THÈSE

présentée à

l'Université de Lille 1

pour optenir le titre de

Docteur en Informatique

par

Maxime Hugues

Un Paradigme de Programmation Multi-Niveaux pour le Calcul Numérique sur les Machines Post-Petascales et Exascales

Thèse soutenue le 13 Mai 2011 devant la commission d'examen

Membres du Jury :

Président:	Nouredine Melab	Professeur Université de Lille 1, France
Rapporteurs:	Mitsuhisa Sato	Professeur Université de Tsukuba, Japon
	Michel Dayde	Professeur ENSEEIHT, France
Examinateurs:	Peter Arbenz	Professeur ETH Zürich, Suisse
	Henri Calandra	Ingénieur Expert TOTAL
Invité:	Jean-Patrick Mascomere	TOTAL
Directeur de thèse:	Serge Petiton	Professeur Université de Lille 1, France

Ph. D. THESIS

presented at

University of Lille 1

to obtain the title of

Doctor of Philosophy in Computer Science

Defended by

Maxime Hugues

A Multi-Level Programming Paradigm for Numerical Computing on Post-Petascale and Exascale Machines

Defended on May 13^{th} 2011 to the Committee

Committee :

Head of Committee:	Nouredine Melab	Professor University of Lille 1, France
Reviewers:	Mitsuhisa Sato	Professor University of Tsukuba, Japan
	Michel Dayde	Professor ENSEEIHT, France
Defense Committee:	Peter Arbenz	Professor ETH Zürich, Switzerland
	Henri Calandra	Expert Engineer TOTAL
Guest:	Jean-Patrick Mascomere	TOTAL
PhD Advisor:	Serge Petiton	Professor University Lille 1, France

To my parents André and Louise-Marie for their support and their love

Acknowledgments

During the last three years, I have enhanced my knowledge in computer science but also my professional and personal experience. I have no regrets to have done this thesis in front of the tremendous contribution to myself. Therefore, nothing could have been possible without the funding of TOTAL that I want to thank in the name of Pascal Dauboin which gives me this opportunity. I would like also to thank Jean-Patrick Mascomère who was my manager during these 3 years at TOTAL, for his advices on the problem analysis and from his industry point of view. I thank also Henri Calandra with which I have worked. He has trusted me and offered the opportunity to spend 6 months in the US to participate in a collaboration project with DataDirect Networks. I acknowledge my Ph.D. advisor Prof. Serge Petiton. I have really appreciated to work with him for his management with liberty and for his precious advices. He has trusted me and I have trusted him. I hope to have been at the measure of his expectations.

I express my recognition to Michel Daydé and Mitsuhisa Sato who have accepted to review my Ph.D. dissertation. I thank also all the members of my committee, Peter Arbenz, Nouredine Melab, Henri Calandra and Jean-Patrick Mascomère. I am very grateful and honored of their presence.

Finally, I would like to thank Matthieu Brucher and Rached Abdelkhalek for our discussions at Total. My greetings are also dedicated to all the guys at DataDirect Networks and especially to Kishore Ramaswamy for his precious help during my stay in the US and Mike Moretti for his efficiency at work and his pleasantness. I thank also a previous member of the MAP team which works now for TOTAL, Laurent Choy, for his precious advices and his patience. I have enjoyed working with him during my master of science. I express also my greetings to Ye Zhang and Ling Shang of the MAP team with which I have shared a good time and have enjoyed to work with. The last but not the least, I would like to thank Pierre-Yves Aquilanti from the MAP team. I have enjoyed to collaborate with him at TOTAL. I thank him for his kindness, our discussions on computer science and other things. Above all, thank you Pierre-Yves to bear me during those 3 years and for the friendship that we have created. And hopefully that will continue.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

High performance computing has been introduced many years ago to accelerate the solution of scientific and engineering problems by modelizing and simulating them on supercomputers. Machines were initially dedicated to defense purposes and were using specific processors. Performances were after raised by the architecture and hardware enhancement of processor which are able to execute simultaneously multiple instructions. That kind of chip are superscalar processors and had the ability to reach a performance over Mflop/s. Another type of processor was also used at this time, the vector processor which executes one instruction on a range of data. Hundred and thousand of processors were after associated to reach a performance order of Tflop/s and to have "parallel computing". Architecture improvement and the frequency increase have mainly contributed to the throughput growth. However, frequencies have hit the wall and cannot be increased any more because of energy limit on a chip and quantum effect. This has conducted processors manufacturer to multiply cores on a chip in order to continue the performance evolution which has lead to create a new level of parallelism inside the chips. Pflop/s barrier has been broken since 2008 with RoadRunner which is an hybrid supercomputer composed of multicore superscalar processors and heterogeneous processors mixing a superscalar core and vector cores. This performance range era has opened new possibilities to make scientific breakthrough and accelerate the solution of engineering problems which were not computable in an acceptable time before. Many areas such as oil and gas, aircraft design or domains like physics, biology, weather science for the climate changes and so on use high performance computing and exascale supercomputer are now on the way. Many scientists are thrilled to apply this computing power to solve complex and large problems and make new breakthrough.

Nevertheless, many petaflops and excascale supercomputers challenge computer sci-

entists by many aspects. These kind of machines will be massively multicore, heterogeneous with GPU, FPGA, Many-Integrated Core or vector processor and large with a high number of cores, average of a billion, which introduces many issues such as fault-tolerance, energy consumption, storage performance and programming. The failure probability will be more important on that type of machine because they will be very large. Fault-tolerant mechanisms must be supported at every level from the operating system, the programming model up to the algorithm in order to run large applications and re-execute a failed tasks and not all the application [1]. Energy consumption of exascale supercomputers may be also a problem with an order of 25MW to 100MW with conventional processor, following the projection of IESP. It is not conceivable that high performance computing centers have the budget to pay for it or the adequate infrastructure to deliver such power. Some solutions have to be explored to reduce energy consumption like the frequency decrease of non-used core or the optimization of data movement [2]. Storage is also a critical aspect of the next generation which will need an order of half an exabyte and a high bandwidth of hundred TB/s. To get these performances, I/O systems must be scalable and guaranty availability when millions of cores access at the same time the storage. If these points can be solved, the problem of how to program that kind of supercomputers remains in order to reach the maximum performance efficiency of the machine with an application. The other issue is how to manage many millions of threads running in parallel with the classical programming paradigm MPI. The use of accelerator should also introduced heterogeneity that must be taken into account and is not the case in the actual paradigm. All these issues mainly challenge computer scientists to find a way to make concrete the use and the running of applications for new scientific breakthrough on exascale supercomputers.

1.2 Motivation

Many issues of Exascale supercomputers must be faced and some are critical such fault-tolerance and power consumption which are currently attacked for 10 Petaflop/s systems. Fault-tolerant mechanisms are integrated in the third standard of the common parallel programming model MPI and computation accelerators are added to get a better flops/W ratio. To exploit this huge computation power, applications must be split among processors to process in parallel sets of data. During the last decade, two standards of programming models have been adopted to develop and run parallel applications such as MPI for distributed memory and OpenMP for shared memory machines.

Programming Exaflops supercomputers with billion cores and heterogeneous hardwares will be a problem with these common models [3]. A flat-MPI approach induces to map one MPI process per core and it will be hard to manage billion of processes and get the highest efficiency. However, the programming of million processes should

be possible with some communication optimizations as described in [4] [5]. These papers also underline that the scaling of global communications such as AllToAll, AllGather,AllGatherv will suffer of latency issues and be not conceivable to get high performances. The other issue is that MPI does not support heterogeneity which is necessary with accelerator such as GPU that has a different programming paradigm. MPI does not also consider the memory hierarchy of multicore processors. OpenMP is dedicated to shared memory architecture and should be more adapted for multicore. Despite this fact, a flat-MPI approach is in the most of cases more efficient than OpenMP [6]. The reason of poor performances is the lack of modularity and expression of locality optimizations which are crucial on multicore to minimize data movements and cache defaults. An other problem of OpenMP is the doubt about the scaling on 100-1000 cores processors of this programming model. Some researchers propose for machines beyond 10 Petaflops a mixed programming paradigm of MPI+OpenMP. This approach is not adapted for reasons that have been underlined but also for non support and convenient programming of accelerators with OpenMP and for the low level programming of MPI which involves a lack productivity. As we have explained, many issues must be solved before running Exascale supercomputers and programming paradigm is the third critical aspects to figure out with resilience and energy. Both are also related to the programming paradigm which has to support fault-tolerance mechanisms and energy consumption awareness.

The architecture of Exascale supercomputers is going to be different from known systems and organized in a hardware hierarchy which has its own distinctiveness of efficiency. The proposed programming paradigm for exaflops must harness those particularities at each level. The massively multicore processors, including accelerator, are going to be the base of computation nodes and thus be at the low level of the system with 100 to 1000 cores per each. They have memories organized in hierarchy of caches with low latency and a main memory shared among cores with a high latency. Hence, data movements are crucial parameters to optimize in order to take advantage of high speed and low latency of caches and avoid cache misses which generate a bottleneck to the shared memory accessed in competition by cores. Data locality and movements are thus a relevant criteria that must be considered to get the most efficiency of massively multicore processors. The interconnection between nodes is at an intermediate level of the hierarchy. This interconnection encompasses various speeds inside the node with a fast optical interconnect of 1TB/s between sockets and offside the node with the interconnect topology of 10GB/s. This difference must be considered to optimize point-to-point communications and avoid the contribution of other nodes in the network that degrades latency and bandwidth beyond one hop. The large cluster composed of a huge number of nodes represents the high level of the hardware hierarchy and should be around an average of 10 millions nodes to reach the

billion cores for the total system. The orchestration and the load balancing of tasks on this cluster are necessary to avoid lazy synchronizations and communications which can be improved by aggregating and making them in an asynchronous way.

In this thesis funded by the oil and gas company Total, we have been attached to take into account this hierarchy by proposing a global programming paradigm based on a multi-level paradigm approach which is hierarchic and convenient from threads of massively multicore processors to a large cluster. This topic is highly important for a company like Total in order to start figuring out on the coming big turns with exascale computing, especially programming paradigm which involves to educate users and re-develop parts of applications. The purpose for the company is to get an idea on interesting paradigms. According to the hardware hierarchy, each level is examined with a proposed paradigm that corresponds to the specific features and the ideal efficiency. In the next chapter, the state-of-art on programming paradigm is presented and explains the advantage and inconvenient by trying to match requirements of a level. The chapter 3 starts exploring the multi-paradigm proposal by focusing on a paradigm adapted to massively multicore processor also known as many-cores to encompass conventional CPU and accelerators. Issues of massively multicore processors are firstly outlined to lead to the proposition of a convenient paradigm which is explored through experiments. The middle level in the hierarchy is not presented because point-to-point communications are very well known through the message passing paradigm. Nevertheless, the penultimate chapter deeply explains the place of this paradigm and its interaction with others. The chapter 4 attacks the last and highest level in the programming hierarchy devoted to the huge number of nodes. It exposes the issue of this large parallelism and outlines the proposed paradigm through its advantage for productivity based on users expertise and some experiments. This user knowledge and expertise are after explored in the chapter 5 around the context of storage and their possible integration into a high programming paradigm to realize optimized I/O in a delegated and transparent way for end-users. After proposing and studying separately the levels, issues of exascale are reminded and are viewed alongside our multi-paradigm programming proposal in the chapter 6. An assembly of the programming stages is realized to demonstrate the feasibility of the approach and a discussion is made on results. To conclude, the last chapter summarizes the contribution of this thesis and proposes some future works to extend this work through a discussion on exascale programming paradigm adoption.

Chapter 2

Paradigm: State of the art

2.1 Message Passing

This programming paradigm has been created for parallel architectures with distributed memory. Those machines have many processors with their own memory and communicate through the network. These interaction are managed by programmers which distribute computations and data among the various processors. To realize communications between the different processing elements, messages are exchanged between a sending and a receiving process. Those processes are identified by the programmer at the development stage. The first Message Passing library was the Parallel Virtual Machine library (PVM) and has been replaced by the well known Message Passing Interface (MPI). The execution model is Single Program Multiple Data (SPMD) which traduces the writing and execution of one program executed by all processors. Different sets of data are assigned and processed on each processor following the rank of the compute element.

2.2 Multi-thread

The multi-thread programming paradigm is also named concurrent model for shared memory and is mainly used for Symmetric Multi-Processing (SMP) machines which have a shared memory between processors. OpenMP is the programming standard for the programming of shared memory machines. This programming paradigm allows to express task parallelism and data parallelism without any locality control. The execution model is named FORK/JOIN in which threads are launched (FORK) to run computation kernel and a barrier synchronization is made to wait the end of all threads (JOIN). Programmer expresses parallelism with *pragma* annotation which are translated by the compiler in parallel directive. Some other language like Cilk [7] uses keyword to express parallelism. This programming paradigm achieves good performances on shared

memory architecture but lack of locality control to realize communications between processors.

2.3 Data Parallel

The data parallel paradigm has been introduced with the massively parallel computing and SIMD machines (Single Instruction Multiple Data) like the CM-5. This programming model allows to express data parallelism. It is focused on data mapping and movements during the development stage. Data are distributed across numerous processors which execute one instruction or operation on a set of data. One of the most known data parallel language is High Performance Fortran (HPF) [8] which wanted to be the successor of Fortran 90. Unfortunately, this language has not been popular because the first version of the compiler was disappointing anf offered poor perfomance which has not appeal to users. Other data parallel programming languages exist like CM-Fortran dedicated to Connection Machine or Data Parallel C Extension (DPCE) or C* [9]. Some other data parallel language are based on a different data parallelism, named Nested Data-Parallelism like NESL [10]. It allows to run a function on a set of values and this function may call another parallel function. Data parallelism has the interest to avoid race conditions or deadlocks which are the most common errors in parallel programming.

2.4 Stream

The stream programming paradigm is usually used in image and video processing. It is based on computation kernels which are run on a stream of data. The parallelism is expressed in pipeline between the computation kernels which are dependent. This paradigm thus exploits data parallelism but also task parallelism for tasks which are not dependent. Moreover, it has the particularity to reuse the local memory of processors and thus to minimize the memory bandwidth. StreamIt [11], Stream-C, Brook [12] are some examples of language which use this programming paradigm.

2.5 Workflow or Graph Description

The workflow paradigm allows to describe dependences between tasks through a graph which represents the control-flow. Each node of the graph is a task and the edge represent task dependences. The parallelism description and the computation are thus segregated. DagMan [13], UNICORE [14] and GridFlow [15] are some projects of workflow framework which are based on Directed Acyclic Graph (DAG) model). In contrast, YML [16] is a framework based on a graph description language which adopts a Di-

rected General Graph model. This graph model allows to express loops, iterations and branching in opposite of other workflow framework. Those approaches are typically for clusters. However, graph based on DAG are recently proposed and used on multicore processors and accelerator such as PLASMA [17] and MAGMA [18].

2.6 Dataflow

Such as workflow, the dataflow programming paradigm allows to express dependences through a graph description. However, the dataflow describes the flow of data and their respective operations. Each node of the graph is a data and edges represent operations. This paradigm is a fine grained parallelism and data oriented which induces a larger graph to process when the amount of data and operations are huge. One of the most known dataflow language is SISAL [19]. The StreamIt language is also associated with the dataflow paradigm because it analyzes the dataflow to manage data locality and transfer.

2.7 Partitioned Global Address Space

The partitioned global address space (PGAS) programming paradigm has the property to consider distributed memory of nodes as a global address space available from each node. It has been developed in the purpose to have a paradigm with the knowledge of data locality. The application parallelization is realized with threads which have a part of the address space. The programmer is in charge to manage data distribution between processors and to express communications through *get* and *put* operations to send or retrieve data. These communications are one-sided communications which offers to avoid CPU participation by a direct access to the remote memory. Two libraries GASNet [20] and ARMCI [21] are used by PGAS language and used for communications. The well known language and candidate for exascale programming are Unified Parallel C (UPC) [22] for C language developed at University of Berkeley, Co-Array Fortran (CAF) [23] for Fortran developed at Rice University and Titanium [24] for Java language developed at University of Berkeley. Other language such as X10 [25] and Chapel [26] are based PGAS programming paradigm but are at a higher level because communications are not expressed by end-user and generated by the compiler. XcalableMP (XMP) [27] is also a PGAS language and standard which proposes to express parallelism with annotation and should improve productivity.

2.8 Object Oriented

In contrast to the message passing programming paradigm which exchanges messages between processes, the object oriented programming paradigm exchanges information between objects. The programmer is in charge to create, manage objects and coordinate interactions between them. Cantor [28], Emerald [29], COOL [30] and Orca [31] are some of the first languages to implement this programming paradigm. Charm++ [32] is also a language based on this paradigm in which parallelism is expressed through threads.

2.9 MapReduce

The MapReduce programming paradigm has been introduced by Google to realize computations on a large set of data and on millions of computing nodes [33]. It takes its inspiration from functional language and intensive computing. Only two functions are available to the programmer, *Map* which generates a key/value pair by analyzing input data and *Reduce* which reduces the set of same values in a list which is identified by an unique key. The program is automatically parallelized and scheduled on a distributed system which takes in account the data locality. This programming paradigm is also currently experimented for multicore programming [34].

2.10 Bulk Synchronous Parallel

The Bulk Synchronous Parallel (BSP) [35] paradigm is a programming paradigm in which communications property among threads are taken into account. These properties are three of kinds: p the number of processors, l the synchronization time of a barrier synchronization and g the communication time to exchange data over the network. A program based on this paradigm is composed of p threads and divided into supersteps. Each superstep executes a computation on a processor by using its local data. After each processor sends its computation results to other processors and a barrier synchronization is realized. So data are available locally for each processor. H-BSP [36] is an example of implement of the BSP programming paradigm.

2.11 Transactional Memory

Transactional Memory is a control paradigm of concurrency for execution of atomic or isolate operation. It is inspired from transactional data base for which concurrent accesses and failure are frequent. For multicore processors, some researchers propose this paradigm with some adaptations for software, named Software Transactional Memory (STM) [37]. One transaction is composed of three steps:

- an initialization step: BeginTransaction
- a read / write step
- a finalizing step: EndTransaction

During the transaction, the data may be only accessed by the controlling thread which has begun the transaction.

Chapter 3

Data Parallelism on Many-Core

3.1 Introduction

Performance progression of processors was due to the improvement of hardware design to execute more operations per clock cycle and mainly to the clock frequency growth to speed computation processing. The increase of this parameter impacts the power consumption of processor that states that has reached its acceptable limit and conducted to stop the usual performance progression by hitting the "frequency wall". Since the Moore's law is still true doubling transistor every 18 months and pursue the performance improvement, chip makers have found a solution to this issue by multiplying processing cores on a chip. These cores were firstly segregated from each other with their own small and fast memory, named cache. Different levels of cache were after added across cores in order to share data between and to avoid frequent accesses to the main memory by reusing data available in cache levels. However, the non-exploitation of these levels leads to cache misses and a slower data access through the main memory. The non-optimized use of cache levels also leads to a more frequently competition between cores to send or retrieve data from the main memory which creates a bandwidth bottleneck, known as the "memory wall". All of these may appear billion times during computations and contributes severely to slow down global performance on processors.

Exascale supercomputers will be composed of massively multicore processors beyond 100 cores per chip for conventional processors (x86, IBM Power) or 1000 cores for accelerators and their performance efficiency are the most significant factor to reach Exaflops. These have the general term of *many-core* processors. For massively multicore processors, data mapping and movements on the chip become highly important in order to take advantage of different cache levels and to optimize the main memory bandwidth which is a critical parameter. Common parallel programming such

as MPI or OpenMP for shared memory have not a data-centric programming and do not offer the possibility to use efficiently specific architecture features of many-core. Programming paradigm of massively multicore should change and this thesis proposes data parallel programming as a good candidate for the lower level of the hardware hierarchy in an Exascale supercomputer. It is a data-centric paradigm by focusing data mapping and data movements and avoids by nature usual parallel programming issues like race conditions and deadlocks.

Graphic Processing Unit (GPU) have made the rebirth and the interest of data parallel programming on massively multicore processors. More recently some initiatives propose data parallel extension such as HTA (Hierarchic Tiled Array) [38] or Ct from Intel [39]. In order to study data parallelism on many-core, GPU is taken as platform because it is the most massively multicore chip available on the market and it is of a high interest for the HPC community. GPU is different in some ways from multicore such as one instruction unit for multiple cores in opposite to one per core for usual multicore. However, GPU has some common features that corresponds of future many-core architecture like the different cache levels with a fast access and a slow global memory which is accessed by multiple cores at a time. Data parallel programming should help to take advantage of cache levels but some computations are more difficult to optimize with this programming paradigm such as sparse matrix computations. They have irregular data structure and generate a memory bottleneck by accessing data in a non-contiguous fashion which make them challenging to implement on GPU.

This chapter is thus focused on sparse matrix computations and mainly on the optimization of the sparse matrix vector product (SpMV) which is the most used kernel in sparse computing. It is also the main performance key of iterative methods that interest and are explored at TOTAL for seismic imaging and reservoir characterization. The main optimization strategy to achieve best performances for sparse matrix computations is to reduce main memory accesses and to make efficient data movements by arranging data in a convenient manner. Sparse matrix formats define this arrangement and are strongly involved in the performance achievement. In the following, sparse matrix formats for SpMV are implemented and studied with a data parallel paradigm on GPU in order to propose optimizations and examine performance of successful sparse matrix formats on previous data parallel machines.

3.2 GPU Architecture

The work presented here is based on a Tesla T10P GPU architecture. It has a peak performance of 1 TFlops in single precision (SP) and 86 GFlops in double precision (DP). GPU has 240 cores which are spread among 30 multiprocessors, named *streaming pro-*

cessor (SMs). Each has 8 *scalar processors* (SPs), two special function units (SFUs) dedicated to perform specific mathematical operations and a multi-threader instruction unit. Different memories are dispatched hierarchically on GPU and have various latencies. All multiprocessors share at a high level the *global memory* which is an off-chip "large memory" of 4GB, with a 512-bit interface, a 102GB/s bandwidth and a high latency of 400-600 clock cycles. At a lower level, scalar processors of a SM share also resources such as 16384 32-bit registers with a low latency of two clock cycles, 16KB of *shared memory* organized in 16 banks and cache for *constant* and *texture* memories. Constant and texture memories are read-only regions of the global memory which offers to speed up reads. The Figure 3.1 gives an overview of the T10P architecture organization.

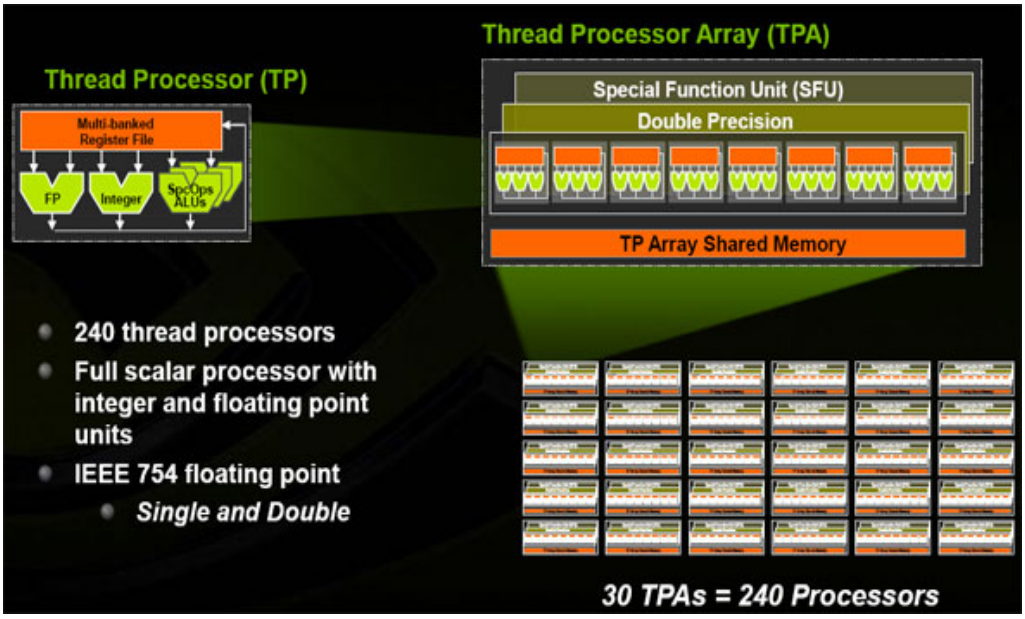

Figure 3.1: T10 GPU Achitecture

3.3 Programming and Performance Keys

The application programming on this architecture consists of a sequential program executed on a host CPU which launches kernels written with the *Compute Unified Device Architecture* (CUDA) technology on GPU. A kernel is run in parallel on a grid of threads which are executed in a *Single Instruction Multiple Threads* (SIMT) and share access to the global memory. Threads are organized into groups, named *thread block* which have access to a shared memory space and are synchronized through the call of a

barrier function. Each thread has its own local memory and register space. Each group and each thread of a group have an unique identification, respectively named *blockIdx* for a group and *threadIdx* for a thread of a group that allow to attribute different set of data to compute per thread. Threads of blocks are managed and mapped on each scalar processor by the multiprocessor and are executed into groups of 32 threads, called *warps*. A single instruction at a time is performed by all threads of a warp. The different latencies underline the memory access are strategic parameters. They can be optimized according to some guidelines given in the CUDA Programming Guide [40]. All threads of a warp access the global memory with arbitrary addresses that may run to multiple memory transactions. To optimize these transactions, memory accesses have to be coalesced which is realized when a *half-warp*, i.e. 16 threads, reads or writes a 128-bytes segment in the global memory. It results in one memory transaction if threads of the half-warp address one segment. Memory transactions are also reduced with the use of *built-in vector types*. They offer the ability to read or write 32, 64, 128 bit words in the global memory in one memory transaction. Both optimizations are the most important to get maximum memory bandwidth and reach best performances.

3.4 Sparse Matrix Computations

Sparse matrix computations have irregular data structures and accesses which get low throughput without any optimizations. Sparse matrix formats define the data organization in such way that get the maximum performance. These formats have been refined during the last years on various type of architecture such as vector processor or data parallel machines [41]. Different computation kernels have also been tailored in relation to sparse formats and especially the SpMV which is mainly used in iterative methods. The arriving of many-core processors has modified the chip architecture and the programming paradigm that make interesting the efficiency re-examination of sparse formats and SpMV. Before explaining our implementations and optimizations of sparse formats on GPU, some related works of sparse matrix computations on multicore are outlined and in the following common sparse matrix formats are listed.

3.4.1 Related Works

Many researches have been conducted to optimize this kernel implementation through various sparse formats on many-core processors. On one core, the OSKI library [42] provides auto-tuned computation kernels for sparse matrix computations which take into account the particular cache size of the processors. On multicore processors, the Block Compressed Sparse Row (BCSR) format has been evaluated and optimized on different platforms like Cell BE, Intel, AMD and Sun processors [43]. It clearly achieves the best performances by reusing sparse matrix elements in the different cache levels

thanks to the register and cache blocking techniques. These techniques allow to reuse data in cache and register levels and in consequence to optimize the use of main memory bandwidth. On massively parallel architectures, some implementations of SpMV kernel have been evaluated by using various sparse formats on NVIDIA GPUs [44]. They have shown that their hybrid format (HYB) achieves the best performances on set of matrices previously used on multicore processors. This format is a mix between Ellpack/ITpack format which uses full-coalescing and the Coordinated format for irregular part of the matrix. They have also demonstrated diagonal sparse format (DIA) and Ellpack/ITpack format are the most efficient sparse formats for finite differences discretization of the Laplacian. In our experiments, we propose a variation of the Ellpack/ITpack format which uses vectorization and a row-major order of elements that corresponds to the real data organization of 2D array in C language. In [45], authors have also proposed an optimized implementation of the Compressed Sparse Row (CSR) format by using vectorization and padding in order to get a multiple of 16 (half-warp) per row and a partial-coalescing. Their implementation increases the efficiency and outperforms the CSR vectorized version of Nvidia. A sparse linear solver [46] method has been developed on GPU and proposed the implementation of the SpMV by using the BCSR format for register blocking. In our evaluation, we have extended this format with vectorization, implemented two blocking version and compared with set of matrices in order to position performances of this sparse format. We also propose implementation and comparison of column sparse formats and sparse formats which has succeed on data parallel machines, like Connection Machine.

3.4.2 Sparse Matrix Formats

A wide variety of sparse matrix formats exists and have been adapted depending of the application problem and processor architecture. This subsection outlines common sparse matrix formats that are supported by the SPARSKIT library [47] and are mostly convenient to general/many matrices for SpMV ($y \leftarrow Ax$) where A is a matrix $m \times n$. Some formats that have been successful on data parallel machines are also presented .

3.4.2.1 Compressed Sparse Row Format

One of the most used formats in sparse matrix applications is the Compressed Sparse Row format (CSR), illustrated on Figure 3.2. This format compresses each row of a matrix A and stores the non-zero values (nnz) in an array *values*. The column information is thus lost for each element in this array. In order to keep column index, an array *columns* is created and stores column index of data. A last array *rowIndex* stores the index of the element in the array *values* which is the first data of a row. The array size is $m + 1$ in which the number of nnz is stored at the last element. An implementation of the CSR format for the SpMV is given in the Algorithm 3.1.

$$A = \begin{bmatrix} 11 & 12 & 0 & 2 \\ 0 & 6 & 0 & 8 \\ 0 & 0 & 0 & 3 \\ 13 & 0 & 14 & 16 \end{bmatrix}$$

$$\begin{aligned} values &= \begin{bmatrix} 11 & 12 & 2 & 6 & 8 & 3 & 13 & 14 & 16 \end{bmatrix} \\ columns &= \begin{bmatrix} 0 & 1 & 3 & 1 & 3 & 3 & 0 & 2 & 3 \end{bmatrix} \\ rowIndex &= \begin{bmatrix} 0 & 3 & 5 & 6 & 9 \end{bmatrix} \end{aligned}$$

Figure 3.2: CSR Format

```
for i = 0 to m − 1 do
  for j = rowIndex[i] to rowIndex[i + 1] − 1 do
    y[i] ← y[i] + values[j] × x[columns[j]]
  end for
end for
```

Algorithm 3.1: Sparse matrix vector product implementation for the compressed sparse row format

3.4.2.2 Compressed Sparse Column Format

The analog format of CSR is the Compressed Sparse Column (CSC), illustrated on Figure 3.3. In opposite of the CSR, it compresses each column of a matrix. The format has also three arrays such as array *values* in which the non-zero values are stored. The second array *rows* stores the row index of each element. The third array *columnIndex* stores the index from the array *values* of the first column element. The size of this array is the number of $n + 1$ in which this index the number of nnz is stored. An implementation of the CSC format for the SpMV is given in the Algorithm 3.2.

3.4.2.3 Block Compressed Sparse Row Format

The Block Compressed Sparse Row format (BCSR) is a blocked variant of the CSR which offers the possibility to take advantage of the register by blocking computations. This format is illustrated on Figure 3.4. The matrix A is divided into block rows of dimension $r \times c$ and each dense block row is stored consecutively in an array *values*. For each block in this array, the first column index is saved in an array *ind* in order

$$A = \begin{bmatrix} 11 & 12 & 0 & 2 \\ 0 & 6 & 0 & 8 \\ 0 & 0 & 0 & 3 \\ 13 & 0 & 14 & 16 \end{bmatrix}$$

$$\begin{aligned} values &= \begin{bmatrix} 11 & 13 & 12 & 6 & 14 & 2 & 8 & 3 & 16 \end{bmatrix} \\ rows &= \begin{bmatrix} 0 & 3 & 0 & 1 & 3 & 0 & 1 & 2 & 3 \end{bmatrix} \\ columnIndex &= \begin{bmatrix} 0 & 2 & 4 & 5 & 9 \end{bmatrix} \end{aligned}$$

Figure 3.3: CSC Format

for $i = 0$ to $n - 1$ **do**
  for $j = columnIndex[i]$ to $columnIndex[i + 1] - 1$ **do**
    $y[rows[j]] \leftarrow y[rows[j]] + values[j] \times x[i]$
  end for
end for

Algorithm 3.2: Sparse matrix vector product implementation for the compressed sparse column format

$$A = \begin{bmatrix} 11 & 12 & 0 & 2 \\ 0 & 6 & 0 & 8 \\ 0 & 0 & 0 & 3 \\ 13 & 0 & 14 & 16 \end{bmatrix}$$

$$\begin{aligned} values &= \begin{bmatrix} 11 & 12 & 0 & 2 & 0 & 0 & 0 & 3 \\ 0 & 6 & 0 & 8 & 13 & 0 & 14 & 16 \end{bmatrix} \\ ind &= \begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix} \\ ptr &= \begin{bmatrix} 0 & 2 & 4 \end{bmatrix} \end{aligned}$$

Figure 3.4: BCSR Format with $r \times c = 2 \times 2$

to find the column index of each element. A last array *ptr* is dedicated to store the index in *values* of the first block in each row. The blocks are considered as dense and incomplete blocks are filled with explicit zeros. An implementation of the 2×2 BCSR format for the SpMV is given in Algorithm 3.3.

for $i = 0$ to $(m/r) - 1$ **do**
  for $j = ptr[i]$ to $ptr[i+1] - 1$ **do**
    $col \leftarrow ind[j]$
    $y[i] \leftarrow y[i] + values[j] \times x[col] + values[j+1] \times x[col+1]$
    $y[i+1] \leftarrow y[i+1] + values[j+2] \times x[col+2] + values[j+3] \times x[col+3]$
  end for
end for

Algorithm 3.3: Sparse matrix vector product implementation for the block compressed sparse row format with $r \times c = 2 \times 2$

3.4.2.4 ELLPACK / ITPACK Format

$$A = \begin{bmatrix} 11 & 12 & 0 & 2 \\ 0 & 6 & 0 & 8 \\ 0 & 0 & 0 & 3 \\ 13 & 0 & 14 & 16 \end{bmatrix}$$

$$values = \begin{bmatrix} 11 & 12 & 2 \\ 6 & 8 & 0 \\ 3 & 0 & 0 \\ 13 & 14 & 16 \end{bmatrix} \quad indice = \begin{bmatrix} 0 & 1 & 3 \\ 1 & 3 & - \\ 3 & - & - \\ 0 & 2 & 3 \end{bmatrix}$$

Figure 3.5: ELLPACK/ITPACK Format

The format ELLPACK/ITPACK (ELL), see Figure 3.5 is a well-suited format for vector architectures and matrices which have approximately the same number of elements per row. The format is composed of two 2D arrays *values* and *indice* of dimension $m \times s$ where s is the maximum number of element in a row. The first array stores each compressed row of A in a row of value and the second array stores the corresponding column of each element. A rows with fewer elements than s is padded with some extra zeros which implies an extra storage. An implementation of the Ellpack/ITpack format for the SpMV is given in the Algorithm 3.4.

```
for i = 0 to m − 1 do
  for j = 0 to s − 1 do
    y[i] ← y[i] + values[i][j] × x[indice[i][j]]
  end for
end for
```

Algorithm 3.4: Sparse matrix vector product implementation for the Ellpack/ITpack sparse format

3.4.2.5 Sparse General Pattern Format

$$A = \begin{bmatrix} 11 & 12 & 0 & 2 \\ 0 & 6 & 0 & 8 \\ 0 & 0 & 0 & 3 \\ 13 & 0 & 14 & 16 \end{bmatrix}$$

$$values = \begin{bmatrix} 11 & 12 & 14 & 2 \\ 13 & 6 & 0 & 8 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 16 \end{bmatrix} \quad rowIndex = \begin{bmatrix} 0 & 0 & 3 & 0 \\ 3 & 1 & - & 1 \\ - & - & - & 2 \\ - & - & - & 3 \end{bmatrix}$$

$$index = \begin{bmatrix} 0 & 1 & 1 & 2 \\ 0 & 0 & - & 1 \\ - & - & - & 0 \\ - & - & - & 0 \end{bmatrix}$$

Figure 3.6: SGP Format

The last format of this evaluation is the Sparse General Pattern (SGP) [48] which has succeeded on Connection Machine. The Figure 3.6 gives an example of SGP format compression. It is composed of three 2D arrays of dimension $nc \times n$ where nc is the maximum number of element in a column. The first array *values* stores the nnz of each compressed column and the second array *rowIndex* stores the row index of each element in *values*. The third array *index* stores the position index of each element in a row. For example, for the 2 value of a row composed of 4 elements, 1 is stored in the array *index*. This last array helps to change the matrix representation from a matrix column to a compressed row matrix. An implementation of the SGP format for the SpMV is given in the Algorithm 3.5.

```
for i = 0 to nc − 1 do
  for j = 0 to n − 1 do
    tmp[i][j] ← values[i][j] × x[j]
  end for
end for
for i = 0 to nc − 1 do
  for j = 0 to n − 1 do
    tmp[i][j] ← tmp[rowIndex[i][j]][index[i][j]]
  end for
end for
for i = 0 to nc − 1 do
  for j = 0 to n − 1 do
    y[i] ← y[i] + tmp[i][j]
  end for
end for
```

Algorithm 3.5: Sparse matrix vector product implementation for the sparse general pattern format

3.4.3 Implementations and Optimizations of Sparse Formats

Many sparse formats exist and have been optimized during the last years [47] [41]. They define the data structure of matrices and thus play an important part in achieving the best performance. On distributed computers, the critical parameter is communications which depend on data mapping on nodes, induced by the format. On past data parallel machines such as CM-5, communications were also an important parameter because of the computation units were interconnected through a high performance network. The GPU is a data parallel architecture which has made the rebirth of data parallel computing. The difference with the CM-5 is has a large shared memory between cores. As we have explained in the previous section, the tricky point is the minimization of transactions to access data in the global memory. We present in this section some implementations of usual formats and others which have greatly succeeded on this type of architecture for the SpMV ($y \leftarrow Ax$) where A is a matrix $m \times n$.

3.4.3.1 CSR

Two implementations of this formats are proposed. The first one is a naive parallelization of the SpMV which assigns one thread to compute a matrix row, named CSR. The second one is a vectorized version that has been proposed by Bell and Garland [44]. It allows to take advantage of coalescing by vectorizing read accesses to the array *values*. The vectorization is realized by using a warp to compute a matrix row and thus process 32 elements of a row at a time. Our implementation named CSR_Vec is different by

assigning an half-warp to the vectorization and by adding extra zeros to each row. This guarantees an entire half-warp execution and a better coalescing to realize in one memory transaction the read of 16 elements. After this step, partial-results are obtained and put into the shared memory. A parallel reduction is performed to sum all partial-results among threads of a half-warp. Then the result of each row is put in the vector y by the first thread of the half-warp. Our implementation also uses caching by mapping the vector x in the texture memory of the GPU in order to reduce the access time to it.

3.4.3.2 CSC

The analogous format to CSR with column compression storage, the CSC format, has also benefited of two implementations with and without vectorization, named respectively CSC and CSC_Vec. The first one assigns one thread to compute one matrix column. The second one is composed of two computation steps. The first step computes all products between the array *values* and the vector x with an half-warp. Then, the results are put in a temporary array which thus contains the product operation of each column element. The second step sums all data in the temporary array which belong to the same row. To make this operation, a fourth array is previously constructed and is organized by row which contains the index of an element in *values* belonging to the same row. The CSC format has the particularity to map one element of the vector x per thread or half-warp which allows to access only one time to this vector at the beginning of computations.

3.4.3.3 BCSR

The parallelization of this format on GPU has been proposed in [46]. It is realized by using the built-in-type vector for computations of one block row per thread in order to minimize the memory transactions. This implementation is referenced as BCSR in the evaluation. However this implementation may be improved with partial coalescing by assigning an half-warp per each block row. Partial-results of a thread are stored in registers to benefit of a faster during the processing of the assigned block rows. This version is named BCSR_Vec and compared in our evaluation with the non-vectorized version and the other sparse formats.

3.4.3.4 ELLPACK/ITPACK

Depending on the point of view, GPU is a data parallel or a vector architecture and the ELLPACK/ITPACK format is well adapted for it as demonstrated in Bell and Garland [44]. Their implementation named ELL is based on a column-major order in memory of the ELLPACK/ITPACK format which offers to get full coalescing memory access and thus optimize memory transactions. Each thread is assigned to multiply

each element of a row by the corresponding element in the vector x and to put the result in the vector y. The column major order used by this implementation is not the order of the C language which orders elements of 2D arrays in row order in memory. This is why we propose an implementation of the ELLPACK/ITPACK format with a row-major order, named ELL_row. In the same way as the CSR_Vec implementation, each row is computed by a warp and a parallel reduction is performed to sum partial-results. The vectorization of computations offers to access the memory efficiently by minimizing transactions. Both implementations also map the vector x into the texture memory in order to cache data.

The analog format of ELL is ELLPACK with a column compression, named ELL_Col. We have implemented ELL_Col on a GPU in the same mind as the CSC format. One element of the vector x is mapped per thread in order to access only once to these values which are reused during computations. All products between *value* and the corresponding element of x are computed and put into a temporary array. Then, all results of the temporary array belonging to the same row are summed and the final result is saved into the vector y.

3.4.3.5 SGP

The interest of this sparse matrix format is that we can convert the matrix from a compressed column representation into a compressed row representation. This feature had a great interest for data parallel machines because it allows to re-map the vector result y on each computation unit for the next SpMV during computations. It is particularly interesting in algorithms where many spmv are used. The implementation on GPU of this sparse format is done by mapping one element of vector x per thread and assigning each thread to a column for the multiplication between the array *values* and the corresponding element of x. After, the result is put into the 2D temporary array in which results are organized in a compressed row representation. Then, each value of a row is summed and put into the vector y.

3.5 Evaluation Methodology

After the overview of various sparse formats and the different ways to implement them on many-core, this section presents a performance comparison of SpMV on GPU with two methodologies. On one hand, a set of matrices previously used in others papers are taken as reference in order to discuss the performances of our improvements and the behavior of compressed column format. On the other hand, we propose to evaluate the efficiency of sparse formats following the distribution of non-zero values with a technique that has been used on data parallel machines.

3.5.1 Reference Matrices

The set of sparse matrices comes from the University of Florida Sparse Matrix Collection [49] and are given in the Table 3.1. They have different structures, sizes and numbers of non-zero values which plays on performances achievement. In addition, two sparse matrices from a finite difference discretization of the Laplace operator in 2D are used with 5-point and 9-point stencil which represent an interest for Total and have a regular data structure. These sparse matrices were also used in the paper [43] to compare optimization and performance of SpMV on different multicore processors. In [44], these sparse matrices are taken as reference to evaluate sparse formats and SpMV on GPU. They are also used in our evaluation in order to get the same references and compare a larger range of sparse formats for SpMV performances.

Matrix Name	Dimensions	NNZ	NNZ/Row
Protein	36K x 36K	4.3M	119.3
FEM Spheres	83K x 83K	6.0M	72.1
FEM Cantilever	62K x 62K	4.0M	64.1
Wind Tunnel	218K x 218K	11.6M	53.3
FEM Harbor	47K x 47K	2.37M	50.6
FEM Ship	141K x 141K	3.98M	55.4
Economics	207K x 207K	1.27M	6.1
Epidemiology	526K x 526K	2.1M	3.9
FEM Accelerator	121K x 121K	2.62M	21.6
Finite Difference 5-point	1M x 1M	5M	5.0
Finite Difference 9-point	1M x 1M	9M	9.0

Table 3.1: Sparse Matrices used for this evaluation

3.5.2 Matrix Patterns to Modify the Distribution of Non-Zero Values

The data structure of matrices also takes part in achieving performances. The distribution of non-zero may unbalance the computation burden per thread and penalize an efficient access to the vector x. On past data parallel supercomputers like Connection Machine, the distribution of data was a critical parameter because it defined communications between computation units. Some matrix patterns, named C-Diagonal and C-Diagonal q-perturbed, were introduced to evaluate SpMV with various sparse formats following the distribution of nnz in order to determine the best and worst performance cases [48]. As GPU is a data parallel architecture, we propose to experiment this type of matrices to compare the different format efficiency following the distribution of nnz. The C-Diagonal matrix is defined as a matrix with nc diagonals from the main diagonal to the right, as shown in Figure 3.7.

The C-Diagonal q-perturbed matrix is built from a C-Diagonal matrix by modifying

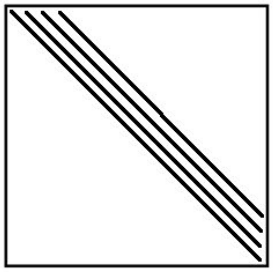

Figure 3.7: C-Diagonal Matrix (nc=4)

the data structure with a random perturbation. Each element of the matrix has a perturbation probability called q which defines whether the value has to be set at a random distance from the main diagonal, the element column index must be modified. During the random perturbation, a homogeneous distribution of the nnz per column is guaranteed in order to have roughly the same number of accesses to the elements of the vector x. An example of a C-Diagonal q-perturbed matrix with $nc = 4$ and $q = 0.2$ is given in Figure 3.8.

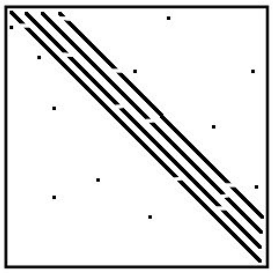

Figure 3.8: C-Diagonal q-perturbed Matrix ($nc = 4$ and $q = 0.2$)

3.6 Performance Analysis

3.6.1 On a GPU

After the overview of various sparse formats and the different ways to implement them, we present in this section a performance comparison on GPU. On one hand a set of matrices previously used in other papers is taken as reference in order to discuss the performances of compressed column format. On other hand, we propose to evaluate

the efficiency of sparse formats following the distribution of nnz values with a technique that has been used on data parallel machines.

3.6.1.1 Comparative Results

The various formats evaluation is based on matrices from the University of Florida Sparse Matrix Collection. The matrices have different structures, sizes and numbers of non-zero values which are reported in Table 3.1. We also use two matrices from a finite difference discretization of the Laplace operator in 2D with different stencil, 5-point and 9-point.

The best results are obtained with the ELL format on the finite difference matrices with 17.98 Gflop/s for 5-point and 19.31 Gflop/s for 9-point in single precision (SP), as the Figure 3.9 shows. The regular structure of these matrices allows to have approximately the same number of values per row and to have a balance of computation load between threads. On the other matrices, the ELL format is also the most efficient thanks to column-major order which permits the coalescing by accessing data in a contiguous way and in consequence to minimize memory transactions. Our version ELL_row with a row-major order is less efficient in spite of the use of entire warp to read data from the array *value*. Data are accessed in a coalescing way by 16 threads but not in a contiguous way in the entire memory. On finite difference, this format achieves a poor performance because rows with a nnz less than a warp are padded with extra zeros. It introduces new values and increases the number of elements to compute. The CSR format performs poorly in this naive implementation. Performances of the CSR format on finite differences represent 20.47% for 5-point and 11.88% for 9-point of ELL results and the format achieves an average of 2.1 Gflop/s with others matrices. The format implementation retrieves data in a non-optimized fashion because threads do not read data in a contiguous access. A contrario, the CSR_Vec implementation uses vectorization which allows to access data contiguously in memory and takes advantage of coalescing. This CSR version performs better and is a little bit more efficient than ELL.

The blocking version of CSR improves performances on non-structured matrices with an average of 5.1 Gflop/s and offers a SpeedUp of 2.5 in comparison of CSR. This improvement comes from the use of the built-in-type vector to access data by reading 4 values of the array *value* in one memory instruction. We have only evaluated two different sizes of blocks, 2×2 and 4×4. The last dimension of blocks has realized the best performances for many matrices but well-suited blocking heavily depends of the matrix structure. For matrices Economics and Epidemiology, most of blocks are not fully filled because the nnz are strongly shared out the matrix what generates blocks with few elements and a lot of zeros. As we have explained previously the full-filled by zeros in order to make dense block increases the number of values to compute and

decreases performance. The BCSR_Vec kernel is the vectorized version which takes advantage of coalescing by accessing 16 blocks of a row at a time. Performances are really improved with a SpeedUp of 2 on all matrices, except on Economics, Epidemiology and finite differences because of adding extra zeros to make dense blocks. We have also proposed and evaluated some column sparse formats which avoids to access many times to the vector x in opposition to compressed row formats. After reads optimization of the array *value*, the vector x is the last non-contiguous access. However, the column formats do not achieve the best performances of our evaluation. The CSC format is the least efficient with 2.6 Gflop/s for the highest result on Epidemiology. Computations are made in two steps. The first multiply each element of the array *value* and the second sums all elements belonging to the same row. The sum operations are the main reason of performance degradation because of irregular accesses to temporary array in which results of the first step are stored. The ELL version with compressed column, ELL_Col, has exactly the same issue but perform much better than CSC because of the coalescing to read values in the temporary array and the array *ptr*. It sustains 7.7 Gflop/s and 8.4 Gflop/s on finite difference matrices what represents twice less than ELL performances. The best results for this format are obtained on these matrices because of the regular structure. The SGP format obtains more interesting results on all matrices and is the best efficient compressed column sparse format. The implementation of this format is also in two steps but the reason of non-achieving a better performance is different. The source comes from the conversion of a compressed column representation into a compressed row which generates irregular access and does not optimize the memory transactions.

Figure 3.10 shows the results in double precision (DP) and the same analyze of performance can be made. The ELL format performs the best results with 10.9 Gflop/s and 12.3 Gflop/s on finite difference matrices. We can observe that performances decrease in double precision by achieving 57% of single precision performance for the best case. Nevertheless, the best performance in SP only reaches 1% of the GPU peak performance in opposite to the double precision which reaches 14.3%. It represents a difference factor of 7.5 and underlines that double precision is more efficient in spite the fact it achieves less performance.

3.6.1.2 Results Following the Distribution of Non-Zero Values

For experiments, the matrix size is set to $N = 1008000$ in order to reach the space limit of shared memory for the most larger format. The number of diagonal nc is fixed at different values, such as 4,8,16 and 32 successively, to increase the amount of non-zero. Finally for each nc count, the perturbation q is varied from 0.0 to 1.0 which respectively represent the best and worst case of distribution. The ELL implementation still achieves the best performance with more than 20 Gflop/s in SP for a non-perturbed

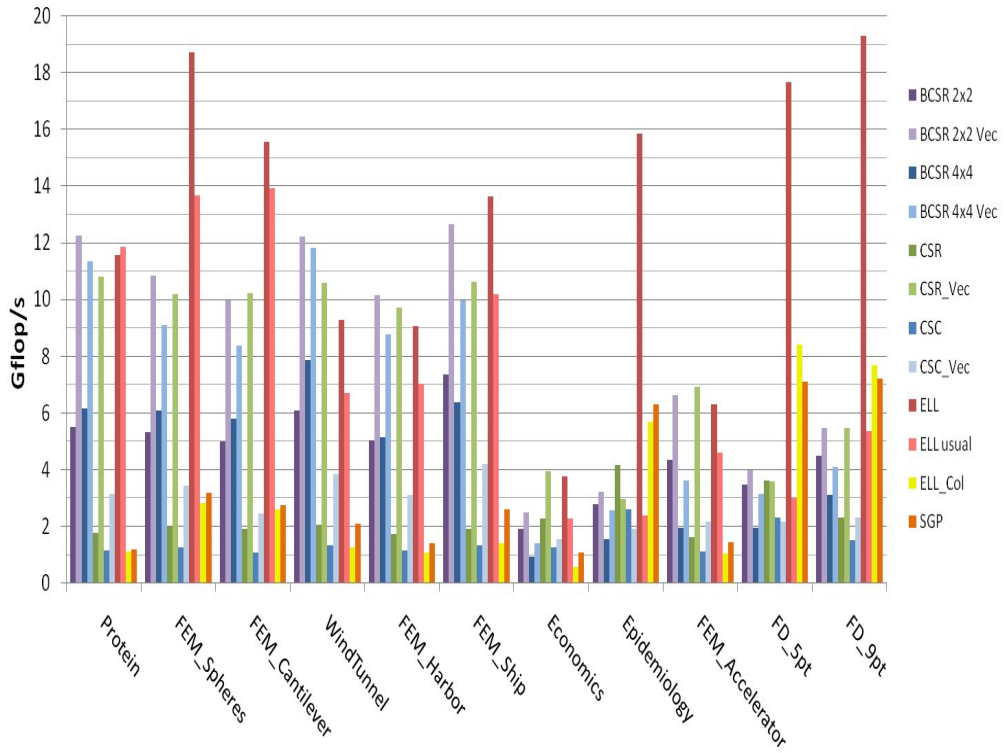

Figure 3.9: Sparse Matrix Vector Product Performance on a GPU in single precision

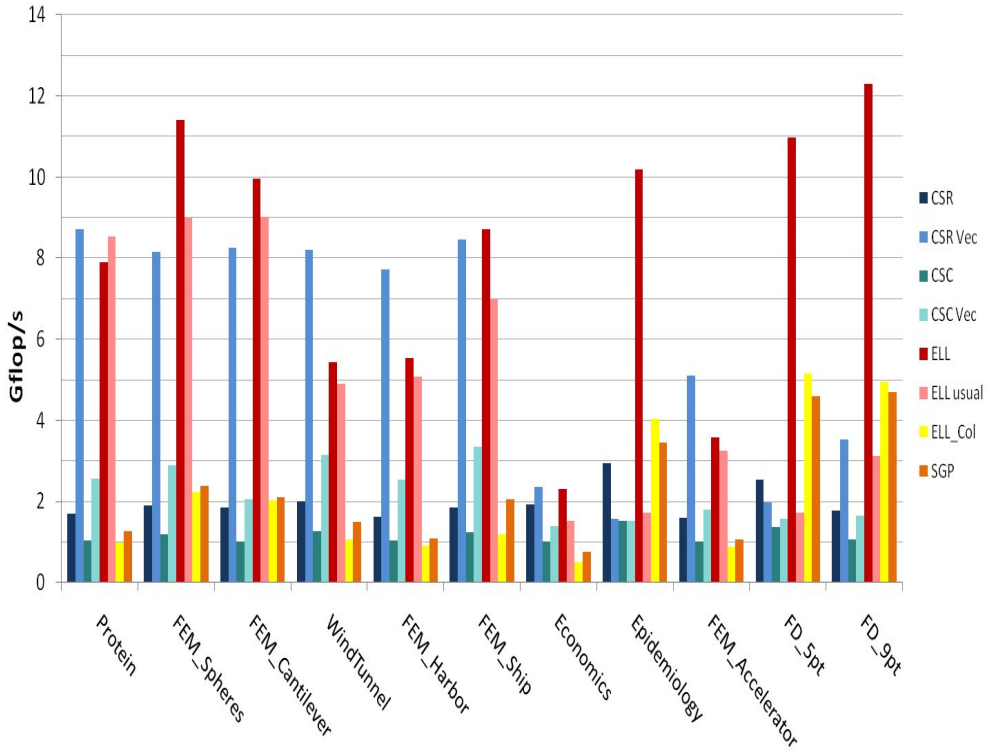

Figure 3.10: Sparse Matrix Vector Product Performance on a GPU in double precision

matrix (q=0.0) with $nc = 16$ or $nc = 32$ diagonals see Table 3.11. The other results for $nc = 4$ and $nc = 8$ of this implementation are a little bit less with respectively 17 Gflop/s and 19 Gflop/s. The nnz per thread are decreased in this case what does not allow to load the computation pipeline and avoid it latency. We can also see that the previous comparison between formats is still correct when $q = 0.0$. However, it is interesting to notice that for $q > 0.5$, the implementations efficiency are going down with a difference of few Gflop/s between formats. When the matrix is totally perturbed for $q = 1.0$ the same level of performance is reached by all formats, around 2 Gflop/s for ELL, ELL_usal and CSR_Vec and around 1 Gflop/s for the others. This result demonstrates that it makes no odds to care about the well-suited format for perturbed matrices on GPU with a perturbation greater than 50%. Moreover, the vectorized formats such as ELL_usual, CSR_Vec and BCSR*_Vec perform very poorly for a number of diagonals under 16. This is because vectorization is the fact to take advantage of coalescing by using a half-warp, 16 threads. For $nc = 4$ and $nc = 8$, all threads of the warp are not involved to compute a row which has less than 16 elements. Then, the coalescing and the minimization of memory transactions are not realized and it decreases performances. This observation is an obvious result and can be generalized about all vectorized formats that do or do not use zero-padding to have 16 elements per row.

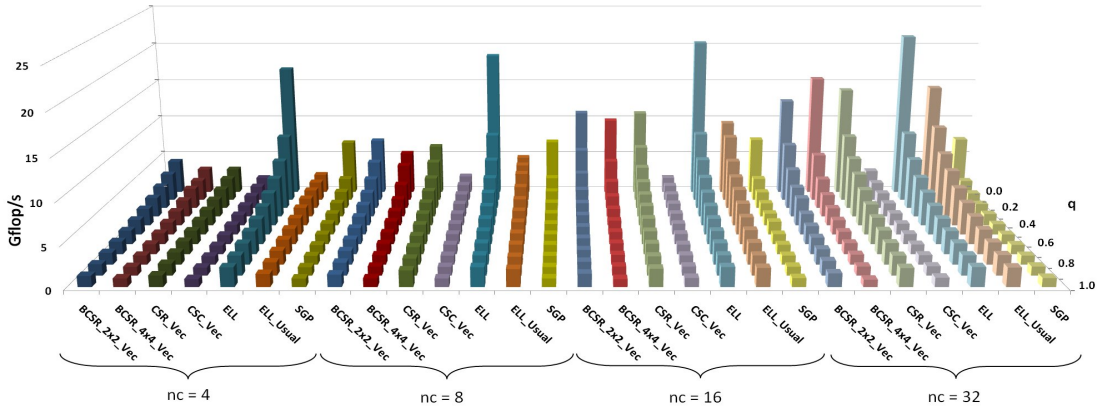

Figure 3.11: SpMV performances with C-Diagonal q-perturbed matrix in single precision

In double precision, the ELL format is also the best implementation on GPU for SpMV by sustaining 10 Gflop/s for $q = 0.0$ and 1.8 Gflop/s for the maximum perturbation, see the Figure 3.12. The behavior of other formats is still the same as in single precision. We observe that for $q < 0.5$ the performance of each format in double

precision represents 50% of the achievement in single precision. This comes from the memory bandwidth use which is more expensive in double precision. For matrices with a perturbation greater than 0.5, the difference between both precision is decreasing and for $q = 0.1$ there are no differences between single and double precision performance. This is an interesting point for iterative methods which need high precision to converge faster without loosing global performance. The comparison of the ratio effective/peak performance gives advantage on GPU to double precision with for the best case ($q = 0.0$) 2.06% in SP and 16.09% in DP and for the worst case ($q = 1.0$) 0.22% in SP and 2.07% in DP.

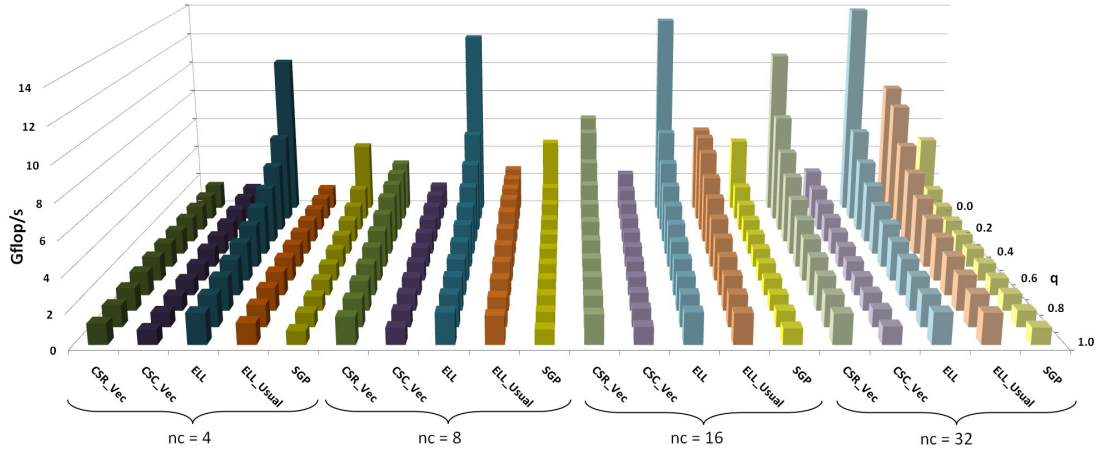

Figure 3.12: SpMV performances with C-Diagonal q-perturbed matrix in double precision

The results are also interesting whether they are compared to past results on Connection Machine. C-diagonal q-perturbed matrices have been introduced to evaluate the impact on communications following the distribution of nnz. GPU and Connection Machine are data parallel architectures but GPU has a shared memory which offers low latency to access and transfer data between threads. In contrry, the Connection Machine had an hypercube network which interconnected the computation units. The distribution of nnz into the matrix should have less impact performances on GPU. Nevertheless, curves show that the efficiency of the various formats used decreases more the nnz are distributed. Results on GPU have the same trend as the Connection Machine and the reason is the same. The access to the vector x is the source of performance degradation. When the nnz are strongly distributed, values are dispatched in a non-contiguous way through columns which does not allow to take advantage of coalescing. Then the amount of memory transactions to read data are increased and performances go down on GPU. The computation burden per thread is not implicated in our evalua-

tion and analysis because only the column index is perturbed and we roughly guarantee during the perturbation a homogeneous distribution of values per columns. For column compressed format, the vector x is not in cause because one value is mapped on each thread. The problem comes from row values which are arranged in a contiguous fashion in a column. This leads to the same consequence such as a non-coalescing access to row data. Except the fact that GPU is a new data parallel architecture with shared memory, it has the same behavior than Connection Machine on strongly distributed matrices. Despite matrices used here are not a realistic case, these experiments show that it is always important to care about data access of the vector x on a low latency architecture as on past data parallel supercomputers.

These results are also presented in [50] and some experiments on Fermi GPU have been published in [51].

3.6.2 On Multi-GPUs

SpMV is one performance key of iterative methods in which multiple SpMV are executed. Performance of the SpMV on a single many-core processor is one part of the achievement when the method is run in parallel. Communications are the second part involved. To parallelize the SpMV, a row wise decomposition is commonly applied across processors. Each processor P_i is responsible to make the computation of a row, as defined by $y_i = A_{i,*}x$. In an iterative methods, this operation is realized multiple times. After each SpMV, each partial result of y_i handled by a processor P_i must be sent to all processors in order to get the full vector y which becomes the vector x for the next SpMV. This operation may be summarized as $y = A(Ax)$ and is executed many times in an iterative method. Communication to send and receive the partial results of y highly contribute in the performance achievement and scalability. However, some values of y are not necessary on all processors because the vector x is not totally accessed during the SpMV. This depends on the problem to solve. Graph [52] [53] and hypergraph [54] decomposition help to minimize communication volume of the SpMV. Each node of the graph represents the result of each row_k by the vector x. Each edge corresponds to the dependence between each row and a column value which are generated by the previous SpMV.

Communications are more crucial in performance achievements of multiple GPU applications. Because the link between host and GPU has a low bandwidth and is considered as the bottleneck. Thereby, minimize data transfer between the GPU and host is highly important for the SpMV. In this subsection, we propose to use a row wise and a graph decomposition in order to reduce communication between processors and between host and GPU for the $y = A(Ax)$ operation on a multi-GPU cluster. Experiments are made to observe the performance and scalability of the SpMV on multi-GPU with and without communications between host and GPU. For this, we

have implemented two version of this operation.

The first one only minimizes the communication with the graph decomposition across processors. In more detail, after transferring data of the matrix on GPU, the first SpMV is executed and data generated, i.e. the partial vector y, are copied from GPU to the host. Next, MPI processes exchange the necessary data between processors based on the graph decomposition. Hence, data received are scattered to x on the host which is after integrally copied from the host to the GPU. To end, the second SpMV is processed.

The second one minimizes both communications, among processors and between host and GPU. In the same way, data of the matrix are firstly transferred to the GPU and the first SpMV is executed. By using the graph decomposition information, only local values of y are copied from the GPU to the host. Based on the index of the value to send, data of y are gathered into a *SendBuffer* which is transferred to the host. The values of the buffer are after sent to the corresponding processors through MPI communications still based on the graph decomposition. Hence, data are received in a *RecvBuffer* which is copied from the host to the GPU. The value of the *RecvBuffer* on the GPU are after scattered to x to the corresponding index of the value received. To end, the second SpMV is executed. Thereby with this version, data transfer between host and GPU are minimized by only exchanging the necessary values.

In order to compare both versions, experiments are run on a cluster composed of 20 GPUs (S1070) which are spread by 2 per host. A third version is added at our evaluation. It is the practical peak achievable by the $y = A(Ax)$ operation without any communications. The addition of this version is to evaluate the efficiency of our fully-optimized communication implementation. For our experiments, the sparse format that achieves the best performance on a GPU is used, i.e. Ellpack/ITpack. Two large matrices are taken for this evaluation. A C-Diagonal matrix with 32 diagonals ($nc = 32$) and a number of row equal to 1008000. A matrix from a 5-pt discretization of the Laplacian operator in 2D of a grid 1000×1000 (number of rows = 1000000).

Our implementation with communications minimized between nodes and host and GPU performs better than the non-optimized version. A speedup of 2 is obtained for a small number of GPU and is growing with the adding of GPUs to reach a speedup of 5 for 20 GPUs, as shown on the Figure 3.13. Those accelerations of the $A(Ax)$ operation traduces the highly importance of communications between host and GPU and their minimization. We also observe that our approach scale better than the version with only MPI communications are reduced. The performance difference between the practical peak achievable and our version is also growing in relation to the adding of GPUs to compute $A(Ax)$. It is explained by the fact that communications also increase with the number of GPUs and take increasingly a higher part in the global execution time.

The C-Diagonal matrix is an ideal in which values are close to the diagonal and has only one reception communication per process of the values handled by the process

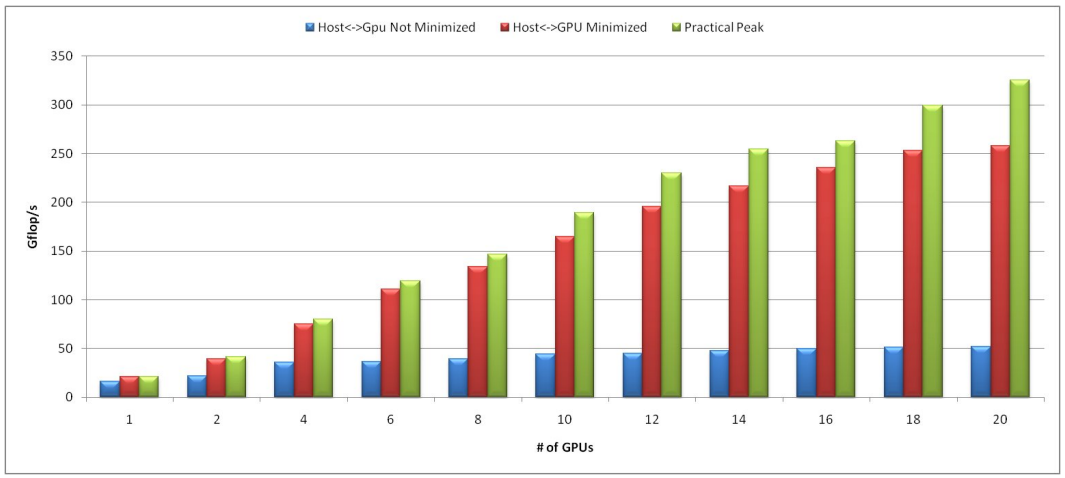

Figure 3.13: A(Ax) performance and scalability with the Ellpack/ITpack sparse format and a C-Diagonal matrix (nc=32) on a multiGPUs-cluster in Single Precision

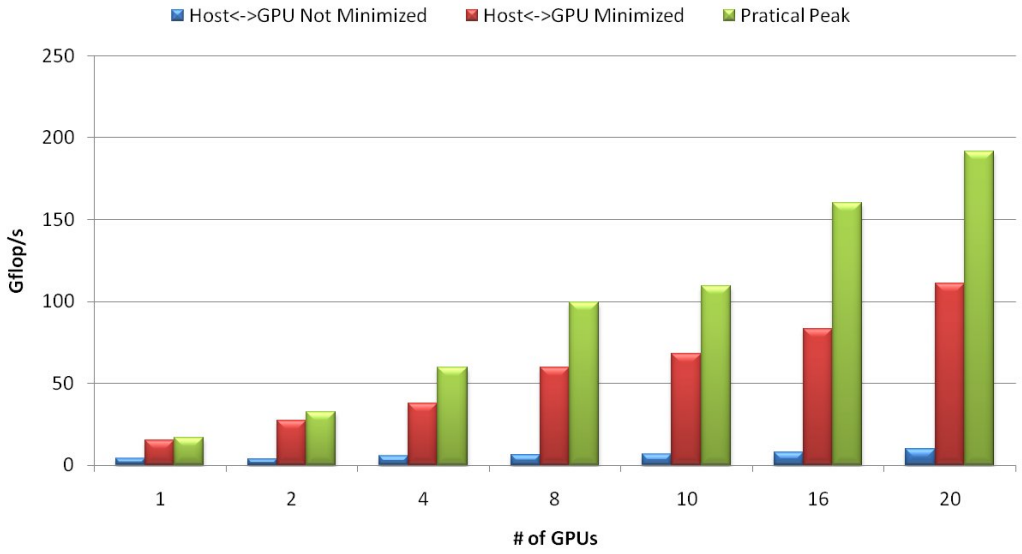

Figure 3.14: A(Ax) performance and scalability with the Ellpack/ITpack sparse format and a 5-point finite difference dicretization of the Laplacian operator in 2D on a multiGPU-cluster in single precision

$rank + 1$, except for the last process. Experiments realized with a matrix which comes from the 5-point discretization of the Laplacian operator 2D is a more real case. All the

case of different GPU number has not been evaluated because the matrix has 1000000 which can not be homogeneously distributed across processors. On the Figure 3.14, we can observe for this matrix that the speedup is higher than the ideal case with a factor of acceleration from 2 to 10. The scalability still true for this matrix bu the performance difference between the practical peak and our implementation is more important than on the C-Diagonal. This traduces the fact that communications are more numerous and take a higher in the performance achievements as the number of contributing GPUs is growing.

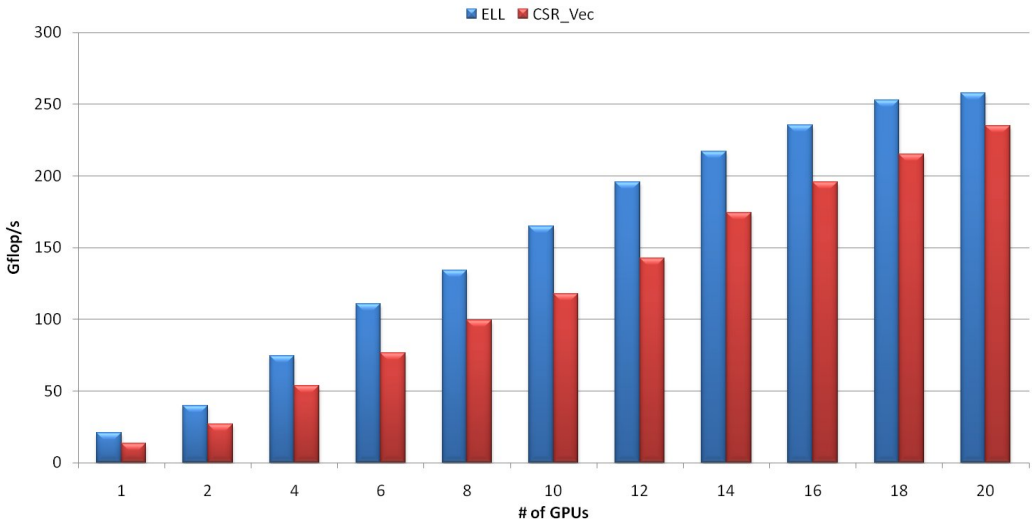

Figure 3.15: Performance comparison between Ellpack/ITpack and CSR Vectorized for A(Ax) with a C-Diagonal matrix (nc=32) on a multiGPUs-cluster in Single Precision

The communication volume is the same for all compressed row format. Thereby, only the performance of the SpMV kernel on a GPU has an importance in the achievement of multiple SpMV operation on a multi-GPUs cluster. As the comparison between the Ellpack/ITpack format shows on the Figure 3.15.

As we have seen, our optimization of communications for the $A(Ax)$ operation offers scalability and high performance speedups comparing to no communication reducing between host and GPU. In iterative methods, the SpMV is executed more times than twice. In consequence a poor SpMV on multi-GPUs may impact performances and scalability of these methods at a higher degree without any communication minimization between host and accelerator.

3.7 Conclusion

In this chapter, the data parallel paradigm has been explored on a many-core processor with sparse matrix computations which are the most difficult to optimize with this paradigm. To illustrate that, we have proposed and evaluated the sparse matrix vector product on a GPU with the common sparse matrix formats which define the data structure and play a role on performance achievements. As results have demonstrated the Ellpack format offers the best performances for common matrices by optimizing accesses to the main memory. Our variant of Ellpack with a row-major order (Ell_Row) also performs well, depending on the matrix structure. For matrices with a nnz per row lower than 16, our proposed implementation and all vectorized formats perform poorly because the vectorization is not completely realized and dos not gives partial-coalescing. The padding with extra zeros to get 16 values per row correct this problem as shown in [45]. However, padding generates extra computations that can degrade performance. Our implementation of BCSR with vectorization also achieves good performances on matrices with a sufficient nnz to have an optimized blocking.

Compressed column formats have the advantage to reduce irregular accesses to the vector x and have achieved good performances on Connection Machine. Unfortunately, results have shown that type of format perform less than compressed row format because of the non-possibility to exchange data directly between cores. The study of sparse formats performance following the distribution of nnz has also shown interesting results about the choice of format for different matrix structure. ELL implementation with column-major order stays the most efficient format. Nevertheless, we have seen for sparse matrices with a strong distribution of nnz that the various formats achieve approximately the same performance by reaching 2.0 Gflop/s. Moreover, for this matrix structure type performances in SP and DP on GPU have a gap of a few flop/s. As we have underlined, this can be interesting for iterative methods which require high precision to keep stability and converge faster without loosing performance.

On multi-GPU clusters, we have investigated a minimization of communications between host and GPU based on a graph decomposition. Experiments show that our proposition offers scalability and a high performance speedup in comparison of an approach with only the minimization of communications between processors. Hence, these results and our proposition contribute to the performance and scalability of iterative methods multi-GPUs cluster.

The interest for TOTAL in this study was to get information about sparse matrix format performances. On one hand these results may be used for seismic imaging where

some applications are based on iterative solvers and a finite-differences discretization of the Laplacian operator which composes the equation. On the other hand, iterative methods are currently explored for reservoir characterization which are based on a finite element discretization. Results following the nnz distribution should contribute to help the decision of the convenient sparse format. The SpMV has not been integrated in these solver because they are currently in development and need some supplementary works on mathematical aspects before any acceleration with GPU.

As we have seen, sparse matrix computations are challenging to optimize. Nevertheless, the past knowledge on data parallel programming has helped to take advantage of the memory hierarchy and to optimize accesses to the main memory which is the main bottleneck. However, experiments have pointed out that it is not easy to use shared caches (shared memory for GPU) for irregular computations. They have been only used for reduction among cores. Some difficulties with some formats such as SGP have also outlined that GPU and in general many-core suffer from the lack of direct communication possibilities between cores. This induces to use the main memory for data exchange and thus degrades considerably performances. This point is highly important for exascale computing because it traduces the impossibility to optimize data movement on the chip. This parameter is critical in order to avoid a bottleneck to the main memory and to minimize the energy consumption. Some chip makers have whereas the consciousness of this issue and are currently releasing research processors with a network-on-chip (NoC) such as Intel with its Single-chip Cloud Computer (SCC) and their 80-cores processor. Nvidia has also some plans to integrate communications possibilities in their GPU that should contribute to reduce the bandwidth consumption of the main memory and optimize data movements on chip.

Chapter 4

A Graph Description Paradigm for Asynchronous Coordination and Communications on Large Clusters

4.1 Introduction

A common trend in supercomputers is the aggregation of many processors to gain computing power. The arrival of multicore processors has not modified this trend but it has changed the base type of the counting from processors to cores. Exascale supercomputers should have billions of cores organized in millions of nodes based on many-core processors. The large parallelism created by these machines should be hard to manage with common programming paradigms such as a flat-MPI or a mixed MPI+OpenMP approach to get the best efficiency, as we have underlined in the chapter 1. For exascale computing, programming paradigms must scale to run applications on millions of nodes and high level to improve productivity. The program scalability is mainly due to communications and essentially on those that involve a large number of cores. Partial or global synchronization of participating elements take a high amount of execution time which slows down the global application performance. Asynchronous communications avoid this coordination and have also the asset of allowing to overlap computations during data transfer. However, their management at a low level by end-user may be complex, inefficient and unproductive for exascale. A support at a high level and a run-time management should help to realize asynchronous and optimized communications in a transparent manner. Typically, a parallel and distributed application is constituted of a succession of tasks. The load balancing and the scheduling of tasks are also important in order to avoid barrier synchronization and start a task as soon as possible. This task orchestration must be also supported at run-time to load-balance computations and use efficiently available computing

resources. Such a system may also contribute to reduce programming complexity by giving transparent and productive parallel programming through a high level description.

In our approach of programming exascale, we have proposed a multi-paradigm programming and have explored for many-core processors a data parallel programming paradigm. The problem is now parallelism at a high level between nodes or node groups following the points underlined previously. Different solutions are possible to program this last level. Domain specific languages are good candidate to improve productivity with an orientation to a specific domain. They may also provide portability by generating convenient code for a targeted platform. For example, Liszt [55] is designed for mesh-based partial differential equation and can be compiled to MPI cluster or GPU. Nevertheless, these languages are restricted to some application areas and are not general purpose parallel programming models. Some other candidates for exascale programming are more general models such as Unified Parallel C (UPC) [56] for C, Co-Array Fortran (CAF) [23] for Fortran and Titanium [24] for Java. These languages rely on a Partitioned Global Address Space (PGAS) model and provide improved performance by exploiting locality and hardware protocol with one-sided communication. They have the shortcoming to be in the same mind of message passing with explicit communications expressed by end-users which are not managed and optimized at run-time. Other approaches are possible for exascale programming to manage large parallelism between nodes and improve productivity. Workflow or graph description languages are more end-user oriented by expressing task dependencies through a graph. They offer many information that may be used at run-time for task orchestration and asynchronous communication.

In this chapter, we propose to explore this graph description paradigm for the highest level of the hardware hierarchy through one implementation,named YML [57], that have developed in relation with our team. The next section outlines the mainly known workflow or graph description languages and their differences with the language proposed. The design and the language of the YML Framework are presented in the third section. The interest and the evaluation of the Framework are demonstrated on a nation-wide cluster of clusters and based on a dense matrix inversion method which are outlined in the fourth section . The contribution of this framework to the Time-To-Solution minimization are presented in the fifth section through two adaptations of a dense matrix inversion method which is implemented with YML and a performance evaluation. Mechanisms of data persistence and data migration anticipation may improve achievement by avoiding or making in advance some communications. A graph description paradigm may be contribute to anticipate data migration, i.e. realize asynchronous communications, without any extra programming for end-users The graph analysis at

run-time may give information to find out future data movements and anticipate them. More details are given in the sixth section in which we explain how this may be supported in YML. Actually, these mechanisms are not supported in the framework because backends which bind up the independent part of YML and middleware are not able to take these optimizations into account. In consequence, a performance pre-evaluation to estimate the YML overhead in this context is made by emulating these mechanisms and comparing to an implementation with these features in OmniRPC which is a middleware supporting the data persistence. The last section concludes and discusses on the contribution of a graph description paradigm to improve productivity, manage task orchestration, realize asynchronous communications in a transparent way and other necessary points that have been explored here for programming exascale machines.

4.2 Related works

To simplify parallel and distributed programming, some high level approach like workflow or graph description languages have been developed for grids and clusters. DAGMan [13] is one of them. It is a workflow engine and a meta-scheduler for the Condor-G middleware [58]. The expression of cycle are not possible and thus does not prevent dead-locks. Data migration must be explicitly managed by the end-user. GridAnt [59] is a workflow system which uses a language based on XML to describe the workflow. It associates an operation to a tag. To be executed, an operation needs to wait the end of all tags on which it depends. Pregel [60] is targeted to run parallel and distributed application on large data-center. The framework is based on a BSP execution paradigm and claims to scales on billions cores. The parallelism is expressed through a C++ API to declare tasks which can exchange messages between them. These and many other workflow approaches such as UNICORE [14] and GridFlow [15] are based on a Directed Acyclic Graph (DAG) which allows to express a static workflow. YML [57] is another graph approach which is developed at the University of Versailles and University of Lille. It is a high level language and framework to develop and run parallel and distributed applications on several middleware. In contrary to the previous workflow mentioned, the graph description language provided by YML is based on Directed General Graph (DGG). This graph model gives the possibility to express loops, iterations and branching. Moreover, YML is independent from middleware with a design composed of two parts. The first is in charge to manage the development and the execution of parallel and distributed applications. The second binds up the first part and the middleware chosen by end-user. This and the component programming approach provide modularity and reusability to allow the reduction of the development time.

4.3 YML Framework

4.3.1 YML Design

YML is a framework dedicated to the development and the execution of parallel and distributed applications on cluster, grids and peer-to-peer middleware. It has a component-oriented programming approach which offers modularity and reusability. Furthermore, YML applications are independent from middleware thanks to the design of YML which is composed of two parts presented in Figure 4.1. The first part of the design is the frontend which encompasses a compiler for the dedicated graph description language, a just-in-time scheduler [61] and a development catalog. The compiler translates applications expressed by the graph description language into an intermediate representation. This representation describes the dependences between tasks through an event mechanism. The combination of events are the pre- and post-conditions and determine whether a task can be executed or not. The just-in-time scheduler is in charge to manage application executions. It resolves tasks dependences at run-time by detecting tasks which are ready to run. Thereafter, the scheduler generates a set of parallel tasks which become computing requests through the backend part. The development catalog stores components and data type information used during the development step. This set contributes to validate the graph description input program. The frontend is the middleware independent part and is associated to a second part, named backend, in charge to generate the binaries and binds up the frontend and the chosen middleware which dispatches tasks with its own scheduler among the computation resources. Actually, YML supports two middleware: XtremWeb and OmniRPC. A multi-backend management [62]. is also implemented and offers the use of both middleware at the same time. Those supported middleware and the multi-backend management are presented in the following of this chapter. The data repository server provides the component binary and data requested by YML workers. They after compute the assigned task on the data set and take over results to the data repository server when the task is finished. A presentation in more details of different YML parts is given in [16].

To describe applications and their executions, YML includes a dedicated graph description language called *Yvette* which is based on control flow expression. The graph of an application is constituted by different elements. The nodes represent tasks and the edges are the dependences between them. Thereby, this high level type of language offers implicit communications which are optimized by the compiler and not managed by end-users in contrary to MPI or OmniRPC for example. The YML language allows to describe scientific application graphs in opposition to Kepler [63] or YAWL [64] which are graphical languages of workflow. The declaration of three or more dimensions for application graphs is difficult with this type of language. *Yvette* offers to express graphs in many dimensions and has also an easy parsing thanks to a LL(1) grammar. The framework has also a component-oriented approach which is different

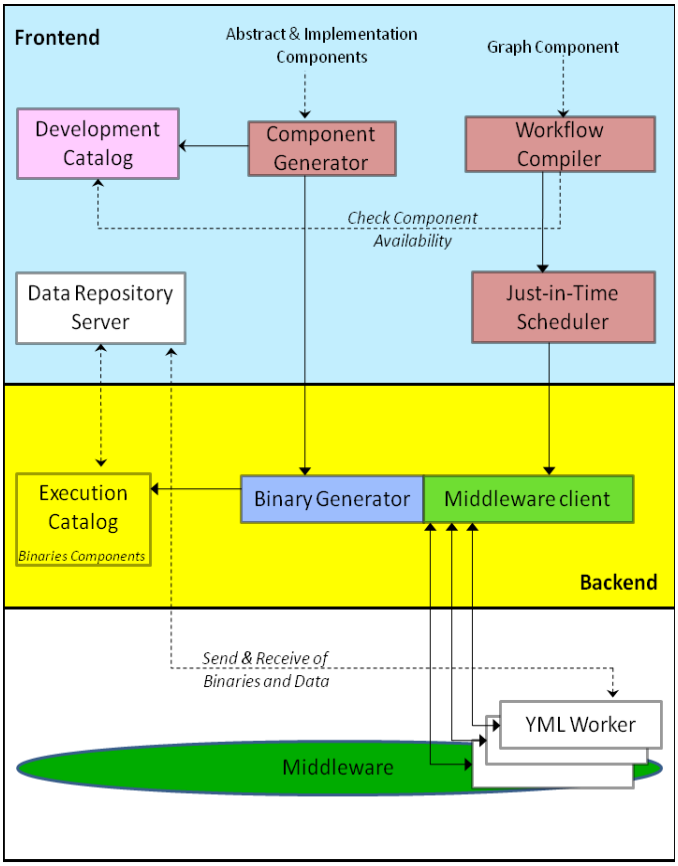

Figure 4.1: YML Design

from other component-oriented workflow like AGWL [65] or GFDL the language used in GridFlow [66]. Thereby, it has two description aspects which are encapsulated in XML document for homogeneity: describing components and the application graph. The development of a YML application is made with components which are three of kinds:

- Abstract component: an abstract component defines the communication interface with the other components. This definition gives the name and the communication channels with other components. Each channel corresponds to a data in input, in output or both and is typed. This component is used during the development and the code generation stages to create the graph. An example is given on the Figure 4.2.

- Implementation component: an implementation component is the implementation

```
<?xml version="1.0"?>
<component type="abstract" name="DGEMM" description="AxB=C">
<params>
<param name="A" type="Matrix" mode="in" />
<param name="B" type="Matrix" mode="in" />
<param name="C" type="Matrix" mode="inout" />
<param name="size" type="integer" mode="in" />
</params>
</component>
```

Figure 4.2: YML Abstract Component Declaration

of an abstract component. It provides the description of computations and it is used during the execution. The implementation is done by using common languages like C or C++. They can have several implementations for the same abstract component. This type of component is illustrated in the Figure 4.3.

```
<?xml version="1.0"?>
<component type="impl" name="DGEMM_impl" description="AxB=C"
abstract="DGEMM">
<impl lang="CXX" libs="">
<header></header>
      <source>
      <![CDATA[
            int i,j,k;


            for(i = 0 ; i < size ; i++)
                  for(j = 0; j < size ; k++)
                        for(k = 0; k < size ; k++)
                        {
                          C[i][j] += A[I][k] * B[k][j];
                        }
        ]]>
        </source>
<footer></footer>
</impl>
</component>
```

Figure 4.3: YML Implementation Component Declaration

- Graph component: a graph component carries a graph expressed in *Yvette* instead of a description of the computation. The graph can contain parallel and sequential sections and usual constructions such as branches and loops. The synchronization between the different steps is done with the event mechanism.

```
<?xml version="1.0"?>
<application name="BLOCK DGEMM">
<description>Double Precision AxB=C</description>
<graph>

    par(k:=1;Size) do
       par(i:=1;Size) do
          par(j:=1;Size) do
             if(k gt 1) then
               wait(Step[i][j][k-1]);
             endif

             compute DGEMM (C[i][j],A[i][k],B[k][j],size);
             notify(Step[i][j][k]);
          enddo
       enddo
    enddo

</graph>
</application>
```

Figure 4.4: Block Matrix-Matrix Product in Yvette

4.3.2 Yvette Language

In more details about *Yvette*, the syntax of the language is similar to Pascal and C. The different existing keywords in Yvette are as follow. Parallel sections are declared to make concurrency and are defined such as ***par*** Section1 // Section2 ***endpar***. The loops are expressed in two ways: ***par do*** ... ***enddo*** structure is for parallel loop and ***seq do*** ... ***enddo*** is for sequential loop. The conditional structure is defined by the use of the usual ***if****(condition)*...***then***... ***else***... ***endif*** structure. The different tasks of the graph are synchronized through events with two keywords: ***wait(event)*** and ***notify(event)***. Components declared previously are called with the ***compute*** keyword followed by the component name. An example of an *Yvette* program is given in the Figure 4.4 through a typical BLAS3 operation, a block matrix-matrix product.

4.3.3 Supported Middleware

4.3.3.1 XtremWeb

XtremWeb is a desktop middleware dedicated to grid computing which is develop in Java language. This middleware has a master-worker programming model which is based on an architecture organized as client, coordinator,worker. The client has tasks to compute in parallel on a set of worker nodes which are put in relation by the coordinator The

binding between workers and client is not the only role of the coordination which is also in charge to collect results and exchange information in the system. In opposite to other master-worker model, XtremWeb has the particularity to have worker in a *pull* model. This mode allows to each work to be not managed by a central manager and to retrieve their computations by requesting the coordinator. This worker model has been integrated into XtremWeb in order to support very large and volatile resources.

4.3.3.2 OmniRPC

OmniRPC is a thread-safe remote procedure call (RPC) system, based on Ninf [67], for cluster and grid environments. It supports the parallelism described in the master/worker model. Then an OmniRPC application contains a client program which expresses the control flow and calls a remote executable. The scheduling is deterministic because it is written by the user through the control flow. A schedule is also done to assign a worker node for a remote execution. The remote executables are programs which contain remote procedure whose the execution is synchronous or asynchronous. The declaration of a remote procedure is defined by an interface definition language (IDL). It defines the interface and its implementation which writes in familiar scientific computation language like FORTRAN, C or C++. Data persistence mechanism is supported with the *handle* function which creates a remote connection to a host and allows to use data as a remote object. Moreover, an agent is in charge to invoke the remote executable and manage communications with client programs.

4.3.3.3 Multi-Backend Management

YML also supports multiple middleware at the same time for running applications on Peer-To-Peer networks and clusters. The binding of YML to middleware is made through a back-end. To support the execution of application on multiple middleware, a new back-end the Multi-backend back-end has been created. In opposite to others this back-end does not communicate directly with middleware but it is connected to the Back-end Manager. The Back-end Manager is in charge to schedule dynamically tasks to middleware and acts as a proxy between YML workflow processes and the multiple middleware through the Back-end Connector. This last component is in charge to relay information between the Back-end Manager and the regular back-end.

4.4 Experimental Platform and a Dense Matrix Inversion Method

For demonstrating and evaluating the interest of YML, experiments have been realized on a heterogeneous and highly reconfigurable platform, Grid'5000. A dense matrix

inversion method is used as demonstrating algorithm which provides the advantage to have different levels of parallelism and is a complete matrix inversion method in opposite to LU.

4.4.1 A National Distributed Cluster of Clusters: GRID'5000

Grid'5000 [68] is a large scale infrastructure for grid research. It is composed of nine geographically distributed clusters and each one has between 100 to 1000 heterogeneous nodes to reach 5000 cores. This cluster of clusters is interconnected by the French national research network RENATER. Grid'5000 provides reconfiguration and monitoring tools to find out grid issues. This platform allows users to make reservation, reconfiguration, run preparation and run experiments by using OAR [69] and Kadeploy [70] for nodes reservation and deployment of specific environment which built by user. Grid'5000 is used to investigate issues at different levels of the grid. This includes network protocols, middleware, fault tolerance, parallel/distributed programming, scheduling and issues in performance.

4.4.2 Block-based Gauss-Jordan Method

We consider the block-based Gauss-Jordan method as a linear algebra method example for our experiments and it is presented below. Let A be a dense real matrix of dimension N and let B the inverse of A, i.e. $AB = BA = I$. Let A and B be partitioned into matrices of p x p blocks of dimension n which n $= \frac{N}{p}$.

4.4.2.1 Intra-Step

This method is described by Agorithm4.1 and has p steps composed of four parts. At each step k, the part (0) is used to get the inverted block matrices B_{kk}. The part (1) assigns the block product of matrix A_{ki} and B_{kk} to the block A_{ki}. The part (2) consists of two parts of matrix products. The part (2.1) computes the column blocks belonging to the pivot and the (2.2) operation computes the corresponding parts of matrix B. The part (3) is composed of two parts which calculates the blocks of all columns of the matrix A with index i above and below that of the pivot row. The part (3.1) calculates the corresponding parts of matrix B. At last, the part (3.2) is used to compute the blocks of the column number k of matrix B except B_{kk}.
Finally, for one step of the loop k, one block inversion, $2(p-1)$ block products and $(p-1)^2$ block triadics are computed. So at maximum, $(p-1)^2$ computations run in parallel. Then $(p-1)^2$ processors are necessary for computations when the k loop is executed sequentially.

Require: A (partitioned into $p \times p$ blocks)
Ensure: $B = A^{-1}$
  for $k = 0$ to $p - 1$ **do**
    $B_{kk} = {A^{-1}}_{kk}$ **(0)**
    for $i = k + 1$ to $p - 1$ **do**
      $A_{ki} = B_{kk} \times A_{ki}$ **(1)**
    end for
    for $i = 0$ to $p - 1$ **(2) do**
      if $(i \neq k)$ **then**
        $B_{ik} = -A_{ik} \times B_{kk}$ **(2.1)**
      end if
      if $(i < k)$ **then**
        $B_{ki} = B_{kk} \times B_{ki}$ **(2.2)**
      end if
    end for
    for $i = 0$ to $p - 1$ **(3) do**
      if $(i \neq k)$ **then**
        for $j = k + 1$ to $p - 1$ **do**
          $A_{ij} = A_{ij} - A_{ik} \times A_{kj}$ **(3.1)**
        end for
        for $j = 0$ to $k - 1$ **do**
          $B_{ij} = B_{ij} - A_{ik} \times B_{kj}$ **(3.2)**
        end for
      end if
    end for
  end for

Algorithm 4.1: Block Gauss-Jordan Matrix Inversion

4.4.2.2 Inter-Steps

There is the parallel possibility existing between different iterative step. The parallelism comes from the data dependence between different operations in different iterative steps. See all the data dependence between two iterative steps through figure 4.5. Its algorithm and adaptive algorithm can be found in [71]. To state conveniently, we call the intra-step and inter-steps based parallel Block Gauss-Jordan algorithm as inter-steps algorithm in the following part of this subsection.

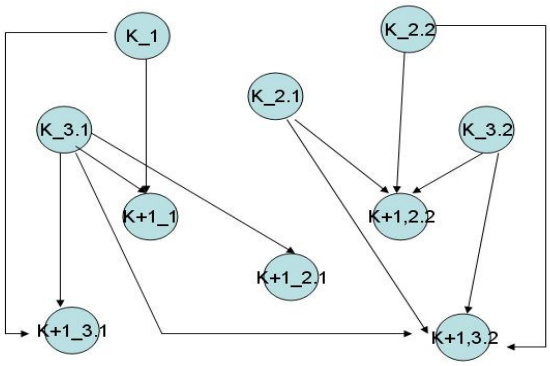

Figure 4.5: All these Data Dependence in Block based Gauss-Jordan algrithm

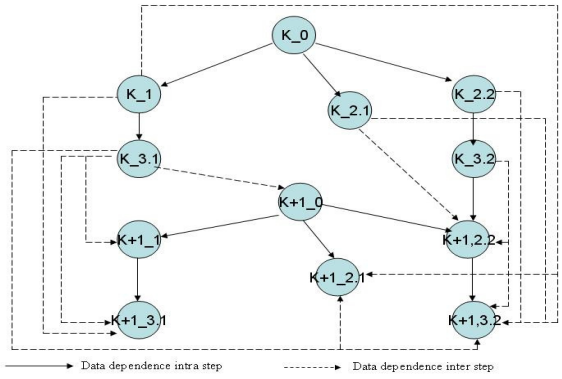

Figure 4.6: Data dependence between operations based on inter-step parallelism

Then all data dependences of the block-based Gauss-Jordan algorithm are summarized in Figure 4.6.

4.4.2.3 Difference between two kinds of parallel block-based Gauss-Jordan Algorithms

Intra and inter-steps algorithm shares the same operations (elementary BLAS3 types). The main difference between these algorithms are the dependence condition of operations. The condition in intra-step based parallel algorithm is two dimensions, one dimension is the index of matrix column and the other is the index of matrix row. The parts available to be executed in parallel are fixed intra an iterative step and operations can be controlled by a serious of two dimensions based precedent conditions. While conditions in inter-steps based parallel algorithm is three dimensions and one more dimension is iterative step. It can not only control parallel execution presented above in intra-step algorithm but also support parallel execution between steps.

4.5 Time-To-Solution Minimization

In this section, both parallel and distributed adaptations of block Gauss-Jordan are implemented with the framework. These implementations are made to show an example in YML and illustrate how it may reduce the development time just by modifying the graph component of application. However, YML is on top of middleware with its high level approach and may generate an overhead in comparison to the direct use of a middleware. We take as reference the intra-step implementation in OmniRPC which is a cluster oriented middleware to compare and evaluate the YML overhead. Different cluster architectures are used to vary some parameters like network or CPU speed and some optimizations are proposed to improve the YML overhead.

4.5.1 Programming using OmniRPC

OmniRPC is a middleware for cluster and grid environment which has a master/worker programming model based on a thread-safe remote procedure call (RPC) system. Users can call remote procedures from the client program without having any knowledge of network programming to parallelize operations. However, users should know in a first place the API functions provided by OmniRPC such as OmniRpcCall, OmniRpcWait or OmniRpcCreateHanlde and how to use them. In a second place, they develop remote libraries which contain a set of remote procedures defined by an interface definition language (IDL). It defines the interface and its implementation which can be written in familiar scientific computation language like FORTRAN, C or C++. To end, users develop the OmniRPC application which contains a client program expressing the control flow and calling remote libraries. The expression of the control flow leads to a static scheduling. To run the application, a host file of computation resources has to be defined with the host name, the number of cores and the type of protocol to use (rsh or ssh). After, the OmniRPC agent is in charge to schedule tasks among processors by

invoking remote procedures and to manage communications with the client program. OmniRPC has also the advantage to support data persistence mechanism through the *handle* function which creates a remote connection to a host and allows to use data as a remote object.

4.5.2 Programming using YML

YML is a component based framework which is used to develop scientific program. As we saw previously, three component types are necessary to create an application. The block Gauss-Jordan method is composed of four different computing tasks. A block matrix inversion (step 0), a block matrix product (step 1 and 2.2), a negative block product (2.1) and a triadic product (step 3.1 and 3.2) are tasks of the algorithm as described above. Four abstract and implementation components are defined in YML:

1. inversion: to inverse one matrix block
2. prodMat: to compute the two blocks products
3. mProdMat: to compute the negative of two blocks products
4. ProdDiff: to compute the triadic product

The last component to define is the graph component in which the task dependences and thus the parallelism are described through the graph description language, *Yvette*. The following gives the intra-step implementation of the block Gauss-Jordan matrix inversion method.

Intra-step algorithm in Yvette
Input: A, B matrices partitioned into $p \times p$ blocks of size n
Output: $B = A^{-1}$

```
seq (k:=0; p-1)
do
# Step 0
compute inversion(A[k][k],B[k][k],n,n);
notify(bInversed[k][k]);

# Step 1 - Parallel loop
par (i:=k + 1; p - 1)
do
  wait(bInversed[k][k]);
  compute prodMat(B[k][k],A[k][i],p);
```

```
  notify(prodA[k][i]);
enddo

par(i:=0; p - 1)
do
  # Step 2.1
  if(i neq k) then
    wait(bInversed[k][k]);
    compute mProdMat(A[i][k],B[k][k],B[i][k],n);
    notify(mProdB[k][i][k]);
  endif # Step 2.2
  if(k gt i) then
    wait(bInversed[k][k]);
    compute prodMat(B[k][k],B[k][i],n);
    notify(prodB[k][i]);
  endif
enddo

par( i:= 0; p - 1)
do
  if (i neq k) then
    # Step 3.1
    par (j:=k + 1;p - 1)
    do
      wait(prodA[k][j]);
      compute prodDiff(A[i][k],A[k][j],A[i][j],n);
      notify(prodDiffA[i][j][k]);
    enddo
    # Step 3.2
    par(j:=0; k - 1)
    do
      wait(prodB[k][j]);
      compute prodDiff(A[i][k],B[k][j],B[i][j],n);
```

Site	Nodes	CPU/Memory
Nancy	53	2x DC Intel Xeon, 1.6GHz/2GB
	47	2x AMD64 Opteron, 2.0GHz/2GB
Rennes	99	2x AMD64 Opteron, 2.0GHz/2GB
	25	2x DC Intel Xeon, 2.33GHz/8GB
Sophia	25	2x AMD64 Opteron, 2.2GHz/4GB
Bordeaux	25	2x Intel Xeon, 3.0GHz/2GB
Toulouse	25	2x AMD64 Opteron, 2.6GHz/8GB

Table 4.1: Computational nodes of Grid'5000 used for experiments

```
      enddo
    endif
  enddo
  endseq
```

In bold, the name of component called by the keyword compute. After compiling all components, the application file (graph component) is set to the YML scheduler to be run. The scheduler gives tasks through the backend to the middleware which is in charge to distribute them among processors. The program example illustrates the separation between the computational code and the parallelism which permits to reuse components for an other application or another adaptation. The second implementation of block Gauss-Jordan that we have realized, the inter-step version, has been made by reusing these components and by modifying the graph. End-user which wants to make these modifications with a lower level approach like a middleware, should rewrite all the application and have a complex system to manage the massive parallelism of the inter-step version. Thus the development time is reduced through the high level programming paradigm of YML.

4.5.3 Performance comparison between YML and OmniRPC

The framework reduces the development time by reusing components but execution time may be also reduced with a massively parallel implementation like the inter-step version. Performances of both adaptations in YML and the intra-step version in OmniRPC are compared in this subsection to show the implementation efficiency and the YML overhead. Experiments are done on Grid'5000 with 100 nodes of the Nancy cluster. The details of used resources are given in the Table 4.1.

p×p	Number of tasks	Matrix size	YML intra-steps	YML inter-steps	Gain
2×2	8	3000	344 s	343 s	0.29 %
3×3	27	4500	559 s	465 s	16.81 %
4×4	64	6000	914 s	748 s	18.16 %
5×5	125	7500	1359 s	992 s	27.00 %
6×6	216	9000	2070 s	1362 s	34.20 %
7×7	343	10500	3103 s	2220 s	28.45 %
8×8	512	12000	5008 s	3122 s	37.65 %

Table 4.2: Time comparison between intra-steps and inter-steps for a cluster of 100 nodes on the site of Nancy, with block size = 1500

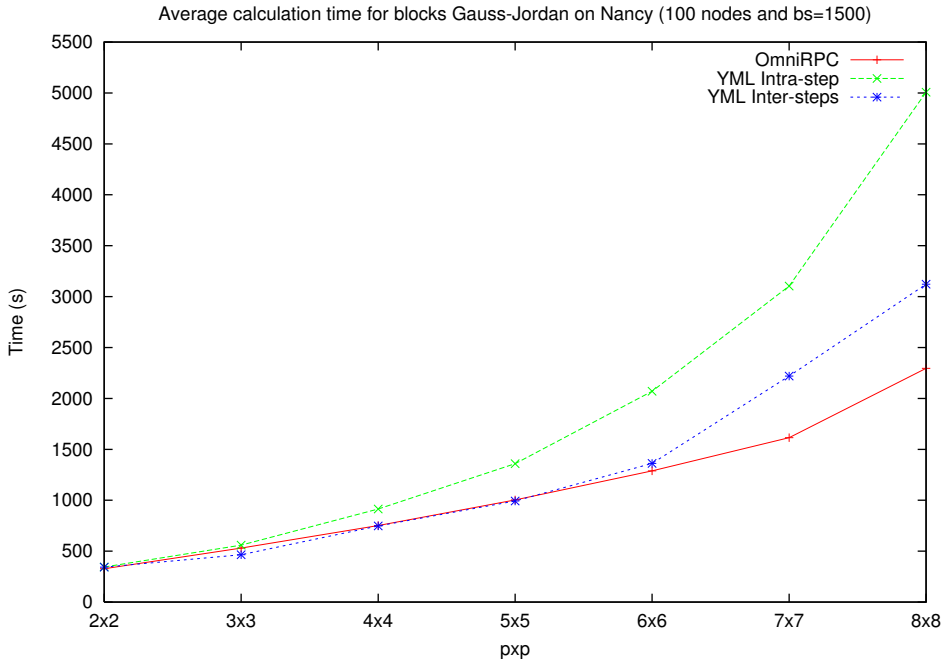

Figure 4.7: Execution time of both adaptations in YML and the intra-step in OmniRPC on Nancy, block size=1500

p×p	Number of tasks	Matrix size	YML intra-steps	YML inter-steps	Gain
2×2	8	3000	274 s	237 s	13.50 %
3×3	27	4500	409 s	346 s	15.40 %
4×4	64	6000	640 s	538 s	15.94 %
5×5	125	7500	824 s	699 s	15.16 %
6×6	216	9000	1114 s	969 s	13.02 %
7×7	343	10500	1530 s	1179 s	22.94 %
8×8	512	12000	2052 s	1645 s	19.83 %

Table 4.3: Time comparison between intra-steps and inter-steps using 100 nodes in Rennes, with block size = 1500

The YML inter-steps version is a massively parallel implementation which should be faster than the YML intra-step. This fact is demonstrated in the Figure 4.7, on which we see that the YML inter-steps performs better than the intra-step version in YML and its performance is closer to the intra-step OmniRPC implementation. The execution time is reduced with this implementation in spite of the YML overhead is still present and is growing with the number of tasks. This is a consequence of the graph analysis by the just-in-time scheduler which are in charge to solve task dependences at run-time and the data repository server which sends and receives data from workers. As the number of blocks is growing the number of tasks is also increased and generates more computing load to find dependences. However, those results are very encouraging because the massively parallel implementation has more task dependences to solve than the intra-step implementation and is faster. The just-in-time scheduler is thus well suited to analyze and find out the dependences in time. Moreover, the improvement in the Table 4.2 between both adaptations in YML is over 25% of execution time, depending on the number of tasks. To get this improvement, end-user has just to modify the graph component by expressing a different graph of dependences with the *Yvette* language. Though, there is an overhead between the OmniRPC program and the massively parallel YML implementation. We can notice that the components reuse for different parallelism and the reduction of execution time decrease the Time-To-Solution by two aspects. Furthermore, YML is independent from middleware that allows YML applications to be reused on a different platform which operates another middleware. One of most important challenges in scientific computing is to get the solution faster and easier. From a computer point of view, faster means good efficiency and easier means less time spent on development of parallel programs. So it is very important for scientific users to modify and improve their parallel program in an easy way by reusing kernels and running them on a different platform conveniently.

As we saw, the high level approach of YML has introduced an overhead comparing to use directly a programming middleware. The graph analysis to find out task

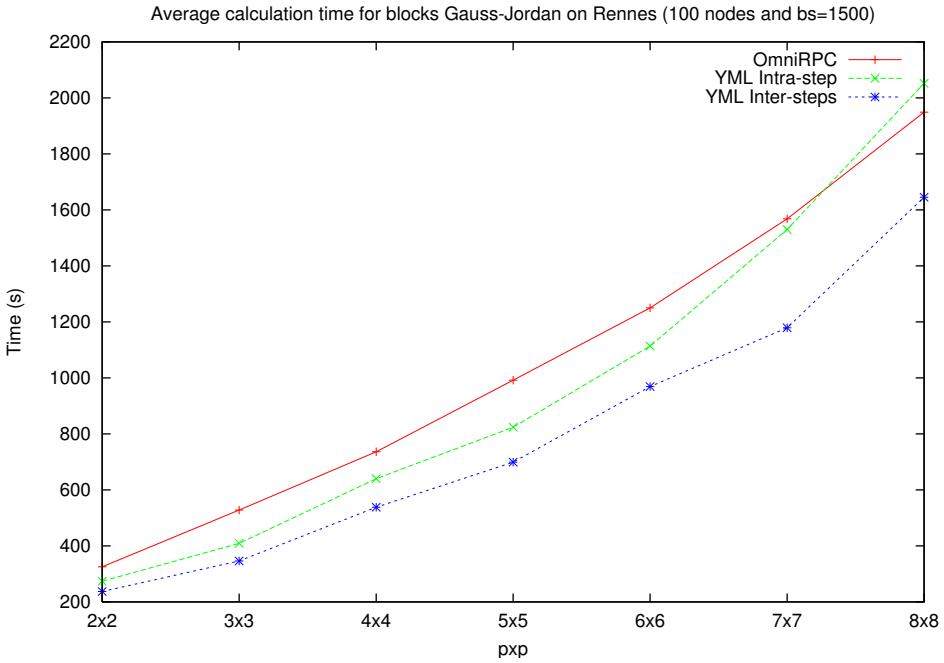

Figure 4.8: Execution time of both adaptations in YML and the intra-step in OmniRPC on Rennes, block size=1500

dependences by the just-in-time scheduler and the data repository server in charge of data exchange between the master and worker nodes are sources of the overhead. It is a system burden which needs computing power to process and answer requests in time. In this part, we propose to change for a more adapted cluster configuration for YML. Experiments are now made on the cluster of Rennes with 100 nodes composed by 99 AMD nodes and one Intel node. The scheduler and the data repository server are placed on a more powerful node, i.e. the Intel node, to respond to the burden. Both adaptations in YML are more efficient compared to results on Nancy because of the global increasing of computing power. The massively parallel implementation is over 15% faster than the intra-step version, see the Table 4.3. The gain is reduced between both implementations compared to previous results. As expected, the YML overhead is considerably reduced that is pointed out on the Figure 4.8. The intra-step version of the block Gauss-Jordan in OmniRPC is slower than the YML implementation, except when the number of tasks becomes higher where the YML overhead reappears.

The increase of computing power has offset the overhead by reducing the consuming resolution of task dependences. However, the difference of scheduling between OmniRPC and YML has also an influence. The first declares dependences through OmniRpcCall and OmniRpcWait functions expressed in a sequential order. This involves a static scheduling in which all dependent tasks must be completed before running the next set of parallel tasks. A more sophisticated implementation could be realized to manage the start and the end of tasks but this is a complex solution for end-users. The second expresses dependences through a graph which is managed at run-time. The just-in-time scheduler is responsible of its management by solving task dependences and allows to run immediately a task when all its dependences are satisfied. This dynamic execution offers the possibility to schedule a tasks when it is ready in an asynchronous way and without carrying end-users. This kind of approach is not recent and has been examined in [72]. YML gives a newer experience of this idea with a higher level programming and its dynamic scheduling has a real advantage in comparison to OmniRPC by running a task as soon as possible. Experiments show this high level approach helps to decrease its overhead when a powerful resource is given to schedule and exchange data.

Results in [73] have shown that the YML overhead is also decreased on a cluster of clusters because of numerous levels of communications which are balanced by the dynamic scheduling of tasks. Our platform distributed over three sites (Lille and Orsay in France and Tsukuba in Japan) and interconnected clusters as TeraGrid, DEISA or NAREGI generate a motivation to evaluate the YML overhead with a more convenient resource configuration on a cluster of clusters. The cluster of clusters architecture is composed of 100 nodes distributed over four clusters: Rennes (Intel nodes), Sophia, Bordeaux and Toulouse. The scheduler and the data repository server are placed on a

p×p	Number of tasks	Matrix size	YML intra-steps	YML inter-steps	Gain
2×2	8	3000	406 s	275 s	32.27 %
3×3	27	4500	490 s	425 s	13.27%
4×4	64	6000	685 s	573 s	16.35 %
5×5	125	7500	936 s	770 s	17.74 %
6×6	216	9000	1210 s	1149 s	05.04 %
7×7	343	10500	1552 s	1532 s	01.29 %
8×8	512	12000	2023 s	1801 s	10.97 %

Table 4.4: Time comparison between intra-steps and inter-steps using 100 nodes distributed over 4 clusters, with block size = 1500

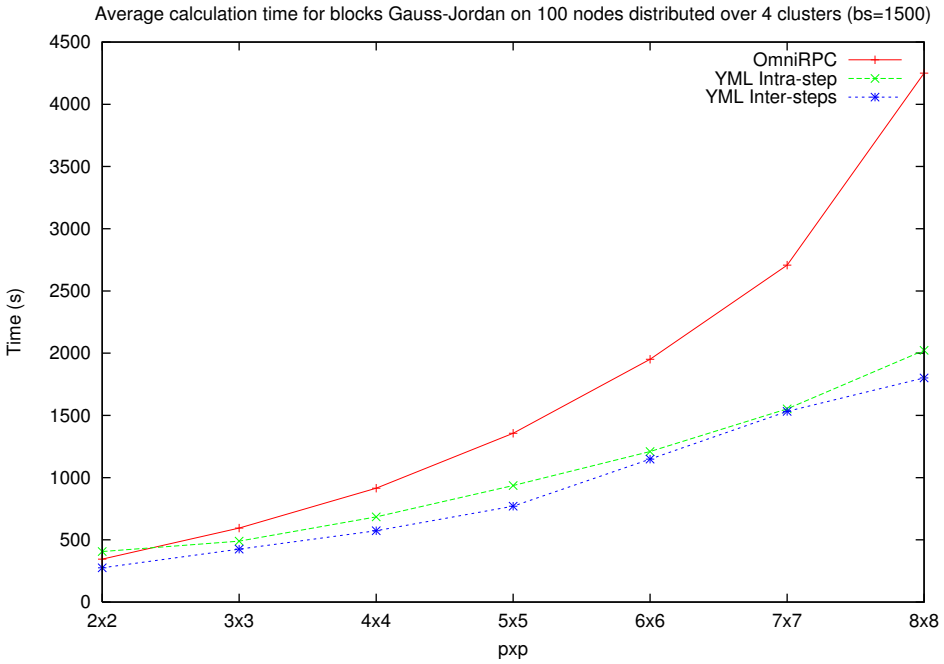

Figure 4.9: Execution time of both adaptations in YML and the intra-step in OmniRPC on a cluster of clusters, block size=1500

node of Rennes. Results in the Table 4.4 show that the gain between both adaptations in YML is approximately the same as the previous experiments on Rennes from 4×4 to 5×5 blocks. The gain is hardly existent as the number of tasks to schedule is growing, for $p > 6 \times 6$. This cluster configuration introduces different levels of networks which have various speeds, latencies and can slow down the execution. The dynamic resolution of dependences by the YML scheduler allows to hide some of these characteristics. Thereby, YML is faster than OmniRPC on this configuration, as shown on the Figure 4.9. Thus the overhead is highly reduced thanks to the use of a more powerful node to schedule and transfer data. The high level approach is helpful in this case of distributed infrastructures and also allows end-users to write easily various parallel adaptation of an application by modifying the dependence graph. All of these contribute to minimize the Time-To-Solution.

4.6 Contribution of a Graph Paradigm for Data Migration Anticipation and Data Persistence

We have seen that Time-To-Solution is decreased with YML through the reusability of components for another parallel implementation. The framework could be also useful to support in a transparent way some mechanisms to reduce execution time and thereby to minimize the development for end-users. Data persistence and data migration anticipation are two optimizations which contribute to reduce the execution time. The first permits to reduce communications by assigning a data set to a node at the beginning of the execution. Computations steps modify them and only the additional data are transfered to achieve the task. The second makes in advance data transfer by migrating or pre-deploying data on a node where they will be needed. Data are migrated when a task ended to the dependent task before it execution and thus overlap communications and computations. Implementation of these techniques often depends on the middleware used. YML has a high level approach which may support them in a transparent way for end-users. The graph description language may be used to anticipate data migration and make data persistence. It describes the control flow from which is induced the data flow after compiling. The just-in-time scheduler could take advantage of these information to anticipate data migration by analyzing the graph. The states of tasks and dependences are known at run-time then it can send in advance data to the core where they are needed to compute the next task. End-users have not to modify their YML programs to use these crucial optimizations. OmniRPC is one of supported middleware by YML and has a data persistence mechanism through *handle* functions which creates a remote connection to a host. The assignment of persistent data to a host and data migration anticipation have to be managed explicitly. The integration in YML of the presented features would generate a system burden to solve dependences and send data

p×p	Number of tasks	Matrix size	OmniRPC	OmniRPC with DP	Gain
2×2	8	3000	328 s	317 s	03.35 %
3×3	27	4500	530 s	508 s	04.15 %
4×4	64	6000	752 s	719 s	04.38 %
5×5	125	7500	1003 s	951 s	05.18 %
6×6	216	9000	1289 s	1210 s	06.12 %
7×7	343	10500	1614 s	1498 s	07.18 %
8×8	512	12000	2294 s	1828 s	20.30 %

Table 4.5: Time comparison between OmniRPC and OmniRPC with DP on the Nancy site, with block size = 1500

by anticipation. We want to make a pre-evaluation of the YML overhead by comparing OmniRPC using data persistence and data migration anticipation and YML having the same behaviours without any concrete management of these techniques. Middleware do not offer a direct support for a direct use of them, thus the actual backends of YML are not able to take into account these mechanisms. That is why they are not really managed in the framework. To make the pre-evaluation, the data persistence (DP) in YML is emulated by regenerating the persistent blocks on the node where the task is executed. The data migration anticipation is really made and explicitly expressed through the graph description language. In consequence, the YML implementation has the same behaviour than the OmniRPC program. The OmniRPC implementation is inspired from [74], each block is generated and referenced on a processor. One block is sent between the steps (1), (2) and (3) of the algorithm. Computations are made where the data are written. In the loop (1), the block A_{ki} is pre-deployed (generated on a node). When the computation of the block B_{kk} is made, it is sent to the loop (1) and (2). In the loop (2), the blocks A_{ki} and B_{ki} are pre-deployed at each step k. In the loop (3), the blocks A_{ik} are pre-deployed at each step k too. The blocks B_{kj} and A_{kj} are sent to the loop (3) when they are ready. For YML, the implementation is the same except that persistent blocks are not referenced but regenerated on nodes where they are needed and one block per task is transferred. Furthermore, the mechanism of data persistence and data migration anticipation in YML are not implemented yet, then in all of these experiments the overhead does not take into account the time to schedule, the analysis of the graph dependences to anticipate data migration and data placement induced by the data persistence. As a result, a value due of these mechanisms is to add to the overhead.

The data persistence allows to reduce communications and data migration anticipation allows to overlap computations and communications. The usual main consequence is the decrease of the execution time. Results in the Table 4.5 compare OmniRPC programs with and without these techniques. They show a gain of performance from 3 to

p×p	Number of tasks	Matrix size	OmniRPC with DP	YML with DP	Overhead
2×2	8	3000	317 s	295 s	-06.94 %
3×3	27	4500	508 s	437 s	-13.97 %
4×4	64	6000	719 s	674 s	-06.26 %
5×5	125	7500	951 s	787 s	-17.24 %
6×6	216	9000	1210 s	1161 s	-04.04 %
7×7	343	10500	1498 s	1516 s	01.20 %
8×8	512	12000	1828 s	1889 s	03.33 %

Table 4.6: Time comparison between YML and OmniRPC with DP on the Nancy site, with block size = 1500

20% in relation to the number of blocks growth. The number of tasks evolves with the number of blocks according to the relation p^3 tasks. Communications decrease by using data persistence and this reduction becomes more important as the number of tasks grows. To have this gain of performance, end-users have to rethink their program and loose time in changing or rewriting the code. If you are not an expert in parallel computing, the balance between the execution time and development can be important to make those modifications. YML could allow to use a mechanism of data persistence and data migration anticipation in a transparent way for end-users. The graph description language could help to anticipate data migration by analysing the graph at run-time as previously explained. The pre-evaluation of YML overhead are given in Table 4.6 . It shows that YML is more or equally efficient than OmniRPC from an overhead of -17% to 1.2% for the test cases presented. The overhead becomes bigger as the number of tasks increases, 3% for 64 blocks ($p = 8$).

The negative overhead comes firstly from the difference of scheduling between the both software. YML is based on a graph description programming paradigm that allows to solve dependences at runtime with the just-in-time scheduler. In contrast, OmniRPC has a static scheduling described by the programmer with some call or wait functions. Secondly, both implementations in OmniRPC and YML have the same parallelism and data movements (one block is only transfered between loops). The data repository server is in consequence less requested for data transfer. Results in [73] have shown that the scheduler and the data repository server are the reasons of the overhead but without a very clear distinction. The data persistence in these new results avoids to use the data repository server and gives to the scheduler a more important role to play in the source of the overhead. Results show the YML overhead is small in this case and demonstrate that the scheduler is quite efficient and the data repository server is the main overhead source. Moreover, YML with this emulation of data persistence and data migration anticipation allows to reduce the processor usage for the send and the receive of data managed by the data repository server. So, the tasks dependences are

solved faster because the program does not need to wait the end of all tasks of one loop before launching an other and the processor is more available to solve dependences. The Figure 4.10 recaps all results and points out that data persistence and data migration anticipation implementation in YML is very efficient.

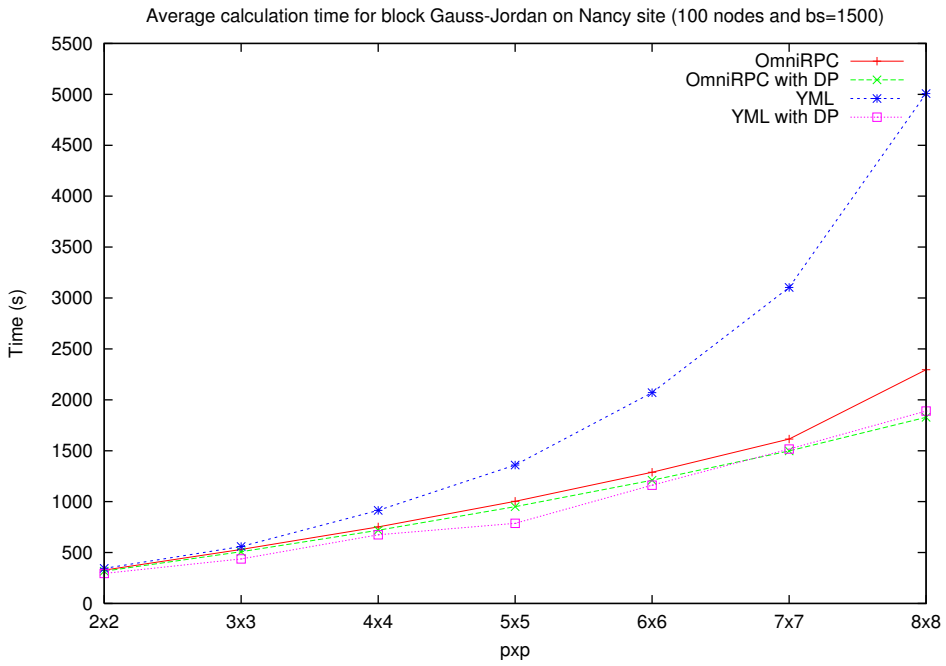

Figure 4.10: Execution time of OmniRPC and YML with and without data persistence on Nancy with block size = 1500

The comparison between YML and YML with data persistence emulation and data migration anticipation have shown that the data repository server is the main source of performance deterioration. The reduce of work for the data repository server with the use of data persistence has balanced the main computational load on the scheduler. Moreover, the growth of tasks increases the overhead because there are more dependences to solve, more computations to schedule and more data exchange between the data repository server and workers. In spite the size of data to send and receive cannot be reduced, the data repository server can be accelerated for an heavy load. It needs computing power because one instance per worker is launched to answer to the requests. A just-in-time scheduling also needs resources to be sufficiently fast enough to process in time the tasks. A lack of computing power is also a factor which has an impact on the overhead.

4.6.1 Overhead of YML in a favorable case

The need of computational power by YML has been introduced in the third section and the previous part to improve performance and decrease the overhead of YML. It is so necessary to evaluate the overhead of YML with a powerful node where the scheduler and the data repository server in the context of data persistence and data migration anticipation to see how much performances can be improved. The same cluster configuration of Rennes is taken then the third section, i.e. a cluster of 100 nodes composed by 99 AMD nodes and one Intel node which is reserved for YML.

p×p	Number of tasks	Matrix size	OmniRPC	YML	Overhead
2×2	8	3000	325 s	274 s	-15.69 %
3×3	27	4500	528 s	409 s	-22.53 %
4×4	64	6000	736 s	640 s	-13.04 %
5×5	125	7500	992 s	824 s	-16.93 %
6×6	216	9000	1250 s	1114 s	-10.88 %
7×7	343	10500	1568 s	1530 s	-02.42 %
8×8	512	12000	1949 s	2000 s	02.61 %

Table 4.7: Time comparison between OmniRPC and YML on Rennes, with block size = 1500

p×p	Number of tasks	Matrix size	OmniRPC with DP	YML with DP	Overhead
2×2	8	3000	272 s	247 s	-09.19 %
3×3	27	4500	450 s	359 s	-20.22 %
4×4	64	6000	635 s	556 s	-12.44 %
5×5	125	7500	856 s	692 s	-19.15 %
6×6	216	9000	1092 s	984 s	-09.89 %
7×7	343	10500	1357 s	1332 s	-01.84 %
8×8	512	12000	1662 s	1660 s	-00.12 %

Table 4.8: Time comparison between OmniRPC and YML with data persistence on Rennes, with block size = 1500

Execution times have been reduced with this new configuration, a few or hundred seconds for OmniRPC and from 50 to 700 seconds for OmniRPC with data persistence, see Table 4.7 and 4.8. The lowest execution time results are obtained by the use of data persistence and data migration anticipation with OmniRPC. They come from the reuse of data on a node which allows to have less data exchange and to decrease the amount of communications. Furthermore, as expected by the increase of the computational power, the performances of YML are improved and the comparison between the results

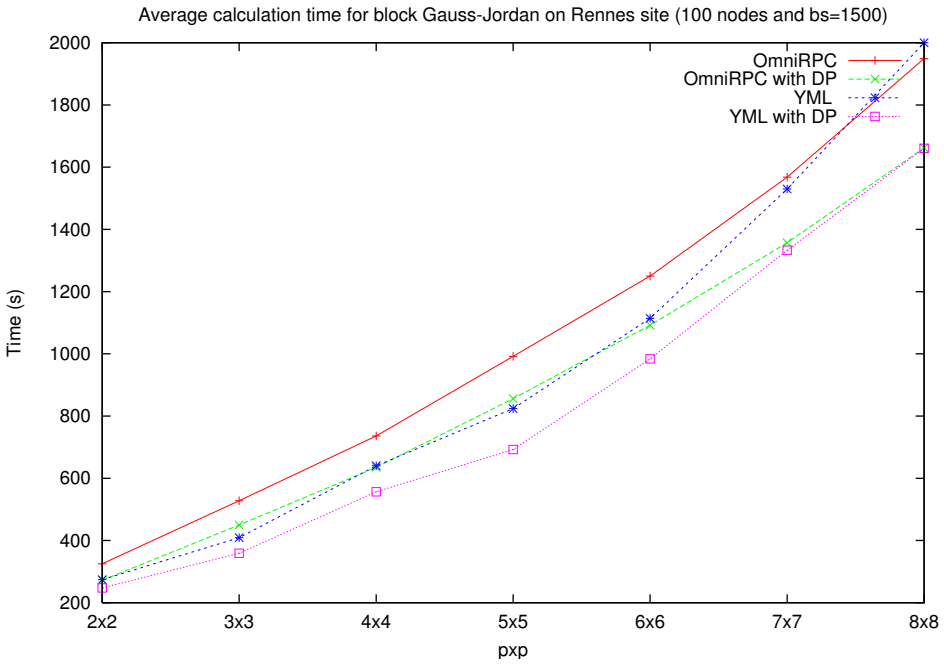

Figure 4.11: Execution time of OmniRPC and YML with and without data persistence on Rennes with block size = 1500

of Nancy on Figure 4.10 and the results of Rennes on Figure 4.11 are more interesting. The execution times of YML without data persistence and data migration anticipation implementation are faster than the previous on Nancy, from 20% to 50% on the new cluster configuration. In the same way, the YML implementation with data persistence and data migration anticipation is a little bit faster from 12% to 17%. These results with a different cluster configuration show the performances of YML can be globally improved when it has more computational power to solve dependences and transfer data. As a consequence, the overhead is decreased with the use of a powerful node and that is demonstrated in the Table 4.7. The overhead is reduced 2% for 64 blocks and is also negative from -10% to -22%. YML is in consequence faster than the OmniRPC implementation. For the use of data persistence and data migration anticipation in YML, overhead is decreased too but of a few percent. The reduction comes from the supplementary power which allows to YML to solve task dependences more efficiently. Moreover, the data repository server can answer to the requests and distribute data faster.

The previous part and this one have demonstrated two elements for YML. The first is YML can be an interesting framework to program an application with the use of data persistence and data migration anticipation because it does not generate a huge overhead without rewriting the application. The second is it has been demonstrated that the data repository server is mainly responsible of the overhead because the use of data persistence has reduced data transfer and has decreased requests to it. This part has confirmed this point with a different cluster configuration which is more adapted for YML with a powerful node. The overhead is also decreased in this case. YML can be interesting for performance because it uses a graph description paradigm with a component approach. This approach offers to end-users the possibility to minimize the Time-To-Solution with a low cost and without any extra programming by reusing components and running the same code on different middleware.

4.6.2 Cluster of clusters

As presented in the third section, we have an interest to evaluate YML on a cluster of clusters. The same platform is now used in the context of data persistence and data migration anticipation to observe the impact of it.

The cluster of clusters architecture introduces different levels of network communications which can slow down the execution of an application. This is pointed out in the Table 4.12 and 4.10 where all-times of execution are increased if they are compared with the results obtained on Rennes. The performances of YML with and without data persistence and data migration anticipation are closer to the execution time get on the cluster of Rennes for p greater than 3. By contrast, OmniRPC performances are

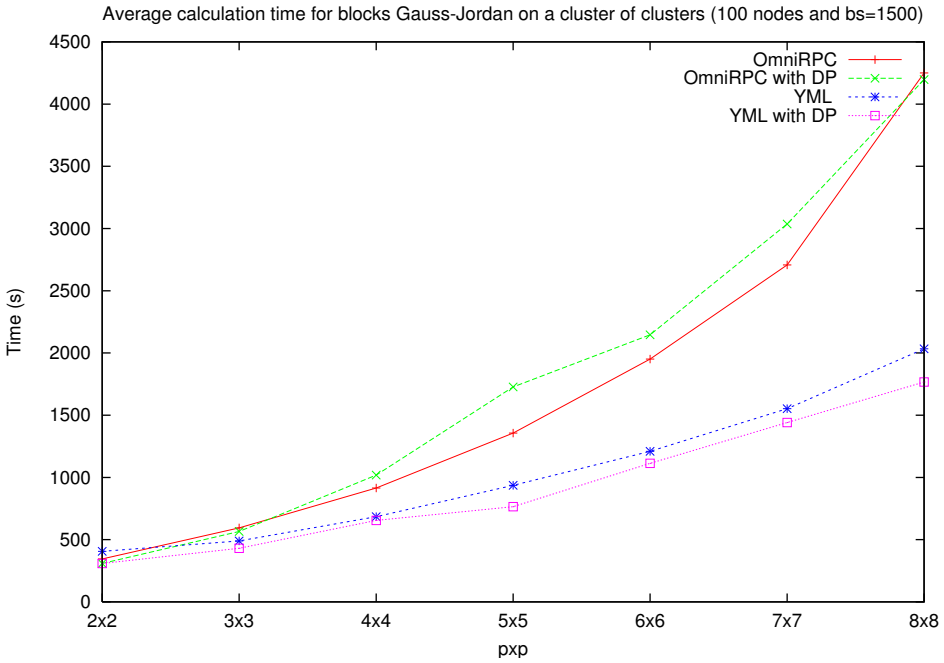

Figure 4.12: Execution time of OmniRPC and YML on a cluster of clusters

p×p	Number of tasks	Matrix size	OmniRPC	YML	Overhead
2×2	8	3000	345 s	406 s	17.68 %
3×3	27	4500	595 s	490 s	-17.64 %
4×4	64	6000	915 s	685 s	-25.13 %
5×5	125	7500	1356 s	936 s	-30.97 %
6×6	216	9000	1951 s	1210 s	-37.98 %
7×7	343	10500	2708 s	1552 s	-42.69 %
8×8	512	12000	4250 s	2034 s	-52.14 %

Table 4.9: Time comparison of OmniRPC and YML on a cluster of clusters of 100 nodes, with block size = 1500

p×p	Number of tasks	Matrix size	OmniRPC with DP	YML with DP	Overhead
2×2	8	3000	309 s	310 s	00.32 %
3×3	27	4500	566 s	431 s	-23.85 %
4×4	64	6000	1019 s	656 s	-35.62 %
5×5	125	7500	1727 s	765 s	-55.70 %
6×6	216	9000	2146 s	1113 s	-48.13 %
7×7	343	10500	3036 s	1441 s	-52.53 %
8×8	512	12000	4197 s	1767 s	-57.89 %

Table 4.10: Time comparison of OmniRPC and YML with data persistence on a cluster of clusters of 100 nodes, with block size = 1500

decreasing drastically. In the same way than previous parts, the dynamic tasks scheduling permits to hide some communications latency which are generated by the different levels of network. Moreover, the use of a more powerful node drastically decreases the overhead and allows to YML to be faster than OmniRPC, see the Table 4.9 and 4.10. For the best case that we have evaluated, the overhead is -52% for YML without data persistence and data migration anticipation, then the execution is divided by two and the worse result is 17% for 4 blocks ($p = 2$). This loose of performance is explainable by the fact that the tasks scheduling takes more time than computations. The overhead for the data persistence and data migration anticipation implementation is also reduced in the same order 57% for the best and 0.32% for the worse. The comparison of YML with and without data persistence and data migration anticipation points out a small gap of execution time, see Figure 4.12. The data repository is the main responsible for this lack of performance but the data persistence implementation where the just-in-time scheduler is mainly used shows that this point can be improved with a more powerful node in this cluster configuration. The overhead is in consequence negative and demonstrates the efficiency of YML when the scheduler and the data repository

server use an adequate powerful resource.

4.7 Conclusion

In this chapter, we have seen that a graph description paradigm is a high level programming approach. It allows end-users to develop parallel program without expressing communications between tasks. Those communications are extracted from the task dependences graph which is expressed separately from computational code. This segregation between the description of parallelism and computational kernel is realized by the component-oriented programming of YML. It also helps to modify parallelism without interfering in the writing of computational code. An illustration of this aspect has been given through two parallel and distributed adaptations of the block Gauss-Jordan method. The first adaptation was the parallelization of the different loops which compose a computation step. The second adaptation was a massively parallel implementation of the matrix inversion method which consists of adding the parallelization of the computation steps. This adaptation has been written by modifying the graph of dependences and reusing components of the first adaptation. Thereby, the development time has been reduced by reusing computational kernels and expressing a different parallelization easily through the graph description. So the graph description paradigm contributes to reduce development time and improve productivity which is a criteria for the programming paradigm candidate for exascale. This paradigm should also increase productivity for debugging by extending implementation component of YML with a graph which should give information on where the program has unexpectedly stopped. For fault-tolerance, the graph description of application and components should offer to restart application exactly where a fault has occurred.

Moreover, the graph description paradigm avoids end-users to manage explicitly communications which are managed at run-time. The just-in-time scheduler is in charge to drive communications and task running by solving task dependencies during the application execution. Unfortunately, communications in YML are only synchronous. However, the just-int-time scheduler may be extended to analyze the graph of dependencies at run-time to know when and where data are necessary and anticipate their migration by making asynchronous communications. Data persistence and data migration anticipation experiments with YML were good example to illustrate this point. Experiments have shown YML is faster than OmniRPC which has a static-scheduling in opposite to YML thanks to the dynamic dependences resolution during the run-time. The use of the graph dependences to take advantage of data persistence and data migration anticipation, asynchronous communications, gives end-users the possibility to exploit these optimizations without any extra programming. This contributes to reduce development and execution time and so to minimize the Time-To-Solution.

For exascale computing, the productivity is an important criteria to satisfy for the programming paradigm candidate but efficiency is the first critical parameter. The management of communications and computation resources by a scheduler should contribute to improve efficiency and discharge end-users to write sophisticated management systems. As we have seen, a graph description paradigm like YML provides many information which help to schedule task and may help to realize asynchronous and optimized communications. These information may also contribute to optimize and get efficient data movement. However, this just-in-time scheduling in YML based on graph suffers of performance degradations that are planned to correct with some research works on grid scheduling and Many-Task Computing. The other problem with this paradigm for exascale is scalability which is highly important. Considering that in our proposal, the graph description paradigm is to program supercomputers at the high level, so at coarse-grain, graph management should scale to million of tasks. Nevertheless, this point remains to demonstrate with YML.

The interesting points underlined for exascale are also valuable for TOTAL. For their applications, a graph description may increase productivity and the run-time management may improve efficiency with task scheduling and asynchronous communications which are frequent in the main used domain decomposition application for seismic imaging, i.e. the reverse time migration [75]. As we have pointed out, the graph description provides many information to optimize communications, restart applications where it stops unexpectedly in case of faults or execute tasks when they are ready to run. Nevertheless, tasks and I/O dependences may be also extracted from the graph in order to anticipate data buffering (prefetching) and move data in advance where they will be needed, in the same way as data migration anticipation. In the next chapter, we propose to explore a such advantage of the graph description paradigm to optimize I/O.

Chapter 5

Asynchronous and Smart IO Delegation System:ASIODS

5.1 Introduction

As computing power growth to reach Exaflops, the amount of data processed and generated by applications should also increase and exascale storage systems should have half an Exabyte of memory to handle this deluge of data. Those storage systems should be scalable to be accessed simultaneously in competition by million of cores and be able to deliver an average bandwidth of tens TB/s. To get this throughput, storage systems should be organized in a memory hierarchy of RAM, Solid State Disks (SSD) and drives which require to be exploited efficiently to unveil their maximum potential. The problem is common storage systems are built to handle the worst case of synchronization, conflicts and coherence, considering I/O as separate activities from computations. Thereby, the various memories of exascale storage systems should only use common caching techniques such as least recently used and most frequently accessed files which do not take advantage of end-users information about application I/O. As the previous chapter has shown, a graph description paradigm gives many information on task dependencies and may be extended to extract I/O and task dependencies. This dependency knowledge should help to perform I/O in advance in order to hide disk latencies and realize asynchronous storage accesses to overlap I/O and computations.

In this chapter, we propose to use end-users knowledge about task and I/O dependencies to optimize and realize "smarter" I/O. The end-user expertise may be also combined with I/O performed in delegation, also known as I/O forwarding, which has demonstrated improved performances on large systems by avoiding disk contention by competing accesses. To demonstrate the interest of such approach, we have developed an

asynchronous and smart IO delegation system (ASIODS) which combines both aspects of end-user expertise and I/O delegation. For the moment, this system has not been integrated in a graph description paradigm such as YML which may help to support I/O into the programming paradigm and offer a transparent processing of I/O to end-users. The next section outlines related work on I/O optimization such as I/O forwarding or collectives I/O and explains the differences with our approach. The architecture and the implementation design of ASIODS are presented in the third and fourth section. A preliminary evaluation on a common BLAS3 operation is realized and presented in the fifth section. An analysis and a discussion around the interest of such approach for exascale supercomputers are made in the last section of this chapter which concludes and outlines the further extension and integration in a graph description paradigm.

5.2 Related Works on I/O Optimizations

To improve storage performance, the main optimization of file system is to keep the most recent used data in system memory to get a faster access. This method is named caching and allows to take advantage of the read output of RAM which is 10 times faster and has a lower latency than hard drives. In parallel and distributed programming, the most common used library (MPI) offers some I/O optimizations with ROMIO [76] such as caching or collective operations. Some other researches have increased performances by delegating I/O operations, also known as I/O forwarding, to a subset of nodes [77] [78] to avoid the disk contention and to not overload disks with a huge number of requests. The mix of both ideas has been tried and is successful to increase storage performances by delegating I/O and caching on a system formed by a group of nodes [79]. All of these techniques rely on collective I/O, caching and I/O forwarding to optimize disk accesses and by keeping most recent used data in a fast memory zone. However, this kind of technique is limited and does not give the possibility to use efficiently the cache for irregular I/O accesses and take in account the user expertise on applications. One solution is to prefetch data in a memory zone which has a fast access rate and a low latency. Our proposal uses this knowledge and in contrary to the proposed approaches does not rely on specific operating system to make I/O forwarding. In [80], a fragment of application is firstly run to identify future I/O references and generate prefetch requests. Our approach does not use a pre-execution to determine prefetch requests but I/O is expressed in relation to tasks separately from the computation code by the end-user. Although our first implementation lacks of maturity and is not completed, it offers to insulate computations and I/O by using the parallelism is expressed at a high level by end-users. These expressions come from the fact that many algorithms of scientific computing are deterministics in number of operations, data exchange and I/O. Therefore, this knowledge may be used to prefetch data in a smart way and may offer

to overlap computations and communications. The memory zone where data are put in advance could be also used to write data asynchronously on the storage. We propose to explore an advanced approach for an asynchronous and smart I/O management based on the I/O delegation for prefetching or write through on a dedicated part by using the I/O dependency graph of applications.

5.3 Architectural Approach

Supercomputers are designed to achieve high performance for intensive computing by running many thousand cores interconnected through a high bandwidth network and a shared high performing storage. However, machines can be seen as a distributed system composed of two subsets. The first one is composed of cores dedicated to computations and the second one is a set of cores and disks of the storage. Both are thus in charge of specific tasks. The I/O delegation allows to take advantage of the specialization by separating concerns of computations on the *Cluster* part and storage accesses on *Storage* part which exchange data through a memory zone named the Global Buffer Memory (GBM), as shown on the Figure 5.1. The *Cluster* part (red circle) is composed of cores dedicated to computations (small pink circles). The computation cores execute scientific kernels of the parallel application. One node of the Cluster or outside, the *Master*, does not run any computations and is in charge of managing the application execution. It has the graph of I/O dependencies, each node of the graph corresponds to a task associated to different files needed, the size of data, the offset, the process ranks and a completion status of the I/O. Each I/O operation is considered as a request and all are sent to the *I/O Manager* of the *Storage* part in order to make a copy. A synchronization is regularly made between both parts to update the completion status which are used to authorize computation cores to read or write data into the GBM. The *Storage* part (green circle)is composed of cores dedicated to the storage (small green circles) and an *I/O Manager* process. The *I/O Manager* is in charge to receive the requests from the *Master* and to dispatch it to the storage cores. Each storage core has its own request list to process and manages the space available of its physical memory which belongs to the GBM. The *I/O Manager* is synchronized with storage cores to know the completion status of each request and the space available into the GBM. To exchange data between computation and storage cores, a memory zone is defined, the Global Buffer Memory. Data are placed in this memory in order to be processed from or to the storage. The properties of the memory zone are very low latency and high read/write performances, in fact better than the storage system drives. The GBM is physically distributed among many devices to load balance the processing burden and reach a large space of exchange which is considered as one memory space managed by the *I/O Manager*. Devices is a generic word to refer to various storage elements which

can be different kind of natures. It can be a bunch of Solid State Disk (SSD) set on each nodes to give a large and fast storage for each processor, some distributed memory on non-used nodes of the *Cluster* to offer to transfer needed data as close as possible of processors or storage nodes to enhance caching capabilities by reading in advance data. All these configurations depend on the most suitable strategy for a determined application because they have some trade-offs.

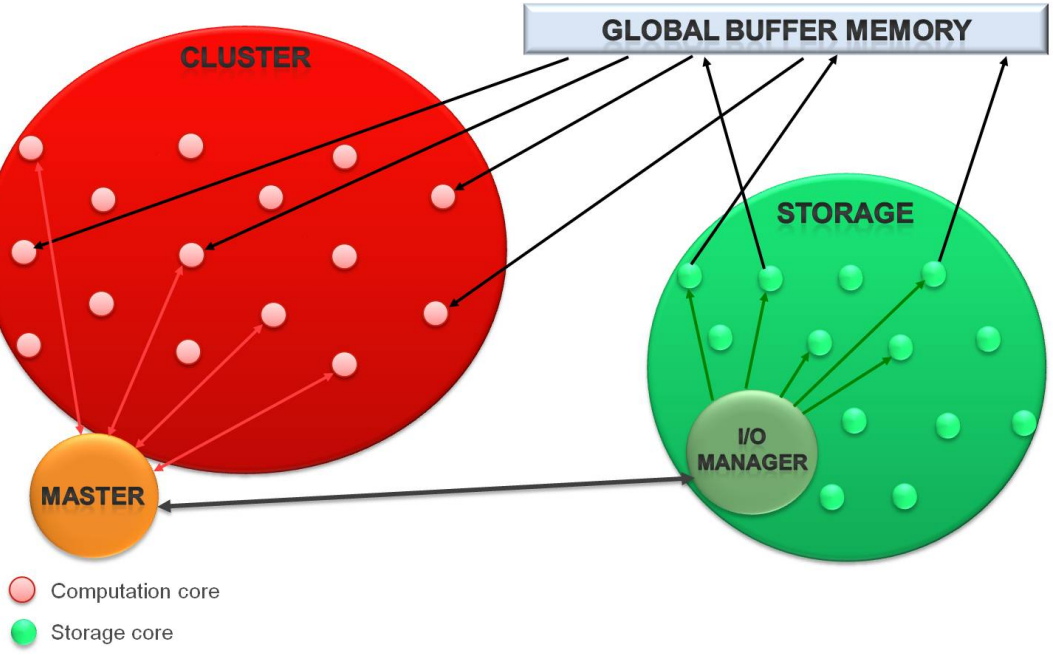

Figure 5.1: Asynchronous and Delegated I/O Architecture

5.4 ASIODS Design

After the presentation of the architecture, this section explains how the storage part and the *I/O Manager* are implemented to form the ASIODS Server. It also details the functioning of the cluster part and the *Master* which form the ASIODS client and are illustrated through an application example of a dense matrix-matrix multiplication.

5.4.1 Server Side

In our implementation of the server, the *I/O Manager* is integrated as a part of the storage core which runs on a multicore processor. Figure 5.2 shows the organization of

parts and illustrates how they interact. The server is in charge to receive requests from the *Master* over the network. These requests are get and placed into the request list to be processed. The I/O Thread Pool is composed of a pool of threads which are in number of two in our case, but a higher number can exist if necessary. Threads retrieve requests in a FIFO order from the list and process I/O. If the request contains an I/O read, data are processed from the storage into the GBM. In opposite, if the request contains an I/O write, data from the GBM are written to the storage. The completion status of the request is after changed to complete in order to update the list handled by the *Master* during the synchronization. At each processing, the available space in the GBM is monitored and updated before any operation on it. Another thread pool, the Data Exchange Thread Pool, is in charge to send or receive data to/from computational cores. The thread pool has three threads in our case. Each thread processed a request received from a computation core and looks for it in the request list. The found request contains the pointer where data are located into the GBM. This pointer is used to send or receive data over the network to or from the computation in demand. For the GBM, we have chosen to allocate the RAM of one node as but it could be partitioned among various nodes which will be managed by the *I/O Manager*.

5.4.2 Client Side

In our implementation of the client, end-user declares I/O and task dependencies through an API which takes as parameters the file name, the offset, the size, the number of accesses and the task name. These dependencies are handled by the *Master* as an ordered list that is planned to be extended by a graph and high level management of tasks in a next version to get a transparent support. The *Master* is composed of two threads. The first one is the Exchange Thread which is dedicated to send requests at start-up and synchronize the completion status of requests with the *I/O Manager*. During the synchronization initiated the *Master*, unnecessary requests are removed from the *I/O manager* in order to get free space in the GBM. These requests are those which are no longer necessary because data have been retrieved by the computation cores. The second thread is the Authorization Thread which is in charge to allow computation cores to access the GBM. It receives request details from a computation core and looks for the completion status in the ordered list of requests. If the I/O of the request has been processed, the thread sends the authorization access to the requesting core. In other case, the computation core is placed in a waiting state until the completion status of the request changes to complete. It will be notified and granted of an authorization access at the next synchronization between the *Master* and the *I/O Manager*, if the completion status has changed. The computation core is only composed of one thread in charge to ask authorization access at the GBM to *Master* and to retrieve data. All these explanations are illustrated and summarized in the Figure 5.3.

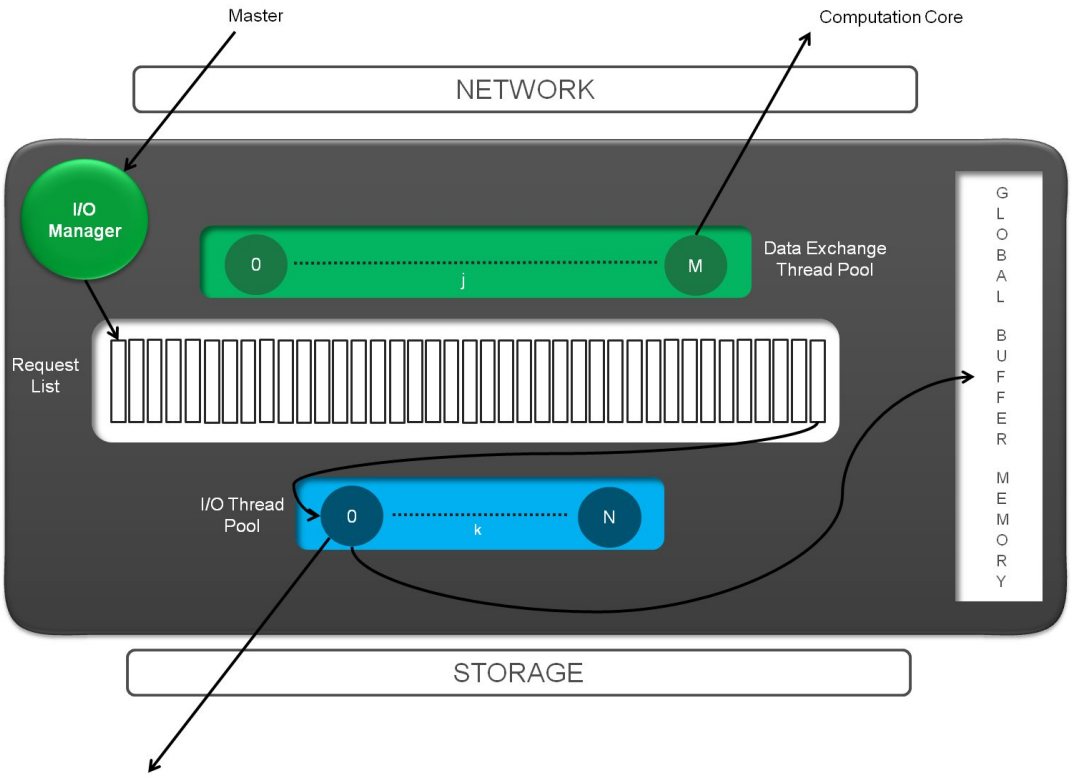

Figure 5.2: ASIODS Server

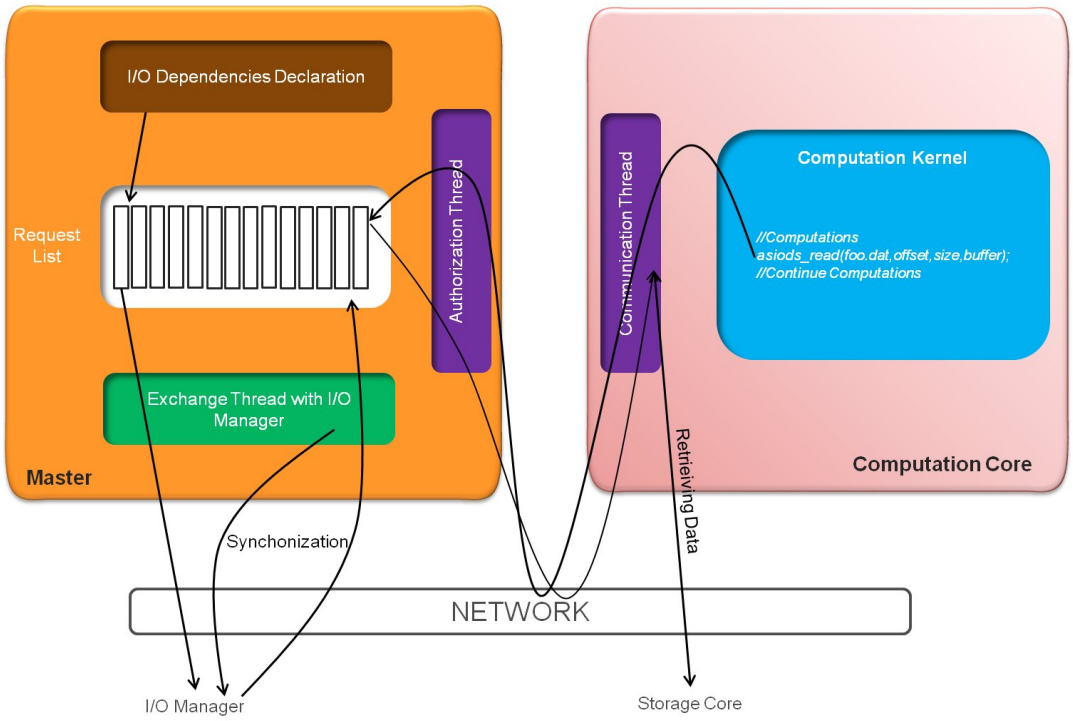

Figure 5.3: ASIODS Client

5.4.3 Example: Matrix-Matrix Product

The classical version of the algorithm computes successively a product between each element of a row and each element of a column of two square dense matrices, A and B, and sums the result of each element product which is an element of a result dense matrix C. For the blocked version, square dense matrices A, B and C are defined and partitioned into $NbBlock \times NbBlock$ blocks of dimension $BlockSize \times BlockSize$. Each block is also divided into sub-blocks of a fixed size (2000×2000) in order to increase the number of read accesses. To process the product, this version uses Algorithm 5.1 which has 3 loops to multiply each block row I of A by each block column J of B, sums the block I,J of C and puts the result into it. We notice that a matrix-matrix product is realized to multiply each block. To run it on many cores, the algorithm is parallelized with MPI (Message Passing Interface) by allocating each block C[I,J] computations on a core such as $rank = I * NbBlock + J$. The algorithm order is of $NbBlock^3$ and runs on p cores such as $p = NbBlock^2$. We have developed three versions of the block matrix multiplication in order to compare the peak performance reachable by the application in different modes. The first one is named *No I/O* which computes

all data from RAM. The second one is named *Classical I/O* reads at each step of the loop K the blocks A[I,K] and B[K,J] and keep in RAM the block C[I,J]. The third one is our approach of I/O delegation and is named *ASIODS*. The Master sends all I/O to the *I/O Manager* which registers it. The computation processes ask for data in the same way as the *Classical I/O* version and receive it when they are loaded in the GSM. Thereby, the *Classical I/O* and *ASIODS* version have the same I/O behavior and read $2 * (NbBlock \times Blocksize/2000)^3$ blocks during the execution.

Require: A (partitioned into $NbBlock \times NbBlock$)
Ensure: $C = A \times B$
  for $I = 1$ to $NbBlock$ **do**
    for $J = 1$ to $NbBlock$ **do**
      for $K = 1$ to $NbBlock$ **do**
        $C_{IJ} = A_{IK} \times B_{KJ}$
      end for
    end for
  end for

Algorithm 5.1: Block Matrix Product Algorithm

5.5 Experimental Platform

After the presentation of the design, this section presents the performance results of ASIODS in order to evaluate our approach and to find improvement points. Experiments are based on a platform composed of one client for computations and one storage node which are directly interconnected through an InfiniBand link. The client and the storage node configurations are given in Table 5.1. The storage node has 120 drives formatted with the GPFS file system to provide a total capacity of 5.5TB. For our evaluation, we have chosen to take advantage of the open space to users on the storage platform and use as GBM the memory available in the storage node, i.e. 10GB. Experiments compare performances of a block matrix product algorithm with classical I/O and with ASIODS. Only the blocks in input are retrieved from the storage. Various number of blocks and block sizes of the algorithm have been set to modify the number of I/O and see the performance behaviors.

5.6 Performance Study

The knowledge of the relation between tasks and their I/O needs offers the opportunity to have a better cache management by moving in advance necessary data from the stor-

Name	Processor	Cores	Memory	Network
Storage	Xeon E5530 2.40 GHz (Nehalem)	4	10GB	Mellanox IB 4xQDR (40 Gb/s)
Client	2x Xeon E5410 2.33 GHz (Harpertown)	8	16GB	Mellanox IB 4xQDR (40 Gb/s)

Storage Configuration			
File System	Capacity	GPFS Read Bandwidth	
GPFS	5.5 TB	2.7 GB/s	

Table 5.1: Hardware Resources

age to a memory zone with a higher throughput. ASIODS is aware of these information and also insulates I/O by delegating these operations on a dedicated node. Thereby, our approach may accelerate accesses to the storage in comparison to a classical I/O use. For a block matrix multiplication partitioned into 2×2, 4 processes are executed in parallel and get back at each computation step two blocks of matrix A and B from the storage. The implementation of the algorithm with ASIODS gives an average of 8% of execution improvement over the version with the classical I/O, as shown in the Figure 5.4. Moreover, the performance gap becomes smaller between *ASIODS* and the *No I/O* version which is the peak performance sustainable by the application.

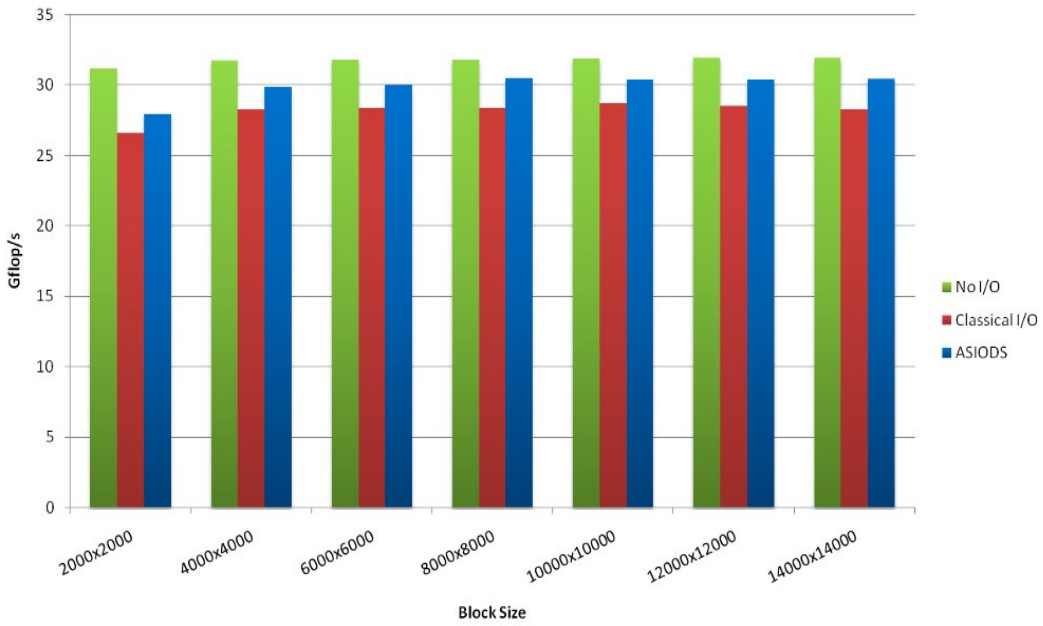

Figure 5.4: Performance for a block matrix product of 2x2 blocks

To evaluate our approach with a higher number of requests at the same time, the block matrix product is now partitioned into 3×3 blocks and run 9 processes in parallel. As shown in the Figure 5.5, ASIODS gives 11% of execution improvement over the *Classical I/O* version for a block size of 2000×2000 and it is still achieving good performances on other cases. However, results decrease as the block size grows, i.e. number of I/O. This comes from the data status checking between the Master and the computation processes in order to give the authorization access to the GBM. In our implementation of ASIODS, only one thread is dedicated to receive requests to check if data are ready or not to be retrieved for computations. It creates a communication bottleneck that we plan to correct in a next version.

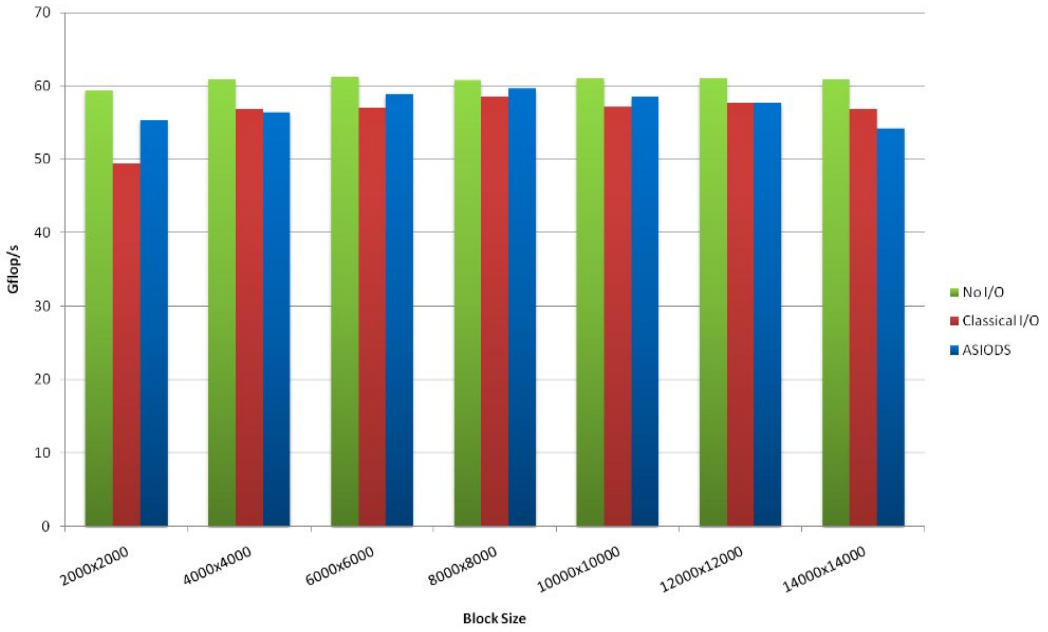

Figure 5.5: Performance for a block matrix product of 3x3 blocks

In previous experiments, both versions have been evaluated by modifying the problem size to increase the number of I/O. Data size to retrieve from the storage was at a fixed dimension that have a constant I/O access time and did not significantly involve the network latency. The ratio between computation time of sub-block products and I/O access time to get data was thus kept at the same value. We now propose to play on these parameters by fixing the block size and by modifying variously the size of sub-blocks. A matrix block in our algorithm is divided into sub-blocks which are read from storage at each computation step. Smaller is the sub-block size, higher is the number of sub-blocks and thus the number of I/O which takes a higher part of the computation

time. ASIODS also achieves good performances with an average of 10% of acceleration in comparison to a classical I/O usage, as shown on the Figure 5.6. This result is especially for small sub-block cases in which more data are read in advance and placed in the global buffer memory. It allows to hide the disk latency that is more important when a small data set must be read and play a higher part in the global execution time, in particularly when the range of data to process has a short computation time. However, ASIODS generates a penalty for larger sub-blocks when blocks have a small division and the amount of I/O decreases. In our model, the Master sends requests to the *I/O Manager* which has to move forward from the storage to the GBM. The transmission of requests and the synchronization between both parts is impacted by the network latency that involves lower performance on large sub-blocks. It was not clear on previous experiments because the amount of data read in advance was sufficient to hide this problem. In these cases also, the computation time of sub-blocks is higher than the I/O access time which involves a lower contribution of I/O in the global computation time. ASIODS should improve the performance, a little bit less because of this point, but the request processing is also in cause of this performance degradation. The request transmission and processing must be improved in a next version in order for our approach to be efficient in most I/O cases. Despite the fast ASIODS suffers from performance loss. The approach offers a segregation between computation and I/O and so a more transparent way for end-users to use efficient I/O by taking advantage of smarter cache management.

5.7 Conclusion and Perspectives

In this chapter, we have presented the design of an asynchronous and smart I/O delegation system and have evaluated the first implementation in order to validate the proof of concept and its potential. Those results have been also presented in [81]. The simple algorithm benchmark with a low I/O bandwidth has demonstrated an execution improvement of 8% than a classical I/O usage by moving in advance data from storage to a high throughput memory zone in order to get faster accesses and hide disk latency. Performance speedup obtained is not very high and without our approach can be balanced with a supplementary I/O rack in order to get more I/O bandwidth. However, results have shown ASIODS helps to go closer to the peak performance of the application and we have explained the source of performance degradation. Problems are the transmission latency of requests between the Master which knows the relation tasks dependencies and I/O and the *I/O Manager* which is in charge to move data from the storage to a memory zone with a higher throughput, the GBM. The request processing by the *I/O Manager* was also detected as a bottleneck. We plan in a second stage to improve these points and extend the evaluation to write performances on a real

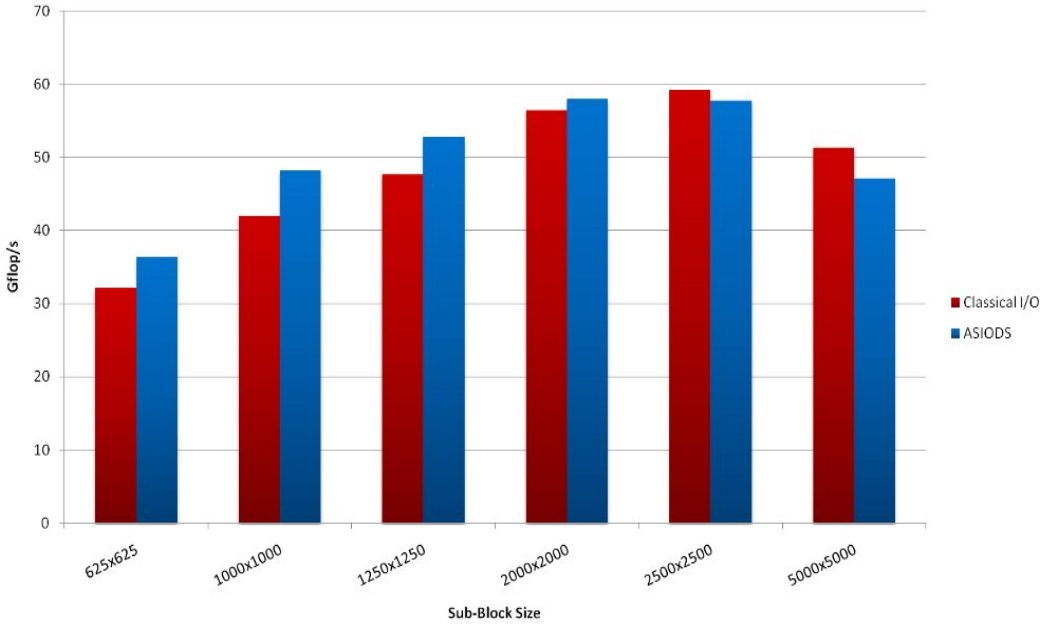

Figure 5.6: Performance for 3x3 blocks with a fixed block size of 10000×10000

application.

Moreover, ASIODS allows to hide disk latency by reading advance data and also avoid disk contention by minimizing the number of clients which access the storage at a time. The relation knowledge between task dependencies and I/O gives a more end-users approach in which I/O are expressed at a high level and are transparently processed and optimized. We have observed during the implementation of our benchmark that it is easy to describe the task dependencies and I/O for a simple application with common programming language but it should be more complex for direct methods. A better way and more end-users oriented to express task dependencies should investigate such as the graph description language of YML [16]. With a high level programming language and the description of the relation between tasks dependencies and I/O offer a way to integrate I/O into a programming model for Excascale supercomputers. The delegation is also an interesting point for some applications which need to apply a pre or post-processing treatment on data, such as some computational fluid dynamic applications. This operation with our approach and the filtering approach of DataCutter [82] could be put together in order to delegate on a node pre and post-processing treatments and thus speedup execution of parallel and distributed applications.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This thesis proposes a global programming paradigm for exascale supercomputers which relies on a multi-level programming paradigm. Those paradigms are proposed in relation to the hardware decomposition of these machines in 3 levels. At the low level, the base computing element should be many-cores processors. We have proposed for this hardware level a data parallel programming paradigm in the third chapter. It has the particularity to be focused on data movements and mapping which are crucial on many-cores. These parameters are typically hard to optimize with irregular data accesses such as sparse matrix computations. Data structures defined by sparse formats play an important role in the performance achievement of sparse matrix computations. To explore data parallelism, we have implemented and evaluated the common sparse formats on a many-core processor (GPU) for the sparse matrix vector product. Our analysis has shown data parallelism offers to exploit the different memory levels on the chip and to optimize accesses to the main memory. However, for matrices with highly irregular data structures and for some sparse formats like column compressed, the lack of direct communication possibilities between cores inevitably generates penalizing accesses and a bottleneck of the main memory. This should be avoided on the future many-cores with the integration of network-on-chip between cores. This feature should allow to optimize communications on the chip, avoid cache misses and thus reduce the main memory bandwidth consumption. The problem of many-cores programming is not very new and is continuing to grow with the multiplication of cores. Unfortunately, we can notice that few initiatives of programming framework and standard have been set out on to attack this issue, except HTA,OpenCL [83], Ct or OpenMP locality extension [84].

At the middle hardware level of excascale machines, socket of many-cores processors should be interconnected by an optical link with a high bandwidth. To take advantage of this fast communication path and network topology between nodes, we have proposed a message passing paradigm to realize point-to-point communications between sockets or groups of node located at one hop. This paradigm and communication aggregation have not been explored in this thesis because it is a well known programming paradigm used for many supercomputers.

At the high hardware level, exascale machines will be composed of millions of nodes to reach a total core counting of a billion. For this level, a graph description paradigm has been proposed in the fourth chapter. The YML framework has been used to explore this paradigm for these particularities to provide a component-oriented programming and a graph description language based on a direct general graph. Through the programming of two parallel and distributed adaptations of a dense matrix inversion method, we have underlined that a graph description paradigm contributes to improve productivity and minimize the Time-To-Solution. The development time is reduced by the segregation between the computational code and the description of parallelism that may be easily modified. The component-oriented programming contributes also to the productivity by giving re-usability and modularity. Furthermore, the support of heterogeneous hardware like GPU can be made easily with multiple implementation for one abstract component with an automatic detection of underlying hardware at run-time. The high level programming approach avoids end-users to explicitly manage communications which may be managed and optimized during the execution. Experiments have pointed out that the graph description gives many information that could help to anticipate data migration with asynchronous communications and also a better coordination of tasks by scheduling them when they are ready to run. For exascale computing, partial or global synchronization are not scalable and will slow down application performances. Asynchronous communications are highly important to make program scalable. They also increase performance by offering to overlap communications and computations. With a graph description paradigm, end-users are not in charge to express communications which are transparently declares through the description of task dependencies. These dependencies can be also used at run-time to realize asynchronous communications by analyzing the graph and taking in account the state of tasks.

Fault-tolerance is critical for exascale computing and can be supported by the last proposed paradigm. The graph description paradigm offers to restart and locate a failed task in the global application execution flow. Considering, the global application state and data available extracted from the graph, the necessary tasks

can be relaunched in order to restore the entering state of the failed task. The task in which a fault has occurred can be also simply restarted and independent tasks can continue their processing. In consequence with graph, faults can be managed at run-time, can avoid checkpoints and the application restart at the last context saving.

As we have pointed out, the potential issue with the graph description paradigm is the application scalability because graph are $NP - hard$ problem to schedule. Moreover, an overhead is also to consider for the scheduling and the run-time management system based on graph. Nevertheless, this paradigm in our proposal is at a high level and thus for coarse-grained tasks. It induces that this paradigm should scale for million of tasks regarding to this paper [85]. However, some research stay to do to get efficient scheduling and tolerable overhead.

In our experiments, we have shown that data migration anticipation may be realized automatically and transprently for end-users by analyzing the graph of dependencies during the execution. However, other operations may be anticipated and extracted from the graph information, such as I/O. The fifth chapter has presented and explored the use of graph dependencies to realize anticipated and delegated I/O. Asynchrony of anticipated storage accesses gives the possibility to get a fast data access and overlap computations and I/O. Delegation participate to reduce disk contentions coming from the massive competition accesses of cores which should be highly present in exascale systems. The exploitation of task dependencies offers a transparent management of I/O to end-users by supporting I/O directly into the programming paradigm. The segregation between I/O and computations gives also a layer to optimize storage accesses with collective I/O at run-time. The graph analysis and the information of task mapping also offers to move data from the storage to the closest place where data are going to be processed. The knowledge of future I/O accesses may also contribute to exploit the memory hierarchy of future storage systems. The high level support and I/O optimizations at run-time should lead to an improvement of productivity and increase performance of applications to sustain many petaflops on exascale machines.

6.2 Consequences and Future Works

Two programming ways are possible for exascale, an uniform programming with one programming paradigm or a hybrid with the aggregation of multiple paradigm. Our proposal is a hybrid form which is based on the exploitation of three different programming paradigms. However, it involves some trade-offs and consequences. The use of multiple paradigm makes more complex the programming for end-users by exposing the

different levels. They must be aware and care about the different levels of parallelism and use the convenient programming paradigm. This point may appropriate to decrease the programming productivity.

As an extension of this work, it will be interesting to study how these paradigms may be integrated together to create a general programming model. This should be investigated during the ANR-JST FP3C project which is a collaboration between France and Japan to work on framework and programming models for post-petascale machines. One of the purpose is to adapt YML to have the possibility to use XMP as a language for implementation components. Moreover, each XMP component must support and interact with a run-time for many-cores processors. The interaction between those three will be studied and integrated in order to propose general programming model. For this project, YML should be extended with an asynchronous global address space support in order to allow interaction between XMP and YML and also in order to realize automatic data migration anticipation. Scheduling strategies for graph must be also improved in YML to support a better scaling and to deliver higher performances.

Another consequence of a multi-level paradigm programming is the redesign of algorithms. They should take into account the hierarchy decomposition to get the maximum performance efficiency. This induces a lot of effort to re-think and redesign numerical libraries and also to rewrite applications. A hybrid programming model is highly possible considering the complexity of the hierarchy and the heterogeneity of exacale machines. Run-time should integrate an automatic data distribution and mapping engine in order to take into account this hierarchy and the various performance and memory size available from the computation elements.

In our opinion, the future programming model for exascale should be not rely on a compiler. They commonly need a long time to reach maturity in order to provide reliability and deliver performances. API or language extension are the best candidates because they provide incremental modifications from actual applications and not a re-program from scratch. We also think that run-time should take advantage of the end-user expertise at the application level in order to provide auto-tuning. This should help to adapt program on targeted machines and exploit the maximum performance.

Despite our approach is not complete and need to be pursued in order to finalize some ideas and demonstrate its potential through the integration of the different levels of paradigms. We believe that we have given some research direction and arguments in the orientation of the future programming paradigm for exascale computing. We estimate also that the education of end-users is highly strategic and they should start to focus on hybrid programming and application. We are also convinced that they should work jointly with computer scientists in order to prepare the turn of exascale which should be the key for many science breakthrough and for the design of revolutionary engineering solutions.

Bibliography

[1] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. Toward Exascale Resilience. *Int. J. High Perform. Comput. Appl.*, 23:374–388, November 2009. 2

[2] John Shalf, Sudip S. Dosanjh, and John Morrison. Exascale Computing Technology Challenges. In José M. Laginha M. Palma, Michel J. Daydé, Osni Marques, and João Correia Lopes, editors, *VECPAR*, volume 6449 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2010. 2

[3] Barbara Chapman, Jesús Labarta, Vivek Sarkar, and Mitsuhisa Sato. Programmability Issues. *Int. J. High Perform. Comput. Appl.*, 23:328–331, November 2009. 2

[4] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Torsten Hoefler, Sameer Kumar, Ewing L. Lusk, Rajeev Thakur, and Jesper Larsson Träff. MPI on Millions of Cores. *Parallel Processing Letters*, 21(1):45–60, 2011. 3

[5] William Gropp. MPI at Exascale: Challenges for Data Structures and Algorithms. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 3–3, Berlin, Heidelberg, 2009. Springer-Verlag. 3

[6] Franck Cappello and Daniel Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society. 3

[7] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*, pages 212–223, 1998. 5

[8] Alok N. Choudhary, Charles Koelbel, and Mary Zosel. High performance Fortran: Implementor and Users Workshop. In *SC*, pages 610–613, 1993. 6

[9] J Rose and G Stelle. C*: An Extended C Language for Data Parallel Programming. In *SuperComputing*, pages 2–16, 1987. 6

[10] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. Implementation of a Portable Nested Data-Parallel Language. *J. Parallel Distrib. Comput.*, 21(1):4–14, 1994. 6

[11] Michael I. Gordon, William Thies, and Saman P. Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In John Paul Shen and Margaret Martonosi, editors, *ASPLOS*, pages 151–162. ACM, 2006. 6

[12] Ian Buck, Tim Foley, Daniel Reiter Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004. 6

[13] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor: A Distributed Job Scheduler. pages 307–350, 2002. 6, 39

[14] Dietmar W. Erwin and David F. Snelling. UNICORE: A Grid Computing Environment. In Rizos Sakellariou, John Keane, John R. Gurd, and Len Freeman, editors, *Euro-Par*, volume 2150 of *Lecture Notes in Computer Science*, pages 825–834. Springer, 2001. 6, 39

[15] Junwei Cao, Stephen A. Jarvis, Subhash Saini, and Graham R. Nudd. GridFlow: Workflow Management for Grid Computing. In *CCGRID*, pages 198–205. IEEE Computer Society, 2003. 6, 39

[16] Olivier Delannoy, Nahid Emad, and Serge G. Petiton. Workflow Global Computing with YML. In *GRID*, pages 25–32. IEEE, 2006. 6, 40, 80

[17] Jakub Kurzak and Jack Dongarra. Implementing Linear Algebra Routines on Multi-core Processors with Pipelining and a Look Ahead. In Bo Kågström, Erik Elmroth, Jack Dongarra, and Jerzy Wasniewski, editors, *PARA*, volume 4699 of *Lecture Notes in Computer Science*, pages 147–156. Springer, 2006. 7

[18] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems. *Parallel Comput.*, 36:232–240, June 2010. 7

[19] Andrew L. Wendelborn and H. Garsden. Exploring the Stream Data Type in SISAL and Other Languages. In Michel Cosnard, Kemal Ebcioglu, and Jean-Luc Gaudiot, editors, *Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, volume A-23 of *IFIP Transactions*, pages 283–294. North-Holland, 1993. 7

[20] Dan Bronachea and Jaein Jeong. GASNet specification. Technical report, University of California, 2002. 7

[21] Jarek Nieplocha and Bryan Carpenter. ARMCI: A Portable Remote Memory Copy Libray for Ditributed Array Libraries and Compiler Run-Time Systems. In José D. P. Rolim, Frank Mueller, Albert Y. Zomaya, Fikret Erçal, Stephan Olariu, Binoy Ravindran, Jan Gustafsson, Hiroaki Takada, Ronald A. Olsson, Laxmikant V. Kalé, Peter H. Beckman, Matthew Haines, Hossam A. ElGindy, Denis Caromel, Serge Chaumette, Geoffrey Fox, Yi Pan, Keqin Li, Tao Yang, G. Ghiola, Gianni Conte, Luigi V. Mancini, Dominique Méry, Beverly A. Sanders, Devesh Bhatt, and Viktor K. Prasanna, editors, *IPPS/SPDP Workshops*, volume 1586 of *Lecture Notes in Computer Science*, pages 533–546. Springer, 1999. 7

[22] Parry Husbands, Costin Iancu, and Katherine A. Yelick. A Performance Analysis of the Berkeley UPC Compiler. In Utpal Banerjee, Kyle Gallivan, and Antonio González, editors, *ICS*, pages 63–73. ACM, 2003. 7

[23] John Reid and Robert W. Numrich. Co-arrays in the next Fortran Standard. *Scientific Programming*, 15(1):9–26, 2007. 7, 38

[24] Arvind Krishnamurthy, Alexander Aiken, Phillip Colella, David Gay, Susan L. Graham, Paul N. Hilfinger, Ben Liblit, Carleton Miyamoto, Geoff Pike, Luigi Semenzato, and Katherine A. Yelick. Titanium: A high performance java dialect. In *PPSC*, 1999. 7, 38

[25] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In Ralph E. Johnson and Richard P. Gabriel, editors, *OOPSLA*, pages 519–538. ACM, 2005. 7

[26] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. The Cascade High Productivity Language. In *IPDPS*, pages 52–60. IEEE Computer Society, 2004. 7

[27] Jinpil Lee and Mitsuhisa Sato. Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW '10, pages 413–420, Washington, DC, USA, 2010. IEEE Computer Society. 7

[28] William C. Athas and Charles L. Seitz. Multicomputers: Message-Passing Concurrent Computers. *IEEE Computer*, 21(8):9–24, 1988. 8

[29] Andrew P. Black, Norman C. Hutchinson, Eric Jul, Henry M. Levy, and Larry Carter. Distribution and Abstract Types in Emerald. *IEEE Trans. Software Eng.*, 13(1):65–76, 1987. 8

[30] Rohit Chandra, Anoop Gupta, and John L. Hennessy. COOL: An Object-Based Language for Parallel Programming. *IEEE Computer*, 27(8):13–26, 1994. 8

[31] Andrew S. Tanenbaum, M. Frans Kaashoek, and Henri E. Bal. Parallel Programming Using Shared Objects and Broadcasting. *IEEE Computer*, 25(8):10–19, 1992. 8

[32] Laxmikant V. Kalé and Sanjeev Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based On C++. In *OOPSLA*, pages 91–108, 1993. 8

[33] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004. 8

[34] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary R. Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *HPCA*, pages 13–24. IEEE Computer Society, 2007. 8

[35] Alexandros V. Gerbessiotis and Leslie G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. *J. Parallel Distrib. Comput.*, 22(2):251–267, 1994. 8

[36] Thomas Cheatham, Amr F. Fahmy, Dan C. Stefanescu, and Leslie G. Valiant. Bulk Synchronous Parallel Computing - A Paradigm for Transportable Software. In *HICSS (2)*, pages 268–275, 1995. 8

[37] Nir Shavit and Dan Touitou. Software Transactional Memory. *Distributed Computing*, 10(2):99–116, 1997. 8

[38] James C. Brodman, Basilio B. Fraguela, María J. Garzarán, and David Padua. New Abstractions for Data Parallel Programming. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, pages 16–16, Berkeley, CA, USA, 2009. USENIX Association. 12

[39] Anwar Ghuloum. Ct: Channelling NeSL and SISAL in C++. In *Proceedings of the 4th ACM SIGPLAN workshop on Commercial users of functional programming*, CUFP '07, pages 5:1–5:3, New York, NY, USA, 2007. ACM. 12

[40] *NVIDIA CUDA Programming Guide 2.3*, 2009. 14

[41] Yousef Saad. Krylov Subspace Methods on Supercomputers. *SIAM J. Sci. Stat. Comput.*, 10(6):1200–1232, 1989. 14, 20

[42] Richard Vuduc, James W Demmel, and Katherine A Yelick. OSKI: A Library of Automatically Tuned Sparse Matrix Kernels. In *Institute of Physics Publishing*, 2005. 14

[43] Samuel Williams, Leonid Oliker, Richard W. Vuduc, John Shalf, Katherine A. Yelick, and James Demmel. Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. In Becky Verastegui, editor, *SC*, page 38. ACM Press, 2007. 14, 23

[44] Nathan Bell and Michael Garland. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM. 15, 20, 21, 23

[45] Muthu Manikandan Baskaran and Rajesh Bordawekar. Optimizing Sparse Matrix-Vector Multiplication on GPUs. IBM Technical Report IBM-2009-004, IBM, april 2009. 15, 35

[46] Luc Buatois, Guillaume Caumon, and Bruno Levy. Concurrent Number Cruncher: a GPU Implementation of a General Sparse Linear Solver. *Int. J. Parallel Emerg. Distrib. Syst.*, 24(3):205–223, 2009. 15, 21

[47] Youcef Saad. SPARSKIT: a Sparse Matrix Basic Tool Kit for Computations, 1994. 15, 20

[48] Serge G. Petiton and Christine Weill-Duflos. Massively Parallel Preconditioners for the Sparse Conjugate Gradient Method. In Luc Bougé, Michel Cosnard, Yves Robert, and Denis Trystram, editors, *CONPAR*, volume 634 of *Lecture Notes in Computer Science*, pages 373–378. Springer, 1992. 19, 23

[49] Timothy A. Davis. University of Florida Sparse Matrix Collection. 23

[50] Maxime R. Hugues and Serge G. Petiton. Sparse Matrix Formats Evaluation and Optimization on a GPU. In *HPCC*, pages 122–129. IEEE, 2010. 31

[51] Maxime Hugues and Serge Petiton. Optimized Sparse Matrix Formats on GT200 and Fermi GPUs. Session 2, May 2011. 31

[52] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20:359–392, December 1998. 31

[53] Maher Kaddoura, Chao-Wei Ou, and Sanjay Ranka. Partitioning Unstructured Computational Graphs for Nonuniform and Adaptive Environments. *IEEE Parallel Distrib. Technol.*, 3:63–69, September 1995. 31

[54] Umit Catalyurek and Cevdet Aykanat. Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Trans. Parallel Distrib. Syst.*, 10:673–693, July 1999. 31

[55] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language Virtualization for Heterogeneous Parallel Computing. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 835–847, New York, NY, USA, 2010. ACM. 38

[56] Christian Bell, Dan Bonachea, Rajesh Nishtala, and Katherine A. Yelick. Optimizing Bandwidth Limited Problems using One-Sided Communication and Overlap. In *IPDPS*. IEEE, 2006. 38

[57] Olivier Delannoy and Serge Petiton. A Peer to Peer Computing Framework: Design and Performance Evaluation of YML. *Parallel and Distributed Computing, International Symposium on*, 0:362–369, 2004. 38, 39

[58] James Frey, Todd Tannenbaum, Miron Livny, Ian T. Foster, and Steven Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *HPDC*, pages 55–. IEEE Computer Society, 2001. 39

[59] Kaizar Amin, Gregor von Laszewski, Mihael Hategan, Nestor J. Zaluzec, Shawn Hampton, and Albert Rossi. GridAnt: A Client-Controllable Grid Work.ow System. In *HICSS*, 2004. 39

[60] Grzegorz Malewicz, Matthew H. Austern, Aart J.C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a System for Large-Scale Graph Processing. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 6–6, New York, NY, USA, 2009. ACM. 39

[61] Sébastien Noël, Olivier Delannoy, Nahid Emad, Pierre Manneback, and Serge G. Petiton. A Multi-level Scheduler for the Grid Computing YML Framework. In Wolfgang Lehner, Norbert Meyer, Achim Streit, and Craig Stewart, editors, *Euro-Par Workshops*, volume 4375 of *Lecture Notes in Computer Science*, pages 87–100. Springer, 2006. 40

[62] Laurent Choy, Olivier Delannoy, Nahid Emad, and Serge G. Petiton. Federation and Abstraction of Heterogeneous Global Computing Platforms with the YML Framework. *The 3rd International Workshop on P2P, Parallel, Grid and Internet Computing*, 2009. 40

[63] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific Workflow Management and the Kepler System. In *Concurr. Comput. : Pract. Exper*, page 2006, 2005. 40

[64] W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245 – 275, 2005. 40

[65] T. Fahringer, J. Qin, and S. Hainzer. Specification of Grid Workflow Applications with AGWL: an Abstract Grid Workflow Language. *Cluster Computing and the Grid, IEEE International Symposium on*, 2:676–685, 2005. 41

[66] Zhijie Guan, Francisco Hernandez, Purushotham Bangalore, Jeff Gray, Anthony Skjellum, Vijay Velusamy, and Yin Liu. Grid-Flow: a Grid-Enabled Scientific Workflow System with a Petri-Net-Based Interface: Research Articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1115–1140, 2006. 41

[67] Mitsuhisa Sato, Hidemoto Nakada, Satoshi Sekiguchi, Satoshi Matsuoka, Umpei Nagashima, and Hiromitsu Takagi. Ninf: A Network Based Information Library for Global World-Wide Computing Infrastructure. In Louis O. Hertzberger and Peter M. A. Sloot, editors, *HPCN Europe*, volume 1225 of *Lecture Notes in Computer Science*, pages 491–502. Springer, 1997. 44

[68] Franck Cappello, Eddy Caron, Michel J. Daydé, Frédéric Desprez, Yvon Jégou, Pascale Vicat-Blanc Primet, Emmanuel Jeannot, Stéphane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Benjamin Quétier, and Olivier Richard. Grid'5000: a large scale and highly reconfigurable grid experimental testbed. In *GRID*, pages 99–106. IEEE, 2005. 45

[69] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron, and Olivier Richard. A Batch Scheduler with High Level Components. In *CCGRID*, pages 776–783. IEEE Computer Society, 2005. 45

[70] Areski Flissi and Philippe Merle. A Generic Deployment Framework for Grid Computing and Distributed Applications. In Robert Meersman and Zahir Tari, editors, *OTM Conferences (2)*, volume 4276 of *Lecture Notes in Computer Science*, pages 1402–1411. Springer, 2006. 45

[71] Ling Shang, Zhijian Wang, and Serge G. Petiton. Solution of Large Scale Matrix Inversion on Cluster and Grid. In *GCC '08: Proceedings of the 2008 Seventh International Conference on Grid and Cooperative Computing*, pages 33–40, Washington, DC, USA, 2008. IEEE Computer Society. 47

[72] S. Petiton. Parallelization on an MIMD Computer with Real-Time Scheduler. *Aspects of Computation on Asynchronous Parallel Processors, North Holland*, 1989. 55

[73] Maxime Hugues and Serge G. Petiton. A Matrix Inversion Method with YML/OmniRPC on a Large Scale Platform. In José M. Laginha M. Palma, Patrick Amestoy, Michel J. Daydé, Marta Mattoso, and João Correia Lopes, editors, *VECPAR*, volume 5336 of *Lecture Notes in Computer Science*, pages 95–108. Springer, 2008. 55, 59

[74] Lamine M. Aouad, Serge G. Petiton, and Mitsuhisa Sato. Grid and Cluster Matrix Computation with Persistent Storage and Out-of-core Programming. In *CLUSTER*, pages 1–9. IEEE, 2005. 58

[75] Caroline Baldassari, Hélène Barucq, and Julien Diaz. High-Order Schemes with Local Time Stepping for Solving the Wave Equation in a Reverse Time Migration Algorithm. In *2010 ISFMA Symposium & Shanghai Summer School on Maxwell' equations: Theoretical and Numerical Issues with Applications*, Shanghaï, China, 2010. 67

[76] Rajeev Thakur, William Gropp, and Ewing Lusk. *Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation*. Mathematics and Computer Science Division, Argonne National Laboratory, October 1997. ANL/MCS-TM-234. 70

[77] Arifa Nisar, Wei-keng Liao, and Alok Choudhary. Accelerating I/O Forwarding in IBM Blue Gene/P Systems. In *SC '10: Proceedings of the 2010 ACM/IEEE conference on Supercomputing*. IEEE Press, 2010. 70

[78] Nawab Ali, Philip Carns, Kamil Iskra, Dries Kempe, Samuel Lang, Robert Latham, Robert Ross, Lee Ward, and P. Sadayappan. Scalable I/O Forwarding Framework for High-Performance Computing Systems. In *IEEE International Conference on Cluster Computing*, 2009. 70

[79] Arifa Nisar, Wei-keng Liao, and Alok Choudhary. Scaling Parallel I/O Performance Through I/O Delegate and Caching System. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press. 70

[80] Javier Blas, Florin Isaila, J. Carretero, Robert Latham, and Robert Ross. Multiple-Level MPI File Write-Back and Prefetching for Blue Gene Systems. In Matti Ropo, Jan Westerholm, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5759 of *Lecture Notes in Computer Science*, pages 164–173. Springer Berlin / Heidelberg, 2009. 70

[81] Maxime R. Hugues, Michael Moretti, Serge G. Petiton, and Henri Calandra. ASIODS - An Asynchronous and Smart I/O Delegation System. *Procedia CS*, 4:471–478, 2011. 79

[82] Systems Michael Beynon, Michael Beynon, Renato Ferreira, Tahsin Kurc, Alan Sussman, Joel Saltz, and Johns Hopkins Medical. DataCutter: Middleware for Filtering Very Large Scientific Datasets on Archival Storage. In *In IEEE Symposium on Mass Storage Systems*, pages 119–133. IEEE Computer Society Press, 2000. 80

[83] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science and Engineering*, 12:66–73, 2010. 81

[84] Barbara M. Chapman and Lei Huang. Enhancing OpenMP and Its Implementation for Programming Multicore Systems. In Christian H. Bischof, H. Martin Bücker, Paul Gibbon, Gerhard R. Joubert, Thomas Lippert, Bernd Mohr, and Frans J. Peters, editors, *PARCO*, volume 15 of *Advances in Parallel Computing*, pages 3–18. IOS Press, 2007. 81

[85] Michel Cosnard, Emmanuel Jeannot, and Laurence Rougeot. Low Memory Cost Dynamic Scheduling of Large Coarse Grain Task Graphs. In *IPPS/SPDP*, pages 524–530, 1998. 83

Résumé:

L'arrivée des supercalculateurs post-petascales et exascales offre la perspective d'accélérer la résolution des problèmes d'ingénierie aux modélisations hautement complexes. Cependant, ces futurs systèmes posent des problèmes aux informaticiens pour construire de telles machines. De nombreux problèmes doivent être résolus comme la tolérance aux pannes, la consommation énergétique et la programmation de ces systèmes complexes composés de milliard de coeurs.

Dans cette thèse, nous nous sommes concentrés sur l'aspect programmation et proposons un paradigme de programmation multi-niveaux composé de trois niveaux. Pour le bas niveau, un paradigme data parallèle est proposé pour programmer les processeurs à nombreux coeurs pour sa focalisation sur la distribution et le mouvement des données. Nous avons implémenté et évalué le produit matrice vecteur creux suivant différents formats de matrice creuse sur un GPU pour illustrer ce point. Pour le niveau intermédiaire, nous proposons un paradigme à passage de messages de manière à optimiser les communications inter-processeurs et inter-noeuds. Pour le haut niveau, un paradigme de description de graphe est proposé pour programmer et gérer le parallélisme entre les noeuds.

Avec une méthode d'inversion matricielle dense développée en YML, nous soulignons l'intérêt des graphes pour la minimisation du temps à la solution et pour le support des communications asynchrones de facon transparente. L'intérêt des graphes est également démontré pour les optimisations d'entrées/sorties et leur support dans un modèle de programmation. Nous concluons finalement en analysant une telle proposition de paradigme de programmation pour les machines exascales et présentons la direction des travaux futurs.

Mots Clés: paradigme, exascale, YML, GPU, data parallèle, description de graphe, délégation d'entrée/sortie

Abstract:

The coming of post-petascale and exascale supercomputers offers the perspective to accelerate the solving of engineering problems which have highly complex modeling. However, these future systems challenge computer scientists to built such machines. Many issues must be faced such as fault-tolerance, energy consumption and the programming of these complex systems composed of billions cores.

In this thesis, we have focused on the programming aspect and propose a multi-level programming paradigm composed of three levels. For the low level, a data parallel paradigm is proposed to program many-cores processors for its focus on data mapping and movements. We have implemented and evaluated the SpMV with various sparse matrix formats on GPU to illustrate this point. For the intermediate level, we propose a message passing paradigm in order to optimize inter-sockets and inter-nodes communications. For the high level, a graph description paradigm is proposed to program and manage the parallelism between nodes.

With a dense matrix inversion method developed in YML, we underline the interest of graph for the Time-To-Solution minimization and for the support of asynchronous communications in a transparent way. The interest of graph is also demonstrated for I/O optimizations and for their direct support into the programming model. We conclude finally by analyzing a such proposition of programming paradigm for exascale machines and outline the future work direction.

Keywords: programing paradigm, exascale, YML, GPU, data parallel, graph description, I/O delegation