UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

INFORMATIQUE

École Doctorale Sciences Technologie Santé

Hybrids Architectures to Reach Exascale

Les Architectures Hybrides pour Atteindre l'Exascale

par **Julien Loiseau**

Thèse de doctorat d'Informatique

Sous la direction de : **M. Michaël KRAJECKI, Professeur des Universités**

JURY

??	??	Rapporteur
??	??	Rapporteur
Benjamin Bergen	Docteur-Ingénieur au Los Alamos National Laboratory, USA	Examinateur
Guillaume Colin de Verdiere	Ingénieur au CEA, France	Examinateur
Michaël Krajecki	Professeur à l'Université de Reims Champagne-Ardenne	Directeur
François Alin	Maître de Conférences à l'Université de Reims Champagne-Ardenne	Co-directeur
Christophe Jaillet	Maître de Conférences à l'Université de Reims Champagne-Ardenne	Encadrant

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

Thanks everyone !

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 main 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 but the 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

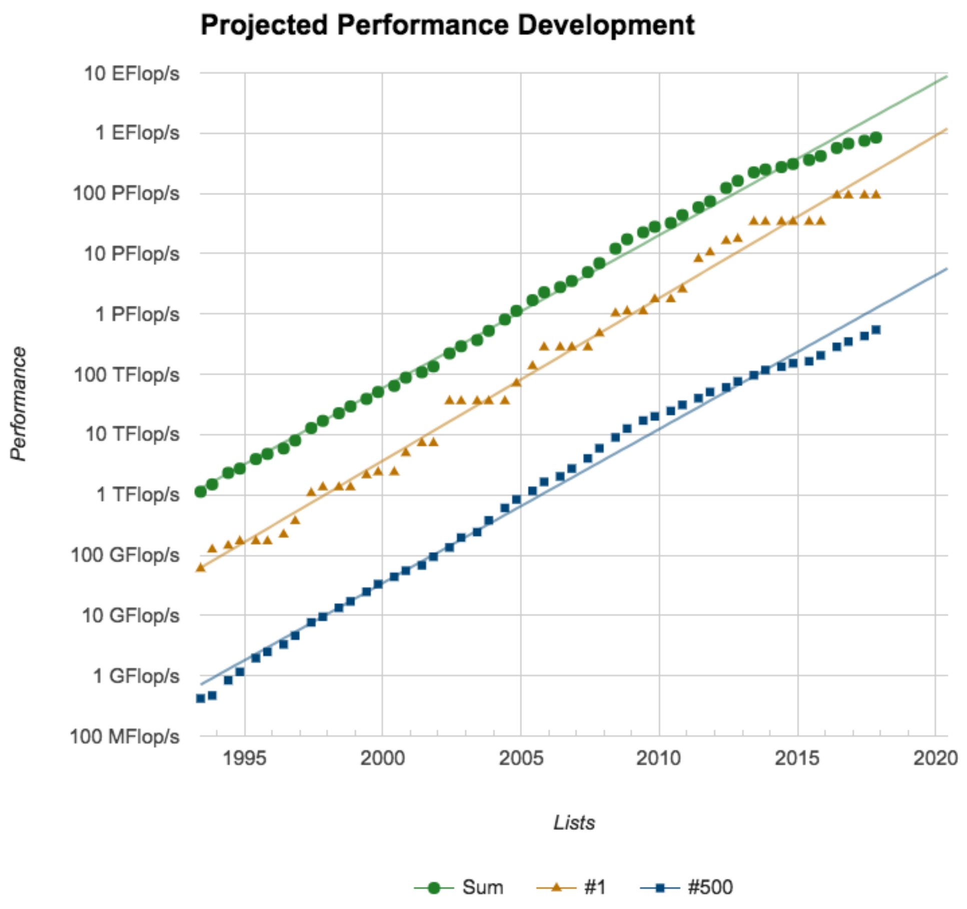


Figure 1: Computational power evolution in the TOP500 list

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 chapters 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 chapter 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 chapter, 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

High Performance Computing, HPC, does not find a strict definition. Since computer creation, first dedicated for balistic purposes, domain scientists developped their tool to perform computations. Then, in front of the complexity of building such machines, HPC became a dedicated field of research.

Computer scientists interested in HPC will have to focus on several domains.

- The energy consumption, mainly directed by the hardware producers.
- The computational power, how to take advantages of the ressources ?
- The communication, because such machines are constructed over several machines or nodes.

Domain scientists are also involved directly in HPC with their software and redefining the structure based on their needs and usages.

Chapter 1

Theory of HPC

1.1 Introduction

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

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

The Amhdal's and Gustafson's laws are presented and detailed and thus 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 create. The most famous of first vacuum tubes supercomputers, The ENIAC, was based on decimal system. It might be the most known of first supercomputers but the real revolution came from its successor. In 1944 the first binary system based computer, called the Electric Discrete Variable Automatic Computer (EDVAC), was created. In the EDVAC team, a physists 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 Fig. 1.1.

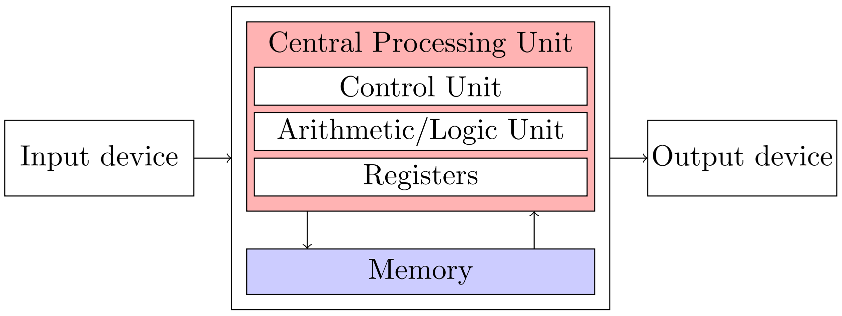


Figure 1.1: Von Neumann model

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

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

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 arithmetique with operations such as ADD, MUL, SUB, etc. Of course those operations are way more complex on modern CPUs. On the other hand, 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 important and called Buses and can be separated between data buses, control buses and adresses buses.

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

Every devices or machines we will describe in the next chapter will have the same architecture as a basis.

1.2.1 Terminology

Before characterizing the execution models, some terminology must be set to describe properly the machines.

Core: A core is a Von Neumann machine.

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, in addition to the CPU, provide additional computation power. We can identify them as GPUs, FPGAs, ASICs, etc. A socket can have access to one or more accelerators and sockets can also share their usage.

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

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

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

Table 1.1: Flynn's taxonomy

1.3 Flynn taxonomy and executions models

The Von Neumann model gives us a generic idea of how a computational unit is fashioned. The constant demand in more powerful computers required the scientists to find more way to provide this computational capacity. 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 SIMD, SISD, MISD and MIMD models represented on Fig. 1.1. Every of those for execution model correspond to a specific machine and function.

1.3.1 Single Instruction, Single Data: SISD

This is the model corresponding to a single core CPU like in the Von Neumann model. This sequential model takes one instruction, operates on one data and the result is then store and the process continues over. SISD is important to consider as a reference for computational time and will be considered in the next part for Amdahl's and Gustafson's laws.

1.3.2 Single Instruction, Multiple Data: SIMD

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

1.3.3 Multiple Instructions, Multiple Data: MIMD

Every element executes its own instructions on its own data set. This can represent the behavior of a CPU using several cores, threads or even the differents nodes of a supercomputer cluster.

1.3.4 Multiple Instructions, Single Data: MISD

This last model can correspond to a pipelined computer but even in this case the data are modified after every operations. This is the least common exection model.

1.3.5 SIMT

We can also find another characterization to describe the new GPUs architecture: Single Instruction, Multiple Threads. This appears in one of NVIDIA's company paper [LNOM08]. This model describes a stack of SIMD architectures, every block of threads is working with the same

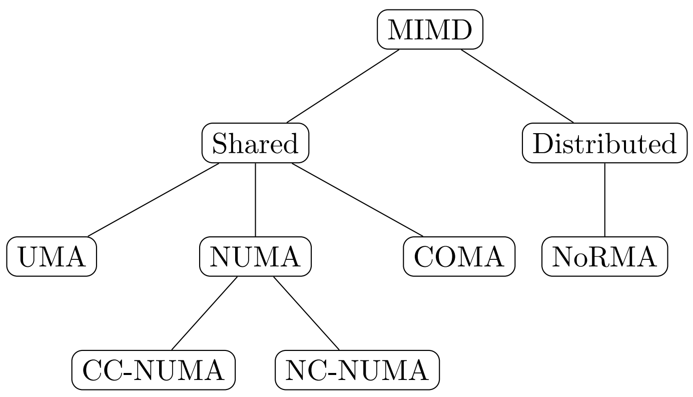


Figure 1.2: MIMD memory models

control processor on different data. This is the model we describe in next chapter used for the *warps* model in NVIDIA CUDA.

1.4 Memory

In addition of the execution model and parallelism the memory accesses parterns have a main role on performances especially in SIMD and MIMD.

Different memory technologies exists and the aim is always greater capacity, better speed and bandwidth while keeping the data integrity.

1.4.1 Memory technologies

We present here the volatile memory, represented in the memory part of the Von Neumann model.

SRAM

The Static Random Access Memory is built using so called ”flip-flop” circuits that can store the data as long as the machine is powered. This kind of memory is very expensive to produce due to the number of component needed and the size of the memory. Therefore it is usually limited for small amount of storage. The SRAM is mainly used for cache memory.

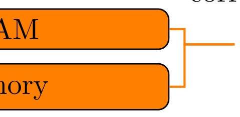
Cache is a memory mecanism that is useful to consider when targeting performance. This little memory is built over several levels. The closer to the CPU is L1, then L2 and generally no more than L3 except on specific architecture. When look for a data the CP will first check the L1 cache, otherwise L2 and L3 to get the data to higher level. This is based on the idea that if a data is used, it shall be use again in the near future.

DRAM

The Dynamic Random Access Memory, at the opposite to the SRAM, is based on transistors and capacitors to store the binary information. This memory is less expansive to produce but needs to be refresh at a determined time however the data are lost. There is several sub categories of DRAM used in different devices.

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

We can also find Error-Correcting Code, ECC, memory which implements a bunch of data correction algorithm be garanty the validity of them when error is not allowed.



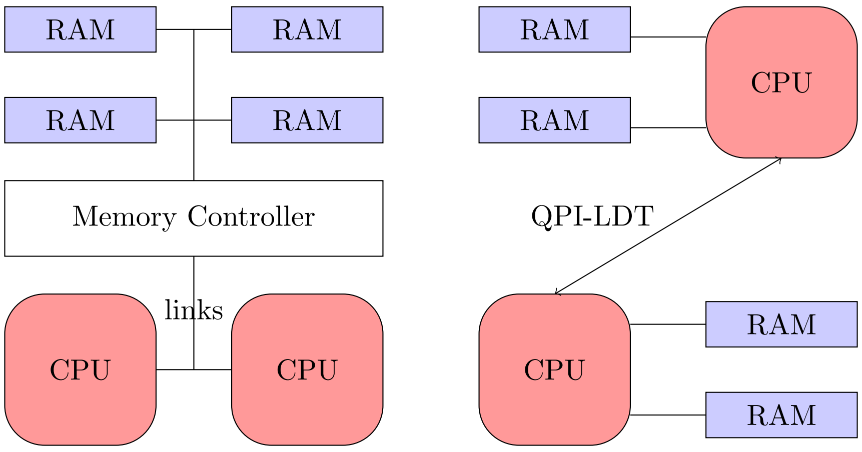


Figure 1.3: UMA vs NUMA

The different types of memory for MIMD model are summed up in Fig.1.2. Two main categories can be extract, share or distributed memories.

1.4.2 Shared memory

In case of the SISD the memory access is just serial and no really rules needs to be set for its usage. When it comes to multi-threaded and multi-cores like MIMD or SIMD execution models several kind of memory models are possible. We give a description of the most common shared memories architecures.

UMA

The Uniform Memory Access is a global memory shared by every threads or cores. In UMA every processors us its own cache as local memory. The addresses can be accessed directly by each processors which make the access time ideal. The downside is that more processors require more buses and thus UMA is hardly scalable. The cache consistancy problem also appears in this context and will be discussed in next part. Indeed, if a data is loaded in one processor cache and modifies, 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 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, a cache coherency is important. Two methods are possible: on one hand, the most used is Cache-Coherent NUMA (CC-NUMA) were protocols are used to keep data coherency through the memory. on the other hand No Cache NUMA (NC-NUMA) force the processes to avoid cache utilization and write results in main memory losing all the benefits of caching data.

COMA

In Cache-Only Memory Accesses, the whole memory is see as a cache from every processes. Attraction memory is setting up and will attract the data near the process that will use those

Name	**FLOPS**	**Year**	**Name**	**FLOPS**	**Year**
kiloFLOPS	10^3		petaFLOPS	10^{15}	2005
megaFLOPS	10^6		exaFLOPS	10^{18}	2020 ?
gigaFLOPS	10^9	$\approx$ 1980	zettaFLOPS	10^{21}	
teraFLOPS	10^{12}	1996	yottaFLOPS	10^{23}	

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

data. This model is less commonly use and lead to, at best, same results as NUMA.

1.4.3 Distributed memory

The previous models are shared memory, in the case where the processes can access memory of their neighbors processes. In some cases, like supercomputer, it would be too heavy for processors to handle the requests of all the others through the network. Each process or node will then 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 copyed in local memory. This distributed memory is called No Remote Memory Access (NoRMA).

1.5 FLOPS, Speedup, efficiency and scalability

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

1.5.1 FLOPS

The Floating point Operation Per Second consider the number of floating-point operation that the system will executes in a second. They are an unit of performance for computers, higher FLOPS is better. This is the scale also use to consider supercomputers computational power. For a cluster we can compute the theoretical FLOPS (peak) with:

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

On Fig.1.2, the scale of FLOPS and the year of the first world machine is presented.

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

1.5.2 Scalability

The scalability express the way a program react to parallelism. When an algorithm is implemented on a serial machine and is ideal to solve a problem, one may consider to use it on more than one core, socket, node or even cluster. Indeed, one may expect less computation time, bigger problem or a combination of both while using more ressources. This completely depend on the algorithm parallelisation and is expressed through scalability. A scalable program will

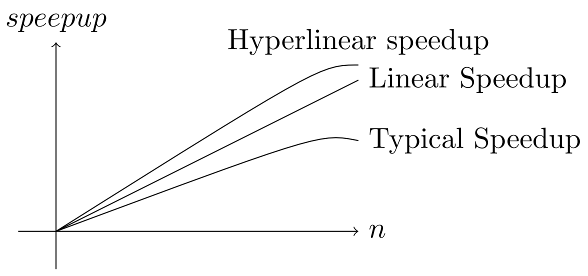


Figure 1.4: Observed speedup

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 approch using speedup and efficiency.

1.5.3 Speedup and efficiency

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

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

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

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

As shown on figure 1.4 several kind of speedup can be observed.

Linear The linear speedup usually represents the target for every program in HPC. Indeed, having the speedup growing exactly as the number of processors grows is the ideal case. Codes fall typical into two cases, typical and hyperlinear 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 reduiced to the sequential part or lose time in communications/exchanges.

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

The efficiency is defined by the speedup devided by the number of workers:

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

The efficiency, expressed in percent, represent the evolution of the code stability to growing number of processors. As the number of processes grows, a scalable application will keep an efficiency near 100%.

1.6 Amdhal's and Gustafson's law

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

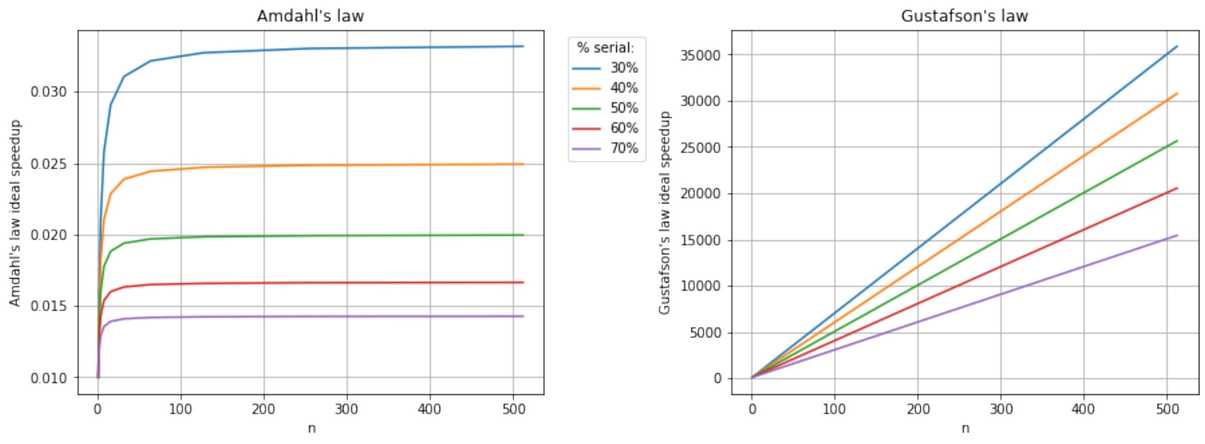


Figure 1.5: Theoretical speedup for Amdahl's (left) and Gustafson's (right) law

1.6.1 Amdahl's law

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

Extracted from the Amdahl paper the law can be writen 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 $n = \inf$ processes, $S_n \leq \frac{1}{Seq}$ the sequential part of the code become the most time consumming.

And the efficiency become:

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

A representation of Amdahl's speedup is presented on Fig. 1.5 with varying percentage of serial part. The parallel part is like $Par = (100 - Ser)\%$.

1.6.2 Gustafson's law

The Amdhal'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 effiencity:

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

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

1.7 Conclusions

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

Chapter 2

Hardware in HPC

2.1 Introduction

Optimization can't be done without a good knowledge of the architecture of device, machine, or computer. Indeed, nowadays software and API try to take care of most of the optimizations but the last percents of gain always need to be architecture dependent. In this chapter we describe the most important devices architecures from classical processors, General Purpose Graphics Processing Units (GPGPUs), Field Programmable Gate Arrays (FPGAs) and Application-specific integrated circuits (ASICs). Then those independants elements are use together in order to build supercomputers. The way they are arranged and the nodes interconnection is something that matters at large scale.

2.2 Architectures

In this section we will describe the main nowadays architecture from HPC world and their specificities.

The CPU, as we know it today, begins its history with *Texas Instruments Inc* and the first patent describing a CPU is "Computing systems cpu" proposed by *Gary Boone* and published in 1973. It is the reflection of the Von Neumann Machine we presented in Chapter I.

From this first version plenty of optimizations arised.

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

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

Prefetching 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. Prefecthing technology is a way to, knowing the next instructions operands, prefetch the data in closer cache. The prefetch can either be hardware or software implemented and can concern data and even instructions.

Vectorization Processors allows the instructions to be executed at the same time in a SIMD manner. If the same instruction is executed on coalescent data they can be executed in the same clock cycle. Of course this tool require specific care during coding.

Those optimizations can be found either in the classical processor model or accelerators.

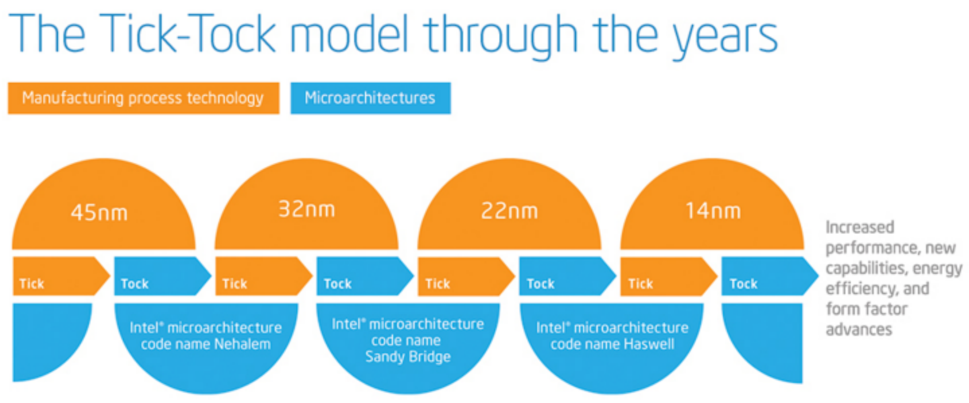

The Tick-Tock model through the years
Manufacturing process technology
Microarchitectures
45nm
32nm
22nm
14nm
Tick
Tock
Tick
Tock
Tick
Tock
Tick
Tock
Intel® microarchitecture code name Nehalem
Intel® microarchitecture code name Sandy Bridge
Intel® microarchitecture code name Haswell
Increased performance, new capabilities, energy efficiency, and form factor advances

Figure 2.1: Intel Tick-Tock model

2.2.1 Classical processors

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

Intel

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

Since its creation the model followed the same fashion, a new manufacturing technology like shrink of the chip with better engraving on a "Tick" and a new microarchitecture 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, Hyperthreading, 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.

Hyperthreading

Another specificity of Intel processor is Hyperthreading (HT). This technology makes a single physical processor appearing as two logical processors for user's level. In fact a processors embedding 8 cores will appear 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 does not fit to all the cases and can be disable for normal use of the processors.

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 parallism 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 Ferry, Knights Corner and Knight Landing [SGC$^+$16]. The last architecture, Knight Hill, was recently canceled by Intel. 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.

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 to Advanced RISC Machine (ARM). ARM is a specific kind of CPU based on RISC architecture as its ISA depsite usual processors using CISC. The downside of CISC machines makes them hard to create and they require way more energy to work. The ISA from the RISC is simplier and requires less transistors to operate. Therefore, the energy required and the heat dissipated is less important. It would then be easier to create massively parallel processors based on ARM. On the other hand, simple ISA impose more work on the source compilation to fit the simple architecture. That makes the instructions sources longer and therefore more single instructions to execute.

The ARM company provide several version of ARM processors named Cortex-A7X, Cortex-A5X and Cortorx-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 finaly reach Exascale near 2020. The third step, Mont-Blanc 3, is about to work on a pre-exascale prototype powered by Cavium’s ThunderX2 ARM chip based on 64-bits ARMv8.

2.2.2 GPGPU

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

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. Its name seems to come from *invidia* the latin word for Envy and vision, for the graphics generation.

Known as the pioner in graphics, cryptocurrency, portable devices and now AI, it seems to be even the creator of the name ”GPU”. It GPU, inspired from visualisation and gaming at a first glance, is available as a dedicated device since the Tesla. The public GPUs can also be use for dedicated computation but does not feature ECC memory, double precision or special functions/FFT cores.

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

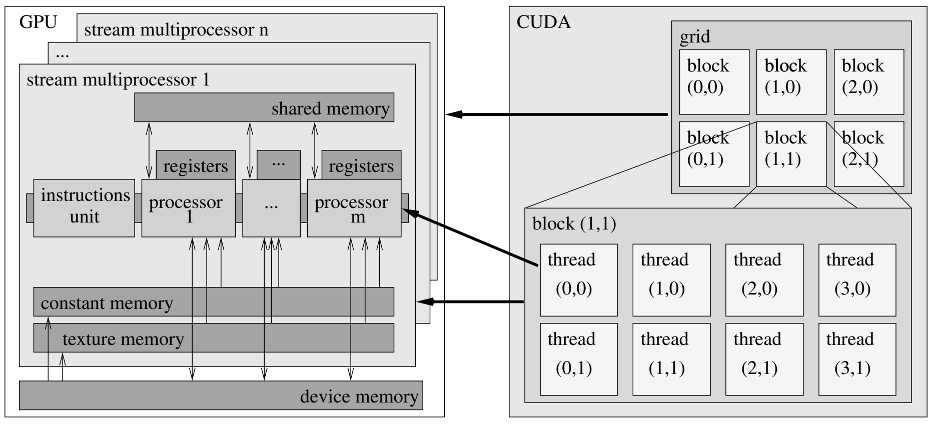


Figure 2.2: NVIDIA GPU and CUDA architecture overview

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

Memory, bandwidth and streams: In order to use data in a CUDA 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.

Threads synchronization: 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. Thus we intend to provide a way to regularize their execution, in order to get good acceleration with GPU computation.

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

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

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

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

We give details on the GPU we mainly used in this study in the ROMEO supercomputer center.

Details on K20X This NVIDIA Tesla Kepler GPU is based on the GK110 graphics processor describes in the whitepaper[Nvi12] on 28nm process. This GPU comes in active and passive cooling with respectively K20Xc and K20Xm. This GPU embedded 2688 CUDA cores distributed in 14 SMX (we note that GK110 normally provides 15 SMX but only 14 are present4 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 place its GPGPUs. Their HPC GPUs are called FirePro. They are targeting using a language near CUDA called OpenCL. An interessant creation of AMD are 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.

2.2.3 FPGA and ASIC

Field Programmable Gates Array are device that can de reprogram to fit the needs of the user after their construction. The leader was historically Altera with the Stratix, Arria and Cyclone FPGAs and is now part of Intel. With the FPGAs the user have access to the hardware itself and can design its own circuit. Nowadays FPGA can be targeted with OpenCL programming language. The arrival of Intel in this market promess 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.

ASICs are dedicated device construct for on purpose. An example of ASIC can be the Gravity Pipe (GRAPE) which is dedicated to compute gravitation given mass/positions. Google leads the way for ASIC and just created its dedicated devices to boost AI bots.

2.3 Interconnection and clusters

2.3.1 Interconnects

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

The Fig.2.1 list the most famous interconnection network and their diameters.

Finir table

Ajouter sche topologies ?

InfiniBand (IB) is the most spread technology used for interconnect with different kind of bandwith presented in Fig.2.2.

2.3.2 TOP500 remarkable supercomputer

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

Type	**SubType**	**Diameter**	**Name**	**Gbs**	**Year**
Trees					
	Fat Tree				
	k-ary tree				
	Extended Fat Tree				
Mesh/Torus					
	k-ary n-mesh				
	k-ary n-cube				
Dragonfly					
HyperX					

Table 2.1: InfiniBand technologies

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 2.2: InfiniBand technologies

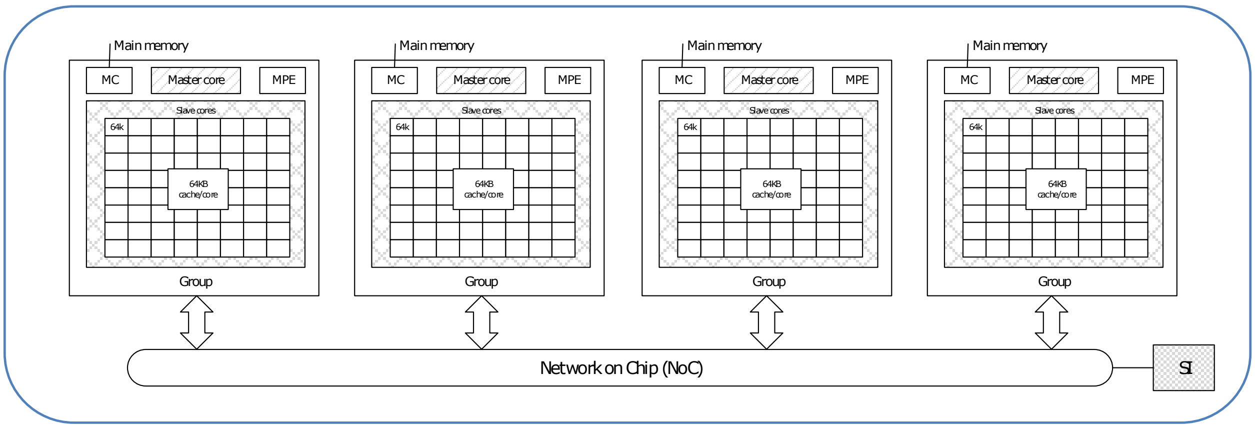


Figure 2.3: Sunway Taihulight node architecture from *Report on the Sunway TaihuLight System*, Jack Dongarra, June 24, 2016.

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 interessant 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 Fig.2.3. The processor is composed of the master core, a Memory Controller (MC), a Management Processing Element (MPE) that manages the Computing Processing Elements (CPE) which are the slaves cores.

The interconnect network is called Sunway Network and connected using Melloanix Host Channel Adapter (HCA) and switches. This is a five level interconnect going through computing nodes, computing board, supernodes 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 GPU accelerated GPU is a most powerful representative of the hybrid architecture featuring processor and accelerators.

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 mixes a 6D Mesh/Torus interconnect.

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.

2.4 ROMEO Supercomputer

The ROMEO supercomputer center is the computation center of the Champagne-Ardenne region in France. Hosted since 2002 by the University of Reims Champagne-Ardenne, this so called meso-center (French name for software and hardware architectures) is used for HPC for theoric research and domain science like applied mathematics, physics, biophysics and chemistry.

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

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

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

2.4.1 ROMEO hardware architecture

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

Each node is composed of two processors Intel Ivy Bridge 8 coeurs @ 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 filesystem.

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 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.

The TOP500 supercomputers presented here are a perfect use case of the tehcnologies presented and reflect the best human can do with nowadays systems. They also show that every architecture is unique in its construction and justify the optimization work dedicated to reach performance.

Chapter 3

Software and Benchmarks

3.1 Introduction

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

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

Parler des pi
load balancin
currence, ...
faire plus ba

3.2 Sofware/API

In this section we present the main runtimes, API and programming language use in HPC and in this study in particular. The considered language will be C/C++, the most present in HPC world.

3.2.1 Parallel programming

PThreads

The POSIX threads API is an execution model available in most of the languages. It allows the user to define threads that will execute concurrently on the processor ressources using shared/private memory. PThreads is the low level handling of threads and the user need to handle concurrency with mutex, 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.

Fine-grain C
grain applica

OpenMP

Open Multi-Processing, OpenMP, is an API for multi-processing shared memory. It is based on the fork-join model. [Cha08, Sup17]

Ohters

3.2.2 Distributed

In the cluster once the code have been developped 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.

MPI

The Message Passing Interface, MPI, is the most and widely spread runtime for distributed computing. [Gro15, Gro14]

Charm++

Legion

3.2.3 Accelerators

CUDA

OpenCL

3.3 Benchmark

3.3.1 TOP500

The most famous benchmark is certainly the TOP500[1]. It gives the ranking of the 500 most powerful, known, supercomputers of the world as its name indicates. Since 1993 the organization assembles and maintains this list updated twice a year in June and November.

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

3.3.2 GRAPH500

3.3.3 GRENN500

[1]http://www.top500.org

Conclusion

Part II

Complex systems

Chapter 4

Computationally heavy

4.1 Introduction

4.2 Background

Present here the combinatorial problems and the ways to solve them.

4.2.1 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, 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 Fig. 4.1.

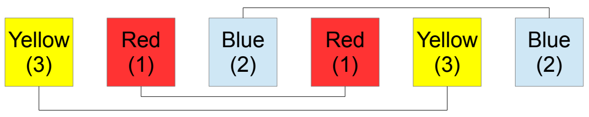


Figure 4.1: L(2,3) arrangement

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 $L(2,n)$ has solutions for all n such that $n = 4k$ or $n = 4k - 1$ ($k \in \mathbb{N} \setminus \{0\}$). 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 Fig. 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 using a specific algorithm. (see Table 4.1); $L(2,27)$ and $L(2,28)$ have just been computed but no details were given.

The main efficient known algorithms are the following: 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

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		-
L(2,28)	1,607,383,260,609,382,393,152		-

Table 4.1: Solutions and time with differents methods

lastest 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 $L(2,28)$.

4.2.2 Miller algorithm

In this part we present our multiGPU cluster implementation of the Miller's algorithm. First, we introduce the backtrack method. Then we present our implementation in order to fit the GPUs architecture. The last section presents our results.

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 multiGPU cluster.

Langford's problem tree representation In [HKS02], we propose to formalize the Langford problem as a CSP (*Constraint Satisfaction Problem*), first introduced by Montanari in [Mon74], and show that an efficient parallel resolution is possible. CSP formalized problems can be transformed into tree evaluations. In order to solve $L(2,n)$, we consider the following tree of height n: see example of $L(2,3)$ in Fig. 4.2.

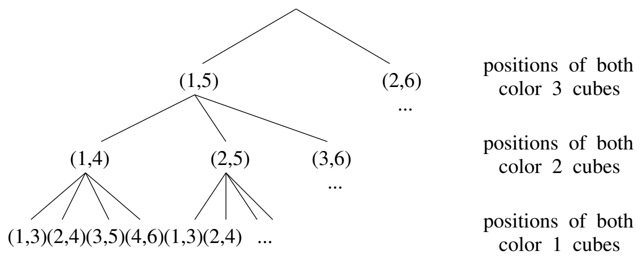

(1,5)
(2,6)
...
positions of both color 3 cubes
(1,4)
(2,5)
(3,6)
...
positions of both color 2 cubes
(1,3)(2,4)(3,5)(4,6)(1,3)(2,4) ...
positions of both color 1 cubes

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

- Every level of the tree corresponds to a 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.

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 Fig. 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 (Fig. 4.4).

```
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

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 interger, one bit to 1 corresponding to a cube presence, a 0 to its absence. This is what we called a *mask*. This way Fig. 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 1s correspond to occupied positions; the assignment is consistent *iff* there are as many 1s 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. Fig. 4.6 represents the tree masks' representation.

	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 $L(2,3)$

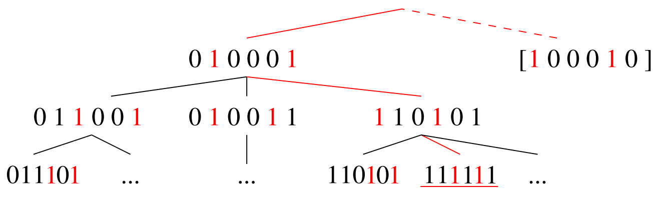


Figure 4.6: Bitwise representation of the Langford $L(2,3)$ placement tree

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 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 Fig. 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 Fig. 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.

Hybrid parallel implementation This part presents our methodology to implement Miller's method on a multiGPU 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 Fig. 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

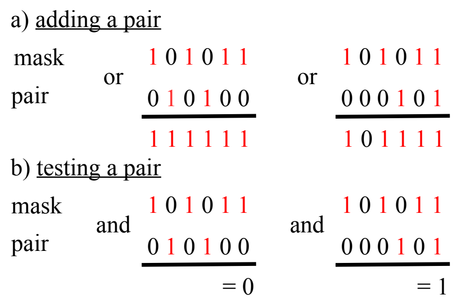


Figure 4.7: Testing and adding position

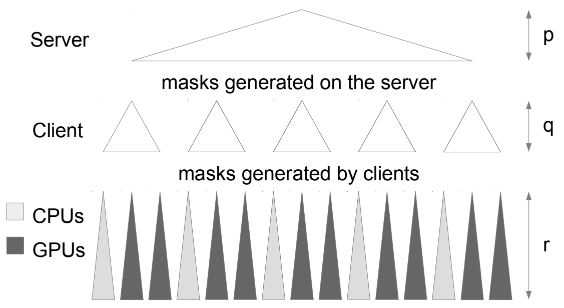


Figure 4.8: Server client distribution

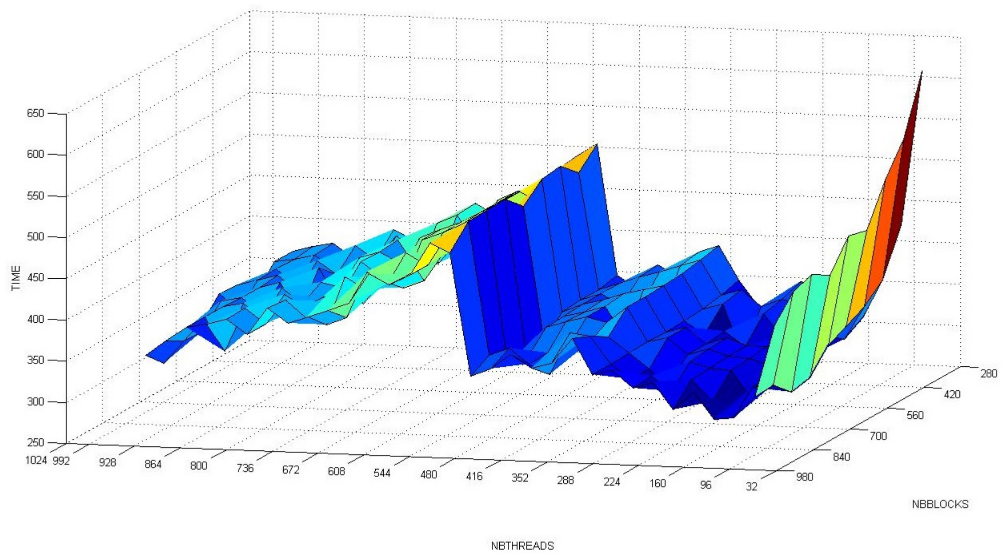


Figure 4.9: Time depending on grid and block size on $n = 15$

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.

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. Fig. 4.9 represents the time in relation with the number of blocks per grid and the number of

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

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.

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 Fig. 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.

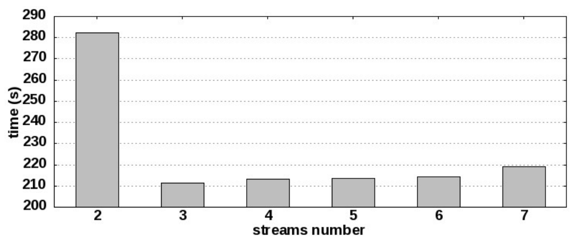


Figure 4.10: Computing time depending on streams number

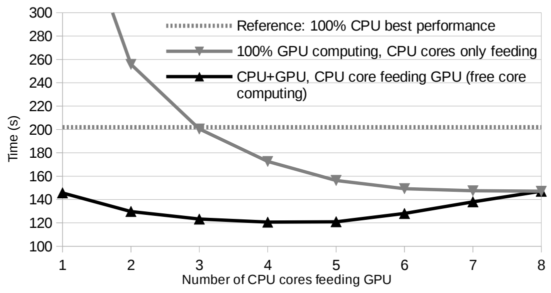


Figure 4.11: CPU cores optimal distribution for GPU feeding

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 Fig. 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 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. Fig. 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.

Results

Regularized method results We now can 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 4.2), 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 200000× more nodes of the search tree than the CPUs, with a faster time.

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

Table 4.2: 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

Table 4.3: Backtrack (seconds)

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 decide to implement the original backtrack method on GPU (see Fig. 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 4.2.2, 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 4.3 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× 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 exhibits the ability of the GPU scheduler to manage highly irregular tasks. It proves that GPUs are adapted even to solve combinatorial problems, which they were not supposed to be.

4.2.3 Godfrey's algebraic method

The previous part presents the Miller algorithm for the Langford problem, this method cannot achieve bigger instances than the $L(2,21)$.

An algebraic representation of the Langford problem has been proposed by M. Godfrey in 2002. In order to break the limitation of $L(2,24)$ we already used this very efficient problem specific method. In this part we describe this algorithm and optimizations, and then our implementation on multiGPU clusters.

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

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 $F(X,3) = (X_1X_3 + X_2X_4 + X_3X_5 + X_4X_6) \times (X_1X_4 + X_2X_5 + X_3X_6) \times (X_1X_5 + X_2X_6)$

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:

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

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

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.

Optimizations

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

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 recomputation. 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.

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 sum 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). Fig. 4.12 represents this big integer handling.

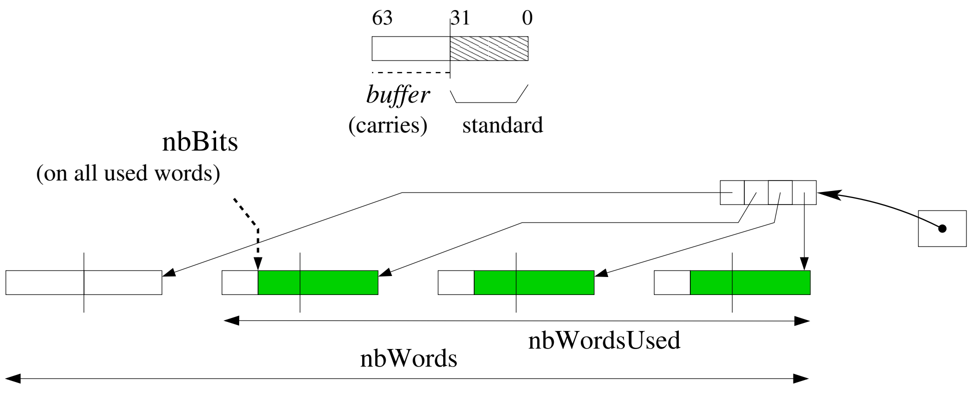


Figure 4.12: Big integer representation, 64 bits words

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 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.

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.

Experimental 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.

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 4.2.3 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. Fig. 4.14 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. Fig. 4.13 presents the tests performed on the $n = 20$ problem: the best distribution is around 128 threads per block.

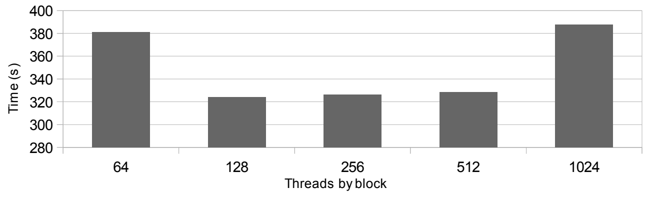


Figure 4.13: $L(2, 20)$, number of threads per block

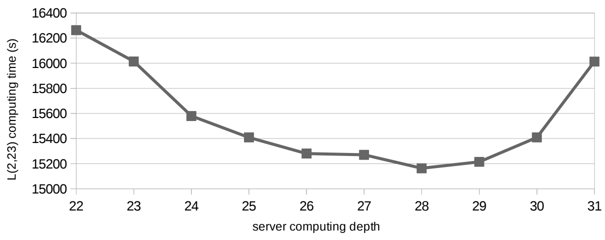


Figure 4.14: Influence on server generation depth

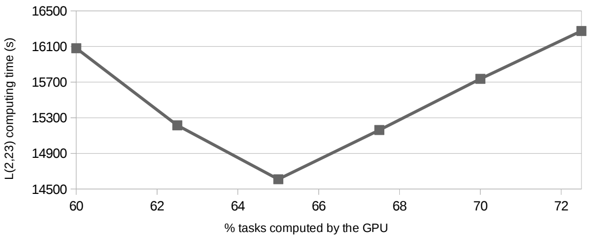


Figure 4.15: Influence of tasks repartition

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. Fig. 4.15 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 **??**, 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.

Static distribution - In this case we used the synchronous distribution presented in 4.2.3. 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 Fig. 4.16. 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 ouput 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.

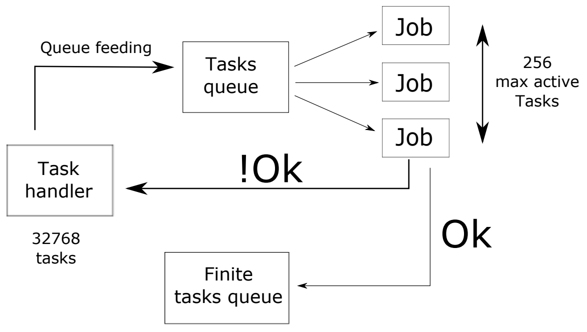


Figure 4.16: Best-effort distribution

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.

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 - Fig. 4.17 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.

Fig. 4.18 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.

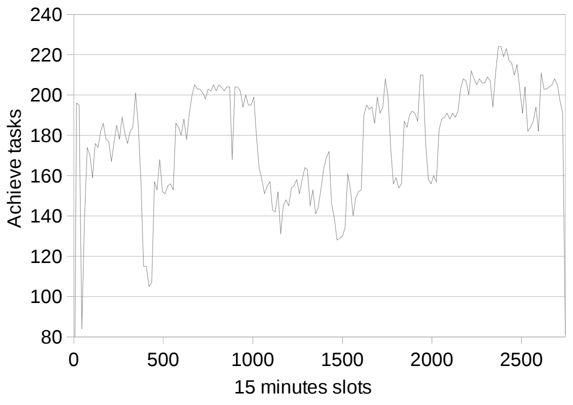


Figure 4.17: $L(2,27)$ tasks grouped by 15" slots

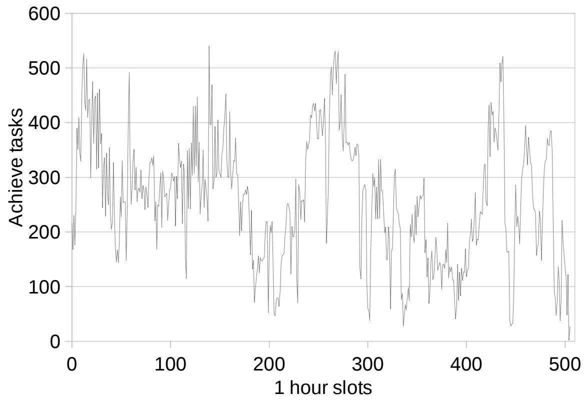


Figure 4.18: $L(2,28)$ tasks grouped by 1 hour slots

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.

4.2.4 Conclusion

This paper presents two methods to solve the Langford pairing problem on multiGPU clusters. In its first part the Miller's algorithm is presented. Then to break the problem limitations we show optimizations and implementation of Godfrey's algorithm.

CSP resolution method - 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.

MultiGPU clusters and best-effort - In addition and with the aim to beat the Langford limit we present a new implementation of the Godfrey method using GPUs as accelerators. 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.

Langford problem results - This study enabled us to compute $L(2,27)$ and $(L2,28)$ in respectively less than 2 days and 23 days on the University of Reims ROMEO supercomputer. 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

Perspectives - This study shows the benefit of using GPUs as accelerators for combinatorial problems. In Miller's algorithm they handle 80% of the computation effort and 65% in Godfrey's. As a near-term prospect, we want to scale and show that it is possible to use the order of 1000 or more GPUs for pure combinatorial problems.
The next step of this work is to generalize the method to optimization problems. This adds an order of complexity since shared information has to be maintained over a multiGPU cluster.

Chapter 5

Complex systems as a benchmark

5.1 Introduction

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.

Algorithm 1 Sequential BFS

```
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:     $In \leftarrow \{v_s\}$;
3:     $Vis \leftarrow \{v_s\}$;
4:     $P(v) \leftarrow \perp \forall v \in V$;
5:     while $In \neq \emptyset$ do
6:         $Out \leftarrow \emptyset$
7:         for $u \in In$ do
8:             for $v | (u, v) \in E$ do
9:                 if $v \notin Vis$ then
10:                    $Out \leftarrow Out \cup \{v\}$;
11:                    $Vis \leftarrow Vis \cup \{v\}$;
12:                    $P(v) \leftarrow u$;
13:                end if
14:            end for
15:        end for
16:        $In \leftarrow Out$
17:    end while
18: end function
```

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 2015, is composed of 201 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

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

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.

In this study we focus on GPU optimization. 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.

This paper is organized as follows. 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 concude by examining the results for different graph scales and load distributions.

5.1.1 Related work

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. This paper [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.

5.1.2 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

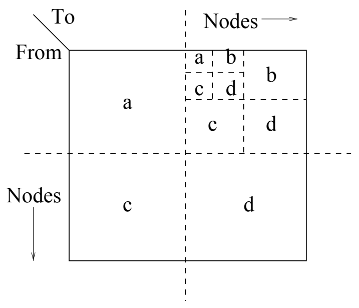


Figure 5.1: Kronecker generation scheme based on edge probability

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.

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 Fig 5.1, 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.

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.

5.1.3 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.

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 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. The following example illustrates the CSR representation:

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$R = \{0, 2, 4, 7, 8\}$$

$$C = \{1, 2, 0, 2, 0, 1, 3, 2\}$$

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.

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.

Algorithm 2 Parallel BFS on 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:     $P(N_{i,j}, v_s) \leftarrow v_s$
5: **end if**
6: **while** *true* **do**
7:     $(Out_{i,j}, Marks_{i,j} \leftarrow$ ExploreFrontier();
8:     $done \leftarrow \bigwedge_{0 \leq k,l \leq n} (Out_{k,l} = \emptyset)$
9:     **if** *done* **then**
10:        **exit loop**
11:    **end if**
12:    **if** $j = 0$ **then**
13:        $prefix_{i,j} = \emptyset$
14:    **else**
15:        **receive** $prefix_{i,j}$ from $N_{i,j-1}$
16:    **end if**
17:    $assigned_{i,j} \leftarrow Out_{i,j} \setminus prefix_{i,j}$
18:    **if** $j \neq n-1$ **then**
19:        **send** $prefix_{i,j} \cup Out_{i,j}$ **to** $N_{i,j+1}$
20:    **end if**
21:    $Out_{i,j} \leftarrow \bigcup_{0 \leq k \leq n} Out_{i,k}$
22:    WritePred()
23:    $Vis_{i,j} \leftarrow Vis_{i,j} \cup Out_{i,j}$
24:    $In_{i,j} \leftarrow Out_{j,i}$
25: **end while**
```

This algorithm 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.

- 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.

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*. Fig 5.2 presents the two approches, whith the time visiting all the edges, and the benefits of their hybridization.

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 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 3 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*.

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.

Algorithm 3 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
```

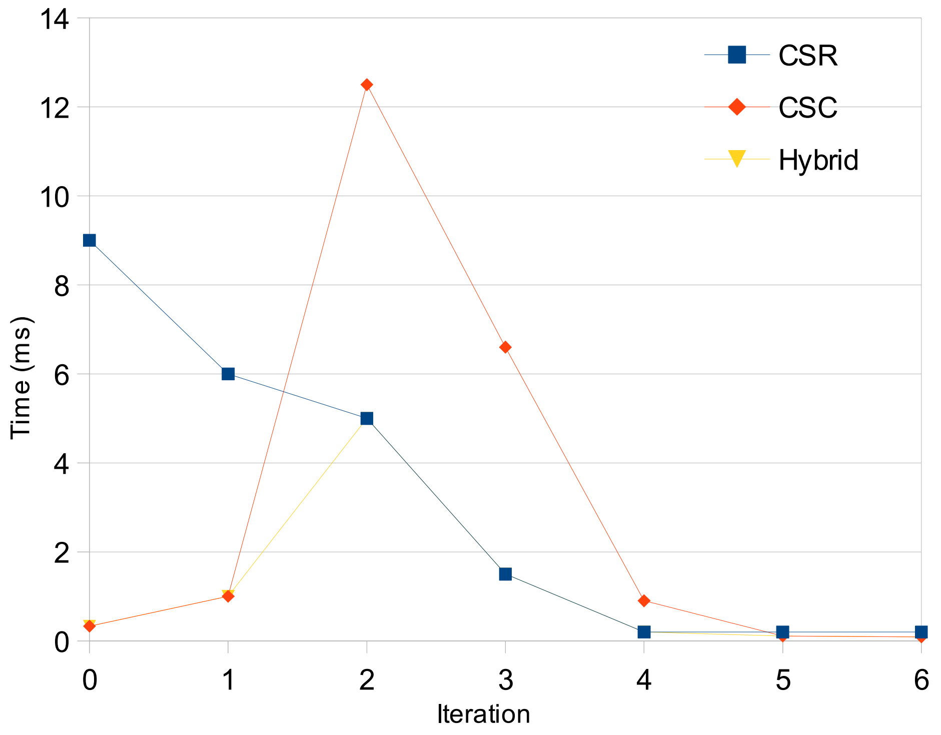


Figure 5.2: CSR and CSC approach comparison. On a 6 iterations BFS, the time with the two method is compared. The hybridization just takes the best time of each method

5.1.4 Results

Working environment

CPU and GPU comparison

On Fig 5.3 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.

Strong and weak scaling

On Fig 5.5 and Fig 5.4 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 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 Fig 5.4, the problem SCALE is presented on each point. With our method we were able to reach up to 12 GTEPS using this scaling.

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

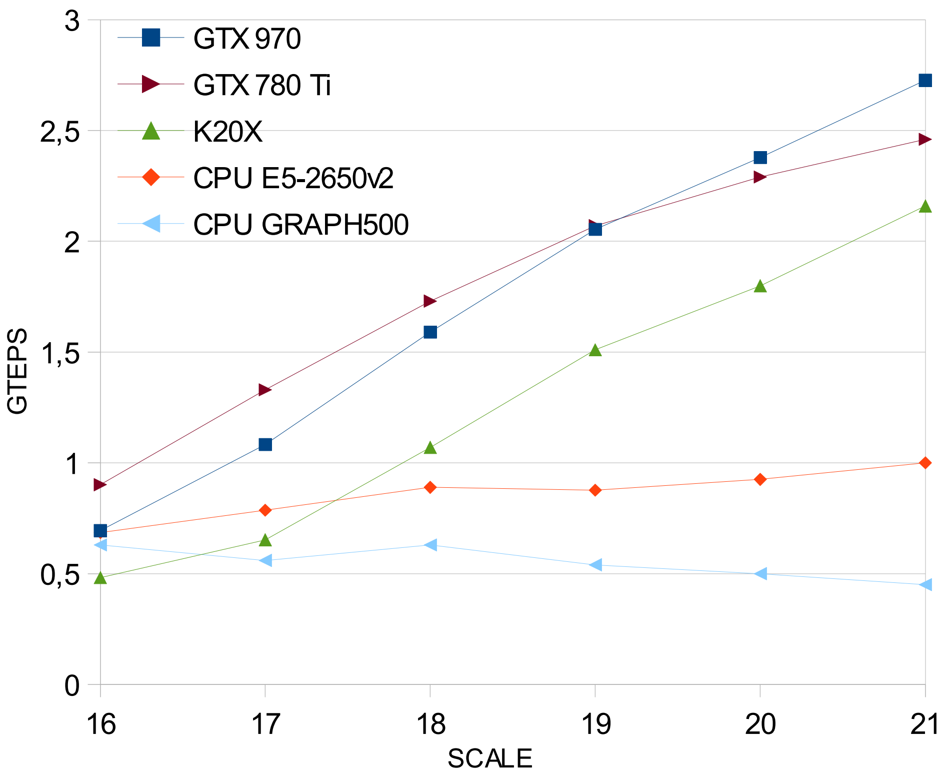


Figure 5.3: Single CPU and accelerators comparison. CPU Graph500 represent the best implementation proposed by the Graph500 website.

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.

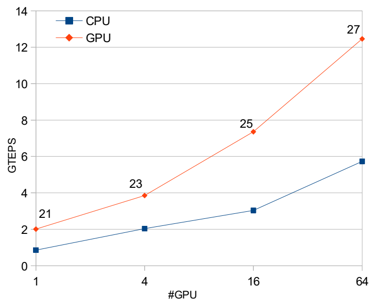


Figure 5.4: CPU *vs* GPU weak scaling. The number of CPUs is the same as the number of GPUs.

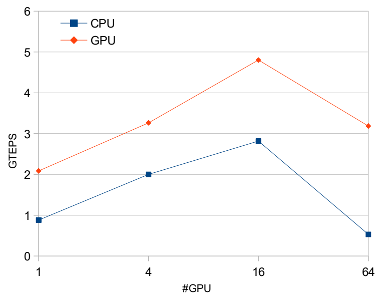


Figure 5.5: CPU *vs* GPU strong scaling. The *SCALE* is showed on the GPU line. The number of CPUs is the same as the number of GPUs.

On Fig 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 Fig 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.

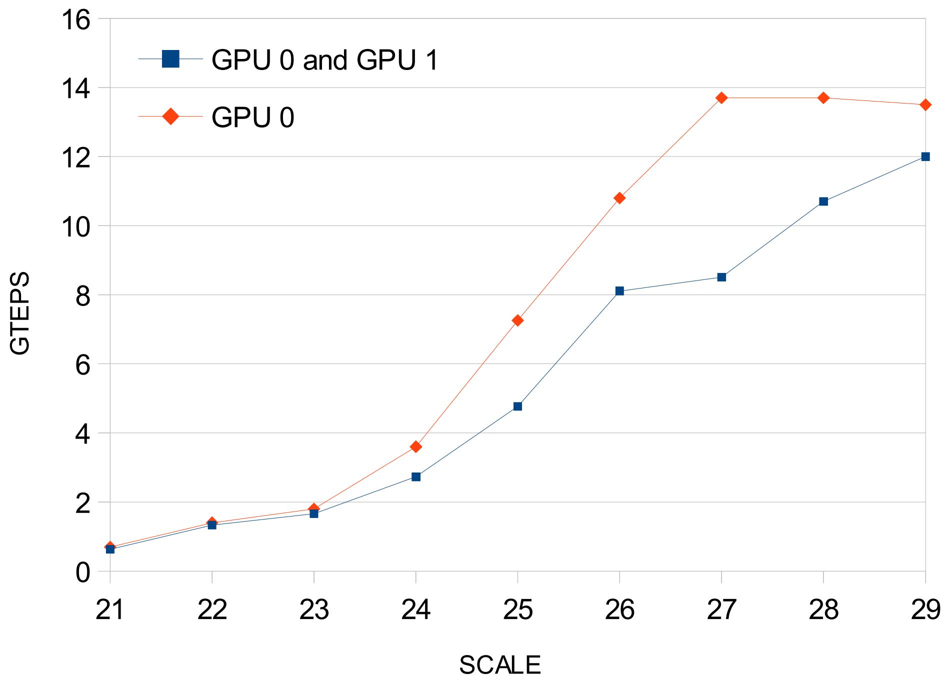


Figure 5.6: Full node GPUs *vs* GPU 0. The GPU 0 implementation not use the QPI link of the two CPU socket.

5.1.5 Conclusions

In this study we present an optimized implementation of the Graph500 benchmark for the ROMEO multi-GPU cluster. 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 representation of graphs using both CSR and CSC. Although this representation requires more memoriy, 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 will be published in the next Graph500 list. With a total of 13.70 GTEPS the ROMEO supercomputer could be ranked at the 91th position.

Today we can identify some interesting perspectives to carry on the study. Communication cost is the major limitation and a better control of load distribution is needed between communication and computation in order to obtain even better performance. Part of the solution might come from new technologies developed by Nvidia, such as the new PASCAL architecture or NVlink buses.

5.2 Conclusion

Part III

Application

Chapter 6

Physics and Astrophysics

6.1 Fluids

6.2 Binary Neutron Stars

Chapter 7

FleCSPH

7.1 FleCSI

7.2 FleCSPH applications

Conclusion

‘

Annexes

Bibliography

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

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

[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.

[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).

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

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

[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.

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

[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.

[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.

[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.

[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.

[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.

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