces, and dependency closures.

FleCSI introduces a functional programming model with control, execution, and data abstractions that are consistent both with MPI and with state-of-the-art, task-based runtimes such as Legion[BTSA12] and Charm++[KK93]. The abstraction layer insulates developers from the underlying runtime, while allowing support for multiple runtime systems including conventional models like asynchronous MPI.

The intent is to provide developers with a concrete set of user-friendly programming tools that can be used now, while allowing flexibility in choosing runtime implementations and optimization that can be applied to future architectures and runtimes.

FleCSI's control and execution models provide formal nomenclature for describing poorly understood concepts such as kernels and tasks. FleCSI's data model provides a low-buy-in approach that makes it an attractive option for many application projects, as developers are not locked into particular layouts or data structure representations.

FleCSI currently provides a parallel but not distributed implementation of Binary, Quad and

[1] `http://github.com/laristra/flecsi`

Oct-tree topology. This implementation is base on space filling curves domain decomposition, the Morton order. The current version allows the user to specify the code main loop and the data distribution requested. The data distribution feature is not available for the tree data structure needed in our SPH code and we provide it in the FleCSPH implementation. The next step will be to incorporate it directly from FleCSPH to FleCSI as we reach a decent level of performance. As FleCSI is an on-development code the structure may change in the future and we keep track of these updates in FleCSPH.

Based on FleCSI the intent is to provide a binary, quad and oct-tree data structure and the methods to create, search and share information for it. In FleCSPH this will be dedicated, apply and tested on the SPH method. In this part we first present the domain decomposition, based on space filling curves, and the tree data structure. We describe the HDF5 files structure used for the I/O. Then we describe the distributed algorithm for the data structure over the MPI processes.

7.3 FleCSPH implementation

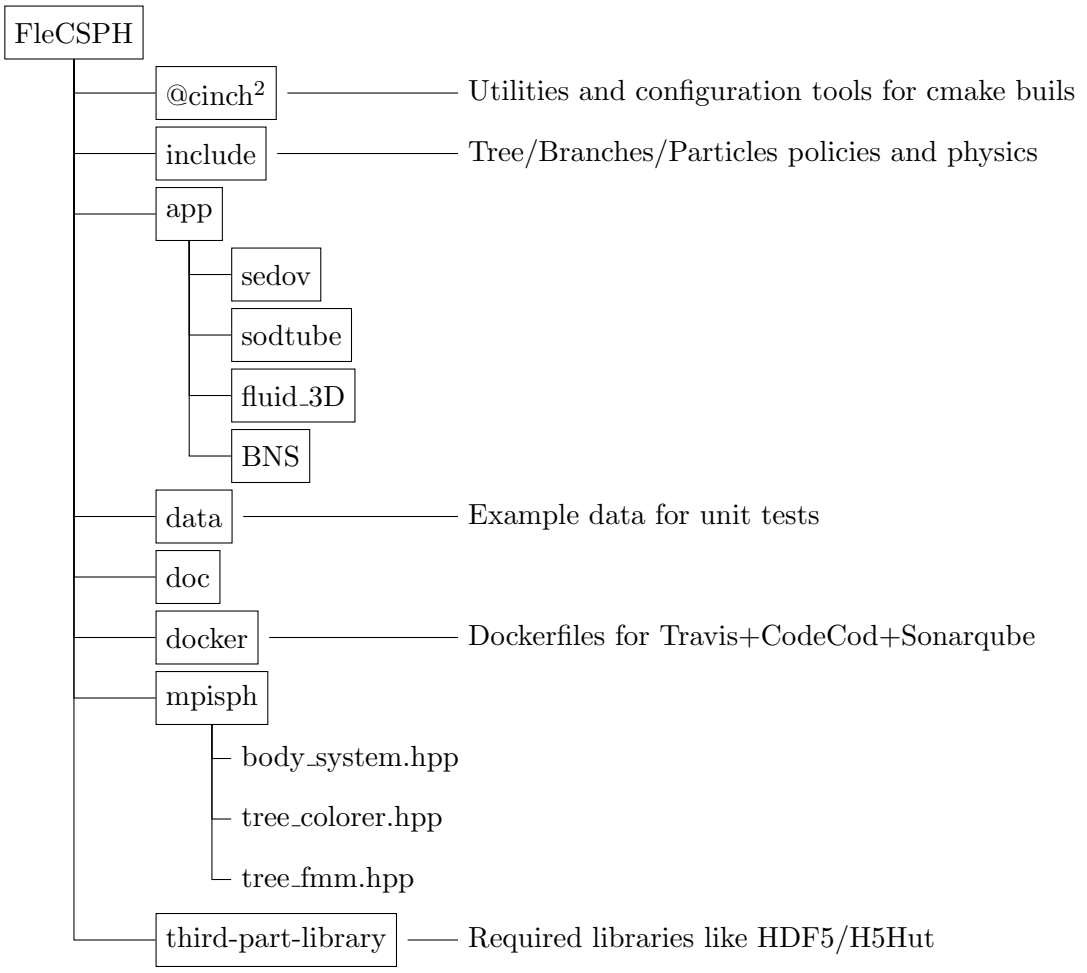


Figure 7.1: FleCSPH structures and files

7.3.1 Domain decomposition

The number of particles can be high and represent a huge amount of data that does not fit in a single node memory. This implies the distribution of the particles over several computational nodes. As the particles moves during the simulation the static distribution is not possible and they have to be redistributed at some point in the execution. Furthermore, this distribution need

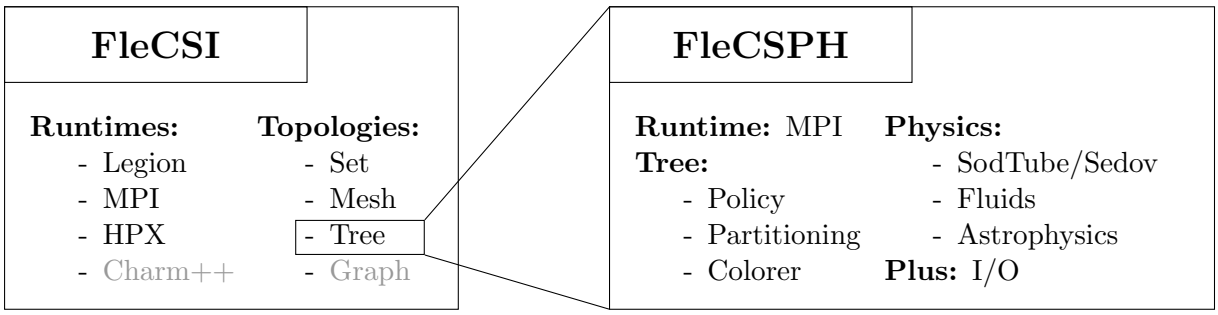


Figure 7.2: FleCSI and FleCSPH frameworks

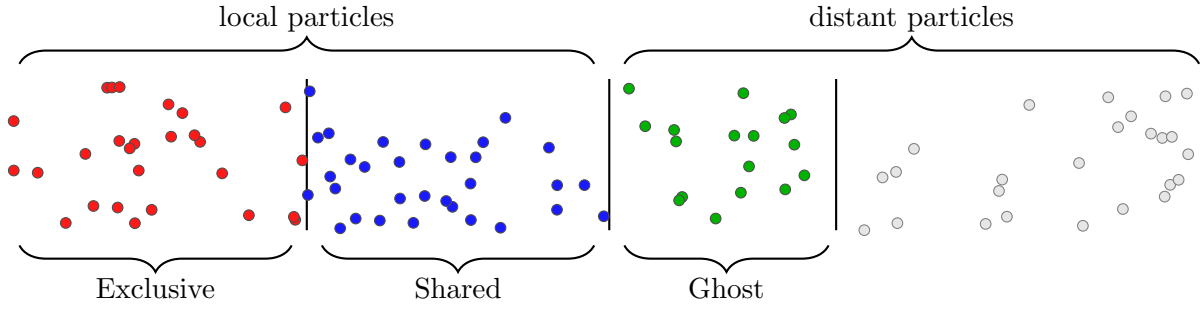


Figure 7.3: Particles "coloring" with local particles: exclusive, shared and distant particles useful during run: ghost particles

to keep local particles in the same computation node to optimize the exchanges and computation itself.

A common approach is to distribute the particles over computational nodes using *space filling curves* like in [War13, Spr05, BGF+14]. It intends to assign to each particle a key which is based on its spacial coordinates, then sorting particles based on those keys keeps particles grouped locally. Many space filling curves exists: Morton, Hilbert, Peano, Moore, Gosper, etc.

This domain decomposition is used in several layers for our implementation. On one hand, to spread the particles over all the MPI processes and provide a decent load balancing regarding the number of particles. On the other hand, it is also used locally to store efficiently the particles and provide a $O(Nlog(N))$ neighbor search complexity, instead of $O(N^2)$, using a tree representation describe in part 7.3.2.

Several space filling curves can fit our purposes:

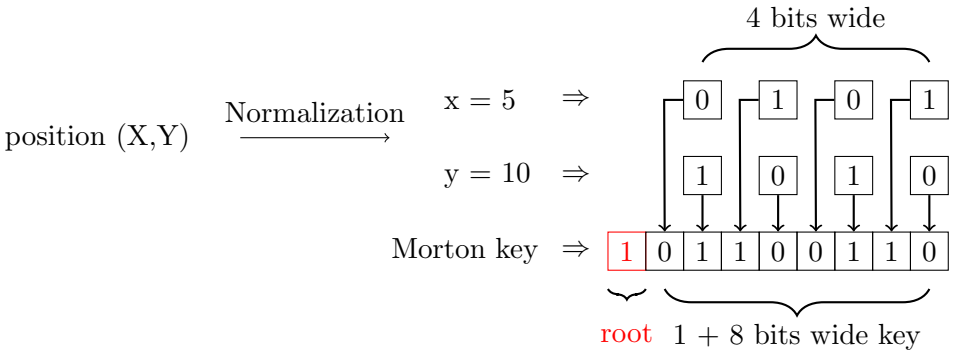


Figure 7.4: Morton order key generation

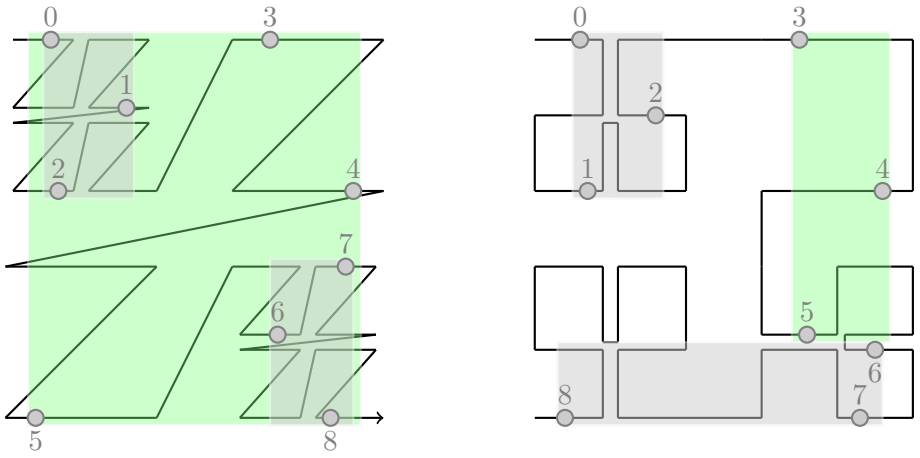


Figure 7.5: Morton and Hilbert space filling curves

The Morton curves

[Mor66], or Z-Order, is the most spread method. This method can produce irregular shape domain decomposition like shown in green on Fig. 7.5. The main advantage is to be fast to compute, the key is made by interlacing directly the X, Y and Z bits without rotations.

The Hilbert curves

[Sag12] are constructed by interlacing bits but also adding rotation based on the Gray code. This work is based on the Peano curves and also called Hilber-Peano. The construction is more complicated than Morton but allows a better distribution.

On Fig. 7.5, the Morton (left) and Hilbert (right) space-filling curves are represented in this example. The particles are distributed over 3 processes. The set of particles of the second process appears in green. As we can see there are discontinuities on the Morton case due to the Z-order "jump" over the space. This can lead to non-local particles and over-sharing of particles that will not be needed during the computation. In the Hilbert curve, the locality over the processes is conserved.

In this first implementation of FleCSPH we used the Morton ordering due to the computational cost. The next step of this work is to compare the computation time of different space filing curves.

Technically the keys are generated for each particle at each iteration because their position is expected to change over time. To be more efficient, the keys can stay the same during several steps and the final comparison can be made on the real particles positions. This increase the search time but allows less tree reconstructions.

We use 64 bits to represent the keys to avoid conflicts. The FleCSI code allows us to use a combination of memory words to reach the desired size of keys (possibly more than 64 bits) but this will cost in memory occupancy. The particle keys are generated by normalizing the space and then converting the floating-point representation to a 64 bits integer for each dimensions. Then the Morton interlacing is done and the keys are created. Unfortunately in some arrangements, like isolated particles, or scenarios with very close particles, the keys can be either badly distributed or duplicate keys can appear. Indeed, if the distance between two particles is less than $2^{-64} \approx 1e{+}20$, in a normalized space, the key generated through the algorithm will be the same. This problem is then handle during the particle sort and then the tree generation. In both case two particles can be differentiate based on their unique ID generated at the beginning execution.

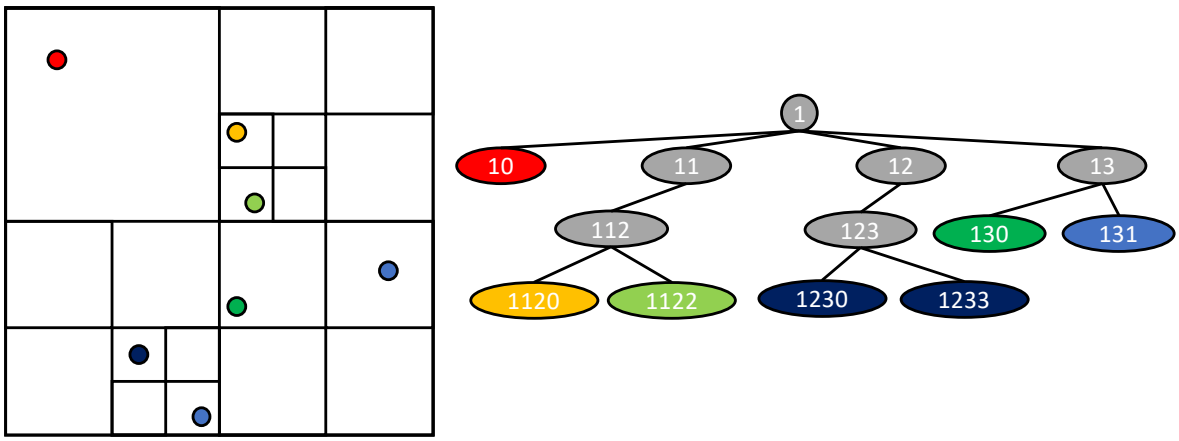


Figure 7.6: Quadtree, space and data representation

7.3.2 Hierarchical trees

The method we use for the tree data structure creation and research comes from Barnes-Hut trees presented in [BH86, Bar90]. By reducing the search complexity from $O(N^2)$ for direct summation to $O(Nlog(N))$ it allows us to do very large simulations with billions of particles. It also allows the use of the tree data structure to compute gravitation using multipole methods.

We consider binary trees, for 1 dimension, quad-trees, for 2 dimensions, and oct-trees, for 3 dimensions. The construction of those trees is directly based on the domain decomposition using keys and space-filling curve presented in section 7.3.1.

As explain in the previous section, we use 64 bits keys. That give us up to 63, 31 and 21 levels in the tree for respectively 1, 2 and 3 dimensions. As presented on Fig. 7.6 the first bit is use to represent the root of the tree, 1. This allows us to have up to 2^{63} different keys and unique particles.

Tree generation

After each particle get distributed on its final process using its space-filling curve key, we can recursively construct the tree. Each particle is added and the branches are created recursively if there is an intersection between keys. Starting from the root of key "1" the branches are added at each levels until the particles are reached. An example of a final tree is shown on Fig. 7.6.

Tree search

When all the particles have been added, the data regarding the tree nodes are computed with a bottom up approach. Summing up the mass, position called Center of Mass (COM), and the boundary box of all sub-particles of this tree node.

For the search algorithm the basic idea would be to do a tree traversal for all the particles and once we reach a particle or a node that interact with the particle smoothing length, add it for computation or in a neighbor list. Beside of being easy to implement and to use in parallel this algorithm requires a full tree traversal for every particle and will not take advantage of the particles' locality.

Our search algorithm, presented on Algorithm 6, is a two step algorithm like in Barnes trees: First create the interaction lists and then using them on the sub-tree particles. In the first step we look down for nodes with a target sub-mass of particles $tmass$. Then for those branches we compute an interaction list and continue the recursive tree search. When a particle is reached, we compute the physics using the interaction list as the neighbors. The interaction list is computing using an opening-angle criterion comparing the boundary box and an user

define angle. In this way we will not need a full tree traversal for each particle but a full tree traversal for every group of particles.

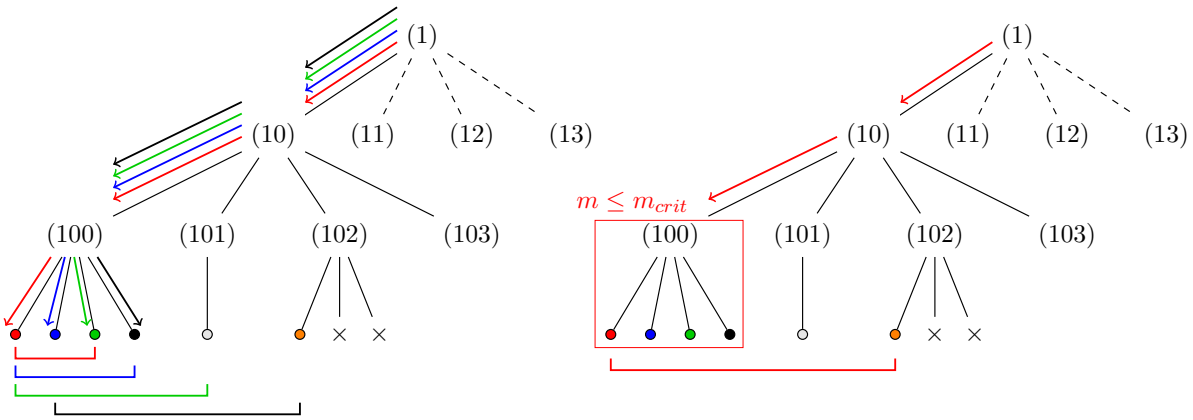


Figure 7.7: Neighbors search using a tree traversal per particle vs a group of particle and computing an interaction list

7.3.3 Distribution strategies

The previous section presented the tree data structure that can be use locally on every node. The distribution layer is added on top of it, keeping each sub-tree on the computation nodes. The current version of FleCSPH is still based on synchronous communications using the Message Passing Interface (MPI).

The main distributed algorithm is presented on algorithm 7:

Particle distribution

The sort step, line 5, is base on a distributed quick sort algorithm. The keys are generated using the Morton order described in part 7.3.1. As we associate a unique number to each particle we are able to sort them using the keys and, in case of collision keys, using their unique ID. This gives us a global order for the particles. Each process sends to a master node (or submaster for larger cases) a sample of its keys. We determined this size to be 256 KB of key data per process for our test cases but it can be refined for larger simulations. Then the master determines the general ordering for all the processes and shares the pivots. Then each process locally sort its local keys and, in a global communication step, the particles are distributed to the process on which they belong. This algorithm gives us a good partition in term of number of particles. But some downside can be identified:

- The ordering may not be balanced in term of number of particles per processes. But by optimizing the number of data exchanged to the master can lead to better affectation.
- The load balance also depend on the number of neighbors of each particle. If a particle get affected a poor area with large space between the particles, this can lead to bad load balancing too.

This is why we also provide another load balancing based on the particles neighbors. Depending on the user problem, the choice can be to distribute the particles on each processes regarding the number of neighbors, having the same amount of physical computation to perform on each processes.

After this first step, the branches are shared between the different processes, line 8. Every of them send to its neighbors several boundaries boxes, defined by the user. Then particles from the neighbors are computed, exchanged and added in the local tree. Those particles are labeled as NON_LOCAL particles. At this point a particle can be referenced as: EXCLUSIVE:

Algorithm 6 Tree search algorithm

```
1: procedure FIND_NODES
2:     stack stk ← root
3:     while not_empty(stk) do
4:         branch b ← stk.pop()
5:         if b is leaf then
6:             for each particles p of b do
7:                 apply_sub_tree(p,interaction_list(p))
8:             end for
9:         else
10:            for each child branch c of b do
11:                stk.push(c)
12:            end for
13:        end if
14:    end while
15: end procedure
16:
17: procedure APPLY_SUB_TREE(node n, node-list nl)
18:    stack stk ← n
19:    while not_empty(stk) do
20:        branch b ← stk.pop()
21:        if b is leaf then
22:            for each particles p of b do
23:                apply_physics(p,nl)
24:            end for
25:        else
26:            for each child branch c of b do
27:                stk.push(c)
28:            end for
29:        end if
30:    end while
31: end procedure
32:
33: function INTERACTION_LIST(node n)
34:    stack stk ← root
35:    node-list nl ← ∅
36:    while not_empty(stk) do
37:        branch b ← stk.pop()
38:        if b is leaf then
39:            for each particles p of b do
40:                if within() then
41:                    nl ← nl + p
42:                end if
43:            end for
44:        else
45:            for each child branch c of b do
46:                if mac(c,angle) then
47:                    nl ← nl + c
48:                else
49:                    stk.push(c)
50:                end if
51:            end for
52:        end if
53:    end while
54: end function
```

Algorithm 7 Main algorithm

```
1:  procedure SPECIALIZATION_DRIVER(input data file f)
2:      Read f in parallel
3:      Set physics constant from f
4:      while iterations do
5:          Distribute the particles using distributed quick sort
6:          Compute total range
7:          Generate the local tree
8:          Share branches
9:          Compute the ghosts particles
10:         Update ghosts data
11:         PHYSICS
12:         Update ghosts data
13:         PHYSICS
14:         Distributed output to file
15:     end while
16: end procedure
```

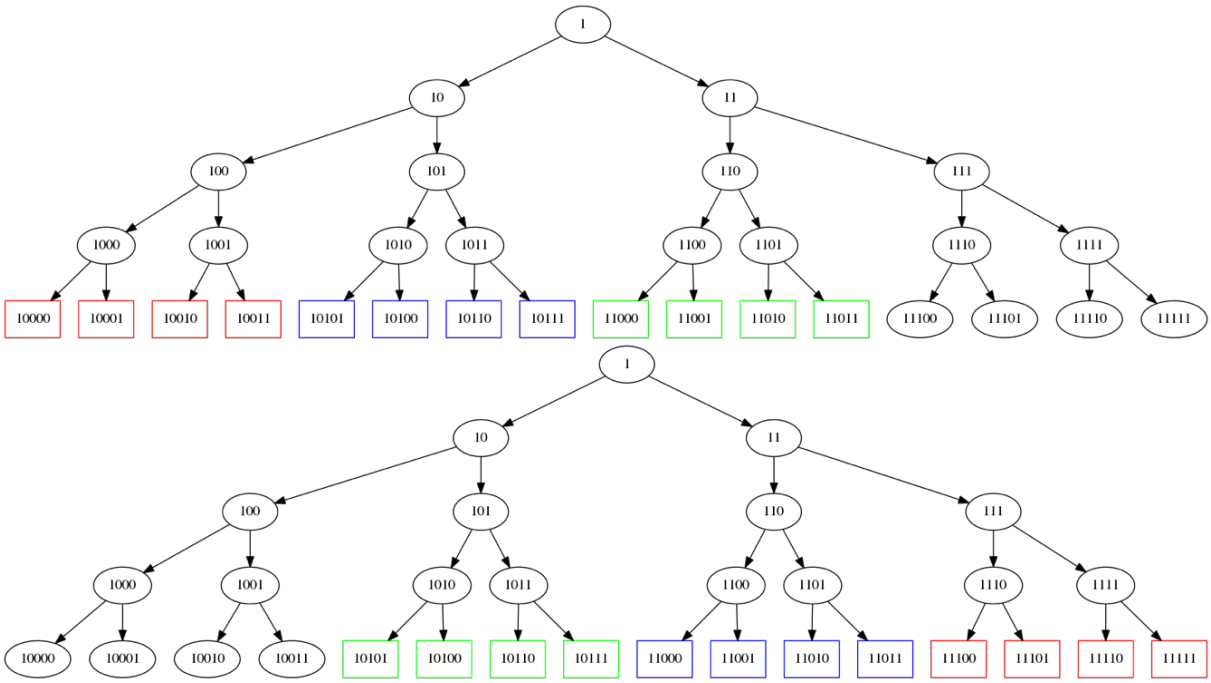


Figure 7.8: Binaries tree for a 2 processes system. Exclusive, Shared and Ghosts particles resp. red, blue, green.

will never be exchanged and will only be used on this process; SHARED: may be needed by another process during the computation; GHOSTS: particles information that the process need to retrieve from another process. An example is given for 2 processes on Fig. 7.8.

Exchange Shared and Ghosts particles

The previous distribution shares the particles and the general information about neighbors particles. Then each process is able to do synchronously or asynchronously communications to gather distant particles. In the current version of FleCSPH an extra step is required to synchronously share data of the particles needed during the next tree traversal and physics part. Then after this step, the ghosts data can be exchanged as wanted several times during the same time step.

7.3.4 I/O

Regarding the high number of particles, an efficient, parallel and distributed I/O implementation is required. Several choices were available but we wanted a solution that can be specific for our usage. The first requirement is to allow the user to work directly with the Paraview visualization tool and splash[3] [Pri07].

We base this first implementation on HDF5 [FCY99] file structure with H5Part and H5Hut [HAB+10]. HDF5 support MPI runtime with distributed read and write in a single or multiple files. We added the library H5hut to add normalization in the code to represent global data, steps, steps data and the particles data for each steps. The I/O code was developed internally at LANL and provides a simple way to write and read the data in H5Part format. The usage of H5Hut to generate H5part data files allows us to directly read the output in Paraview without using a XDMF descriptor like requested in HDF5 format.

7.4 Conclusion

[3] http://users.monash.edu.au/~dprice/splash/

Conclusion

In this part we presented the

Conclusion

Bibliography

[AAA+17a] Benjamin P Abbott, R Abbott, TD Abbott, F Acernese, K Ackley, C Adams, T Adams, P Addesso, RX Adhikari, VB Adya, et al. Gw170814: A three-detector observation of gravitational waves from a binary black hole coalescence. *Physical review letters*, 119(14):141101, 2017.

[AAA+17b] Benjamin P Abbott, Rich Abbott, TD Abbott, Fausto Acernese, Kendall Ackley, Carl Adams, Thomas Adams, Paolo Addesso, RX Adhikari, VB Adya, et al. Gw170817: observation of gravitational waves from a binary neutron star inspiral. *Physical Review Letters*, 119(16):161101, 2017.

[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.

[ABL15] Ali Assarpour, Amotz Barnoy, and Ou Liu. Counting the number of langford skolem pairings. 2015.

[AC14] Alejandro Arbelaez and Philippe Codognet. A gpu implementation of parallel constraint-based local search. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 648–655. IEEE, 2014.

[AHA12] S Adami, XY Hu, and NA Adams. A generalized wall boundary condition for smoothed particle hydrodynamics. *Journal of Computational Physics*, 231(21):7057–7075, 2012.

[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.

[BAP13] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. *Scientific Programming*, 21(3-4):137–148, 2013.

[Bar90] Joshua E Barnes. A modified tree code: don't laugh; it runs. *Journal of Computational Physics*, 87(1):161–170, 1990.

[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.

[BG97] Rick Beatson and Leslie Greengard. A short course on fast multipole methods. *Wavelets, multilevel methods and elliptic PDEs*, 1:1–37, 1997.

[BGF+14] Jeroen Bédorf, Evghenii Gaburov, Michiko S Fujii, Keigo Nitadori, Tomoaki Ishiyama, and Simon Portegies Zwart. 24.77 pflops on a gravitational tree-code to simulate the milky way galaxy with 18600 gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 54–65. IEEE Press, 2014.

[BH86] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. *nature*, 324(6096):446–449, 1986.

[BM06] David A Bader and Kamesh Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *Parallel Processing, 2006. ICPP 2006. International Conference on*, pages 539–550. IEEE, 2006.

[BMC16] Ben Bergen, Nicholas Moss, and Marc Robert Joseph Charest. Flexible computational science infrastructure. Technical report, Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), 2016.

[BP07] Ulrik Brandes and Christian Pich. Centrality estimation in large networks. *International Journal of Bifurcation and Chaos*, 17(07):2303–2318, 2007.

[Bra01] Ulrik Brandes. A faster algorithm for betweenness centrality*. *Journal of mathematical sociology*, 25(2):163–177, 2001.

[BTSA12] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 66. IEEE Computer Society Press, 2012.

[CDB+11] Alejandro C Crespo, Jose M Dominguez, Anxo Barreiro, Moncho Gómez-Gesteira, and Benedict D Rogers. Gpus, a new tool of acceleration in cfd: efficiency and reliability on smoothed particle hydrodynamics methods. *PloS one*, 6(6):e20685, 2011.

[CDK+00] Rohit Chandra, Leo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.

[CDPD+14] Federico Campeotto, Alessandro Dal Palu, Agostino Dovier, Ferdinando Fioretto, and Enrico Pontelli. Exploring the use of gpus in constraint solving. In *Practical Aspects of Declarative Languages*, pages 152–167. Springer, 2014.

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

[CPW+12] F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A. R. Choudhury, and Y. Sabharwal. Breaking the speed and scalability barriers for graph exploration on distributed-memory machines. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–12, Nov 2012.

[DBGO14] Andrew Davidson, Sean Baxter, Michael Garland, and John D Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 349–359. IEEE, 2014.

[Deh01] Walter Dehnen. Towards optimal softening in 3-D n-body codes: I. minimizing the force error. *Mon. Not. Roy. Astron. Soc.*, 324:273, 2001.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 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.

[DRZR17] Zili Dai, Huilong Ren, Xiaoying Zhuang, and Timon Rabczuk. Dual-support smoothed particle hydrodynamics for elastic mechanics. *International Journal of Computational Methods*, 14(04):1750039, 2017.

[DTC+14] H. Djidjev, S. Thulasidasan, G. Chapuis, R. Andonov, and D. Lavenier. Efficient Multi-GPU Computation of All-Pairs Shortest Paths. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 360–369, 2014.

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

[FCY99] Mike Folk, Albert Cheng, and Kim Yates. Hdf5: A file format and i/o library for high performance computing applications. In *Proceedings of Supercomputing*, volume 99, pages 5–33, 1999.

[FDB+14] Zhisong Fu, Harish Kumar Dasari, Bradley Bebee, Martin Berzins, and Bradley Thompson. Parallel breadth first search on gpu clusters. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 110–118. IEEE, 2014.

[FKF03] Olivier Flauzac, Michaël Krajecki, and Jean Fugère. Confiit: a middleware for peer to peer computing. In *International Conference on Computational Science and Its Applications*, pages 69–78. Springer, 2003.

[Fly72a] M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, Sept 1972.

[Fly72b] 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.

[Gar56] Martin Gardner. *Mathematics, magic and mystery*. Dover publication, 1956.

[GGRC+12] Moncho Gomez-Gesteira, Benedict D Rogers, Alejandro JC Crespo, Robert A Dalrymple, Muthukumar Narayanaswamy, and José M Dominguez. Sphysics–development of a free-surface fluid solver–part 1: Theory and formulations. *Computers & Geosciences*, 48:289–299, 2012.

[GJ79] M. R. Garey and D. S. Johnson. *Computer and Intractability*. Freeman, San Francisco, CA, USA, 1979.

[GLG+12] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, 2012.

[GM77] Robert A Gingold and Joseph J Monaghan. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly notices of the royal astronomical society*, 181(3):375–389, 1977.

[GM82] RA Gingold and JJ Monaghan. Kernel estimates as a basis for general particle methods in hydrodynamics. *Journal of Computational Physics*, 46(3):429–453, 1982.

[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.

[GSS08] Robert Geisberger, Peter Sanders, and Dominik Schultes. Better approximation of betweenness centrality. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 90–100. Society for Industrial and Applied Mathematics, 2008.

[GW99] I.P. Gent and T. Walsh. Csplib: a benchmark library for constraints. Technical report, Technical report APES-09-1999, 1999. Available from http://csplib.cs.strath.ac.uk/. A shorter version appears in the Proceedings of the 5th International Conference on Principles and Practices of Constraint Programming (CP-99).

[HAB+10] Mark Howison, Andreas Adelmann, E Wes Bethel, Achim Gsell, Benedikt Oswald, et al. H5hut: A high-performance i/o library for particle-based simulations. In *Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*, pages 1–8. IEEE, 2010.

[Hac86] I. Hachisu. A versatile method for obtaining structures of rapidly rotating stars. *Astrophysical Journal Supplement Series*, 61:479–507, 1986.

[HKK07] Takahiro Harada, Seiichi Koshizuka, and Yoichiro Kawaguchi. Smoothed particle hydrodynamics on gpus. In *Computer Graphics International*, pages 63–70. SBC Petropolis, 2007.

[HKS00] Z. Habbas, M. Krajecki, and D. Singer. Parallel resolution of csp with openmp. In *Proceedings of the second European Workshop on OpenMP*, pages 1–8, Edinburgh, Scotland, 2000.

[HKS02] Z. Habbas, M. Krajecki, and D. Singer. Parallelizing Combinatorial Search in Shared Memory. In *Proceedings of the fourth European Workshop on OpenMP*, Roma, Italy, 2002.

[HNY+09] Tsuyoshi Hamada, Tetsu Narumi, Rio Yokota, Kenji Yasuoka, Keigo Nitadori, and Makoto Taiji. 42 tflops hierarchical n-body simulations on gpus with applications in both astrophysics and turbulence. In *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, pages 1–12. IEEE, 2009.

[Hop14] Philip F Hopkins. Gizmo: Multi-method magneto-hydrodynamics+ gravity code. *Astrophysics Source Code Library*, 2014.

[HS00] H. Hoos and T. Stutzle. Satlib: An online resource for research on sat. In *SAT2000*, pages 283–292, 2000.

[HVN09] Pawan Harish, Vibhav Vineet, and P. J. Narayanan. Large graph algorithms for massively multithreaded architectures. *International Institute of Information Technology Hyderabad, Tech. Rep. IIIT/TR/2009/74*, 2009.

[IABT11] Markus Ihmsen, Nadir Akinci, Markus Becker, and Matthias Teschner. A parallel sph implementation on multi-core cpus. In *Computer Graphics Forum*, volume 30, pages 99–112. Wiley Online Library, 2011.

[IOS+14] Markus Ihmsen, Jens Orthmann, Barbara Solenthaler, Andreas Kolb, and Matthias Teschner. Sph fluids in computer graphics. 2014.

[Jai05] Christophe Jaillet. *In french: Résolution parallèle des problèmes combinatoires.* Phd, Université de Reims Champagne-Ardenne, France, December 2005.

[JAO+11] John Jenkins, Isha Arkatkar, John D Owens, Alok Choudhary, and Nagiza F Samatova. Lessons learned from exploring the backtracking paradigm on the gpu. In *Euro-Par 2011 Parallel Processing*, pages 425–437. Springer, 2011.

[JG03] Morris Jette and Mark Grondona. *SLURM : Simple Linux Utility for Resource Management.* U.S. Departement of Energy, June 23, 2003.

[JHC+10] Shuangshuang Jin, Zhenyu Huang, Yousu Chen, Daniel Chavarría-Miranda, John Feo, and Pak Chung Wong. A novel application of parallel betweenness centrality to power grid contingency analysis. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–7. IEEE, 2010.

[JK04a] Christophe Jaillet and Michaël Krajecki. Solving the langford problem in parallel. In *International Symposium on Parallel and Distributed Computing*, pages 83–90, Cork, Ireland, July 2004. IEEE Computer Society.

[JK04b] Christophe Jaillet and Michaël Krajecki. Solving the langford problem in parallel. In *International Symposium on Parallel and Distributed Computing*, pages 83–90, Cork, Ireland, July 2004. IEEE Computer Society.

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

[JTB] Christopher J Riley High-Performance Java and Gabrielle Keller Transformation-Based. Irregular parallel algorithms.

[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.

[KFM04] M. Krajecki, O. Flauzac, and P.-P. Merel. Focus on the communication scheme in the middleware confiit using xml-rpc. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 160–, April 2004.

[KK93] Laxmikant V Kale and Sanjeev Krishnan. Charm++: a portable concurrent object oriented system based on c++. In *ACM Sigplan Notices*, volume 28, pages 91–108. ACM, 1993.

[Kra99] Michaël Krajecki. An object oriented environment to manage the parallelism of the fiit applications. In *Parallel Computing Technologies*, pages 229–235. Springer, 1999.

[Kre02] Valdis E Krebs. Mapping networks of terrorist cells. *Connections*, 24(3):43–52, 2002.

[KWm12] David B Kirk and W Hwu Wen-mei. *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[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.

[Lar09] J Larsen. Counting the number of skolem sequences using inclusion exclusion. 2009.

[LCK+10] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *The Journal of Machine Learning Research*, 11:985–1042, 2010.

[LGK+14] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[LGS+09] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast bvh construction on gpus. In *Computer Graphics Forum*, volume 28, pages 375–384. Wiley Online Library, 2009.

[LL59] L. D. Landau and E. M. Lifshitz. *Fluid mechanics*. 1959.

[LL10] MB Liu and GR Liu. Smoothed particle hydrodynamics (sph): an overview and recent developments. *Archives of computational methods in engineering*, 17(1):25–76, 2010.

[LLP+12] Min-Joong Lee, Jungmin Lee, Jaimie Yejean Park, Ryan Hyun Choi, and Chin-Wan Chung. Qube: a quick algorithm for updating betweenness centrality. In *Proceedings of the 21st international conference on World Wide Web*, pages 351–360. ACM, 2012.

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

[LP91] Larry D Libersky and AG Petschek. Smooth particle hydrodynamics with strength of materials. In *Advances in the free-Lagrange method including contributions on adaptive gridding and the smooth particle hydrodynamics method*, pages 248–257. Springer, 1991.

[LSR16] SJ Lind, PK Stansby, and Benedict D Rogers. Incompressible–compressible flows with a transient discontinuous interface using smoothed particle hydrodynamics (sph). *Journal of Computational Physics*, 309:129–147, 2016.

[Luc77] Leon B Lucy. A numerical approach to the testing of the fission hypothesis. *The astronomical journal*, 82:1013–1024, 1977.

[MAB+10] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

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

[MB14a] Adam McLaughlin and David A Bader. Revisiting edge and node parallelism for dynamic gpu graph analytics. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 1396–1406. IEEE, 2014.

[MB14b] Adam McLaughlin and David A. Bader. Scalable and high performance betweenness centrality on the GPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 572–583. IEEE Press, 2014.

[MG83] J.J Monaghan and R.A Gingold. Shock simulation by the particle method sph. *Journal of Computational Physics*, 52(2):374 – 389, 1983.

[MGG15] Duane Merrill, Michael Garland, and Andrew Grimshaw. High-performance and scalable gpu graph traversal. *ACM Transactions on Parallel Computing*, 1(2):14, 2015.

[Mil99] J.E. Miller. Langford's problem: http://dialectrix.com/langford.html, 1999.

[Mon74] U. Montanari. Networks of Constraints: Fundamental Properties and Applications to Pictures Processing. *Information Sciences*, 7:95–132, 1974.

[Mon92] Joe J Monaghan. Smoothed particle hydrodynamics. *Annual review of astronomy and astrophysics*, 30(1):543–574, 1992.

[Mor66] Guy M Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company New York, 1966.

[MU17] Yohei Miki and Masayuki Umemura. Gothic: Gravitational oct-tree code accelerated by hierarchical time step controlling. *New Astronomy*, 52:65–81, 2017.

[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.

[ND10] John Nickolls and William J. Dally. The GPU Computing Era. *IEEE Micro*, 30(2):56–69, 2010.

[NVI] NVIDIA. *CUDA Occupancy calculator*. http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls.

[Nvi07] CUDA Nvidia. Compute unified device architecture programming guide. 2007.

[Nvi08] CUDA Nvidia. Programming guide, 2008.

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

[NVI13] NVIDIA. *CUDA C PROGRAMMING GUIDE*, jul 2013. http://docs.nvidia.com/cuda/pdf/ CUDA_C_Programming_Guide.pdf.

[Ope97] OpenMP Architecture Review Board. *OpenMP C and C++ Application Program Interface*, October 1997. http://www.openmp.org.

[Pac96] Peter Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1996.

[PB11] P Pande and David A Bader. Computing betweenness centrality for small world networks on a gpu. In *15th Annual High Performance Embedded Computing Workshop (HPEC)*, 2011.

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

[Pri07] Daniel J Price. Splash: An interactive visualisation tool for smoothed particle hydrodynamics simulations. *Publications of the Astronomical Society of Australia*, 24(3):159–173, 2007.

[Pro93] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational intelligence*, 9(3):268–299, 1993.

[RDZR16] Huilong Ren, Zili Dai, Xiaoying Zhuang, and Timon Rabczuk. Dual-support smoothed particle hydrodynamics. *arXiv preprint arXiv:1607.08350*, 2016.

[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.

[Ros09] Stephan Rosswog. Astrophysical smooth particle hydrodynamics. *New Astronomy Reviews*, 53(4):78 – 104, 2009.

[RS90] Sanjay Ranka and Sartaj Sahni. *Hypercube Algorithms with Applications to Image Processing and Pattern Recognation*. Springer-Verlag, 1990.

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

[Sag12] Hans Sagan. *Space-filling curves*. Springer Science & Business Media, 2012.

[Sed46] Leonid I Sedov. Propagation of strong shock waves. *Journal of Applied Mathematics and Mechanics*, 10:241–250, 1946.

[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.

[Sim83] James E. Simpson. Langford sequences: perfect and hooked. *Discrete Math*, 44(1):97–104, 1983.

[SKF10] Luiz Angelo Steffenel, Michaël Krajecki, and Olivier Flauzac. Confiit: a middleware for peer-to-peer computing. *Journal of Supercomputing*, 53(1):86–102, July 2010.

[SKN10] Jyothish Soman, Kothapalli Kishore, and PJ Narayanan. A fast gpu algorithm for graph connectivity. 2010.

[SKSÇ13] Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit V Çatalyürek. Betweenness centrality on gpus and heterogeneous architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 76–85. ACM, 2013.

[SL06] Michael Süß and Claudia Leopold. Implementing irregular parallel algorithms with openmp. In *European Conference on Parallel Processing*, pages 635–644. Springer, 2006.

[Smi00] B. Smith. Modelling a Permutation Problem. In *Proceedings of ECAI'2000, Workshop on Modelling and Solving Problems with Constraints, RR 2000.18*, Berlin, 2000.

[SN11] J. Soman and A. Narang. Fast Community Detection Algorithm with GPUs and Multicore Architectures. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 568–579, 2011.

[Sod78] Gary A Sod. A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws. *Journal of Computational Physics*, 27(1):1 – 31, 1978.

[Spr05] Volker Springel. The cosmological simulation code gadget-2. *Monthly Notices of the Royal Astronomical Society*, 364(4):1105–1134, 2005.

[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.

[SZ11] Zhiao Shi and Bing Zhang. Fast network centrality analysis using GPUs. *BMC Bioinformatics*, 12:149, 2011.

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

[VJO+14] Oreste Villa, Daniel R Johnson, Mike Oconnor, Evgeny Bolotin, David Nellans, Justin Luitjens, Nikolai Sakharnykh, Peng Wang, Paulius Micikevicius, Anthony Scudiero, et al. Scaling the power wall: a path to exascale. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 830–841. IEEE, 2014.

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

[Vol10] Vasily Volkov. Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference, GTC*, volume 10, page 16. San Jose, CA, 2010.

[Wal01] T. Walsh. Permutation problems and channelling constraints. Technical Report APES-26-2001, APES Research Group, January 2001.

[War13] Michael S Warren. 2hot: an improved parallel hashed oct-tree n-body algorithm for cosmological simulation. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 72. ACM, 2013.

[WDP+15] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the gpu. In *ACM SIGPLAN Notices*, volume 50, pages 265–266. ACM, 2015.

[WKQ17] James W Wadsley, Benjamin W Keller, and Thomas R Quinn. Gasoline2: a modern smoothed particle hydrodynamics code. *Monthly Notices of the Royal Astronomical Society*, 471(2):2357–2369, 2017.

[WL15] J. Willcock and A. Lumsdaine. A Unifying Programming Model for Parallel Graph Algorithms. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 831–840, 2015.

[WM95] Wm A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[WS95] Michael S Warren and John K Salmon. A portable parallel particle program. *Computer Physics Communications*, 87(1):266–290, 1995.

[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.

[YB11] Rio Yokota and Lorena A Barba. Treecode and fast multipole method for n-body simulation with cuda. In *GPU Computing Gems Emerald Edition*, pages 113–132. Elsevier, 2011.

Index