Distributed and parallel programming paradigms using graphs of tasks for post-petascale supercomputers

Jérôme Gurhem

Université de Lille

UNIVERSITÉ DE LILLE

École doctorale Sciences Pour l'Ingénieur Université Lille Nord-de-France
Centre de Recherche en Informatique, Signal et Automatique de Lille

THÈSE

préparée et soutenue publiquement

PAR

Jérôme Gurhem

à Saclay

le 16 mars 2021

pour obtenir le grade de Docteur en **Informatique**

Paradigmes de Programmation Répartie et Parallèle Utilisant des Graphes de Tâches pour Supercalculateurs Post-Pétascale

---

Thèse dirigée par Serge G. Petiton, Université de Lille

Membres du jury:

Président du jury	William Jalby	Professeur à l'Université Versailles Saint-Quentin-en-Yvelines
Rapporteurs	Michel Daydé	Professeur à l'ENSEEIHT, France
	Ewa Deelman	Professeure à l'Université de Southern California, États-Unis
Examinateurs	Henri Calandra	Docteur, Total SA, France
	Barbara Chapman	Professeure à l'Université de Stony Brook, États-Unis
	Clarisse Dhaenens	Professeure à l'Université de Lille, France
	Miwako Tsuji	Docteure, RIKEN, Japon
Directeur	Serge G. Petiton	Professeur à l'Université de Lille, France

Université de Lille

UNIVERSITY OF LILLE

École doctorale Sciences Pour l'Ingénieur Université Lille Nord-de-France
Centre de Recherche en Informatique, Signal et Automatique de Lille

Ph.D. Thesis

prepared and defended

BY

Jérôme Gurhem

in Saclay

the March 16h, 2021

to obtain the grade of Ph.D. in **Computer Science**

Distributed and Parallel Programming Paradigms Using Graphs of Tasks for Post-Petascale Supercomputers

---

Thesis advised by Serge G. Petiton, University of Lille

Committee members:

Committee president	William Jalby	Professor at University of Versailles Saint-Quentin-en-Yvelines
Reporters	Michel Daydé	Professor at ENSEEIHT, France
	Ewa Deelman	Professor at University of Southern California, USA
Examinators	Henri Calandra	Ph.D., Total SA, France
	Barbara Chapman	Professor at University of Stony Brook, USA
	Clarisse Dhaenens	Professor at University of Lille, France
	Miwako Tsuji	Ph.D., RIKEN, Japan
Advisor	Serge G. Petiton	Professor at University of Lille, France

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisors Prof. Serge G. Petiton and Dr. Henri Calandra for the continuous support of my Ph.D study and related research, for their patience, motivation, and immense knowledge. Their guidance helped me during my research and the writing of this dissertation. I could not have imagined having a better mentors for my Ph.D study.

I would like to thank Total SA for the funding that allowed me to conduct this thesis.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. William Jalby for presiding the committee, Prof. Michel Daydé, and Prof. Ewa Deelman for reviewing my Ph.D. dissertation, Prof. Barbara Chapman, Prof. Clarisse Dhaenens and Dr. Miwako Tsuji for participating in the committee,

I would like to thank all the members of the *Maison de la Simulation* for welcoming me in the laboratory and for the stimulating discussions, especially Dr. Martial Mancip for his help with Docker and improving my Ph.D. defense.

My appreciation also goes out to my family and friends for their encouragement and support all through my studies. I would also like to thank my girlfriend for her support in my work and every day life.

Résumé

Depuis le milieu des années 1990, les bibliothèques de transmission de messages sont les technologies les plus utilisées pour développer des applications parallèles et distribuées. Des modèles de programmation basés sur des tâches peuvent être utilisés, par exemple, pour éviter les communications collectives sur toutes les ressources comme les réductions, les diffusions ou les rassemblements en les transformant en multiples opérations avec des tâches. Ensuite, ces opérations peuvent être planifiées par l'ordonnanceur pour placer les données et les calculs de manière à optimiser et réduire les communications de données.

L'objectif principal de cette thèse est d'étudier ce que doit être la programmation basée sur des tâches pour des applications scientifiques et de proposer une spécification de cette programmation distribuée et parallèle, en expérimentant avec plusieurs représentations simplifiées d'applications scientifiques importantes pour TOTAL, et de méthodes linéaire classique dense et creuses. Au cours de la thèse, plusieurs langages de programmation et paradigmes sont étudiés. Des méthodes linéaires denses pour résoudre des systèmes linéaires, des séquences de produit matrice vecteur creux et la migration sismique en profondeur pré-empilement de Kirchhoff sont étudiées et implémentées en tant qu'applications basées sur des tâches. Une taxonomie, basée sur plusieurs de ces langages et paradigmes est proposée. Des logiciels ont été développés en utilisant ces modèles de programmation pour chaque application simplifiée. À la suite de ces recherches, une méthodologie pour la programmation de tâches parallèles est proposée, optimisant les mouvements de données en général et, en particulier, pour des applications scientifiques ciblées.

Abstract

Since the middle of the 1990s, message passing libraries are the most used technology to implement parallel and distributed applications. However, they may not be a solution efficient enough on exascale machines since scalability issues will appear due to the increase in computing resources. Task-based programming models can be used, for example, to avoid collective communications along all the resources like reductions, broadcast or gather by transforming them into multiple operations on tasks. Then, these operations can be scheduled by the scheduler to place the data and computations in a way that optimize and reduce the data communications.

The main objective of this thesis is to study what must be task-based programming for scientific applications and to propose a specification of such distributed and parallel programming, by experimenting for several simplified representations of important scientific applications for TOTAL, and classical dense and sparse linear methods. During the dissertation, several programming languages and paradigms are studied. Dense linear methods to solve linear systems, sequences of sparse matrix vector product and the Kirchhoff seismic pre-stack depth migration are studied and implemented as task-based applications. A taxonomy, based on several of these languages and paradigms is proposed. Software were developed using these programming models for each simplified application. As a result of these researches, a methodology for parallel task programming is proposed, optimizing data movements, in general, and for targeted scientific applications, in particular.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivations

Previously, the computing power of processors was mainly increasing with the frequency of the processors which cannot be increased easily anymore due the heat generation and the energy consumption increasing alongside. Therefore, a major change in the architecture occurred in order to still increase the performances of the new supercomputers : more processors with tens of cores were introduced in supercomputers. Moreover, graphic processing unit (GPU) also made their appearance in supercomputer nodes due to their high performances considering their low energy consumption compared to regular processes. This leads to an increasing number of different architectures from which the users have to obtain performances.

Furthermore, in the recent massively parallel architectures, the users have to take advantage of multiple nodes constituted of several multi-core processors connected by network to achieve performance, which is done by implementing parallel applications. Moreover, taking advantage of multiple nodes containing several multi-core processors involves the use of the network to send data from one node to another since the memory is local to the nodes. Besides, new processors with a network on chip are also appearing on supercomputers like Fugaku A64FX CPUs and Sunway TaihuLight SW26010. In these processors, the cores are not sharing the same memory. Instead, there is a network that can be used to send data from one subset of core to the others. These processors have hierarchical memory and network in which distributed memory is introduced at several level. There is distributed memory at the level of supercomputer nodes and at the level of the groups of cores in the processor.

Nowadays, parallel and distributed applications are based on MPI+X where MPI is the Message Passing Interface [1], that is used to send the data between the nodes, and X is a parallel programming model that is used to take advantage of the computing resources of the nodes. Usually, X may be OpenMP [2], that takes advantage of shared memory and multi-threading to achieve performance on CPU and GPU and/or CUDA [3], that addresses parallelism on NVidia GPUs. Applications can also implemented with pure MPI where MPI is used to spawn multiple processes on each node. With the increase of available architectures (a combination of different generation of CPU and GPU), the portability of highly optimized applications for a given architecture is not assured on another architecture. If the architecture is too different, it may also be possible that the application cannot be installed due to missing libraries dependent on a given architecture.

However, MPI may not be a solution efficient enough on exascale machines, especially in terms of fault tol-

erance and check-pointing [4]. Task-based approach can help in managing fault tolerance and check-pointing since the tasks could be restarted on another location and data from tasks saved at any moment. Task based programming models can implement fault tolerance by restarting the failed tasks on other computing resources and even stop executing tasks on processors with high fault rate. Check-pointing could be implemented by storing the state of the output of the task in a way that allow to restart the application at the last check-point and reconstruct the lost data. Moreover, task based programming models can also be used to avoid collective communications like reductions, broadcast or gather by transforming them into multiple operations on tasks. Then, these operations can be scheduled by the programming scheduler to place the data and computations in a way that optimize and reduce the data communications. Therefore, with the merging of parallel and distributed programming, graph of tasks and efficient scheduler are a very interesting way to improve performances while reducing communications. Schedulers that can optimize data migrations and task execution while keeping good performances are necessary.

While MPI focus on exchanging data between processes, other programming models may be efficient to parallelize applications with good scalability without being held back by global synchronizations. For instance, Partitioned Global Address Space Languages (PGAS) programming models, that let the users see the distributed memory as a global memory address space that is partitioned across each processing element [5], are an alternative to MPI + X. Task-based programming models, which allow to define fine-grain tasks (computations) on a specific set of data (input and output), are also an alternative to MPI. Runtime systems which can optimize execution are another one. Moreover, task-based programming models (first level of parallelism) combined with coarse-grain tasks implemented in a PGAS language (second level of parallelism) can also be one of them. Usually, fine-grain tasks use one process (eventually multi-threaded) whereas coarse-grain tasks perform on several processes (eventually with distributed memory).

1.2 Objectives and Contributions

As programming models are evolving rapidly, it is important to have a clear view of the main capabilities of the current programming models in order to use the most suitable programming model for the targeted architectures and the implemented application. Some have a high level of abstraction, others, more pragmatic, are based on adding pragmas to existing software. The existence of several levels of parallelism also generates programming paradigms mixing several approaches. Moreover, the development of realistic scientific applications in this context is very complex and cannot be considered for each experiment related to this research. The main objective of this thesis is to study what must be task-based programming for scientific applications on post-petascale supercomputers and to propose a specification of such distributed and parallel programming, by experimenting for several simplified representations of important scientific applications for TOTAL. The optimization of data movements is studied and scheduling strategies are proposed and evaluated. During the dissertation, several programming languages and paradigms are studied. Software is developed using these programming models for each simplified application. As a result of this research, a methodology for parallel task programming is proposed, optimizing data movements, in general, and for targeted scientific applications, in particular. A taxonomy of these languages and a strategy of evolution between the current codes and those respecting this methodology is introduced in the dissertation.

The first contribution of this dissertation is an in-depth analyze of several task based parallel and dis-

tributed programming models. We compare and evaluate PaRSEC [6], Legion [7], Regent [8], TensorFlow [9], HPX [10], YML+XMP [11], Pegasus [12] and Swift [13]. TensorFlow is not a general purpose programming model since it mainly focuses on linear algebra and machine learning. However, it proposes tasks definition interfaces similar to the other task based programming models. PaRSEC, Legion, Regent and HPX are fine grained task based programming models where the tasks run on only one computing unit (a thread or a process) whereas YML+XMP, Pegasus and Swift are large grain task based programming model where the tasks can run on distributed resources (from one process to multiple processes on different nodes of a supercomputer). Some of those task based programming models use graphs to express the dependencies between the tasks. Thus, the applications implemented with them are based on a graph of task that are executed during the execution of the application.

These task based programming models are used to implement three applications which are the second contribution of this dissertation. The first application is a block based dense linear solver based on three algorithms : the block LU factorization, the block Gaussian elimination and the block Gauss-Jordan elimination. This application has been implemented in XMP, MPI, YML+XMP, Regent, TensorFlow, Pegasus and HPX. A block based task sparse matrix vector product has been implemented in MPI, HPX and YML+XMP. It can perform the sparse matrix vector product with the COO, CSR and ELLPACK sparse matrix storage format as well as the dense format. Moreover, this application also supports multiple data distribution. Indeed, the matrix can distributed by block of rows, by block of columns and a rectangular block of rows and columns. The last considered application is a simplified version of the Kirchhoff seismic pre-stack depth migration which is implemented with MPI and HPX.

Finally, the last contribution of this dissertation is a taxonomy of the task based programming models where the features are extracted from the comparisons and analyzes of the task based programming models. This taxonomy summarizes the expertise gained on the task programming models encountered during the work on this dissertation. We analyze the introduced algorithms and task based programming models features in order to highlight the most suited features to use in implementing the applications. We conclude on which category of algorithms can be efficiently implemented into an application relying on parallel and distributed task based programming models. We also provide recommendations and a methodology for parallel and distributed programming based on graph of tasks.

1.3 Outline

This dissertation is organized as follows. In Chapter 2, we present the start-of-the-art of the task based high performance computing including programming models currently used in order to achieve performances and a presentation of the recent task based programming models. Finally, we discuss the challenges the HPC community is facing for the exascale supercomputers.

In Chapter 3, the algorithms of the methods implemented as a benchmark for the task based programming models are covered. The three considered methods are a block based dense linear solver, the block based task sparse matrix vector product and the Kirchhoff seismic pre-stack depth migration.

A selection of task based programming models are introduced with more detail in Chapter 4. We outline the main features of each programming model studied and how to use them to define task, the dependencies between them, how to register the data used in the tasks and how to execute the tasks to perform the

intended computations.

Afterwards, we detail the implementation of the block LU factorization, the block Gaussian elimination and the block Gauss-Jordan elimination. In Chapter 5, we also present the experiments performed on the petascale K computer and the Poincare cluster as well as the results obtained with our applications compared to our MPI scalar implementation as well as ScaLAPACK. We also perform experiments with the block LU factorization on the Poincare cluster in which we compare our implementations of the block LU. We implemented it in Regent, HPX, YML+XM, PaRSEC, MPI and XMP.

In Chapter 6, we introduce the data distribution we use to split the sparse matrices on the multiples nodes available during the run of the application. We present how we distribute the matrix by block of rows, by block of columns and a rectangular block of rows and columns. Then, we explain how the tasks are defined depending of the data distribution and the different sparse matrix storage we use. Finally, we present the numerical results we obtained on PangeaII, the petascale supercomputer from Total located in Pau on which we ran our experiments.

Furthermore, the algorithm of the task based Kirchhoff seismic pre-stack depth migration are introduced in Chapter 7. Its implementation is also detailed in this chapter as well as the numerical experiments performed.

The taxonomy with the features we extracted from the analyze of the programming models in the previous chapters is presented in Chapter 8. We highlight these features in the other programming models and find how these features are implemented in the considered programming models. We also provide recommendations to improve task based programming models and their usage. Finally, we propose a preliminary methodology for parallel and distributed programming based on graphs of tasks which optimizes data migrations and computations scheduling for post-petascale supercomputers.

To conclude, the key results obtained in this thesis are summarized in Chapter 9 and the concluding remarks are presented. Finally, we suggest some possible paths to future researches.

Chapter 2

Task Based High Performance Computing

High Performance Computing (HPC) is one of the most active research areas in computer science since it contributes to solving of large scale problems in science, engineering and business. HPC is more and more used with the arrival of Big Data applications and more recently to improve the increasing number of applications relying on Artificial Intelligence and Deep Learning. The improvement of HPC is due to the effort of several discipline including computer architecture design, parallel algorithms as well as programming models. This chapter gives a state-of-the-art of the task based programming models available to implement HPC applications and discuss the key challenges of the upcoming exascale.

2.1 Timeline of Task Based High Performance Computing

HPC or High Performance Computing is a computer science field which consists in aggregating computing power in order to obtain higher performances than a regular desktop computer. This power is currently achieved by connecting several computers to form a cluster for the small cases or a supercomputer for the very large cases. In the current supercomputers, the computing resources are connected to allow mixing parallel computing (by taking advantage of multi-cores processors and large amount of processors) and distributed computing (the processor memory caches are not directly connected and must be accessed by performing communications through the network). Initially, the increase in performance of an application was obtained from the improvement of the processors used to run the application and especially from the increase of the frequency of the processors. Indeed, the processors were becoming faster with each new generation and thus, the applications ran faster on them. Then, the frequency became impossible to increase without the processors producing too much heat in the early 2000s [14]. Therefore, either the processors had to be cooled at the cost of more power and infrastructure or setting the frequency of processors to a reasonable temperature so that the processors do not produce an excessive amount of heat and shut down so that they do not melt. Moreover, the data transfer speed between the memory and the processor lowest cache is slower than the time needed by the cores to use it. Thus, the frequency of processors has stabilized since then. At this point, high performance application developers could not rely on the improvement of individual processors to achieve performance. Instead of using one processing unit to run applications, several of them could be used at the same time to run multiple operations at the same time and create faster applications. This introduced the use of parallelism in order to achieve high performances on the HPC computing resources. Using multiple

computing resources at the same time was not new since it was already researched in the 1980s with the Connection Machine [15] for instance.

The change from the use of single core processors to the use of several multi-core processors also made a shift in the programming models used to run efficient applications on those architectures. Indeed, using several processors means that data from one processor has to be made available somehow to the other processors if they need this piece of data during their computation. Usually, this is done by sending message containing the data through the network connecting the processors between themselves. This greatly changed the way to implement applications for supercomputers since the developers also have to manage the data mapping on the different computing resources and the eventual data migrations to perform in order to obtain the intended results.

Parallel and distributed programming models such as Message Passing Interface (MPI) [1], Parallel Virtual Machine (PVM) [16], Linda [17] [18] and P4 [19], which are based on message-passing, made their apparition in the early 1990s to address communications between several processors of supercomputers. In this programming model, the application uses several processes to make multiple computations at the same time on different cores and uses MPI or PVM to send data from one process to another one. They provide point-to-point and collective communication operations to help the developers to reorganize, migrate or perform operations on their distributed data.

However, developing a parallel high performance application with such a programming model requires the knowledge of parallel and distributed programming as well as the specificities of the targeted hardware. It takes time and may not be portable to other architectures without investing more time to make the structural modifications that may be necessary, for instance, to run on GPUs. An alternative that aims to reduce those costs and efficiently use the available hardware is task based programming models. Moreover, MPI may not be a solution efficient enough on exascale machines, especially in terms of fault tolerance and check-pointing [4]. Fault tolerance allows applications to manage hardware or software errors on computing resources either by continuing to run the application without this resource if possible or by cleanly stopping the application. Check pointing consists in saving a snapshot of the application a regular interval, so that applications can restart from the last snapshot in case of failure or stop. They are commonly used in long running applications to avoid re-running the application from the start in case something happens. It can also be used to see the evolution of data during the run of the application. For instance, this could be used to check the evolution of a simulation or a neural network training.

Besides, global operations like reductions, gathers and broadcasts are very expensive due to the high number of resources partaking in the operation and the cost of sending information to distant resources on the network connecting the nodes. Task-based approach can help in managing fault tolerance and check-pointing since the tasks could be restarted on another location and data from tasks saved at any moment. Tasks allow to separate the expression of the parallelism from its parallel implementation by letting the developer express the tasks and their dependencies while the runtime of the programming models tries to run as much tasks as possible at the same time, respects the dependencies and tries to obtain the best performances possible. This means that application experts can express algorithms through graph of tasks without being required to understand the hardware in detail. Furthermore, the task-based approach can help to eliminate large scale collective communications by encapsulating them inside tasks. Then, these tasks can run on a subset of the resources allocated to the application and execute collective communications

on a smaller scale. The graph of tasks can be efficiently scheduled so that the execution of tasks optimizes data migrations, IOs tasks and data check-pointing. In task based programming models, data can only be exchanged between tasks as their input and output parameters as opposed to exchanging data during the execution of the task. Therefore the algorithms using collective communications have to be redesigned in order to avoid them or rewrite them as task operations.

Several programming models which support the usage of tasks have appeared over the years. They implemented the task definition and management in different ways and have their unique features. Some of them will be introduced in the following section.

2.2 Current Task Based Programming Models

A task can be defined as an atomic set of operations, with a defined set of data as input and output, which can be asynchronously executed while enforcing data and/or control dependencies. This section introduces programming models supporting the definition and the scheduled execution of tasks.

2.2.1 Task Based Programming Models on Shared Memory

First, task based programming models that are designed to work with shared memory architectures are introduced. Usually, the programming models are based on multi-threading where threads can be considered as workers which can execute the tasks. The scope of the task is limited to the thread although QUARK [20] supports multi-threaded tasks. Some of the programming models introduced here can also offload data to accelerators like GPUs and execute tasks on them.

TensorFlow [9] is introduced in this dissertation since it based on tasks. However, tasks are limited to algebra and tensor operations implemented by the TensorFlow developer team since TensorFlow is Artificial Intelligence and Deep Learning oriented. PyTorch [21] and MindSpore [1] are alternative frameworks to TensorFlow with similar interfaces and functioning. They will not be covered in this dissertation since they have concepts similar to TensorFlow and are only providing Artificial Intelligence based tasks.

2.2.1.1 Cpp-Taskflow

Cpp-Taskflow [22] [23] is a C++ parallel programming library based on the task dependency graph model. In Cpp-Taskflow, a tasks is an instance of the C++ *Callable* object on which the operation *std::invoke* is applicable and is used to run the tasks. Then, they can be declared into a taskflow object from the class *tf::Taskflow* which allow to create a task dependency graph and schedule them for execution. The user can express task dependency with the method *precede* applied on the task handler returned from the task creation. Tasks are executed through *tf::Executor* which runs the taskflow and executes the tasks on threads through a work-stealing algorithm. With work-stealing algorithms, workers without tasks to execute can steal tasks scheduled in other worker queues which respects dependencies in order to execute as much tasks in parallel as possible.

Cpp-Taskflow has the feature of dependency graph composition. It allows the user to create dependency graphs and reuse them to compose larger graphs. It is also possible to make recursive and nested compositions

[1] https://www.mindspore.cn/en

[24]. Taskflow objects can be composed multiple times and the result can also be composed. Therefore, it allows to easily create large and complex parallel workloads.

2.2.1.2 OpenMP

OpenMP [2] is an API which provides a portable and scalable model to develop shared memory parallel applications. OpenMP is based on a fork-join programming model where the parallel regions and loops are specified by pragmas. Team of threads are spawned during parallel regions. Parallel loops are split and mapped on multiple threads so that several iterations of the loop can be executed at the same time.

Tasks were introduced in OpenMP 3 [25] in which tasks can be created with the pragma *omp task* and synchronized with the pragmas *omp taskwait* and *omp barrier*.

In OpenMP 4, the task model was extended with data dependencies. It introduced keywords for data dependencies : *in* for consumed data, *out* for produced data and *inout* for data that will be modified during the task. It allows lock-less and more fine-grained synchronizations between tasks.

2.2.1.3 QUARK

QUeueing And Runtime for Kernels (QUARK) [20] is a runtime environment designed to schedule and execute applications that consist of precedence-constrained kernel routines on shared memory systems. The main principle behind QUARK is the implementation of the dataflow model where the scheduling depends on data dependencies between tasks (routines) in a task graph. The routines have parameters as input and output for computations which are used by QUARK to form an implicit DAG connecting the routines. Each parameter has to be extended by its size and its access mode (in, out, inout) depending if the data are consumed, produced or modified by the routine. This allows QUARK to enforce the data dependencies between the routines while preserving the nature of the original code. QUARK also supports multi-threaded tasks in order to manage large bottleneck tasks which would lock the computations by needing a large portion of the data as parameters.

2.2.1.4 Kokkos

Kokkos C++ Performance Portability Library [26] implements a C++ programming model for writing performance portable applications targeting all major HPC platforms. To do so, it provides abstractions for parallel execution and data management. Kokkos is designed to target complex architectures with many level of memory and multiple type of execution resources. Kokkos can use CUDA, HPX [10], OpenMP [2] and Pthreads programming models as backend.

Kokkos also provides a tasking interface to create and execute a directed acyclic graph of task which supports execution of tasks on GPU [27]. Kokkos previous programming model was limited to data parallelism with multidimensional array data structures. It has been enhanced with DAG of tasks to support performance critical algorithms which could not be effectively implemented with data parallelism only.

The Kokkos C++ Performance Portability Programming EcoSystem also provides maths kernels as well as profiling and debugging tools.

2.2.1.5 Cilk

Cilk [28] is a C-based runtime system for multi-threaded parallel programming. It extends the C language in order to ease the developing of Cilk based programs. Cilk programs consists in a set of procedures themselves composed of a sequence of threads. It is based on two main keywords : *spawn* and *sync* which are used to spawn fine grain tasks and synchronize spawned tasks with their parent. The keywords of a Cilk program can be removed and it leaves a serial version of the program that can be used for debugging purposes. In Cilk, tasks are scheduled using a work-first scheduling strategy as well as a randomized work stealing load balancing strategy. They implemented the strategy found optimal in [29].

Cilk++ [30], similarly, extends the C++ programming language and introduce Cilk keywords. As for the C interface, removing the keywords produces a serial application. Cilk++ also supports parallel loops by adding the keyword *cilk_for* which allows the iterations of the loop to be executed in parallel.

2.2.1.6 TBB

Intel Threading Building Blocks (TBB) [31] [32] is a C++ library for shared memory parallel programming. It is a library for task based parallelism which provides concurrent containers, a memory allocator, a work-stealing task scheduler and low-level synchronization primitives. In TBB, tasks are expressed through the Task Flow which is a set of classes to express parallelism as a graph of compute dependencies or data flows. The tasks can be implemented as C++ lambda expressions as they reduce the time and code needed to implement the tasks by removing the requirement for separate objects or classes. Its scheduler is based on a work-stealing engine. TBB provides tread safe scalable containers, timers and exception classes. It also provides synchronization primitives like atomic operations, mutexes and condition variables. TBB supports nested parallelism and allow building parallel components from smaller ones. It emphasizes data-parallel programming by enabling several threads to operate on different parts of a collection. TBB can address CPUs, GPUs and FPGAs.

2.2.1.7 TensorFlow

TensorFlow [9] is "open source software library for high performance numerical computation". A TensorFlow program consists of a set of Operations arranged into a graph and run by a Session. The Tensors contain the data and are used by the Operations. A Tensor is a set of primitive values shaped into a multidimensional array. An Operation runs computations on the provided Tensors. TensorFlow deduces the graph from the used Operations. It provides a lot of built-in Operations in its Low Level API. It also provides a higher-level API which can be used to implement Machine Learning and AI algorithms. Most of the algorithms are related to model training and layers of neural networks. Thus, the high-level API cannot be used to implement non-AI based applications.

By itself, TensorFlow does not support distributed architecture as it is designed to run on GPU and shared memory. A solution to use TensorFlow across multiple nodes is to use Dask [33], a library for parallel computing in Python. It could be used to schedule TensorFlow tasks on distributed resources as well as manage data through its Big Data collections which extends arrays to distributed environments.

2.2.1.8 OmpSs

OmpSs [34] is based on OpenMP [2] and StarSs [35]. StarSs is a parallel programming model that manage tasks. Each task is a piece of code which can be executed asynchronously in parallel. OmpSs uses the Mercurium source-to-source compiler and the Nanos++ Runtime Library. Mercurium provides the support to transform the high-level directives into a parallelized version of the application. Nanos++ manages the parallelism in the application, including task creation, synchronization and data movement. The tasks are expressed through the task construct with the in, out, inout clauses to express data dependencies. A mix with the OpenMP directives may give the information on the control graph but we didn't find any example of that. OmpSs stays close to the sequential code but a higher level language and more software engineering may be required to face the challenges of the exascale.

2.2.1.9 CnC

CnC [36] [37] (Current Collections) is a graph parallel programming model. A CnC graph uses three types of nodes : the step collections is a computation task, the data collections stores the data needed in the computations, and the control collections creates instances of one or more step collections. With this structure, it is possible to represent both the control and data flow graphs. The control flow graph represents the dependencies between the computing operations whereas the data flow graph represents the data dependencies between the operations. In CnC, there are two ordering requirements : producer/consumer linked to data dependencies and controller/controlee linked to computation dependencies. It allows important scheduling optimizations with the two graphs directly available. CnC has a lot of explanations and possibilities but the development of an application becomes complex very quickly.

2.2.2 Task Based Programming Models on Distributed Memory

We now present some task based programming models in which tasks are scheduled to run on distributed memory. In these programming models, the data dependencies are also taken into consideration to execute single thread (or process) tasks on multiple nodes. The data used by the tasks are described to the programming models so that the schedulers can execute the tasks where the data are stored to reduce data movement if it is supported or send the data where they will be consumed by tasks.

2.2.2.1 Uintah

Uintah [38] [39] is an open-source asynchronous many-task runtime system. In Uintah, data dependencies and parallel computations are expressed with an abstract task graph representation. This representation is a directed acyclic graph of tasks. Uintah uses a Data Warehouse to store distributed data in which all data transfer takes place. It is an abstraction that ensures that the user-coded tasks are independent of the hidden MPI communication layer. Each Uintah application specifies a list of tasks and their data dependencies in a declarative style. Each task is implemented as a C++ function which performs the actual computation, consumes the input and produces an output. Uintah analyzes the DAG to automatically enable load-balancing, data communication, parallel I/O, and check-pointing/restarting. MPI communication are automatically set up for data dependencies between MPI processes. Uintah provides fault tolerance mechanisms [40].

Kokkos [26] is used in Uintah to extend the code base to many-core architectures instead of adopting OpenMP [41]. It also allows to extend Uintah to GPU-based, many-core, and multi-core architectures although Uintah already supports multi-core and GPU-based systems by directly using architecture specific programming models (for instance, CUDA) in the tasks.

2.2.2.2 Charm++

Charm++ [42] [43] is a parallel object-oriented programming language based on C++. It uses an asynchronous message driven execution model. *Chares* are the basic parallel unit in Charm++. It is similar to a lightweight process. A *chare* is similar to a C++ object which contains the data and the computations of an application. A Charm++ program consists in a large number of *chares* distributed on the available computational resources. They interact with each other through asynchronous method invocations. When a *chare* receives a message, the associated method is executed. The *chares* are assigned to a core by the Charm++ runtime system.

2.2.2.3 X10

X10 [44] is an open-source programming language developed by IBM. It aims to provide a programming model that can address the architectural challenge of multiples cores, hardware accelerators, clusters, and supercomputers in a manner that provides scalable performance in a productive manner. X10 is a strongly-typed and garbage-collected object-oriented language. X10 uses the Asynchronous Partitioned Global Address Space programming model (APGAS) with its two main concepts : places and asynchronous tasks. They are used to express both regular and irregular parallelism, message-passing-style and active-message-style computations, fork-join and bulk-synchronous parallelism.

Resilient X10 [45] [46] builds on X10 by exploiting the strong separation provided by places to provide a coherent semantics for execution in the presence of failures.

2.2.2.4 Legion (Regent, Pygion)

Legion [7] is a data-centric parallel programming model. It aims to make the programming system aware of the structure of the data in the program. Legion provides explicit declaration of data properties (organization, partitioning, privileges, and coherence) and their implementation via the logical regions. They are the fundamental abstraction used to describe data in Legion applications. Logical regions can be partitioned into sub-regions and data structures can be encoded in logical regions to express locality describing data independence.

A Legion program is executed as a tree of tasks spawning sub-tasks recursively. Each task specifies the logical region they will access. With the understanding of the data and their use, Legion can extract parallelism and find the data movement related to the specified data properties. Legion also provides a mapping interface to control the mapping of the tasks and the data on the processors during the execution of the application.

Regent [8] is a programming model which simplifies Legion. Regent compiler translates Regent programs into efficient implementations for Legion. It results in programs that are written with fewer lines of codes and at a higher level.

Pygion [47] is a Python interface for Legion. It aims to provide the same functionalities as Regent while being more flexible than Regent and more concise and easier to use than the C++ Legion code.

2.2.2.5 PaRSEC

PaRSEC [6] [48] (Parallel Runtime Scheduling and Execution Controller) is an engine for scheduling tasks on distributed hybrid environments.

It offers a flexible API to develop domain specific languages. It aims to shift the focus of developers from repetitive architectural details toward meaningful algorithmic improvements. Two domain specific languages are supported by Parsec, the Parameterized Task Graph [49] (PTG) and Dynamic Task Discovery [50] (DTD)

In PTG [49], users would use a parameterized expression of task dependencies combined with PaRSEC which would implicitly infer the communication between nodes and the accelerators. The users have to provide a description of the data flow in their application and the tasks which are applied on the data. They also have to provide the tasks that are the source of the data and the tasks that are the destination. Therefore, this is a compressed representation of the task graph. Afterwards, it is transformed into C code using a pre-compiler. Users need to understand and provide all the data flow of their algorithm to use this model.

DTD [50] is a task-based programming paradigm which provides an alternative way to express task dependency in PaRSEC that achieves a similar purpose to PTG. On the contrary of PTG where users had to express tasks in a parameterized manner, DTD allows them to write sequential constructs (ifs, loops, etc. . .) to insert tasks in PaRSEC. The tasks are given to the runtime with the data they will use and their mode of usage. Then, the runtime will compute dependencies out of data pointers used by the tasks. In a distributed system, the data movements among the nodes are completely implicit.

2.2.2.6 HPX

High Performance ParalleX (HPX) [10] is a C++ Standard Library for Concurrency and Parallelism. It implements the facilities defined by the C++ Standard and functionalities proposed as part of the ongoing C++ standardization process. It also extends the C++ Standard APIs to the distributed case. The goal of HPX is to create a high quality, freely available, open source implementation of a new programming model for conventional systems.

HPX API implements the interfaces defined by the C++11/14/17/20 ISO standard and respects the programming guidelines used by the Boost collection of C++ libraries. It aims to improve the scalability of current applications. It also tries to expose new levels of parallelism which are necessary to take advantage of the future systems.

HPX is an open-source implementation of the ParalleX execution model. This model focuses on overcoming the four main barriers to achieve scalability (Starvation, Latencies, Overhead, Waiting for contention resolution).

2.2.2.7 StarPU

StarPU [51] is a runtime system. It schedules tasks on computing resources with accelerators. The tasks can have several implementations depending on the architecture of the processing units and data transfers are

handled by StarPU. StarPU uses auto-tuning to predict execution time and data transfer overhead. It allows StarPU's dynamic scheduler to avoid load imbalance.

Data has to be registered into StarPU so that it can transfer them. The algorithm has to be described as a set of tasks (usually kernels). A task is defined by a codelet, the data used by the task and their access mode. The dependencies between the tasks can be given explicitly or inferred from the data dependencies.

2.2.2.8 ClusterSs

Cluster Superscalar (ClusterSs) [52] is also based on StarSs [35] and designed for large scale clusters. In ClusterSs, tasks are asynchronously created and spawned on the worker nodes as the application is executed. ClusterSs runtime is built on IBM APGAS runtime. APGAS is used to handle the inter-nodes communications and its Active Message paradigm allows to spawn tasks on remote nodes efficiently. The user is required to identify the methods that can be executed in parallel whereas the parallel execution and the data placement is managed by the runtime. The global data is automatically distributed according to computation needs. The execution model is based on one main node that generates tasks and workers that execute the tasks. The main node runs the user application and the selected functions marked by the user are replaced by a call to the ClusterSs runtime. Then, the runtime creates a task which is added to a DAG of dependency. When a task can be executed, the task is scheduled and sent to a node for execution. When choosing the node, ClusterSs tries to exploit data locality then executes the task on the selected node. It sends the data on the node if necessary.

2.2.2.9 Chapel

Chapel [53] is a parallel programming language designed to run on large-scale systems by Cray Inc and their collaborators. It is also designed to be portable and run from multicore desktop to large supercomputers. It was created from principles rather than being an extension of another language. It is a object-oriented programming model with type inference and features for generic programming. Chapel offers high-level abstractions for data parallelism, task parallelism, concurrency, and nested parallelism. It supplies a type called *locale* which enables users to specify and reason about the placement of data and tasks. Finally, Chapel provides interoperability features to integrate other languages into Chapel or vice-versa.

Chapel is an open-source project hosted on GitHub under the Apache v2.0 license. It uses third-party open-source packages under their own licenses.

2.2.3 Task Based Programming Models where Tasks run on Distributed Memory

Finally, task based programming models in which tasks are parallel and distributed functions or applications capable of running on several nodes are introduced. In this case, data are not always managed by the scheduler. Indeed, some of the programming models with parallel and distributed tasks use the file system to transmit data from one task to another by explicitly writing and reading data from files.

2.2.3.1 YML+XMP

YML [54] is a development and execution environment for scientific workflow applications over various platforms, such as HPC, Cloud, P2P and Grid with multilevel of parallelism. YML defines an abstraction over the different middlewares, so the user can develop an application that can be executed on different middlewares without making changes related to the middleware used. YML can be adapted to use different programming models to implement tasks. Currently, the proposed back-end [55] uses OmniRPC-MPI [56], a grid RPC which supports master-worker parallel and distributed programs based on multi SPMD programming paradigms. This back-end is developed for large scale clusters such as Japanese K-Computer [55]. A back-end for peer to peer networks is also available. Moreover, YML also supports fault tolerance [57].

XMP can be used to implement YML components as introduced in [55]. This allows two levels programming. The higher level is the graph (YML) and the second level is the PGAS component (XMP). In the components, YML needs complementary information to optimize the management of the computational resources and the data : the number of XMP processes for a component and the distribution of the data in the processes (template). With this information, the scheduler can anticipate the resource allocation and the data movements. The scheduler creates the processes that the XMP components need to run the component. Then each process will get the piece of data which will be used in the process from the data repository.

2.2.3.2 Swift

Swift [13] [58] is a scripting language for executing many instances of ordinary application programs on distributed and parallel resources. Swift runs applications as soon as their inputs are available. Swift expressions are evaluated in dataflow order to run the workflows with the most concurrency possible. It only depends on the data dependencies and the available resources. Moreover, Swift allow to wrap applications into functions to create more complex scripts. The functions take files as input and produce files as output.

Swift/T [59] is a new implementation of Swift dedicated to HPC. It translates the Swift script into a MPI program using Turbine [60] and ADLB (Asynchronous Dynamic Load Balancer) [61] libraries. Turbine is a Tcl library that assemble MPI, ADLB and the Turbine dataflow library. ADLB is a software library designed to build scalable parallel programs. Swift/T has two main level of programming : the Swift script and the Leaf functions (the task implementations).

The extension functions are the first type of leaf functions. They consist in Tcl or native code functions like C, C++, Fortran or MPI which operate on in-memory data, and are appropriate for high-performance computing. There are also application functions. They are used to call a command-line program which process files, and are appropriate for ordinary workflows. Finally, external scripting functions are used to call an in-memory interpreter in another scripting language, such as Python, R or Julia.

2.2.3.3 HTCondor DAGMan

The DAGMan (Directed Acyclic Graph Manager) [62] is designed to run complex sequences of long-running jobs with dependencies on the Condor [63] [64] middleware. DAGMan is fault-tolerant. It keeps private logs, allowing it to resume a DAG where it left off, even in the the case of failure. The language accepted by DAGMan is constituted of a few keywords. The JOB statement is used to describe a Condor job by associating a job name to a file. The PARENT-CHILD statement describes the dependencies between jobs.

Jobs without dependencies between them can be executed at the same time. DAGMan also allows the usage of pre- and post-scripts by using the keywords PRE and POST to specify the program that will be executed. The RETRY keywords specify DAGMan that a given job can be retried if the job fails.

2.2.3.4 Pegasus

Pegasus [12] is a Workflow Management System. It allows the user to express multi-step computational tasks through a directed acyclic graph (DAG), where the nodes are tasks and the edges denote the task dependencies. The tasks can be everything from short serial tasks to very large parallel tasks (MPI for example) surrounded by a large number of small, serial tasks used for pre- and post-processing.

Pegasus provides helpers to execute workflow-based applications in different environments like desktops, clusters, grids and clouds. It automatically maps high-level workflow descriptions onto distributed resources. It also locates the necessary input data and computational resources needed during the workflow execution. Pegasus enables scientists to construct workflows in abstract terms. It can be linked with several middlewares (Condor, Globus, or Amazon EC2).

Pegasus is fault tolerant. Indeed, when there are errors, it can retry the tasks, retry the entire workflow, checkpoint its execution, re-map part of the workflow, try alternative data sources and as last resort, provide a list of the remaining tasks. Storage is cleaned up during the execution of the workflow. Thus, data-intensive workflows can get enough space to run their tasks on resources with low-capability storage. Pegasus also keeps track of what has been done, where, which software was used and their parameters.

2.3 Exascale Challenges of Supercomputers

The upcoming exascale supercomputers are raising both hardware and software challenges. On one side, computing power is now achieved with the multiplication of computing resources. This leads to highly hierarchical supercomputer with hundreds of high level computing nodes which are made of several multi-core processors and eventually one or more accelerators. In each node, there are also several levels of memory with the RAM and the different cache level of each processor and accelerator. These machines tend to be both massively parallel with the large amount of diverse cores available and massively distributed with their multiple memory levels only accessible directly by a few cores whereas the local memory can be remotely accessed through data communications. Furthermore, there is the network connecting the nodes between themselves. Each node cannot be directly connected to all the others since it would be very costly with the large amount of computing nodes available in the supercomputers. Therefore, the network is also becoming more complex in order to support the communications between nodes. This increases the distance between the furthest nodes on the network and the cost of sending message to far away nodes.

Finally, there are multiple variant of processors, accelerators and networks available on the market. Thus, a lot of different combinations of these components are now incorporated on the current supercomputers. These different architectures require different approaches to implement applications that can use the components available on the nodes efficiently. The first challenge is to select a fitting architecture among all the architectures available.

As there are multiple programming models that address one or more of the available architecture, the

second challenge will be to find a parallel programming model that can take advantage of the targeted architecture while being compatible with the application to execute on the supercomputer. As the current machines tend to become massively parallel and distributed, system faults will happen more frequently on large scale applications since, even with a small probability of an error occurring on one of the computing resource, the large number of resources used increase the probability of an error occurring during the execution of the application. To avoid losing the computations already performed, check pointing allow the save of predefined application variable state regularly on disk and thus recovering the last state saved if an error occur. This is the second challenge. Task based programming models can help to address these challenges by providing an easy way to regularly save the state of an application and recover the computations. Indeed, the data of task based applications are managed by the scheduler so it is easy for it to save the data. Moreover, the recovery of the application means restarting to execute the graph of task where the last data were saved. Therefore, task based programming models with scheduler optimizing data movement and managing fault tolerance as well check pointing, while avoiding global operation by encapsulating them into smaller tasks, seem well adapted for these machines.

The third challenge concerns the kind of application that will be run on supercomputers and the efficiency of their parallel algorithm as well as its implementation. The algorithm having the most operation that can be executed in parallel while reducing the number of communications will most likely obtain the best performances on this kind of systems even if there more computations locally. Moreover, highly optimized common kernels will be very valuable since many applications heavily rely on them. A few examples of such kernels are present in linear algebra such as direct (for instance, algorithms based on the Gaussian elimination) and iterative (Conjugate Gradient, Krylov subspaces...) methods to solve linear systems as well as in Artificial Intelligence (matrix vector product for neural network training, PCA, SVD...).

The following chapter will introduce some of the methods we chose as examples to implement and compare the programming models considered in this dissertation. We chose dense linear algebra methods (solution of dense linear systems) and sparse linear algebra (sparse matrix vector product) as well as a real world application, the Kirchhoff seismic pre-stack depth migration.

Chapter 3

Methods

This chapter introduces the basic algorithms from which the task-based algorithms in the next chapters are derived. The dense linear algebra algorithms will be presented then the sparse linear algebra and finally, the Kirchhoff seismic pre-stack depth migration.

3.1 Dense Linear Algebra

This section focus on the dense linear methods we consider as benchmark to experiment on the programming models. We introduce the LU factorization, the Gaussian elimination, the Gauss-Jordan elimination and the backward and forward substitution. We introduce the regular point to point methods as well as the block based methods.

These methods to solve dense linear systems two very useful algebra operations. They are used as base for more complex operations like linear solvers. For instance, the LU factorization to solve a linear system is used a benchmark which is called LINPACK [65] [66] for the TOP 500 which ranks the performances of the supercomputers. The LU factorization is also used to invert a matrix in LAPACK [67] and its distributed counterpart ScaLAPACK [68] [69].

These implementations are considered as using block algorithms. However, the term "block" as several meanings in the literature as shown in [70]. According to [70], the algorithms used in LINPACK, LAPACK and ScaLAPACK as explained in [71] are better termed as *partitioned algorithms* which "is a scalar algorithm in which the operations have been grouped and reordered as matrix operations" whereas block algorithms are "a generalization of a scalar algorithm in which the basic scalar operations become matrix operations ($\alpha + \beta, \alpha\beta, \alpha/\beta$ become $A + B, AB, AB^{-1}$)"

In this section, we are interested in the block algorithms referring to a generalization of the scalar algorithms. Therefore, this explain why our algorithms are different from ScaLAPACK's.

3.1.1 Block-Based LU factorization to Solve Linear Systems

The block LU factorization problem is introduced in [70] [72] [73]. It consists in finding L and U such as $A = L \cdot U$ where L is a block lower triangular matrix of size $p \times p$ with identity on the diagonal and U is a block upper triangular matrix of size $p \times p$ (see Equation 3.1). However, in general, the U_{ii} blocks are not triangular. They may be complete matrices.

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix} = \begin{bmatrix} I & & \\ L_{2,1} & I & \\ L_{3,1} & L_{3,2} & I \end{bmatrix} \begin{bmatrix} U_{1,1} & U_{1,2} & U_{1,3} \\ & U_{2,2} & U_{2,3} \\ & & U_{3,3} \end{bmatrix} = LU \tag{3.1}$$

We consider LU factorization $A = LU \in \mathbb{R}^{n \times n}$ partitioned as follow where the diagonal blocks are square but are not necessarily of the same dimension. If A_{11} is nonsingular, we can write

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} = \begin{bmatrix} I & \\ L_{2,1} & I \end{bmatrix} \begin{bmatrix} A_{1,1} & A_{1,2} \\ & S \end{bmatrix} \tag{3.2}$$

Equation 3.2 describes a step of a recursive algorithm to compute a block LU factorization where $S = A_{2,2} - A_{2,1}A_{1,1}^{-1}A_{1,2}$ is a Schur complement of A. If we partition S and its $(1,1)$ block is nonsingular, we can factorize S in a similar manner. This process can be repeated until a complete LU factorization is obtained. Algorithm 1 shows this process.

Algorithm 1: Recursive Block (Generalized) LU Factorization

Partition the matrix A : $U_{1,1} = A_{1,1}, U_{1,2} = A_{1,2}$
Solve $L_{2,1}A_{1,1} = A_{2,1}$ for $L_{2,1}$
$S = A_{2,2} - A_{2,1}A_{1,1}^{-1}A_{1,2}$
Repeat this process on S

Algorithm 2 introduces a non-recursive algorithm for the block LU factorization.

Algorithm 2: Block (Generalized) LU Factorization

For k **from** *0* **to** *p-2* **do**
  (1) $Inv^{(k)} = [A_{k,k}^{(k)}]^{-1}$
  For i **from** *k+1* **to** *p-1* **do**
    (2) $A_{i,k}^{(k+1)} = A_{i,k}^{(k)} \cdot Inv^{(k)}$
    For j **from** *k+1* **to** *p-1* **do**
      (3) $A_{i,j}^{(k+1)} = A_{i,j}^{(k)} - A_{i,k}^{(k+1)} \cdot A_{k,j}^{(k)}$
    End for
  End for
End for

The goal here is to solve the linear system $Ax = b$ by replacing A by its LU factorization. Therefore, the problem becomes solving the system $LUx = b$. There are two triangular linear systems to solve : $Ly = b$ then $Ux = y$. Algorithm 3 solves the linear system $Ax = b$ to obtain x given the LU factorization of A stored in one matrix and b.

The block LU algorithm can also be obtained from the scalar algorithm since the it can be generalized by changing from scalar to block matrices and using the appropriate operations. The scalar algorithm with pivot for the LU factorization can be found in [74] [75]. However, there is no pivoting in the block algorithm

Algorithm 3: Backward and forward substitution to solve a triangular linear system with a block-based LU factorization

For i **from** *0* **to** *p-2* **do**
  For j **from** *i+1* **to** p **do**
    (5) $b_j^{(i+1)} = b_j^{(i)} - A_{j,i}^{(i+1)} \cdot b_i^{(p)}$
  End for
End for

For k **from** *p-1* **to** *0* **step** *-1* **do**
  (6) solve $b_k^{(p)} = A_{k,k}^{(k)} \cdot b_k^{(p-1)}$
  For i **from** *1* **to** *k-1* **do**
    (7) $b_i^{(p-k+i)} = b_i^{(p-k+i-1)} - A_{i,k}^{(i)} \cdot b_k^{(p)}$
  End for
End for

Algorithm 4: Scalar LU Factorization

For k **from** *0* **to** *p-2* **do**
  For i **from** *k+1* **to** *p-1* **do**
    (1) $a_{i,k}^{(k+1)} = a_{i,k}^{(k)} / a^{(k)}$
    For j **from** *k+1* **to** *p-1* **do**
      (2) $a_{i,j}^{(k+1)} = a_{i,j}^{(k)} - a_{i,k}^{(k+1)} \cdot a_{k,j}^{(k)}$
    End for
  End for
End for

although pivoting may be used to increase the stability during the inversion of the diagonal blocks. Algorithm 4 is a scalar version without pivot of the LU factorization [72].

The LU factorization is based on the Gaussian elimination which will can also be used to solve linear systems.

3.1.2 Block-Based Gaussian Elimination to Solve Linear Systems

The scalar Gaussian elimination algorithm with pivoting is discussed in [76] and [77]. The pivoting increase the numerical stability of the operations. However, in the case of block algorithms with a generalization of the scalar operations into a matrix operation, pivoting the rows of sub-matrices means that we need to find a way to determine which matrix could be used as a pivot. Moreover, the increase in numerical stability of such method is unknown. This topic is not discussed in this work. Furthermore, the algorithm which is used as base for generalization in the LU factorization does not use pivoting on the sub-matrices. Although, pivoting may be used to improve the numerical stability in the inversion of the sub-matrices. Algorithm 5 introduces a scalar Gaussian elimination without pivoting [72] which will be generalized into a block algorithm in

Algorithm 6. These two algorithms are used to solve a linear system and include the back substitution which allow to compute the solution of the linear system.

Algorithm 5: Scalar Gaussian elimination and back substitution

For k **from** *0* **to** *p-2* **do**
  (1) $b_k^{(k+1)} = b_k^{(k)} / a_{k,k}^{(k)}$
  For j **from** *k+1* **to** *p-1* **do**
    (2) $a_{k,j}^{(k+1)} = a_{k,j}^{(k)} / a_{k,k}^{(k)}$
  End for
  For i **from** *k+1* **to** *p-1* **do**
    (3) $b_i^{(k+1)} = b_i^{(k)} - a_{i,k}^{(k)} \cdot b_k^{(k+1)}$
    For j **from** *k+1* **to** *p-1* **do**
      (4) $a_{i,j}^{(k+1)} = a_{i,j}^{(k)} - a_{i,k}^{(k)} \cdot a_{k,j}^{(k+1)}$
    End for
  End for
End for
(5) $b_{p-1}^{(p)} = b_{p-1}^{(p-1)} / a_{p-1,p-1}^{(p-1)}$
For k **from** *1* **to** *p-1* **do**
  For i **from** *0* **to** *p-k-1* **do**
    (6) $b_i^{(k+i+1)} = b_i^{(k+i)} - a_{i,p-k}^{(i+1)} \cdot b_{p-k}^{(p)}$
  End for
End for

The Gaussian elimination has a similar number of block operations as the block LU factorization but the critical path of the graph of operations is significantly shorter than the one in the block LU factorization since there is only one solution of a triangular system after the operations on the matrix whereas there are two in the block LU factorization. This means that the block LU factorization and the block Gaussian elimination have different performances.

Algorithm 6: Block (Generalized) Gaussian elimination and back substitution

For k **from** *0* **to** *p-2* **do**
  (1) $Inv^{(k)} = [A_{k,k}^{(k)}]^{-1}$
  (2) $b_k^{(k+1)} = Inv^{(k)} \cdot b_k^{(k)}$
  For j **from** *k+1* **to** *p-1* **do**
    (3) $A_{k,j}^{(k+1)} = Inv^{(k)} \cdot A_{k,j}^{(k)}$
  End for
  For i **from** *k+1* **to** *p-1* **do**
    (4) $b_i^{(k+1)} = b_i^{(k)} - A_{i,k}^{(k)} \cdot b_k^{(k+1)}$
    For j **from** *k+1* **to** *p-1* **do**
      (5) $A_{i,j}^{(k+1)} = A_{i,j}^{(k)} - A_{i,k}^{(k)} \cdot A_{k,j}^{(k+1)}$
    End for
  End for
End for
(6) $b_{p-1}^{(p)} = Inv^{(p-1)} \cdot b_{p-1}^{(p-1)}$
For k **from** *1* **to** *p-1* **do**
  For i **from** *0* **to** *p-k-1* **do**
    (7) $b_i^{(k+i+1)} = b_i^{(k+i)} - A_{i,p-k}^{(i+1)} \cdot b_{p-k}^{(p)}$
  End for
End for

3.1.3 Block-Based Gauss-Jordan Elimination to Solve Linear Systems

The Gauss-Jordan elimination [78] [79] is similar to the Gaussian elimination except that the computations made under the diagonal are also performed above it while directly outputting the solution of the linear system. Algorithm 7 is a scalar Gauss-Jordan elimination algorithm without pivoting [79]. It is indeed very similar to Algorithm 5 which is the Gaussian elimination with the mentioned modifications. This algorithm is also converted into a generalized block Gauss-Jordan elimination [80] in Algorithm 8.

The Gauss-Jordan elimination has more block operations than the block Gaussian elimination and the block LU factorization but the critical path of the block operations is even shorter than the Gaussian Elimination since there is no solution of triangular systems. Although, there are more block operations than the two other block algorithms but the critical path is shorter. This means that there are more block operations that can be performed in parallel during the elimination. This also means that this algorithm may be able to run faster than the two other block algorithms on a parallel machine.

In this section, we introduced the different meaning of block algorithm : the algorithms where the scalar operations are generalized into matrix operations or algorithms where the operations are reorganized in order to perform matrix operations (usually matrix product) on subset of the initial matrix. Moreover, we introduced the scalar and block algorithms for the LU factorization, the Gaussian elimination and the Gauss-Jordan elimination.

Dense linear system solution is widely used in solvers for simulations so it is important to create robust

Algorithm 7: Scalar Gauss-Jordan elimination to solve a linear system

For k **from** *0* **to** *p-1* **do**
  (1) $b_k^{(k+1)} = b_k^{(k)} / a_{k,k}^{(k)}$
  For j **from** *k+1* **to** *p-1* **do**
    (2) $a_{k,j}^{(k+1)} = a_{k,j}^{(k)} / a_{k,k}^{(k)}$
    For i **from** *0* **to** *p-1* **do**
      If $k \neq i$ **then**
        (3) $a_{i,j}^{(k+1)} = a_{i,j}^{(k)} - a_{i,k}^{(k)} \cdot a_{k,j}^{(k+1)}$
      End if
    End for
  End for
  For i **from** *0* **to** *p-1* **do**
    If $k \neq i$ **then**
      (4) $b_i^{(k+1)} = b_i^{(k+1)} - a_{i,k}^{(k)} \cdot b_k^{(k+1)}$
    End if
  End for
End for

and efficient library providing these operation since it will save time and computational resources. Simulations can also create sparse matrices to represent the simulated environment. In this case, direct method to solve linear systems based on the Gaussian elimination are not suitable since the factorization modify the matrices and may introduce new values at places where there are zeros in the sparse matrix. However, it exist methods to limit the filling of the zeros values in the direct methods to solve linear systems but these methods are difficult to set up and parallelize. Moreover, they are not in the scope of this dissertation and, thus, they will not be covered. An alternative to that are iterative methods, which usually, do not modify the input matrix and are used to deal with the sparse matrices. Most of these iterative methods are based on the sparse matrix vector product which will be discussed in the next section.

Algorithm 8: Block (Generalized) Gauss-Jordan elimination to solve a linear system

For k **from** *0* **to** *p-1* **do**
  (1) $Inv^{(k)} = [A_{k,k}^{(k)}]^{-1}$
  (2) $b_k^{(k+1)} = Inv^{(k)} \cdot b_k^{(k)}$
  For j **from** *k+1* **to** *p-1* **do**
    (3) $A_{k,j}^{(k+1)} = Inv^{(k)} \cdot A_{k,j}^{(k)}$
    For i **from** *0* **to** *p-1* **do**
      If $k \neq i$ **then**
        (4) $A_{i,j}^{(k+1)} = A_{i,j}^{(k)} - A_{i,k}^{(k)} \cdot A_{k,j}^{(k+1)}$
      End if
    End for
  End for
  For i **from** *0* **to** *p-1* **do**
    If $k \neq i$ **then**
      (5) $b_i^{(k+1)} = b_i^{(k+1)} - A_{i,k}^{(k)} \cdot b_k^{(k+1)}$
    End if
  End for
End for

3.2 Sparse Linear Algebra

There is a large amount of applications that use sparse matrices. For instance, the discretization of partial differential equation in numerical simulations often leads to the creation of a linear system which has to be resolved in order to solve the equation. Another example is the representation of the dependency graph of web pages on which the PageRank method can be applied to rank them. The linear system is most of the time stored as a matrix. When the matrix is dense, direct methods can be used to solve the linear system as seen in the previous section. However, the discretization of the partial differential equation can produce linear system where the matrix contains a large amount of zero. A matrix can be considered sparse when storing the matrix in a sparse storage format takes less memory space than using the dense storage format. In this case, the matrix is considered sparse and is stored using sparse storage formats. Some of the most commonly used sparse matrix storage formats are discussed in this section. Moreover, the direct methods to solve sparse linear systems are complex to implement since they introduce new values in the matrix during the Gaussian elimination. Indeed, modifying a sparse matrix is difficult since depending on the storage format the row or the columns are compressed (the zero are removed from the arrays) so inserting new values means creating a new array and copying the previous values while adding the new one, except for a few sparse storage formats like coordinates (COO introduced later). Therefore, avoiding insertion in sparse matrix improves the performances greatly since the copy is very expensive. This leads to the use of iterative sparse methods to solve linear systems. These methods heavily rely on sparse matrix vector product. There are also several other methods that rely on sparse matrix vector product like the conjugate gradient, the Lanczos methods, the Krylov subspaces [81] [82], Page Rank [83] and an essential part of many graph kernels

[84]. Even hardware prototypes [85] have been designed to improve the graph computational throughput compared to conventional hardware.

In this section, we will focus on the different sparse matrix storage and their corresponding matrix vector product implementation.

3.2.1 Sparse Matrix Storage and Sparse Matrix-Vector Multiplication

Sparse matrix format are used to store and process sparse matrices in a wide range of domains and libraries. For instance, SPARSKIT [86] is a library which provides routines to convert matrices in one sparse format to another one. It also provides a set of routines to make operation on sparse matrices like matrix vector product, eigenvalues computations and linear system solving. The following sections will describe several widely used sparse matrices formats introduced in SPARSKIT and give the algorithm to compute a matrix vector product with those formats.

3.2.1.1 Dense

Dense is the classical and most straightforward format used to store matrices. It is usually used for matrices with a small number of null values. In the case of sparse matrices, it is not efficient since it stores a large amount of unused values (the zeros). Those zeros will also be part of computations while making operations on the matrix. Therefore, storing the sparse matrices into a more adapted format will reduce memory footprint and save operations on zeros. Figure 3.1 shows a 5×5 sparse matrix stored as a dense matrix.

1	2	0	0	3
0	0	4	0	0
0	5	0	0	0
0	6	0	7	0
8	0	9	0	10

Figure 3.1: Sparse matrix stored in dense format

Algorithm 9: Matrix vector multiplication - dense

For *i = 1, N_{row}* **do**
  y[i] = 0.0
  For *j = 1, N_{col}* **do**
    $y[i] \mathrel{+}= val[i,j] \times x[i]$
  End for
End for

3.2.1.2 Coordinates - COO

COO are the first letters of the word coordinates which summarize how the sparse matrices will be stored in this format. It uses two vectors to store the two dimensional coordinates of the value in the matrix and

another vector which contains the values. It is a key-value storage type. The key is the coordinates of the value in the matrix.

The COO format is used in TensorFlow to store sparse matrices into tensors. This format is also used in Big Data libraries and, in particular, MapReduce [87] where the index tuple is used as key.

The COO format has several useful properties. The values can be easily split and distributed by dividing the three vectors while keeping the coordinates-values pairs. It is also a format that allows easy construction of matrices since the addition of new values is easy. Indeed, they can easily be added at the end of the three vectors. The transposition is fast. Only, the row and the column vectors have to be exchanged.

Figure 3.2 shows an example of the storage of the example matrix in COO format.

row	0	0	0	1	2	3	3	4	4	4
col	0	1	4	2	1	1	3	0	2	4
val	1	2	3	4	5	6	7	8	9	10

Figure 3.2: Sparse matrix stored in COO format

Algorithm 10: Matrix vector multiplication - COO

For $i = 1, N_{row}$ **do**
  y[i] = 0.0
End for
For $i = 1, NNZ$ **do**
  $y[row[i]] \mathrel{+}= val[i] \times x[col[i]]$
End for

3.2.1.3 Compressed Sparse Row - CSR

The Compressed Sparse Row format (CSR) [88] [89] or Yale format [90] is similar to the COO format in which the vector containing the index of the rows is compressed. The array giving the row index of the value at the same position in the array of values is replaced by an array which gives the position of the beginning of each row in the column array. In COO, the row array has NNZ values (the number of non zero values) whereas there are only $N + 1$ values in the CSR row array (N is the dimension of the matrix) where N is smaller than NNZ. Therefore, it saves space in memory. A column version of this format called Compressed Sparse Column (CSC) exists. In CSC, the index array for the columns is compressed.

However, inserting new values in a matrix stored in the CSR format implies to move the values in the column and values arrays after the inserted value.

CSR is the default format used in most of the HPC libraries performing operations on sparse matrices. For instance, Intel MKL and Nvidia CuSparse are using the CSR format as input for the sparse matrices used in their routines.

Figure 3.3 shows an example of the storage of the example matrix in CSR format.

idx	0	3	4	5	7	10				
col	0	1	4	2	1	1	3	0	2	4
val	1	2	3	4	5	6	7	8	9	10

Figure 3.3: Sparse matrix stored in CSR format

Algorithm 11: Matrix vector multiplication - CSR

For $i = 1, N_{row}$ **do**
  y[i] = 0.0
  For $j = idx[i], idx[i+1]$ **do**
    $y[i] \mathrel{+}= val[j] \times x[col[j]]$
  End for
End for

3.2.1.4 Diagonal - DIA

The diagonal format (DIA) is very useful in the case of matrices which have a pattern where the non-zero are located in a small amount of diagonals. It is composed of two arrays : a one dimensional array which represents the offsets from the diagonal of the values in the associated column and a two dimensional array with the values in each row in the column associated to their offset from the diagonal. The values array needs to be padded with zeros where there are no values from the initial matrix in order to be used in the algorithms. This format is designed for matrices where diagonals are very populated since it needs a column in the values array for each diagonal where there are values.

offset	-4	-2	-1	0	1	4
values	0	0	0	1	2	3
	0	0	0	0	4	0
	0	0	5	0	0	0
	0	6	0	7	0	0
	8	9	0	10	0	0

Figure 3.4: Sparse matrix stored in diagonal format

Figure 3.4 shows an example of the storage of the example matrix in diagonal format. However, this example is not adapted to the diagonal storage format since it leaves a lot of padding values in the values matrices. For this format, a matrix where is non-zero values are located on the same diagonals is more interesting.

Algorithm 12: Matrix vector multiplication - DIA

For $i = 1, N_{row}$ **do**
  y[i] = 0.0 **For** $j = 1, n\_diag$ **do**
    $y[i] += values[i, j] \times x[i + offset(j)]$
  End for
End for

3.2.1.5 Ellpack-Itpack - ELL

The Ellpack-Itpack format (ELL) is a generalization of the diagonal format. It uses two arrays with two dimensions to store the position of the values in the columns and the values. Each row of those arrays store the values in the rows of the original matrix without zeros and the column array store the column index of the value in the original matrix. The values array needs to be padded with zeros where there no values from the initial matrix in order to be used in the algorithms.

values	1	2	3
	4	0	0
	5	0	0
	6	7	0
	8	9	10
columns	0	1	4
	2	*	*
	1	*	*
	1	3	*
	0	2	4

Figure 3.5: Sparse matrix stored in Ellpack-Itpack format

Figure 3.5 shows an example of the storage of the example matrix in Ellpack-Itpack format.

As for CSR and CSC, the Ellpack-Itpack format has also a column variant in which the row index is stored instead of the index of the column index.

This format also supports a way to split very large rows or columns depending on the version considered. It allows to reduce the memory occupied by the values. However, these methods are not described in this dissertation.

3.2.1.6 Sparse General Pattern - SGP

The Sparse General Pattern format (SGP) was introduced in [91] and there is an example of such format in [92]. The SGP sparse storage format can be used to compress matrices by rows (Figure 3.6) or by columns (Figure 3.7) depending on the needs. This format is similar to Ellpack with the addition of another matrix that stores the position of the value in the column (resp. row) for the compressed row (resp. column) format. This allows to easily change between row and column compressed formats.

Algorithm 13: Matrix vector multiplication - ELL

For $i = 1, N_{row}$ **do**
  y[i] = 0.0
  For $j = 1, max\_col$ **do**
    $y[i] \mathrel{+}= values[i,j] \times x[columns[i,j]]$
  End for
End for

values	1	2	3
	4	0	0
	5	0	0
	6	7	0
	8	9	10
columnindex	0	1	4
	2	*	*
	1	*	*
	1	3	*
	0	2	4
columnpos	0	0	0
	0	*	*
	1	*	*
	2	0	*
	1	1	1

Figure 3.6: Sparse matrix stored in Sparse General Pattern with compressed rows format

3.2.1.7 Hybrid - HYB

The hybrid method consist in using two types of format to store a matrix. For instance $A = B + C$ where B and C are not stored with the same format.

Algorithm 14: Matrix vector multiplication - SGP row compression

For $i = 1, N_{row}$ **do**
  y[i] = 0.0
  For $j = 1, max\_col$ **do**
    $y[i] \mathrel{+}= values[i,j] \times x[columnindex[i,j]]$
  End for
End for

	1	2	4	7	3
	8	5	9	0	10
values	0	6	0	0	0
	0	0	1	3	0
	4	2	4	*	4
rowindex	*	3	*	*	*
	0	1	0	1	2
	0	0	1	*	2
rowpos	*	0	*	*	*

Figure 3.7: Sparse matrix stored in Sparse General Pattern with compressed columns format

Algorithm 15: Matrix vector multiplication - SGP column compression

For $i = 1, N_{row}$ **do**
  y[i] = 0.0
End for
For $j = 1, N_{col}$ **do**
  For $i = 1, max\_row$ **do**
    $y[rowpos[i,j]] \mathrel{+}= values[i,j] \times x[rowindex[i,j]]$
  End for
End for

3.2.2 Parallelization

CSR is the most commonly used sparse storage format in the sparse libraries. Usually, the matrix is divided by its rows [93] since the CSR format provides an easy way to separate the rows of a matrix in the case of distributed memory computing. In the case of shared memory, the matrix is shared by all the threads. To parallelize the sparse matrix vector, each computing unit performs a subset of the computations with a subpart of the input matrix and vector [94]. Then, each computing unit computes a subpart of the output vector. Depending on the matrix division, the output vectors will have to be gathered (matrix split by rows) or summed (matrix split columns) or both (matrix split in both dimensions). This post-processing on the output vector may not be necessary depending on how the output vector will be used afterwards. For instance, the output vector may not have to be gathered completely if the next operation performed on the computing only need the subpart computed by the sparse matrix vector product.

To summarize, the parallelization of the sparse matrix vector product consist on splitting the input matrix and input vector on the available distributed computing resources then perform the sparse matrix vector product on the local operations and finally, construct, if necessary, the full output vector according to the data distribution used. The exact computations performed locally [95] depends on the sparse storage format of the matrix.

3.2.3 Optimizations

As the sparse matrix vector product is a key component in graph kernels and sparse linear solvers which are often used in scientific applications, improving its efficiency could enhance the performances of those scientific applications. In this subsection, we will mention most of the common optimizations implemented to improve the performances of the sparse matrix vector product.

One of the main optimization used is to properly take advantage of the hardware. For instance, some processors support vectorization instructions which allow the processors to apply operations on a array of elements instead of only one element. With the random memory access necessary to access elements in sparse matrices, it is hardly possible to find contiguous data in the matrix to match the contiguous data of the vector and use vectorization on them. A solution to this issue is using block compressed sparse row storage (BCSR) [96] which consists in creating a sparse matrix of dense blocks and adding zero where there are missing values in order to increase locality [97] and allow the usage of vectorization. Moreover, AVX-512 instructions and a bit mask can be used to encode the zero [98] and thus avoiding to put zero in the blocks.

Another solution is to use cache and register blocking as well as multiplication by multiple vectors [99] [100] in order to reuse the data stored in the cache and the registers to the maximum.

Moreover, using storage format optimized to the type of matrix like the diagonal storage (DIA) [86] or the jagged diagonal storage (JAD) [82] can also improve the performance but highly depends on the shape of the matrix considered. Furthermore, the use of GPUs can be considered since they are widely used in the current architectures. In [101], CUDA [3] is used to implement the sparse matrix vector product and run on GPUs. On GPUs, the ELLPACK storage format seems to obtain very good performances as shown in [102] and is a popular storage format to use to perform sparse matrix operations on GPU. The ELLPACK format is also used to obtain better performances than the widely used CSR on Intel Xeon Phi Co-processor Knights Corner (KNC) [103].

Auto-tuning can also be used to find the best implementation and parameters to obtain the best performance on a given architecture [104] while trying different parameters for the optimizations available.

Finally, the sparse matrix can be reordered by applying permutation on it as to improve the performances of the sparse matrix vector product afterwards [96] [105].

3.2.4 Test Matrices

The test matrices we will use to compare the performances are the C-diagonal Q-perturbed matrices introduced in [106] [102]. These matrices consist in C values above the diagonal including the diagonal with a probability of Q to change the column position of the value. Figure 3.8 represent a 30×30 C-diagonal Q-perturbed sparse matrix Q=0.05 and C=4 where each black square is a value.

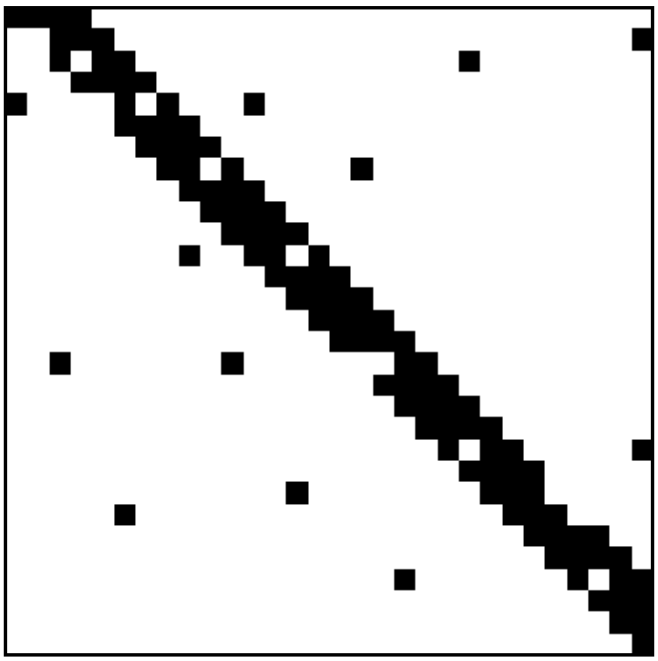


Figure 3.8: C-diagonal Q-perturbed sparse matrix with Q=0.05 and C=4

In this section, we introduced commonly used sparse matrix storage formats and the corresponding algorithm to perform the sparse matrix vector product as well as a way to parallelize the sparse matrix vector product and classical optimizations implemented in the sparse matrix vector product to improve its performances. We also introduced C-diagonal Q-perturbed sparse matrices which are matrices that can be used as input to test the performances of the sparse matrix vector product. In the last sections, we introduced the linear algebra methods we consider to implement to evaluate the task based programming models. However, scientific applications cannot be reduced to only linear algebra methods so we also choose to study the Kirchhoff seismic pre-stack depth migration which will be introduced in the next section.

3.3 Kirchhoff seismic pre-stack depth migration

This section will present the Kirchhoff migration and continue the previous work made in [107]. This report explain the migration and gives hint to implement new algorithms for the migration. This work take advantage from the discussions with Henri Calandra. This section will detail the steps for the Kirchhoff migration.

3.3.1 Overview

The seismic migration is a technique used to visualize the underground. Data are acquired at the surface during an acquisition campaign. Data are processed to geometrically re-locate seismic events either in space or in time. They are re-located to the location the event occurred in the subsurface rather than the location where it was recorded at the surface. Migration moves dipping reflectors to their true subsurface positions and collapses diffractions, resulting in a migrated image that typically has an increased spatial resolution and resolves areas of complex geology much better than non-migrated images. A form of migration is one of the standard data processing techniques for reflection-based geophysical methods (seismic reflection and ground-penetrating radar).

The Kirchhoff migration [108] [109] is a depth migration. It is applied to seismic data in depth (regular Cartesian) coordinates, which must be calculated from seismic data in time coordinates. This method does therefore require a velocity model, making it resource-intensive because building a seismic velocity model is a long and iterative process (Figure 3.9). The significant advantage to this migration method is that it can be successfully used in areas with lateral velocity variations, which tend to be the areas that are most interesting to petroleum geologists.

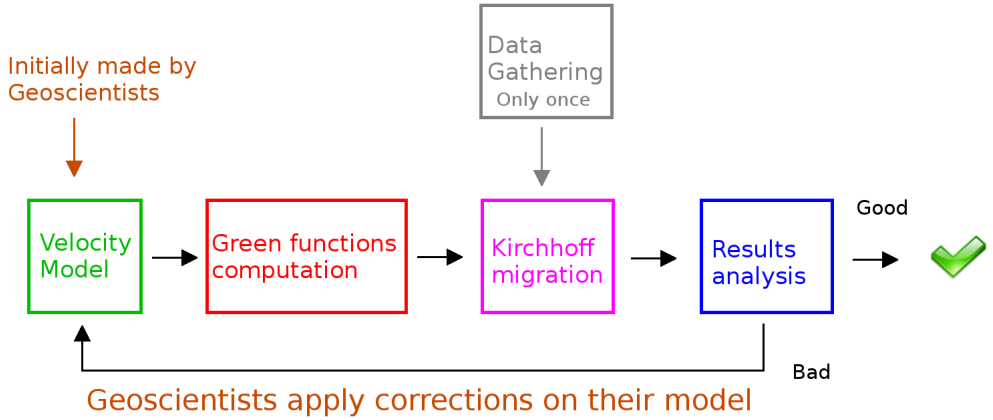


Figure 3.9: Building process of a seismic velocity model

3.3.2 Velocity model

The velocity model describe the propagation speed of the waves into the different layer of the underground. Geophysicists create a model and want to verify its correctness. Their goal is to find the best model that explain the data. They use the Kirchhoff migration that determine where are the limits between layers. This method processes data called traces. They are the output from a seismogram (receiver) at the surface

after the emission (source) of a wave into the ground. If the model is coherent with the image generated by the migration, the model is correct. Otherwise, the model is adjusted by geophysicists and the migration is re-used to obtain a new image. This process is repeated until the image and the model are coherent.

3.3.3 Data gathering

Data are produced during an acquisition campaign. A campaign contains several shots. A shot is a seismic impulsion launched through the ground. It is the source (see Figure 3.10). There are several receivers. They are seismograms placed on a 2D grid on the ground (see Figure 3.11). The source is moved on the 2D grid and produces shots. So a trace is the result for a given shot (source) and a given seismogram (receiver).

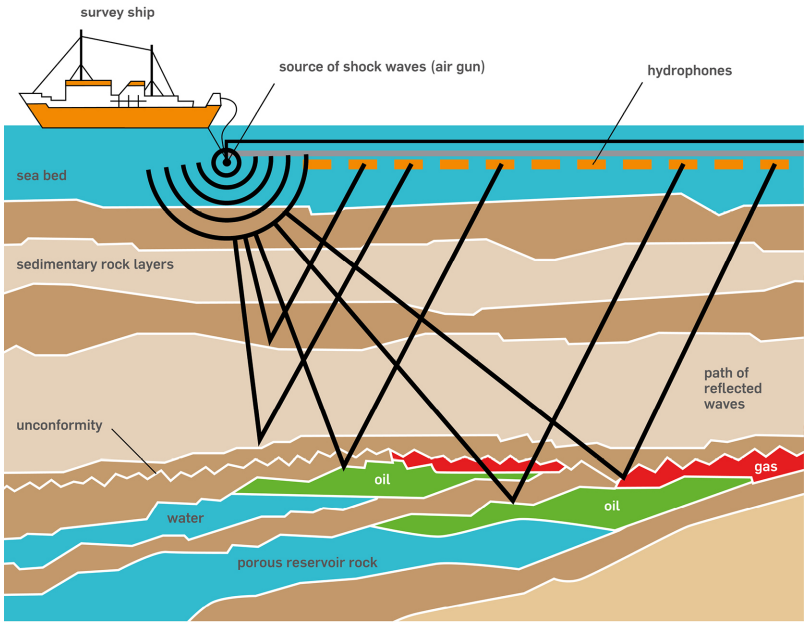


Figure 3.10: Shot during Data acquisition

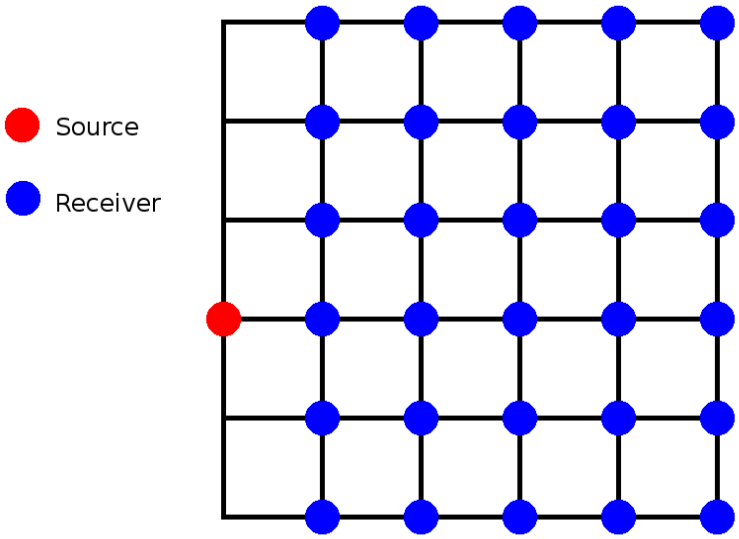


Figure 3.11: Shot and seismograms

The acquisition campaign gives a huge amount of traces since there is a lot of receiver and several shots. A trace contains the coordinates of the source and the receiver and the data from the seismogram.

3.3.4 Green functions

The Green functions represent the response and the behavior of the wave when the source is a Dirac impulse. They allow to solve the wave equation using an integral formula in function of the source of the wave. The behavior of the wave in the ground depends on the velocity model. Usually, Green functions are precomputed and stored on disk for a 2D grid on surface and a 3D grid underground. During the building of the model and the migration, Green functions are retrieved from disk.

In the Kirchhoff migration, the Green functions are used to estimate the travel time between 2 points depending on the velocity model. We estimate the travel time between a source S and a point P of the image (T_{SP}) then between P and a receiver R (T_{PR}). So we can deduce $T_{SPR} = T_{SP} + T_{PR}$ (see Figure 3.12).

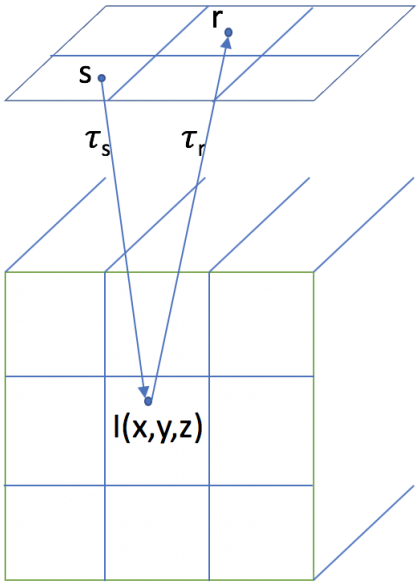


Figure 3.12: Wave travel between S and R through P from [107]

Green functions are precomputed for a 2D grid at the surface and for à 3D grid underground (see Figure 3.13). Moreover, a block of the 3D grid contains an image at a finer grain. Therefore, it is necessary to interpolate or extrapolate the Green functions for the points into the finer grid (see Figure 3.14).

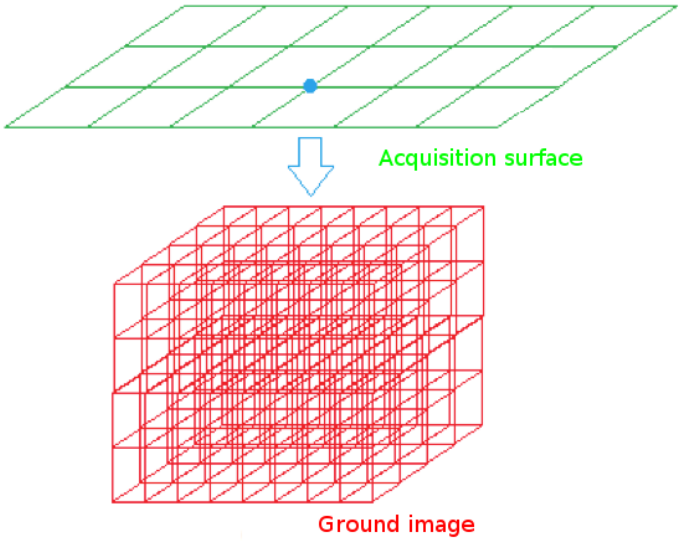


Figure 3.13: 2D and 3D grids from [107]

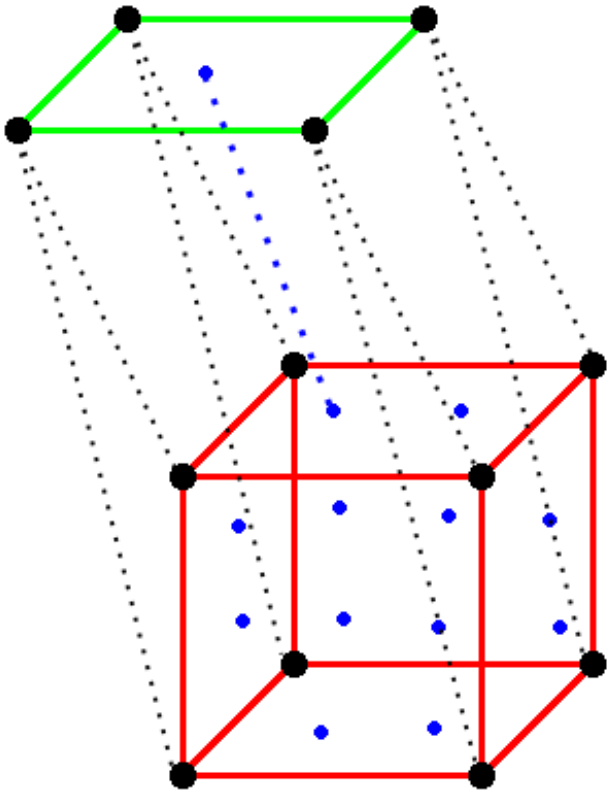


Figure 3.14: Coarse grid and points from the finer grid

3.3.5 Kirchhoff migration

The Kirchhoff migration produces a 3D image of the subsurface by retrieving the position of the reflection points in order to show the different layers of the ground. To do so, find the time a wave need to travel from a source to a point (x, y, z) in the image and travel back from this point to the receiver is necessary. The Green functions in the fine grid give this time. Then, the trace[1] (Figure 3.15) has a peak when the wave arrives to the receiver. This corresponds to the time it needs for the trace to travel from the source to the receiver into the ground. For a given source and receiver, if the time a wave need to travel trough the point (x, y, z) and the time to the peak of the trace match then (x, y, z) can be a candidate to a reflection point.

[1] http://www.iris.washington.edu/ds/nodes/dmc/manuals/irisfetchm/

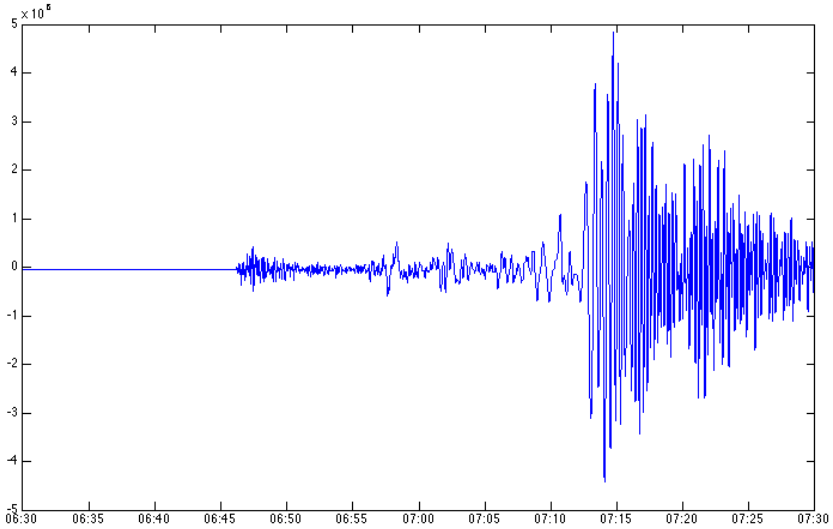


Figure 3.15: A trace from the IRIS-DMC repository

Moreover, we know the Green functions for a coarse grain of the image, so it is possible to know if the points in the sub-domain of the coarse grid will match the time from the trace. If the time matches, the points will be studied at a finer grain. Otherwise, the points of the part of the coarse grid will not be candidates to a reflection point and those points will be ignored for the selected trace.

At the finer grain level, when the block of coarse grid matches with the time of the trace, the Green functions are interpolated or extrapolated for each point in the block. This give the travel time $T(x, y, z)$ between the source and the receiver passing by the point (x, y, z). Then, only the points matching the time of the trace are kept. Those points are likely true reflection points. An arbitrary value (the aperture) $A_{s,r}(x, y, z)$ [110] that favours points with less awkward reflection angles, and positions that are more likely to match the true reflection point, can be calculated. This value depends on the coordinates of the receiver 'r', the coordinates of the source 's' and the point (x, y, z). The value $A_{s,r}(x, y, z)t_{s,r}(x, y, z)$ expresses the intensity of the contribution of the trace for the point (x, y, z).

The image at a point is generated by summing the contribution of all the traces that can be a true reflection point :

$$I(x, y, z) = \sum_{t_{s,r} \in T} A_{s,r}(x, y, z)t_{s,r}(\tau_s(x, y, z) + \tau_r(x, y, z)) \tag{3.3}$$

$I(x, y, z)$ is the point (x, y, z) of the image. $t_{s,r}$ is the trace which has s as source and r as receiver. T is the ensemble of traces. $A_{s,r}$ is the amplitude associated to s and r. $\tau_j(x, y, z)$ is the time need for a wave to travel from j (a source or a receiver) to (x, y, z).

In the existing application, the Green functions (the time needed by the sound wave to go from the source or the receiver to the point inside the grid, the values of τ) are pre-computed from the velocity model which is evaluated and imported from the file system when the application is using them.

Moreover, two granularities of grids are used. The values of A and τ are computed for the coarse grid and they are interpolated from the computed values for the fine grid.

3.3.6 Analysis of the output by the geophysicists

The resulting image is analyzed by geophysicists. If they think that the velocity model is coherent with the image, the model can be exploited. Otherwise, the velocity model is modified and the process of the Kirchhoff migration is reused until the geophysicists judge that the velocity model corresponds to the data.

We presented the three kinds of algorithms we will consider in this dissertation : the solution of linear system with the LU factorization, the Gaussian elimination and the Gauss-Jordan elimination, the sparse matrix vector product with different storage format as well as the Kirchhoff seismic pre-stack depth migration. They will be used as examples and implemented with some of the task based programming model introduced in Chapter 2. The following chapter will introduce in details the programming models that have been selected to implement the methods discussed in this chapter.

Chapter 4

Languages

In Chapter 2, we introduced most of the current task based programming models. In this chapter, we select a few of them in order to use them in our experiments and implement parallel and distributed applications based on the methods presented in Chapter 3. In this chapter, we study several programming models in three classes of programming models in greater details. The first class is task based programming models in which tasks are fine grain and running inside only one thread or process. The second one contains task based programming models that use parallel and distributed tasks that run on multiple processes. The last class is message passing libraries that are the current commonly used programming model to implement parallel and distributed applications.

4.1 Message Passing Interface

The Message Passing Interface (MPI) [1] is a standardized norm designed to work on a wide variety of parallel computers. It defines the syntax of the core library routines to write message-passing applications. The application uses several processes to make computation at the same time on different cores and uses MPI to send data from one process to another one. MPI has several implementations (OpenMPI, MPICH2, IntelMPI, ...) which consist of a set of routines that can be called from C, C++ and Fortran.

The MPI library interface includes point-to-point send/receive operations, aggregate functions involving communication between all processes, synchronous nodes (barrier operations), one-way communication, dynamic process management and I/O operations. MPI provides synchronous and asynchronous routines as well as blocking and non-blocking operations. Collective routines involve communications between all processes in a group, for instance MPI_Bcast (broadcast that sends an array to all of the other processes).

MPI can be used with shared memory programming models like OpenMP or with libraries to send data and make computations on CPUs like CUDA.

4.2 Task Based Programming Models

4.2.1 PaRSEC

PaRSEC [6] [48] (Parallel Runtime Scheduling and Execution Controller) is an engine for scheduling tasks on distributed hybrid environments.

It offers a flexible API to develop domain specific languages. It aims to shift the focus of developers from repetitive architectural details toward meaningful algorithmic improvements. Two domain specific languages are supported by Parsec, the Parameterized Task Graph [49] (PTG) and Dynamic Task Discovery [50] (DTD)

In PTG [49], users would use a parameterized expression of task dependencies combined with PaRSEC which would implicitly infer the communication between nodes and the accelerators. The users has to provide a description of the data flow of their application and the tasks which are applied on the data. They also has to provide the tasks that are the source of the data and the tasks that are the destination. So, this is a compressed algebraic representation of the task graph. Then, it is transformed into C code using a pre-compiler. Users need to understand and provide all the data flow of their algorithm to use this model.

DTD [50] is a task-based programming paradigm which provides an alternative way to express task dependency in PaRSEC that achieves a similar purpose to PTG. Contrary to PTG where users had to express tasks in a parameterized manner, DTD allows them to write sequential constructs (ifs, loops, etc...) to insert tasks in PaRSEC. The tasks are given to the runtime with the data they will use and their mode of usage. Then, the runtime will compute dependencies out of data pointers used by the tasks. In a distributed system, the data movements among the nodes are completely implicit.

Listing 4.1: Task in PaRSEC

```
1  PMM(i, k)
2
3  k = 0 .. bm - 2
4  i = k + 1 .. bm - 1
5
6  : dcA(i, k)
7
8  RW      A <- (k == 0) ? dcA(i, k) : Aij PMM_D(i, k, k)
9            -> Aik PMM_D(i, k + 1 .. bm - 1, k + 1)
10           -> dcA(i, k)
11 READ    Inv <- Inv Inv(k)
12
13 BODY
14 {
15       double *Ap = (double *)A;
16       double *Bp = (double *)Inv;
17       pmm_core(Ap, Bp, dcA->super.mb);
18 }
19 END
```

4.2.1.1 PaRSEC Task definition

Listing 4.1 shows how to define a task in PaRSEC. Line 1 defines the name of the task and the integer parameters in input. This task is called *PMM*. Line 3 and 4 give the range of those parameters. In line 6, *dcA*

is a data descriptor and is used to align the task on the resources from the descriptor. Line 8 to 11 describe the dependencies of the task. Line 13 to 19 are the body of the task. They contain the computations on the data. This task calls the *pmm_core* to process the data stored in the *Ap* and *Bp* pointers.

4.2.1.2 PaRSEC Dependency definition

Line 8 to 11 from Listing 4.1 shows how to describe the dependencies of a task in PaRSEC. The *PMM* task uses two pieces of data *A* and *Inv*. They are temporary pointers that hold the data that will be used in the task.

A is in read-write mode (RW). It means *PMM* expects a pointer in input that will be called A in the body of the task. That pointer can be modified and will be transfered into another task. The data in *A* come from the part (i, k) in the data descriptor *dcA* when $k = 0$ and from the data referenced as A_{ij} in the task *PMM_D*. When the task is finished, the data in *A* will be sent as input of A_{ik} from the task *PMM_D*. Indeed, Listing 4.2 shows the corresponding dependencies definition. We find *A* from *PMM* as input for A_{ik} from *PMM_D* and A_{ij} as input for *A* from *PMM*.

Inv is in read mode (READ). It only uses the piece of data named *Inv* from the *Inv* task.

There is also a write mode (WRITE) that output a new piece of data.

Control dependencies can also be added through the *CTL* keyword. It allows the user to create control flow data which can tell the runtime if a task has to be done before another one.

Listing 4.2: Expression of PMM_D dependencies

```
1 READ    Aik <- A PMM(i, k - 1)
2 RW      Aij <- (k == 1) ? dcA(i, j) : Aij PMM_D(i, j, k - 1)
3             -> (i > k && j == k) ? A PMM(i, j)
4             -> (i > k && j > k) ? Aij PMM_D(i, j, k + 1)
```

This way of expressing the dependencies is error-prone. Indeed, the user can easily be lost in the definition of the dependencies since he has to replicate them in two tasks (for instance, the input of a task is the output from another one). Moreover, it can become very complicated when there is a lot of different tasks and condition on the parameters.

4.2.1.3 PaRSEC Data definition

dcA is a *two_dim_block_cyclic_t**. This is a matrix data type provided by PaRSEC. It helps to store two dimensional arrays of matrices in memory. We use it to store sub-matrices in our block-based LU factorization. It extends the initial data type *parsec_ddesc_t* as for each data type used by PaRSEC. Listing 4.3 shows how to extend *parsec_ddesc_t* to create a data type called *my_datatype_t*.

This data distribution descriptor stores only one data block per MPI [1] process in the field *ptr*, of size *size* bytes. However, the user's memory is not manipulated directly by PaRSEC. It uses *parsec_data_t** as well as *parsec_data_copy_t**. *parsec_data_t* is an abstract representation of user data. *parsec_ddesc_t* class instantiates at most one per MPI process per user data element. *parsec_data_copy_t* is a representation of a copy of the user data and multiple instances can exist at the same time. Distributed Data Descriptors usually

manipulate parsec_data_t objects, and provide parsec_data_copy_t objects, using PaRSEC functions to instantiate them.

Listing 4.3: PaRSEC custom datatype definition

```
1  parsec_ddesc_t *create_and_distribute_data(int rank, int world, int size,
2                                             int seg) {
3      my_datatype_t *m = (my_datatype_t*)calloc(1, sizeof(my_datatype_t));
4      parsec_ddesc_t *d = &(m->super);
5
6      d->myrank = rank;
7      d->nodes  = world;
8      d->rank_of      = rank_of;
9      d->rank_of_key = rank_of_key;
10     d->data_of      = data_of;
11     d->data_of_key = data_of_key;
12     d->vpid_of      = vpid_of;
13     d->vpid_of_key = vpid_of_key;
14     d->data_key     = data_key;
15
16     m->size = size;
17     m->data = NULL;
18     m->ptr  = (uint8_t*)calloc(size, 1);
19     return d;
20 }
```

PaRSEC uses all fields from the *parsec_ddesc_t* object. The user has to fill them in order to implement a new data type. *myrank* is the rank of the local MPI process. *nodes* is the number of nodes in the MPI Communicator. Then, the other obligatory fields are function pointers. *rank_of* and *rank_of_key* return the rank of a given data element, based on its multidimensional index or its flat key. *vpid_of* and *vpid_of_key* return the identifier of the Virtual Process that "owns" a data identified by its multidimensional index or its flat key. *data_of* and *data_of_key* return the parsec_data_t* associated with the user data at this multidimensional index position, or for this flat key.

Line 3 to 9 in Listing 4.4 shows how to initialize *dcA* with a built-in function dedicated to that and then how to allocated the memory in each MPI process with *parsec_data_allocate* function.

With the definition of these functions, PaRSEC is able to manage efficiently the data from the user.

4.2.1.4 PaRSEC Graph execution

PaRSEC compiler generates code to launch the tasks. In this case, it is the *parsec_lu_new* function as shown in Listing 4.4. It takes *dcA* the data descriptor and *bm* which is the number of tiles in each dimension as parameter. The preprocessor finds them in the source file and put them as input parameters of the function to launch the tasks.

Listing 4.4: PaRSEC tasks launching

```
parsec = parsec_init(cores, &argc, &argv);

two_dim_block_cyclic_init( &dcA, matrix_RealDouble, matrix_Tile,
                           world, rank, mb, mb, lm, lm, 0, 0, lm, lm,
                           1, 1, rows );
dcA.mat = parsec_data_allocate((size_t)dcA.super.nb_local_tiles *
                               (size_t)dcA.super.bsiz *
                               (size_t)parsec_datadist_getsizeoftype(
                                           dcA.super.mtype));

srand(rank + 1);
double * matp = (double *)dcA.mat;
for(int i = 0; i < dcA.super.nb_local_tiles * dcA.super.bsiz; i++) {
      matp[i] = (double)((rand() % 2000) - 1000) / 100;
}

tp = parsec_lu_new(&dcA, bm);
```

4.2.1.5 PaRSEC Granularity

PaRSEC assigns computation threads to the cores, overlaps communications and computations. Each task is run into a thread. Furthermore, PaRSEC tasks are fine grain.

4.2.1.6 PaRSEC Tasks re-usability

Tasks cannot be reused as they are since their dependencies, parameter ranges and alignment are stored in the task definition. They have to be rewritten for each new application. However, core computations can be stored into a routine which can be reused into other tasks.

4.2.1.7 PaRSEC Scheduling

The design of the PARSEC runtime system focuses on scalability. PaRSEC tries to limit task knowledge to processes that are responsible for their execution. The scheduling decisions are entirely distributed. Algorithmic correctness should not require control synchronization, and synchronous collective synchronization should be avoided.

4.2.1.8 PaRSEC Type of graph

PaRSEC is mainly based on the expression of data flow dependencies but the user can also give additional control flow dependencies if needed.

4.2.1.9 PaRSEC GPU support

In PTG, the tasks are implemented for CPU by default. However, PaRSEC is able to move data to the GPU and to call tasks on it. The code to run on GPU is specified in the *BODY* as for a CPU task and adding the option *[type=CUDA]* to after the keyword *BODY*. A function which operates on the data on the GPU can be called in the body of the task.

4.2.2 Legion

Legion [7] is a data-centric parallel programming model. It aims to make the programming system aware of the structure of the data in the program. Legion provides explicit declaration of data properties (organization, partitioning, privileges, and coherence) and their implementation via the logical regions. They are the fundamental abstraction used to describe data in Legion applications. Logical regions can be partitioned into sub-regions and data structures can be encoded in logical regions to express locality describing data independence.

A Legion program executes as a tree of tasks spawning sub-tasks recursively. Each task specifies the logical region they will access. With the understanding of the data and their use, Legion can extract parallelism and find the data movement related to the specified data properties. Legion also provides a mapping interface to control the mapping of the tasks and the data on the processors during the execution of the application.

Legion installation is a bit different than other libraries. It is not installed as a compiled library. The user has to compile it each time he compiles an application. A static library is generated while executing the Makefile provided to compile an application and is included in the executable. Therefore, compiling an application is taking some time.

Listing 4.5: Legion task

```
1  void stencil_task(const Task *task,
2                    const std::vector<PhysicalRegion> &regions,
3                    Context ctx, Runtime *runtime) {
4    assert(regions.size() == 2);
5    assert(task->regions.size() == 2);
6    assert(task->regions[0].privilege_fields.size() == 1);
7    assert(task->regions[1].privilege_fields.size() == 1);
8    assert(task->arglen == sizeof(int));
9    const int max_elements = *((const int*)task->args);
10   const int point = task->index_point.point_data[0];
11
12   FieldID r_fid = *(task->regions[0].privilege_fields.begin());
13   FieldID w_fid = *(task->regions[1].privilege_fields.begin());
14
15   const FieldAccessor<READ_ONLY,double,1> read_acc(regions[0], r_fid);
16   const FieldAccessor<WRITE_DISCARD,double,1> write_acc(regions[1], w_fid);
17
```

```
18    Rect<1> rect = runtime->get_index_space_domain(ctx,
19                     task->regions[1].region.get_index_space());
20      for (PointInRectIterator<1> pir(rect); pir(); pir++)
21      {
22        double l2, l1, r1, r2;
23        double result = (-l2 + 8.0*l1 - 8.0*r1 + r2) / 12.0;
24        write_acc[*pir] = result;
25      }
26  }
27
28  int main(int argc, char **argv) {
29    Runtime::set_top_level_task_id(TOP_LEVEL_TASK_ID);
30    {
31      TaskVariantRegistrar registrar(STENCIL_TASK_ID, "stencil");
32      registrar.add_constraint(ProcessorConstraint(Processor::LOC_PROC));
33      registrar.set_leaf();
34      Runtime::preregister_task_variant<stencil_task>(registrar, "stencil");
35    }
36    return Runtime::start(argc, argv);
37  }
```

4.2.2.1 Legion Task definition

Legion tasks are implemented as a tree of tasks. The main function calls Legion runtime to register the top level task which will spawn other tasks. Those task will also spawn other tasks. Line 36 from Listing 4.5 shows how to call Legion runtime to launch the tasks. Tasks have to be registered into Legion using a *Registrar*. Line 31 to 34 shows how to do that and bind the task to use CPUs.

Legion tasks need to have generic parameters so that the runtime can call them and pass the data and pieces of information they need. Line 1 to 3 shows the prototype of a function which is used as a task in Legion. This task expect two regions which store data that will be accessed in this task and a integer as argument since the task asserts that the size of the region is two (line 4 and 5) and that the length of the arguments is of the size of an integer (line 8).

Data regions are accessed through *FieldId* and *FieldAccessor*. Line 12 and 15 shows how to get accessors to a region in read mode. It can be used with an *Iterator* on an index space (as defined on line 18). This iterator is used as index to the accessors so that pieces of data can be retrieved or set as shown on line 24.

Listing 4.6: Legion data

```
1  void top_level_task(const Task *task,
2                      const std::vector<PhysicalRegion> &regions,
3                      Context ctx, Runtime *runtime) {
4    int num_elements = 1024;
```

```
5    int num_subregions = 4;
6
7    Rect<1> elem_rect(0,num_elements-1);
8    IndexSpaceT<1> is = runtime->create_index_space(ctx, elem_rect);
9    FieldSpace fs = runtime->create_field_space(ctx);
10   {
11     FieldAllocator allocator =
12       runtime->create_field_allocator(ctx, fs);
13     allocator.allocate_field(sizeof(double),FID_VAL);
14     allocator.allocate_field(sizeof(double),FID_DERIV);
15   }
16   LogicalRegion stencil_lr = runtime->create_logical_region(ctx, is, fs);
17
18   Rect<1> color_bounds(0,num_subregions-1);
19   IndexSpaceT<1> color_is = runtime->create_index_space(ctx, color_bounds);
20
21   IndexPartition disjoint_ip =
22     runtime->create_equal_partition(ctx, is, color_is);
23   const int block_size =(num_elements + num_subregions - 1) /num_subregions;
24   Transform<1,1> transform;
25   transform[0][0] = block_size;
26   Rect<1> extent(-2, block_size + 1);
27   IndexPartition ghost_ip =
28     runtime->create_partition_by_restriction(ctx, is, color_is,
29                         transform, extent);
30
31   LogicalPartition disjoint_lp =
32     runtime->get_logical_partition(ctx, stencil_lr, disjoint_ip);
33   LogicalPartition ghost_lp =
34     runtime->get_logical_partition(ctx, stencil_lr, ghost_ip);
35 }
```

4.2.2.2 Legion Dependency definition

Dependencies are implicitly defined with the access mode of the regions and the tree of tasks. Indeed, a task can launch another tasks when it is finished.

4.2.2.3 Legion Data definition

Data are defined through regions of any given type. To define a region and divide it in sub-regions then run tasks on the sub-regions, an index space and a field space for the region have to be defined. The index

space keeps the coordinates of the values in the multi-dimensional array represented by a region (line 8 from Listing 4.6). The field space defines which type of data there will be in the region (line 9 to 15 from Listing 4.6). It is represented by a number of byte for each coordinate of the field space. Furthermore, they can be combined into a *LogicalRegion* (line 16 from Listing 4.6). It will allocate the region of the designed type in memory. Custom mappers can be used to optimize the data allocation.

This *LogicalRegion* is split into two partitions; a regular block partition and a partition with ghost borders (duplicated values between neighbor sub-regions). It is done by defining another index space which will record the coordinates of the sub-regions (line 19 from Listing 4.6). Then, the index partitions can be created from the data index space and the sub-regions index space (line 21 to 29 from Listing 4.6). Finally, the *LogicalPartition* which contains the split data can be created from the initial *LogicalRegion* and the newly defined sub-regions index partitions (line 31 to 34 from Listing 4.6).

4.2.2.4 Legion Graph execution

The tasks are registered into the runtime system then the user can use the runtime to start executing the tasks from the tree of tasks.

Listing 4.7: Legion launch task

```
1  ArgumentMap arg_map;
2
3  IndexLauncher stencil_launcher(STENCIL_TASK_ID, color_is,
4       TaskArgument(&num_elements, sizeof(num_elements)), arg_map);
5  stencil_launcher.add_region_requirement(
6      RegionRequirement(ghost_lp, 0/*projection ID*/,
7                        READ_ONLY, EXCLUSIVE, stencil_lr));
8  stencil_launcher.add_field(0, FID_VAL);
9  stencil_launcher.add_region_requirement(
10     RegionRequirement(disjoint_lp, 0/*projection ID*/,
11                       READ_WRITE, EXCLUSIVE, stencil_lr));
12 stencil_launcher.add_field(1, FID_DERIV);
13 runtime->execute_index_space(ctx, stencil_launcher);
14
15 runtime->destroy_logical_region(ctx, stencil_lr);
16 runtime->destroy_field_space(ctx, fs);
17 runtime->destroy_index_space(ctx, is);
```

There is several way of launching tasks in Legion. Tasks can be launched on a *LogicalRegion* and the task will process all the data of the region. They can also be launched on all the sub-regions of a *LogicalPartition* at the same time then the scheduler manage the resources on which launch the tasks. Listing 4.7 shows how to process sub-regions in this way. An *IndexLauncher* need to be created to do so. It needs the identifier of the task which is defined when registering the task, the sub-regions index space on which launch the tasks and arguments for the tasks. The *LogicalPartition* have to be given to the index launcher so that the tasks

can access the data. This is performed by using the *add_region_requirement* and the *add_field* methods. These parameters are passed to the task via the runtime as we saw in Listing 4.5. Then we can tell the runtime to execute the tasks with the *execute_index_space* method (Line 13 in Listing 4.7).

4.2.2.5 Legion Granularity

Tasks are run as thread by the runtime so tasks are fine-grain.

4.2.2.6 Legion Tasks re-usability

Tasks can be registered and launched in another application if it provides the regions and parameters expected by the task.

4.2.2.7 Legion Scheduling

Legion uses a software out-of-order processor (SOOP) to schedule tasks. It dynamically schedules a stream of tasks. It is constrained by region dependences. It is also pipelined, distributed, and extracts nested parallelism from subtasks. Legion uses a *deferred execution model* which separates the issuing of the operations from when operations are executed. An issued operation waits for other operations on which it is dependent to complete before executing without blocking the scheduler.

4.2.2.8 Legion Type of graph

Legion tasks are defined as a tree of tasks.

4.2.2.9 Legion GPU support

Legion supports data migrations to GPUs. Data are accessed through Legion data structure. Tasks are implemented in C++ and can be launched on GPU. The user has to create tasks (a C++ class) that extends a Legion *Launcher* (for instance, *IndexLauncher*) and to implement a *cpu_base_impl* function to run the task on CPU and a *gpu_base_impl* to run on GPU. When the user register the task, he has to specify that the task will run on GPU.

4.2.3 Regent

Regent [8] is a programming model which simplifies Legion. Regent compiler translates Regent programs into efficient implementations for Legion. It results in programs that are written with fewer lines of codes and at a higher level.

4.2.3.1 Regent Task definition

A Regent task is similar to a function. It is defined with the *task* keyword until the keyword *end*. It has parameters after the name of the task (*pmm_d* in Listing 4.8). Regent tasks also need *coherence modes* to specify how the data will be accessed. They can be in *reads*, *writes* or both modes. On line 4 from Listing 4.8, there is an example of read access mode for B as well as read and write access mode for A. Line 5 to 12 are the computations in the task.

Listing 4.8: Regent task

```
1  task pmm_d(A : region(ispace(int2d), double),
2             B : region(ispace(int2d), double),
3             C : region(ispace(int2d), double), n : int)
4  where reads(B), reads(C), reads writes(A) do
5    for i = 0, n do
6      for j = 0, n do
7        for k = 0, n do
8          A[int2d({i, j}) + A.bounds.lo] = A[int2d({i, j}) + A.bounds.lo]
9               -B[int2d({i, k}) + B.bounds.lo] * C[int2d({k, j}) + C.bounds.lo]
10       end
11     end
12   end
13 end
```

4.2.3.2 Regent Dependency definition

Dependencies between tasks are automatically inferred by Regent compiler from input modes. Line 35 to 43 from Listing 4.9 show examples on how to call a task in Regent. This limits the parallelism that can be detected and create additional dependencies when the system is not sure that the tasks are independent.

Listing 4.9: Regent application

```
1  import "regent"
2  local c = regentlib.c
3  local std = terralib.includec("stdlib.h")
4  require("bla_common")
5
6  task make_partition_mat(points : region(ispace(int2d), double),
7                          tiles : ispace(int2d), np : int32)
8    var coloring = c.legion_domain_point_coloring_create()
9    for i in tiles do
10     var lo = int2d { x = i.x * np, y = i.y * np }
11     var hi = int2d { x = (i.x + 1) * np - 1, y = (i.y + 1) * np - 1 }
12     var rect = rect2d { lo = lo, hi = hi }
13     c.legion_domain_point_coloring_color_domain(coloring, i, rect)
14   end
15   var p = partition(disjoint, points, coloring, tiles)
16   c.legion_domain_point_coloring_destroy(coloring)
17   return p
18 end
19
```

```
20 task main()
21   var nt : int32 = 4
22   var np : int32 = 4
23
24   var gridA = ispace(int2d, { x = nt * np, y = nt * np })
25   var tilesA = ispace(int2d, { x = nt, y = nt })
26   var A = region(gridA, double)
27   var Ap = make_partition_mat(A, tilesA, np)
28
29   var gridA_inv = ispace(int2d, { x = nt * np, y = np })
30   var tilesA_inv = ispace(int2d, { x = nt, y = 1 })
31   var A_inv = region(gridA_inv, double)
32   var A_inv_p = make_partition_mat(A_inv, tilesA_inv, np)
33
34   init_mat(A)
35   for k = 0, nt-1 do
36     inversion(Ap[int2d({k, k})], A_inv_p[int2d({k, 0})], np)
37     for i = k + 1, nt do
38       pmm(Ap[int2d({i, k})], A_inv_p[int2d({k, 0})], np)
39       for j = k + 1, nt do
40         pmm_d(Ap[int2d({i, j})], Ap[int2d({i, k})], Ap[int2d({k, j})], np)
41       end
42     end
43   end
44 end
45
46 regentlib.start(main)
```

4.2.3.3 Regent Data definition

Data types are stored in regions. The regions can be partitioned and tasks can be executed on subregions. Custom types can be defined and used as base in regions. Regions are multi-dimensional arrays.

Line 24 to 27 from Listing 4.9 define a partitioned region A. First, the index space of the region itself is created. Then, the index space of the partition is created. It is a 2D array of size $nt * nt$ where nt is the number of sub-region in each dimension. The region A can be created out of the index space of the region in line 26. It creates a 2D region of doubles. Finally, A can be divided in sub-regions. To do that, the function *make_partition_mat* which is defined in line 6 is used. This function puts each point in A into a sub-region. It is done by using coloration. Each point is given a color and goes into one of the sub-region. This function separates the initial region into regular blocks.

The data in an index space are accessed using an *int2d* data structure. It represents the coordinates of the point. Each sub-region keeps the index space of the initial region. Therefore, the lower bound is added to the

int2d while accessing the values in the task (line 8 in Listing 4.8). *int2d* are also used to access sub-regions as in line 40 of Listing 4.9.

4.2.3.4 Regent Graph execution

The tasks are launched through the main function. Regent compiler converts code into Legion code so the tasks are executed using Legion execution model. Main function is registered into Regent using *start* function (line 46 from 4.9). It is the starting point of the application.

4.2.3.5 Regent Granularity

Tasks are run as Legion tasks. They are run into a thread. Furthermore, they are fine-grain tasks.

4.2.3.6 Regent Tasks re-usability

Tasks can be reused. They can be stored in another regent file which can be included in another application. Then, tasks can be called as they are from the other application.

4.2.3.7 Regent Scheduling

Legion scheduling policy is used here.

4.2.3.8 Regent Type of graph

As Regent code is converted in to Legion code, Regent uses Legion tree of tasks that are spawn recursively starting from the main task.

4.2.3.9 Regent GPU support

Regent supports generation of CUDA code for Legion. A task can be converted to GPU code instead of CPU code by annotating the task with *__demand(__cuda)*. Regent will then generate CUDA code for Legion that will be executed by Legion runtime.

4.2.4 TensorFlow

TensorFlow [9] (https://www.tensorflow.org) is "open source software library for high performance numerical computation". A TensorFlow program consist of a set of Operations arranged into a graph and run by a Session. The Tensors contain the data and are used by the Operations. A Tensor is a set of primitive values shaped into a multidimensional array. An Operation runs computations on the provided Tensors. TensorFlow deduces the graph from the used Operations. It provides a lot of built-in Operations in its Low Level API. It also provides a higher-level API which can be used to implement Machine Learning and AI algorithms. Most of the algorithms are related to model training and layers of neural networks. Thus, the high-level API cannot be used to implement non-AI based applications.

The Python package of TensorFlow is easy to install through pip. For the other languages (C, Java, Go, ...), binaries can be downloaded and have to be linked in the program. There is also the possibility to build it from sources.

Listing 4.10: TensorFlow block-based LU factorization

```
import tensorflow as tf
import sys

N = 4
matsize = 10
A = {}
inv = {}
sess = tf.Session()

for i in range(N):
    for j in range(N):
        A[i,j] = tf.Variable(tf.random_uniform([matsize, matsize],
                                               seed = i * N + j))

init = tf.global_variables_initializer()
sess.run(init)

for k in range(N):
    inv[k] = tf.matrix_inverse(A[k, k])
    for i in range(k + 1, N):
        A[i,k] = tf.matmul(A[i,k], inv[k])
        for j in range(k + 1, N):
            A[i,j] = A[i,j] - tf.matmul(A[i,k], A[k,j])

for i in range(N):
    for j in range(N):
       sess.run(A[i,j])
```

4.2.4.1 TensorFlow Task definition

In TensorFlow, Operations are tasks which run on a subset of the problem. New Operations can be added into TensorFlow. They have to be implemented in C++ and inserted into the source code of TensorFlow. The user has to compile it from sources in order to have the new Operations available. There is several steps to follow : register the Operation, implement the Operation into a kernel, make it compatible with the devices (CPU and GPU) and re-build TensorFlow with the new Operation.

4.2.4.2 TensorFlow Dependency definition

Dependencies between the Operations are computed by TensorFlow system. They are inferred from the data accessed and the order of the calls. This limits the parallelism that can be detected and create additional

dependencies when the system is not sure that the Operations are independent.

4.2.4.3 TensorFlow Data definition

In TensorFlow, data are managed through Tensors. It is a multi-dimensional array that store several pieces of data of the same type. Those data can be integers, doubles, complexes or strings. Other data structure are not supported. However, other data structures can be serialized into strings and stores into Tensors as strings.

TensorFlow also provides Variables. They are used to represent shared and persistent data. They can be created with Tensors as base.

TensorFlow Python binding offers the possibility to import data from a *numpy* array and create a Tensor out of it.

4.2.4.4 TensorFlow Graph execution

The graph is built on the run when TensorFlow has inferred the dependencies between the Operations. It is executed through a Session. The user has to create a Session and use it to run the computation on the different Tensors to launch the Operations and obtain the output of the computations.

Listing 4.10 shows on line 8 how to create a Session. It also show on line 15 and 16 how to initialize Variables through the Session. line 25 to 27 shows how to use the Session to perform the Operations on the Tensors.

4.2.4.5 TensorFlow Granularity

TensorFlow Operations run on a CPU or on a GPU depending on the resources available and the availability of the implementation of the kernel for the type of device targeted.

4.2.4.6 TensorFlow Tasks re-usability

Operations can be reused on different architectures if their implementation is available for this architecture.

4.2.4.7 TensorFlow Type of graph

TensorFlow uses a data flow graph inferred from the Operations on Tensors used in the application.

4.2.4.8 TensorFlow GPU support

TensorFlow is designed to support GPU. Indeed, TensorFlow Operations can be implemented to run on GPU and/or CPU.

4.2.5 HPX

High Performance ParalleX (HPX) [10] [111] is a C++ Standard Library for Concurrency and Parallelism. It implements the facilities defined by the C++ Standard and functionalities proposed as part of the ongoing C++ standardization process. It also extends the C++ Standard APIs to the distributed case. The goal of

HPX is to create a high quality, freely available, open source implementation of a new programming model for conventional systems.

HPX API implements the interfaces defined by the C++11/14/17/20 ISO standard and respects the programming guidelines used by the Boost collection of C++ libraries. It aims to improve the scalability of current applications. It also tries to expose new levels of parallelism which are necessary to take advantage of the future systems.

HPX is an open-source implementation of the ParalleX execution model. This model focuses on overcoming the four main barriers to scalability (Starvation, Latencies, Overhead, Waiting for contention resolution).

Listing 4.11: HPX task

```
1  struct stepper
2  {
3      typedef std::vector<partition> space;
4
5      //C = C - A * B
6      static partition_data pmm_d_core(
7          partition_data const& A, partition_data const& B,
8          partition_data const& C) {}
9
10     static partition pmm_d_part(
11         partition const& A_p, partition const& B_p, partition const& C_p)
12     {
13         using hpx::dataflow;
14         using hpx::util::unwrapping;
15
16         hpx::shared_future<partition_data> A_data = A_p.get_data();
17         hpx::shared_future<partition_data> B_data = B_p.get_data();
18         hpx::shared_future<partition_data> C_data = C_p.get_data();
19         return dataflow(hpx::launch::async,
20             unwrapping([C_p](partition_data const& A, partition_data const& B,
21                         partition_data const& C) -> partition {
22                 partition_data r = stepper::pmm_d_core(A, B, C);
23                 return partition(C_p.get_id(), r);
24             }),
25             A_data, B_data, C_data);
26     }
27
28     space do_lu(std::size_t T, std::size_t N);
29 };
30
31 HPX_PLAIN_ACTION(stepper::pmm_d_part, pmm_d_part_action);
```

4.2.5.1 HPX Task definition

In distributed HPX applications, tasks are defined through *HPX_PLAIN_ACTION* macro. Actions can be launch on a local node or a remote node by the scheduler depending on the available resources and the location of the data. Actions are used to retrieve the data and apply computations on them. Those data are managed by a data server (*partition_server* in Listing 4.13 which will be discussed later) which is used to keep track of the data and is used by the runtime to move data around.

partition_data is a class used to store data. It uses a more efficient and specialized implementation than the standard *Vector*. It stores the data on which tasks run computations. The *partition_server* returns futures on calling the *get_data* function. The futures are placeholders that are used to ask for data that will be delivered when they are available. The *dataflow* (on line 19 in Listing 4.11) will wait for the data from the future to be available then execute the function given to it asynchronously (*pmm_d_core* on line 22) and pass the data to it.

The *pmm_d_core* function will make the actual computations on the data.

4.2.5.2 HPX Dependency definition

Dependencies on tasks are expressed with futures on data used in the tasks and actions. The actions previously defined helps to wait for the data from the futures to be available. The actions can now be called with actual data in order to have a complete application. Listing 4.12 shows an example of using actions to launch tasks on data held by futures.

First, *spaces* (vectors of *partition_data*) which contains the data are created then initialized (on line 8 to 17 in Listing 4.12). *locidx* is an in-lined function that compute the locality on where to put data in a round robin way. Actions are instantiated so that they can be used.

To launch actions, an *Operation* has to be defined by binding a locality (a position on where the task should be run) and placeholders which will be used to pass data to the action later on. Finally, the *dataflow* function can be used to launch the action while waiting for the data held by the future. Line 27 to 30 in Listing 4.12 is an example on how to create the *Operation* and call the *dataflow* function to launch the *inv_part_action* which wrap a function that will process the data.

Listing 4.12: HPX dependencies

```
1  stepper::space stepper::do_lu(std::size_t T, std::size_t N,
2                                bool print_matrices) {
3      using hpx::dataflow;
4
5      std::vector<hpx::id_type> localities = hpx::find_all_localities();
6      std::size_t nl = localities.size();     // Number of localities
7
8      space tiles, invs;
9      tiles.resize(T * T);
10     invs.resize(T);
11
```

```
12     for (std::size_t i = 0; i != T; ++i)
13         for (std::size_t j = 0; j != T; ++j)
14             tiles[idx(i, j, T)] =
15                 partition(localities[locidx(i, j, T, nl)], N, T, i, j);
16     for (std::size_t i = 0; i != T; ++i)
17         invs[i] = partition(localities[locidx(i, 0, T, nl)], N, T, i, 0);
18
19     inv_part_action act_inv;
20     pmm_part_action act_pmm;
21     pmm_d_part_action act_pmm_d;
22     using hpx::util::placeholders::_1;
23     using hpx::util::placeholders::_2;
24     using hpx::util::placeholders::_3;
25
26     for (std::size_t k = 0; k < T - 1; ++k) {
27         auto Op =
28             hpx::util::bind(act_inv, localities[locidx(k, 0, T, nl)], _1, _2);
29         invs[k] =
30             dataflow(hpx::launch::async, Op, tiles[idx(k, k, T)], invs[k]);
31         for (std::size_t i = k + 1; i < T; ++i) {
32             auto Op = hpx::util::bind(
33                 act_pmm, localities[locidx(k, 0, T, nl)], _1, _2);
34             tiles[idx(i, k, T)] =
35                 dataflow(hpx::launch::async, Op, tiles[idx(i, k, T)], invs[k]);
36             for (std::size_t j = k + 1; j < T; ++j) {
37                 auto Op = hpx::util::bind(
38                     act_pmm_d, localities[locidx(i, j, T, nl)], _1, _2, _3);
39                 tiles[idx(i, j, T)] =
40                     dataflow(hpx::launch::async, Op, tiles[idx(i, k, T)],
41                         tiles[idx(k, j, T)], tiles[idx(i, j, T)]);
42             }
43         }
44     }
45
46     // Return the LU factorization
47     return tiles;
48 }
```

4.2.5.3 HPX Data definition

Data are stored into a *partition_data* structure. Those are held by futures to wait for their availability and create dependencies between the tasks. Then the actions make request to the *partition_server* to get the data in a future and wait for them. The partition server is implemented and registered to HPX runtime as a component. It generates code that can be used to get the data and use them on the tasks. The *get_data* function is also registered here. Listing 4.13 has the implementation of the *partition_server*. It shows the use of HPX macros to register and set up the *partition_server*.

Listing 4.13: HPX data

```
1  struct partition_server : hpx::components::component_base<partition_server> {
2     partition_server() {}
3
4     partition_server(partition_data const& data)
5       : data_(data) {}
6
7     partition_server(std::size_t n,std::size_t t,std::size_t i,std::size_t j)
8       : data_(n, t, i, j) {}
9
10    partition_data get_data() const { return data_; }
11
12    HPX_DEFINE_COMPONENT_DIRECT_ACTION(
13        partition_server, get_data, get_data_action);
14
15 private:
16    partition_data data_;
17 };
18
19 typedef hpx::components::component<partition_server> partition_server_type;
20 HPX_REGISTER_COMPONENT(partition_server_type, partition_server);
21
22 typedef partition_server::get_data_action get_data_action;
23 HPX_REGISTER_ACTION(get_data_action);
```

4.2.5.4 HPX Graph execution

There is several ways to initialize HPX runtime and execute applications. One of the way is shown in Listing 4.14. The main function has to initialize HPX via the function *hpx::init* which takes Boost command line options description which helps to parse command line arguments as input. HPX runtime launch the *hpx_main* function which takes Boost command line options as argument so that the user can get command line arguments.

In the *hpx_main* function, a few parameters are extracted from the Boost command line option map. Then the *stepper* is instantiated so that the function *do_lu* which initializes the data, launches the tasks and outputs futures containing the processed data. Furthermore, the data in the future are waited to make sure that every computation was done and use them later. They could have been used without waiting for them before hand so that any computation that would depend on the results of *do_lu* could have been tried before every computation from *do_lu* are finished in order to maximize the use of the available resources. Finally, *hpx::finalize* function is called to shutdown HPX runtime at the end of the application.

Listing 4.14: HPX launch

```
1  int hpx_main(boost::program_options::variables_map& vm)
2  {
3      std::uint64_t N = vm["N"].as<std::uint64_t>();
4      std::uint64_t T = vm["T"].as<std::uint64_t>();
5
6      // Create the stepper object
7      stepper step;
8
9      // Execute nt time steps on nx grid points and print the final solution.
10     stepper::space solution = step.do_lu(T, N);
11     for (std::size_t i = 0; i != T * T; ++i)
12         solution[i].get_data().wait();
13
14     return hpx::finalize();
15 }
16
17 int main(int argc, char* argv[])
18 {
19     using namespace boost::program_options;
20
21     options_description desc_commandline;
22     desc_commandline.add_options()(
23         ("N", value<std::uint64_t>()->default_value(10),
24         "Dimension of the submatrices")
25         ("T", value<std::uint64_t>()->default_value(10),
26         "Number of subblocks in each dimension");
27
28     // Initialize and run HPX
29     return hpx::init(desc_commandline, argc, argv);
30 }
```

4.2.5.5 HPX Granularity

The tasks are single threaded functions called on the local node or on a remote node by the runtime. They are fine-grain tasks.

4.2.5.6 HPX Tasks re-usability

The *stepper* structure can be reused to store tasks and implement another function which will use the tasks as base. Tasks and actions may be stores in a better environment so that they can be used more easily.

4.2.5.7 HPX Scheduling

HPX scheduler coordinates execution of tasks. It tries to maximize throughput of the cores, prioritize work according to need and minimize waiting time. HPX creates *worker* (hardware) thread per core on startup on which it runs its own Task Scheduler. HPX tasks are executed on the HPX worker (OS) thread. Each HPX 'task' is referred to as a lightweight thread.

4.2.5.8 HPX Type of graph

The data flow graph is provided by the user through futures and the *dataflow* function.

4.2.5.9 HPX GPU support

HPX provides GPU support through CUDA futures. They work as the regular futures but allows to execute functions on GPUs. By default, the user has to use a future to execute CUDA functions to allocate memory on the device with CUDA, copy data to the device, make operations on the data and copy the results to the host. HPX also provides allocators which helps to manage memory on the device and copy data between the host and the device.

4.3 Parallel and Distributed Task Based Programming Models

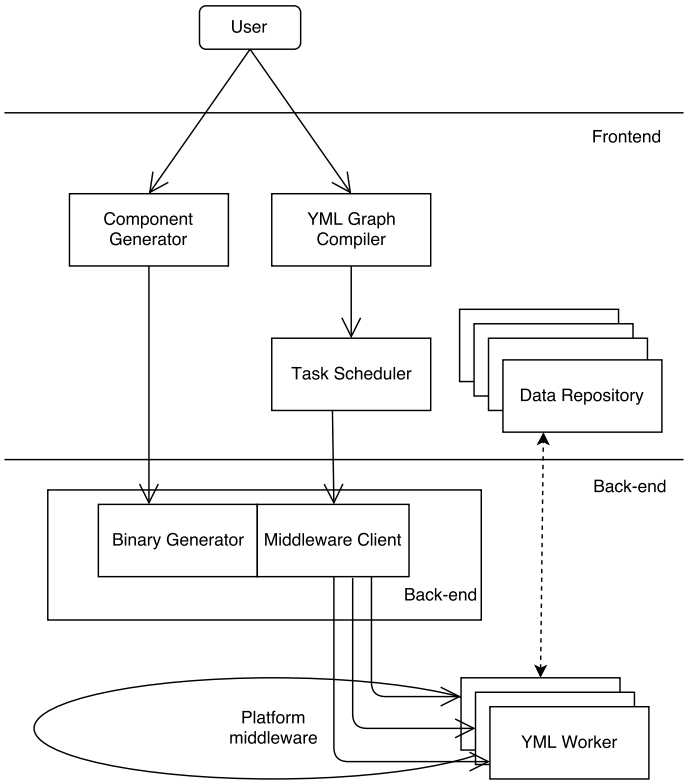
4.3.1 YML+XMP

4.3.1.1 YML

YML [11] [54] is a development and execution environment started in 2000 for scientific workflow applications over various platforms, such as HPC, Cloud, P2P and Grid. YML defines an abstraction over the different middlewares, so the user can develop an application that can be executed on different middlewares without making changes related to the middleware used. YML can be adapted to different middlewares by changing its back-end (Figure 4.1). Currently, the proposed back-end [55] uses OmniRPC-MPI [56] [112], a grid RPC which supports master-worker parallel programs based on multi SPMD programming paradigm. This back-end is developed for large scale clusters such as Japanese K-Computer and Poincaré, the cluster of the *Maison de la Simulation*. A back-end for peer to peer networks is also available. Moreover, YML also supports fault tolerance [57]. The Figure 4.1 shows the internal structure of YML.

YML applications are based on a graph and a component model. They represent the front-end. YML defines components as tasks represented by nodes of a graph which is expressed using the YvetteML language. It expresses the parallelism and the dependencies between components. The application is a workflow of components execution. Components can be called several times into one application. The graph can be seen as the global algorithm of the application and the components as the core code.

Figure 4.1: YML software architecture



The component model defines three classes of components : Graph, Abstract and Implementation. They are encapsulated using XML. They are made to hide the communication aspect and the code related to data serialization and transmission.

- Graph components contain the name, a description of the application and the graph written using YvetteML. This component also defines global parameters. They are data that can be used in components. Abstract components are called with the keyword *compute*, their name and the list of parameters they need.

- Abstract components contain the name, the type (abstract) and a description of the component. They also define the parameters of the data coming from the global application and describe their use in the component (if the data are consumed, modified or created).

- Implementation components contain the name, the type (implementation), a description, the name of the abstract component associated, the language used, the external libraries that YML has to deploy and the source code. The links of the libraries are installed in YML then they can be used in the implementation component and YML deploys them when needed. The source code can also be written

in several programming languages, including, for the moment, C/C++, Fortran or XscalableMP (XMP) [113].

These categories of component allow re-usability, portability and adaptability.

The Component generator registers the Abstract and Implementation components in YML and the data they need. The Component generator starts to check if the abstract components exist then it extracts the source code and the information needed to create a standalone application. The generator also adds the code to import and export the data from the file system. Afterwards, the Component generator calls the Binary Generator (Figure 4.1) to compile the generated application. It uses the compiler associated to the language filled in the Implementation component.

The Graph compiler generates the YML application by compiling the Graph component. It checks that all the components called exist, then verifies that the YvetteML graph is correct. It also extracts the control graph and the flow graph from the Graph component and the Abstract components. It creates a binary file containing the application executable with the YML scheduler.

The scheduler manages the computational resources and the data of the application during its execution. A set of processes is allowed to YML to run the application. The scheduler explores the graph at runtime (Figure 4.1) to determine which component has to be run. A component can be run if the execution of the previous components is finished, the data and the computational resources are available. The scheduler runs the component of the application on a subset of the processes through a worker. The scheduler sends the component (as shown in the Figure 4.1) to a worker through the middleware. It executes the component on the subset of processes that manages the worker. Several workers can run at the same time if there are enough processes available.

Data are stored in a repository created in the current repository at the launch of the application. The data are read by the components that need them. The components produce or/and create data and write them in the repository.

4.3.1.2 XcalableMP

XcalableMP (XMP) [113] is a directive-based language extension for C and Fortran, which allows users to develop parallel programs for distributed memory systems easily and to tune the performance by having minimal and simple notations. XMP supports (1) typical parallelization methods based on the data-/task-parallel paradigm under the "global-view" model and (2) the co-array feature imported from Fortran 2008 for "local-view" programming.

The Omni XMP compiler translates an XMP-C or XMP-Fortran source code into a C or Fortran source code with XMP runtime library calls, which uses MPI and other communication libraries as its communication layer.

Figure 4.2 is an example of XMP programming. A dummy array called **template** indicates data index space and is distributed onto the nodes. Each element of the array is allocated to the node where corresponding template element is distributed.

```
#pragma xmp nodes(4)
#pragma xmp template t(0:7)
#pragma xmp distribute t(block) onto p

double a[8];
#pragma xmp align a[i] with t(i)

int main()
{
   int i;
#pragma xmp loop on t(i)
   for(i=0; i<8;i++){
     a[i]=0;
   }
}
```

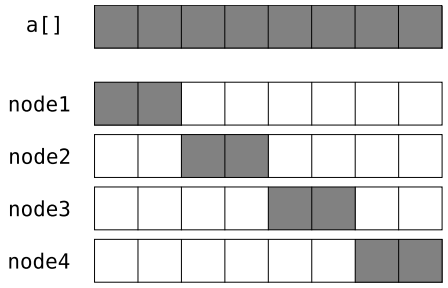


Figure 4.2: Example of XMP programming [114]

4.3.1.3 YML+XMP

For the experiments, we use XMP to develop the YML components as introduced in [55]. This allows two levels programming. The higher level is the graph (YML) and the second level is the PGAS component (XMP). In the components, YML needs complementary information to manage the computational resources and the data at best : the number of XMP processes for a component and the distribution of the data in the processes (template). Line 5 to 11 from Listing 4.15 shows the additional information provided to YML by the user in the implementation component in XMP. With this information, the scheduler can anticipate the resource allocation and the data movements. The scheduler creates the processes that the XMP components need to run the component. Then each process will get the piece of data which will be used in the process from the data repository.

Listing 4.15: XMP implementation component for YML

```
1  <?xml version="1.0"?>
2  <component type="impl" name="prodMat_impl"
3           abstract="prodMat" description="A = B x A">
4  <impl lang="XMP" nodes="CPU:(32,32)">
5    <templates>
6      <template name="t" format="block,block" size="512,512"/>
7    </templates>
8    <distribute>
9      <param template="t" name="B0(512,512)" align="[i][j]:(j,i)"/>
10     <param template="t" name="A0(512,512)" align="[i][j]:(j,i)"/>
```

```
11    </distribute>
12  <header>
13   <![CDATA[
14  #include<xmp.h>
15   double A[512][512];
16   double B[512][512];
17
18  #pragma xmp align A[i][j] with t(j,i)
19  #pragma xmp align B[i][j] with t(j,i)
20
21  #pragma xmp shadow A[*][0]
22  #pragma xmp shadow B[0][*]
23   ]]>
24  </header>
25  <source>
26   <![CDATA[
27     // XMP code
28   ]]>
29  </source>
30  <footer />
31  </impl>
32  </component>
```

In this approach, there are no global communications along all the processors (like scalar products or reductions) in the application since applications are divided on component calls that use a subset of the computational resources allowed to the application. The components running at the same time don't communicate with each other and are run independently. Global communications take time and consume a lot of energy so reducing them and finding other ways to get the same results saves time and is more power efficient.

Listing 4.16: Abstract component for YML

```
1  <?xml version="1.0" ?>
2  <component type="abstract" name="prodMat" description="A0 = B0 * A0">
3  <params>
4  <param name="B0" type="Matrix" mode="in" />
5  <param name="A0" type="Matrix" mode="inout" />
6  </params>
7  </component>
```

4.3.1.4 YML+XMP Task definition

In YML, implementing a task means to create a component. There is two type of components to implement; an abstract component and an implementation component which will be be implemented with XMP in this case. Listing 4.16 shows an example of abstract component. Line 2 gives the type of component (abstract), the name of the component and a brief description. This name will be used in the implementation component and to call the task. Line 4 and 5 define 2 *Matrix* parameters; one is in input and the other one is used in input and output. The second matrix will be modified by this task.

Listing 4.15 contains the actual implementation of the component. Line 2 sets the name of this component and also tells which abstract component it implements. Line 4 gives the language used to implement this component. The languages can also be C++ and Fortran as detailed previously. It uses XMP in this case. Line 4 also sets how much cores will be used to run the task on. In this case, we use a 2D array of 32×32 CPUs. Line 5 to 11 describe how to distribute the Matrix *A0* and *B0* (defined in the abstract component) on the allocated cores. It also defines the size of the matrices used (512×512).

The code of the task is contained in the *header*, *source* and *footer* tags. The code inside the *header* tag will be put at the end of the header part of the generated C source file of the component. The code from *source* tag is put in the main and the code from the *footer* tag will be put at the end of the generated C file.

4.3.1.5 YML+XMP Dependency definition

Dependencies are defined while creating the graph of tasks. Listing 4.17 shows a YML graph with dependencies between the tasks. Tasks are called with the keyword *compute*, the name of the component provided in the abstract component and the parameters of the task. They can be called in bulk with loops. *for* and *par* are the two statements which can be used to repeat statements. The *for* loop can be used to call task one after the other. It creates dependencies to wait for the previous task to be finished before launching the next. The *par* loop launch every task in the loop body at the same time. A *par* loop is used on lines 10 and 13 in Listing 4.17. There is also a *par* statement which can be used alongside with // to launch several tasks at the same time.

Additional dependencies can be expressed by using YML locking event system. The *wait* and *notify* keywords are used to manage the events. *wait* locks the following operations until the event is released by a *notify* on the same event. For instance, on line 22, the *XMP_inversion* task depends on the *p[i][i][i]* which is locked by a *wait*. This event will be released by the *notify* on line 17 when $k = i + 1$ and $j = i + 1$. It means that the task *compute XMP_inversion(A[i][i],B[i])* depends on the task *compute XMP_prodDiff(A[k][i],A[i][j],A[k][j])*. This event system is used to infer a graph of dependencies. It is used to schedule the tasks.

Listing 4.17: YML Graph

```
1 <?xml version="1.0"?>
2 <application name="block LU">
3 <description>Block LU decomposition of a matrix.</description>
4 <graph>
5 par(i:=0;blockcount-1)(j:=0;blockcount-1) do
```

```
6       compute XMP_genMat(A[i][j],i,j);
7       notify(p[i][j][0]);
8   enddo
9
10  par(i:=0;blockcount-1)(j:=0;blockcount-1) do
11      if (j gt i) then
12          par(k:=i+1;blockcount-1) do
13              wait(p[k][j][i] and p[k][i][i+1] and p[i][j][i]);
14              compute XMP_prodDiff(A[k][i],A[i][j],A[k][j]);
15              notify(p[k][j][i+1]);
16          enddo
17      else
18          if (i eq j) then
19              if(i neq blockcount-1) then
20                  wait(p[i][i][i]);
21                  compute XMP_inversion(A[i][i],B[i]);
22                  notify(p[i][i][i+1]);
23              endif
24          else
25              wait(p[j][j][j+1] and p[i][j][j]);
26              compute XMP_prodMat2(A[i][j],B[j]);
27              notify(p[i][j][j+1]);
28          endif
29      endif
30  enddo
31  </graph>
32  </application>
```

4.3.1.6 YML+XMP Data definition

In YML, data types have to be registered so that it is able to manage them. They are stored in the *DefaultExecutionCatalog/generators* directory in YML configuration files. Listing 4.18 shows an example of the functions to implement with the *Matrix* used in the abstract component. There is three functions to implement. The first is *<type_name>_MPI_Type* which return the *_MPI_Datatype* used to import and export the data associated to the type in a task. It can be a custom MPI type. *<type_name>_import* and *<type_name>_export* are the two other functions to implement. They are used to import and export the type from the file system. MPIIO is used to do so in the Matrix type.

Listing 4.18: Matrix XMP type

```
1  #include <stdlib.h>
2  #include <stdbool.h>
```

```
3 #include <mpi.h>
4
5 typedef double XMP_Matrix;
6 typedef double* Matrix;
7
8 static MPI_Datatype Matrix_MPI_Type() { return MPI_DOUBLE;}
9
10 static bool Matrix_import(Matrix param, char* filename,
11           const MPI_Datatype motif, const int size) {}
12 static bool Matrix_export(const Matrix param, char* filename,
13           const MPI_Datatype motif, const int size, MPI_Comm Communicator) {}
```

4.3.1.7 YML+XMP Graph execution

The graph is executed by launching the *.yapp* application with the *yml_scheduler*. This application is produced by *yml_compiler*. With XMP back-end, *yml_scheduler* is launched with *mpirun -n 1 yml_scheduler my_app.yapp*. It will enable YML to launch tasks in the distributed environment.

4.3.1.8 YML+XMP Granularity

With the XMP back-end, tasks are MPI based applications implemented with XMP so they use multi-processes technology. Each XMP task can run on several nodes of the global number of nodes associated to the application. These tasks are coarse-grain.

The C back-end provides multi-threaded tasks. These tasks can be run on one node.

4.3.1.9 YML+XMP Tasks re-usability

Implementation and abstract components can be reused in another application by calling them again with the *compute* keyword. However, components implemented in XMP have their number of cores used and size of data fixed. Unfortunately, they cannot be changed dynamically. It means that the components have to be recompiled each time one of those parameters is changed or that multiple versions if the component with different parameters have to coexist.

4.3.1.10 YML+XMP Scheduling

When a task dependencies are satisfied, the task is put in the execution queue. The tasks in this queue are executed when the resources are available. Furthermore, the tasks are launched on the resources they ask in their implementation.

4.3.1.11 YML+XMP Type of graph

The user defines the control flow graph while giving the dependencies between the tasks. However, the data flow graph can be extracted from the control flow graph and the access mode of the data provided in the abstract component.

4.3.1.12 YML+XMP GPU support

Support for GPUs depends on the backends used. The XMP backend does not directly support GPUs. The user has to use an external library to address the use of GPUs.

Moreover, there is a backend that used XMP and StarPU that can address the usage of GPUs.

4.3.2 Pegasus

Pegasus [12] [115] is a Workflow Management System. It allows the user to express multi-step computational tasks through a directed acyclic graph (DAG), where the nodes are tasks and the edges denote the task dependencies. The tasks can be everything from short serial tasks to very large parallel tasks (MPI for example) surrounded by a large number of small, serial tasks used for pre- and post-processing.

Pegasus provides helpers to execute workflow-based applications in different environments like desktops, clusters, grids and clouds. It automatically maps high-level workflow descriptions onto distributed resources. It also locates the necessary input data and computational resources needed during the workflow execution. Pegasus enables scientists to construct workflows in abstract terms. It can be linked with several middlewares (HTCondor DAGMan [63], Globus, or Amazon EC2).

Pegasus is fault tolerant. Indeed, when there is errors, it can retry the tasks, retry the entire workflow, checkpoint its execution, re-map part of the workflow, try alternative data sources and as last resort, provide a list of the remaining tasks. Storage is cleaned up during the execution of the workflow. Thus, data-intensive workflows can get enough space to run their tasks on resources with low-capability storage. Pegasus also keeps track of what has been done, where, which software was used and their parameters.

Pegasus has been used in a number of scientific domains including astronomy, bioinformatics [116], earthquake science , gravitational wave physics, ocean science, and others.

4.3.2.1 Pegasus Task definition

In Pegasus, tasks are called *Job*. Listing 4.19 shows how to define and register a *Job* into Pegasus. DAX Python API is used to generate the XML file that will be treated by Pegasus to launch the jobs on the available resources. Line 1 defines the graph which will be populated with the jobs. Then, line 3 to 7 define the Job with its name space, name and version. The name of the job corresponds to the name of the Executable defined and registered in Listing 4.20. An executable is registered into a graph with the *addExecutable* method.

Listing 4.19: Pegasus task

```
1 diamond = ADAG("diamond")
2
3 frl = Job(namespace="diamond", name="findrange", version="4")
4 c1 = File("f.c1")
5 frl.addArguments("-a findrange","-T60","-i",b1,"-o",c1)
6 frl.uses(b1, link=Link.INPUT)
7 frl.uses(c1, link=Link.OUTPUT)
8 diamond.addJob(frl)
```

A job can also pass arguments to the executable with the *addArgument* method. String and files can be passed as arguments. A file *c1* is created and will be used as output as stated on line 7. An already existent file *b1* will be used as input. The files have registered into the job with the *uses* method so that Pegasus runtime will be able to manage is and move it across the network if needed. Finally, the job is registered into Pegasus graph by using the *addJob* method from the graph (on line 8 from 4.19).

A file could be used as input and output however, the *Link.INOUT* property cannot be used. It does not seem to be implemented. Therefore, a task cannot modify a file and needs to have an output distinct from the input. Moreover, each file is considered as an output by default. They will be transfered back to the user by the runtime, even the temporary files that has to be created due to the impossibility to use a file name as input and output. The user has to manually set the temporary files as not wanted as output instead of modifying a file. This decreases the complexity of the runtime since the data files will be unique.

Listing 4.20: Register executable in Pegasus

```
1 e_findrange = Executable(namespace="diamond", name="findrange", version="4",
2                          os="linux", arch="x86_64", installed=True)
3 e_findrange.addPFN(PFN("file://" + sys.argv[1] + "/bin/pegasus-keg",
4                          "TestCluster"))
5 diamond.addExecutable(e_findrange)
```

4.3.2.2 Pegasus Dependency definition

Dependencies are simple to add into a Pagasus graph with the Python interface. Indeed, the graph has a method called *addDependency* that takes the dependency as parameter. A dependency can be declared by providing if the parent and child jobs. Listing 4.21 shows an example of such a dependency.

Listing 4.21: Dependecy in Pegasus

```
1 diamond.addDependency(Dependency(parent=frl, child=analyze))
```

4.3.2.3 Pegasus Data definition

Pegasus jobs call executables that process files so the user has to use data files that match the format used by the executables to read or write data from file. The user has to register data files either as intermediary to transfer data from a job to another as described above or registered to the graph which will be used as input and/or output of the jobs performed by the application. Listing 4.22 shows an example of files registered to the graph.

Listing 4.22: File registration in Pegasus

```
1 a = File("f.a")
2 a.addPFN(PFN("file://" + os.getcwd() + "/f.a", "local"))
3 diamond.addFile(a)
```

4.3.2.4 Pegasus Graph execution

Pegasus Python API generates a DAX which can be executed by Pegasus *pegasus-plan* command. It will submit the jobs to the resources available to the application and manage the input and output data files.

4.3.2.5 Pegasus Granularity

The jobs can be anything between short one threaded application to large scale multi-node applications based on MPI.

4.3.2.6 Pegasus Tasks re-usability

Jobs can be reused in another application if they use the same arguments and files as input otherwise, they will have to be rewritten.

4.3.2.7 Pegasus Scheduling

By default, Pegasus uses HTCondor DAGMan [63] as underline work-flow manager. Therefore, it uses HTCondor Schedd to schedule the jobs submitted to HTCondor DAGMan.

Pegasus can use other back-ends as engine. The scheduling policy will depend on how the engine manage the jobs submitted to it.

4.3.2.8 Pegasus Type of graph

The user only defines dependencies between Jobs so the graph is a control flow graph.

4.3.2.9 Pegasus GPU support

Pegasus does not seem to support GPU by itself. The user should be able to tell Pegasus which tasks need GPU to run on and which resources have GPU available. Pegasus tasks are implemented on top of an existing executable so this executable has to be able to manage GPUs to run applications on them.

4.3.3 Swift

4.3.3.1 Swift

Swift [13] [58] is a scripting language for executing many instances of ordinary application programs on distributed and parallel resources. Swift runs applications as soon as their inputs are available. The programming is simpler since the user doesn't have to manage the dependencies. The variables can be assigned only once. This allows Swift to easily know when to launch a function. Figure 4.3 is an example[1] of code which can be translated to the graph of the Figure 4.4. If the variable has been set then the function can use the data. Otherwise, Swift wait for the variable to be set before launching functions that need this variable. It simplifies the expression of the parallelism for the system. Thus, all code will be executed in parallel unless there is a variable unset in a function call. The function using the unset variable will wait for the variable to be set by another call. Therefore, there will be sequencing between the two calls. Swift expressions are

[1] http://swift-lang.org/Swift-T/index.php

evaluated in dataflow order to run the workflows with the most concurrency possible. It only depends on the data dependencies and the available resources. Moreover, Swift allow to wrap applications into functions to create more complex scripts. The functions take files as input and produce files as output.

```
int X = 100, Y = 100;
int A[][];
int B[];
foreach x in [0:X-1] {
   foreach y in [0:Y-1] {
      if (check(x, y)) {
         A[x][y] = g(f(x), f(y));
      } else {
         A[x][y] = 0;
      }
   }
   B[x] = sum(A[x]);
}
```

Figure 4.3: Example of Swift code

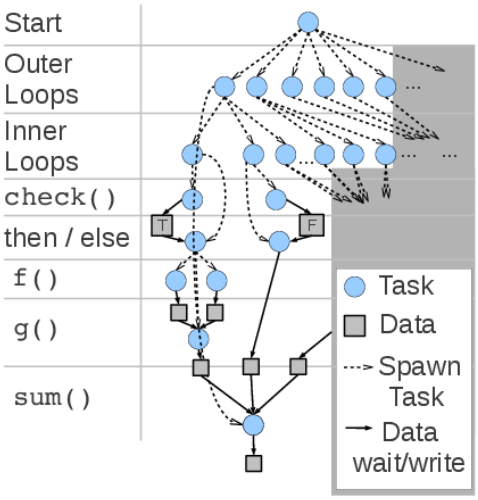


Figure 4.4: Graph translation of the code

Swift is suitable for Grid.

We found several applications which use Swift. For instance, [117] uses Swift for proteins modelling or [118] for parallel climate data analysis.

4.3.3.2 Swift/T

Swift/T [59] is a new implementation of Swift dedicated to HPC. It translates the Swift script into a MPI program using Turbine [60] and ADLB (Asynchronous Dynamic Load Balancer) [61] libraries. Turbine is a Tcl library that assemble MPI, ADLB and the Turbine dataflow library. ADLB is a software library designed to build scalable parallel programs.

Swift/T has two main level of programming : the Swift script and the Leaf functions. The Swift parallel script correspond to the high grain graph and the leaf functions to the tasks. The heavy computations are located in the leaf functions. Since Swit/T uses Swift scripts, the variable assignments and the expression of the parallelism works the same way.

There is three types of leaf functions in Swift/T :

1. Extension functions : Tcl (language used in Turbine) or native code functions like C, C++, Fortran or MPI (possibility to call them as parallel functions). These functions primarily operate on in-memory data, and are appropriate for high-performance computing.

Listing 4.23: C function

```
#include <stdio.h>
#include <stdlib.h>
```

```
double* b(double* v, int length) {
  int i;
  double sum = 0.0;
  printf("length: %i\n", length);
  for (i = 0; i < length; i++) {
    sum += v[i];
  }
  printf("sum: %f\n", sum);
  double* result = malloc(sizeof(double));
  result[0] = sum;
  return result;
}
```

Listing 4.23 is the code of the b function used in the example[2] of a C leaf function.

Listing 4.24: Creation of the Tcl package

```
rm *.o
swig -module b b.i
gcc -c -fPIC b.c
gcc -c -fPIC $TCL_INCLUDE_SPEC b_wrap.c
gcc -shared -o libb.so b_wrap.o b.o
tclsh make-package.tcl > pkgIndex.tcl
```

Listing 4.24 shows how to create the wrapper of the C function which will be called in the Tcl function.

Listing 4.25: Tcl wrapper around the C funtion

```
namespace eval b {
    # v is formatted as a Turbine blob a list of [ pointer length ]
    # The pointer is a simple Tcl integer
    # The length is the byte length
    proc b_tcl { v } {

        # Unpack the list
        set ptr [ lindex $v 0 ]
        set len [ lindex $v 1 ]

        # Get the number of numbers to sum
        set count [ expr $len / [ blobutils_sizeof_float ] ]

        # Convert the pointer number to a SWIG pointer
```

[2] http://swift-lang.github.io/swift-t/leaf.html#_complete_example_2_simple_c_function

```
        set ptr [ blobutils_cast_int_to_dbl_ptr $ptr ]

        # Call the C function
        set s [ b $ptr $count ]

        # Pack result as a Turbine blob and return it
        set r [ blobutils_cast_to_int $s ]
        return [ list $r 8 ]
    }
}
```

Listing 4.25 shows how to use the C function inside a Tcl function. This Tcl function is used in the Swift application. Listing 4.26 shows how to call a Tcl function from a Swift application.

Listing 4.26: Call of the Tcl wrapper by a Swift application

```
import blob;
import io;

(blob sum) b(blob v) "b" "0.0"
[ "set <<sum>> [ b::b_tcl <<v>> ]" ];

file data = input_file("input.data");
blob v = blob_read(data);
blob s = b(v);
float sum[] = floats_from_blob(s);
printf("sum (swift): %f", sum[0]);
```

2. app functions: Functions that call to a command-line program (the shell). These functions primarily operate on files, and are appropriate for ordinary workflows.

3. External scripting functions: Functions that call into an in-memory interpreter in another scripting language, such as Python (Listing 4.27), R (Listing 4.28), or **Julia** (Listing 4.29).

Listing 4.27: Call of python code by a Swift application

```
import io;
import python;
import string;

global const string numpy = "from numpy import *\n\n";

typedef matrix string;
```

```
(matrix A) eye(int n)
{
  string command = sprintf("repr(eye(%i))", n);
  matrix t = python_persist(numpy, command);
  A = replace_all(t, "\n", "", 0);
}

(matrix R) add(matrix A1, matrix A2)
{
  string command = sprintf("repr(%s+%s)", A1, A2);
  matrix t = python_persist(numpy, command);
  R = replace_all(t, "\n", "", 0);
}

matrix A1 = eye(3);
matrix A2 = eye(3);
matrix sum = add(A1, A2);
printf("2*eye(3)=%s", sum);
```

Listing 4.28: Call of R code by a Swift application

```
import io;
import string;
import R;

global const string template =
"""
  x <- %i
  a <- x+100
  cat("the answer is: ", a, "\\n")
""";

code = sprintf(template, 4);
s = R(code, "toString(a)");
printf("the answer was: %s", s);
```

Listing 4.29: Call of Julia code by a Swift application

```
import io;
import julia;
import string;
import sys;
```

```
start = clock();
f =
"""
begin
 f(x) = begin
          sleep(1)
          x+1
        end
 f(%s)
end
""";
s1 = julia(sprintf(f, 1));
s2 = julia(sprintf(f, 2));
s3 = julia(sprintf(f, 3));
printf("julia results: %s %s %s", s1, s2, s3);
wait (s1, s2, s3) {
  printf("duration: %0.2f", clock()-start);
}
```

4.4 Analysis and First Evaluation of the Languages

We introduced three classes of programming models : the currently used message passing libraries, fine grain task based programming models in which tasks run on one thread or process and large grain task based programming models in which tasks are distributed and parallel and run on multiple processes. We studied several task based programming models in each category in detail. In particular, we studied the properties of HPX, PaRSEC, TensorFlow, Legion, Regent, YML+XMP, Swift and Pegasus.

HPX is a task based programming model. The tasks are run as lightweight processes on the resources managed by HPX runtime. The data and the tasks can be transfered across distributed nodes. It uses modern C++ which starts to be used more often in HPC applications. It extends C++ future-based (tasks) multi-threaded programming to distributed. Moreover, HPX is very close to C++ standards so it provides an interface which is supported and documented. Those are the points for which we are choosing HPX.

PaRSEC is a runtime which manages data and launches tasks. It also provides several DSLs to define and implement tasks. Two DSLs are already available. They allow the user to describe the dependencies between the tasks and how the tasks are implemented. PaRSEC has good performances compared to the one obtained with HPX as shown in [119] .

YML+XMP is a distributed and parallel task based programming model where each task is also distributed and parallel. This programming has been selected because it has two level of programming, the graph which is the high level and the parallel and distributed task which is the second level. [119] shows that YML+XMP has a better scalability compared to the other programming models. However, the performances

of YML+XMP are its weak points due the use of the file system to perform the communications. YML+XMP is not the only programming model to use is to make the communications. Indeed, Pegasus also use files to transfer data from a task to the others.

Legion will not be used due to its way of defining tasks via a tree of tasks which is less natural than a graph of tasks. Moreover, Legion verbose API is difficult to use. Finally, Legion performances with the default data mapper are not as good as the performances obtained with the other languages as shown in [119] while using Regent to generate Legion code. Therefore, Regent will not be used too.

TensorFlow is focused in Machine Learning and Neural Networks. It is well adapted to train models and to process data with those models. It offers a high level API to easily create and train models. In TensorFlow, the Operation is the main way to apply computations on data stored into Tensors aside the Machine Learning API. However, Operations can only be implemented by adding it directly into TensorFlow code and by recompiling it. Therefore, it is not well adapted to other kind of applications and will not be used to implement the Kirchhoff Migration.

CnC does not seem to be used a lot and does not seem maintained anymore. Thus, CnC will not be used to implement applications.

OmpSs uses the same kind of approach as OpenMP. It uses directives to parallelize sections of code. However, the execution model is different but it still produces multi-threaded applications whereas distributed programming models are considered in this study.

Swift is a parallel programming model where the tasks are also parallel and distributed. However, the data are sent to the tasks as blobs, an array of bytes, which is copied in each process of the tasks. Therefore, the distribution of the data is not optimal since each process of the task has a copy of the whole data instead of a part of it. Furthermore, the user has to write interface with the libraries that will be used as tasks. This interface consists of several Tcl wrappers and automatic interface generation. This process could be could be fully hidden from the user. Swift will not be studied further fo those reasons.

4.5 Application Deployment with Containers

Applications which use an uncommon software stack may be difficult to deploy on supercomputers. YML+XMP is an example of such an application. It depends on libraries that are not commonly installed on supercomputers like *Util*, a library implemented for YML or *OmniRPC* and the *omnicompiler* which are used to compile and launch parallel and distributed XMP based tasks. Moreover, XMP tasks are compiled into an application which is executed by YML scheduler through *OmniRPC* with *MPI_Comm_spawn*. Very few MPI implementations support *MPI_Comm_spawn* as used by YML and *OmniRPC* so installing the whole software stack with the right versions and support on a supercomputer is challenging. Furthermore, containerization technologies are not well supported yet on current supercomputers especially for distributed applications. The integration of those technologies on supercomputers is a interesting opportunity to make the deployment of complex software stacks easier.

To address this deployment issue, we created containers which can hold the recipe to compile and install YML+XMP for a given architecture. These containers can be built on the supercomputers if the user has the rights to do so or build locally then sent on the supercomputer for execution. The build recipe are used to create a lightweight system image containing all the libraries needed to run the applications. The YML+XMP

containers are made with Docker and Singularity which are lightweight virtualization systems. They allow to build and install applications in a controlled environment similar to virtual machines but lighter and with better performances.

Moreover, the use of container images is not limited to deployment on supercomputers. They can also be used to easily access complex software stack on the developers machines without having to fully install it. Therefore, they can gain in productivity and use the same environment as the one used on the supercomputer to develop the application locally. Finally, containers can be used to replicate software stack and application development on other supercomputers and reproduce experiments on machine which support containers.

We introduced several task based programming models and how to use them to implement task, define dependencies, register distributed data and how to execute the graph of tasks. This classification and the information on how to use the programming models to define tasks, express dependencies as well as register data are a contribution of this dissertation. We also selected YML+XMP, PaRSEC and HPX to implement out applications and make out experiments. These programming models will be used in the following chapter to experiment on dense linear algebra on petascale supercomputers. We will discuss our numerical experiments with the LU factorization and the solution of dense linear systems.

Chapter 5

Task-Based, Parallel and Distributed Dense Linear Algebra Applications

The task based programming models used to make experiments in this dissertations have been introduced and selected in the previous chapter. In this chapter, we introduce the dependency graphs of the block based dense linear algebra algorithms to solve linear systems previously introduced in Chapter 3. Afterwards, these graphs are converted into with YML+XMP applications that are used to perform experiments on the petascale K computer. Then, we focus on the block based LU factorization and implement it with YML+XMP, PaRSEC, Regent and HPX due to the fact that implementing the three algorithms to solve linear systems with every task based programming model would have taken too much time and is not the purpose of this dissertation. We perform experiments to compare and analyze the performances we obtain with the different task based implementations. We also compare the task based implementations to our MPI implementation and ScaLAPACK since MPI is the current library used to implement parallel and distributed applications and ScaLAPACK is a well optimized dense linear algebra library.

5.1 Task-Based Graphs of Methods to Solve Dense Linear Systems

In this section, we present the dependency graphs related to the block based dense linear algebra algorithms introduced in Chapter 3. These graphs are used in this chapter to transform the block based methods into task based applications implemented with multiple task based programming models.

Let A be a block matrix of size $np \times np$ and $p \times p$ blocks so each block is of size $n \times n$. Let b and x be block vectors of size np and p blocks so each block is of size n. The goal here is to solve the linear system $Ax = b$.

In the following algorithms, $A_{i,j}^{(k)}$ represents the k^{th} step of the block at the i^{th} row and j^{th} column of the block matrix A. Thus, $b_i^{(k)}$ represents the k^{th} step of the block at the i^{th} row of the block vector b. The algorithms are expressed using assignments on the blocks of $A_{i,j}^{(k+1)}$ and $b_i^{(k+1)}$. The block based algorithms introduced in Chapter 3 are presented again with a color for each type of operation performed. The red color is for the inversion of a matrix. The blue (magenta) color is for the product between two matrices (a matrix and a vector). The dark (light) green color is for a matrix (vector) minus the product between two matrices (a matrix and a vector). The cyan color is for solution of a linear system. The black is for any type

of operation that may produce the block matrix in input of the LU factorization, for instance generating the data, reading them from disc or data coming from other tasks. These operations are represented in the following dependency graphs. The graphs can be used to schedule the operations in task based programming models. They will be expressed to form application with task based programming models as best as the programming models allow to do so. Each bubble represents an assignment and an operation on the data. In Figure 5.1, the color of the bubbles are associated to the operation they represent.

Figure 5.1: Graphs legend

5.1.1 Block-Based Gaussian Elimination

In Algorithm 16, colors are used to represent the different operations performed in Algorithm 6. These colors are also used in Figure 5.2 to represent the task associated to the operations in the dependency graph.

Algorithm 16: Block Gaussian elimination and back substitution

For k **from** *0* **to** *p-2* **do**
- (1) $Inv^{(k)} = [A_{k,k}^{(k)}]^{-1}$
- (2) $b_k^{(k+1)} = Inv^{(k)} \cdot b_k^{(k)}$
- **For** j **from** $k+1$ **to** *p-1* **do**
  - (3) $A_{k,j}^{(k+1)} = Inv^{(k)} \cdot A_{k,j}^{(k)}$
- **End for**
- **For** i **from** $k+1$ **to** *p-1* **do**
  - (4) $b_i^{(k+1)} = b_i^{(k)} - A_{i,k}^{(k)} \cdot b_k^{(k+1)}$
  - **For** j **from** $k+1$ **to** *p-1* **do**
    - (5) $A_{i,j}^{(k+1)} = A_{i,j}^{(k)} - A_{i,k}^{(k)} \cdot A_{k,j}^{(k+1)}$
  - **End for**
- **End for**

End for

(6) $b_{p-1}^{(p)} = Inv^{(p-1)} \cdot b_{p-1}^{(p-1)}$

For k **from** *1* **to** *p-1* **do**
- **For** i **from** *0* **to** *p-k-1* **do**
  - (7) $b_i^{(k+i+1)} = b_i^{(k+i)} - A_{i,p-k}^{(i+1)} \cdot b_{p-k}^{(p)}$
- **End for**

End for

In Figure 5.2, the dependency graph of the Block Gaussian elimination with the back substitution to solve a linear system is given for $p = 4$. This dependency graph can be used to understand the dependencies

between the operations. Then, they can be expressed as a task based applications where the operations are the tasks and the dependency graph can be used to order the tasks.

Furthermore, the dependency graph can help find the parallelism between the operations and the critical path of the graph which represents the operations that cannot be delayed in order to execute the operations the fastest. For instance, the operations $C - AB$ in A-2-2-1 and A-3-3-1 can be executed at the same time since they depend on completely different data. However, the inversion in inv-0 and the matrix vector product in B-0-1 cannot be executed at the same time since the operation in B-0-1 needs the output of the inversion. Moreover, the inversion and the following matrix vector product could be reunited in the same operation that output both results but it requires the addition of another piece of data for this operation.

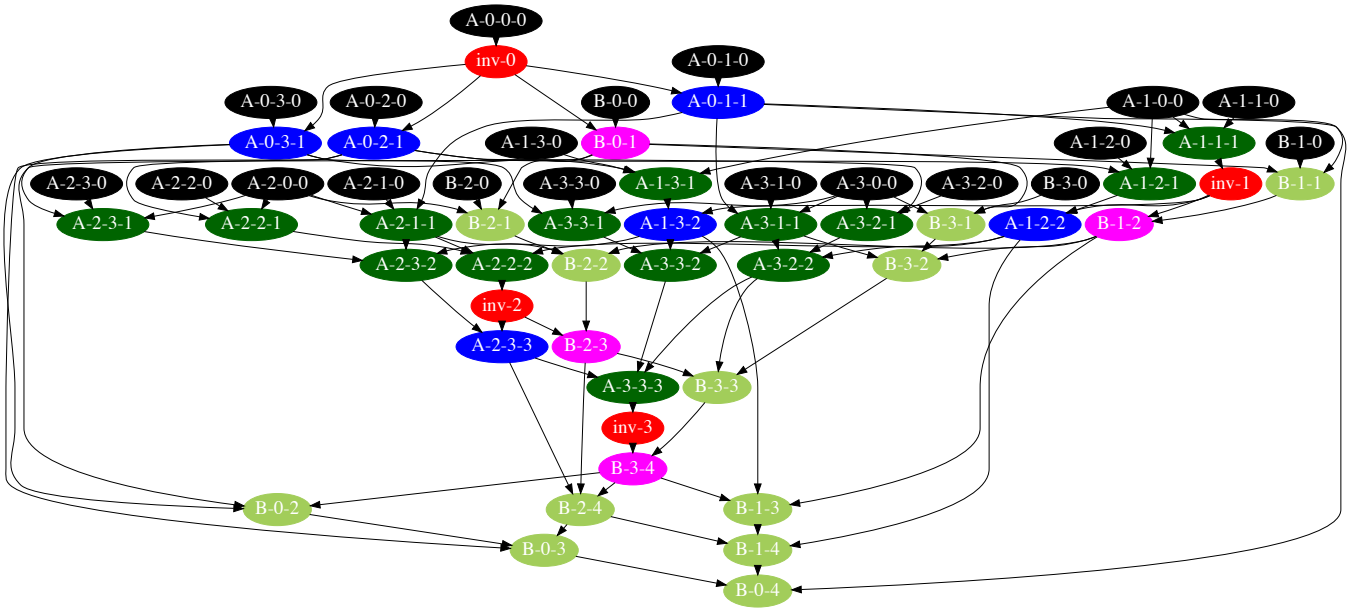


Figure 5.2: Block-based Gaussian Elimination graph with back substitutions for p = 4

5.1.2 Block-Based Gauss-Jordan Elimination

In Algorithm 17, colors are used to represent the different operations performed in Algorithm 8. These colors are also used in Figure 5.3 to represent the task associated to the operations in the dependency graph

In Figure 5.3, the dependency graph of the Block Gauss-Jordan elimination to solve a linear system is given for $p = 4$.

As for the Gaussian elimination, the inversion could be reunited with the matrix matrix product that is executed afterwards with the same advantages and drawbacks.

Algorithm 17: Block (Generalized) Gauss-Jordan elimination to solve a linear system

For k **from** *0* **to** *p-1* **do**
- (1) $Inv^{(k)} = [A_{k,k}^{(k)}]^{-1}$
- (2) $b_k^{(k+1)} = Inv^{(k)} \cdot b_k^{(k)}$
- **For** j **from** $k+1$ **to** *p-1* **do**
  - (3) $A_{k,j}^{(k+1)} = Inv^{(k)} \cdot A_{k,j}^{(k)}$
  - **For** i **from** *0* **to** *p-1* **do**
    - **If** $k \neq i$ **then**
      - (4) $A_{i,j}^{(k+1)} = A_{i,j}^{(k)} - A_{i,k}^{(k)} \cdot A_{k,j}^{(k+1)}$
    - **End if**
  - **End for**
- **End for**
- **For** i **from** *0* **to** *p-1* **do**
  - **If** $k \neq i$ **then**
    - (5) $b_i^{(k+1)} = b_i^{(k+1)} - A_{i,k}^{(k)} \cdot b_k^{(k+1)}$
  - **End if**
- **End for**

End for

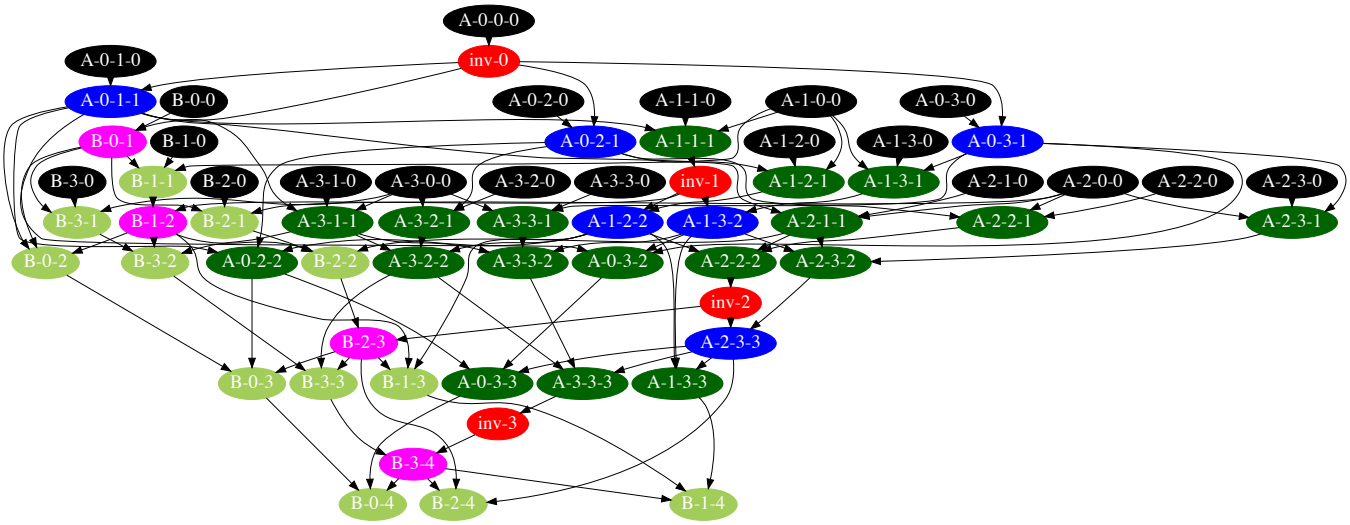


Figure 5.3: Block-based Gauss-Jordan Elimination graph for p = 4

5.1.3 Block-Based LU factorization

In Algorithm 18, colors are used to represent the different operations performed in Algorithm 2. These colors are also used in Figure 5.4 to represent the task associated to the operations in the dependency graph In this algorithm, the LU factorization is shown alone due to the fact that we implemented applications that only perform it.

Algorithm 18: Block-based LU Factorization

For i **from** *0* **to** *p-1* **do**
  For j **from** *0* **to** *p-1* **do**
    $\text{init}(A_{i,j}^{(0)})$
  End for
End for
For k **from** *0* **to** *p-2* **do**
  $Inv^{(k)} = [A_{k,k}^{(k)}]^{-1}$
  For i **from** *k+1* **to** *p-1* **do**
    $A_{i,k}^{(k+1)} = A_{i,k}^{(k)} \cdot Inv^{(k)}$
    For j **from** *k+1* **to** *p-1* **do**
      $A_{i,j}^{(k+1)} = A_{i,j}^{(k)} - A_{i,k}^{(k+1)} \cdot A_{k,j}^{(k)}$
    End for
  End for
End for

In Figure 5.4, the dependency graph of the Block Gauss-Jordan elimination to solve a linear system is given for $p = 4$.

As for the Gaussian elimination and Gauss-Jordan elimination, the inversion could be reunited with the matrix matrix product that is executed afterwards with the same advantages and drawbacks.

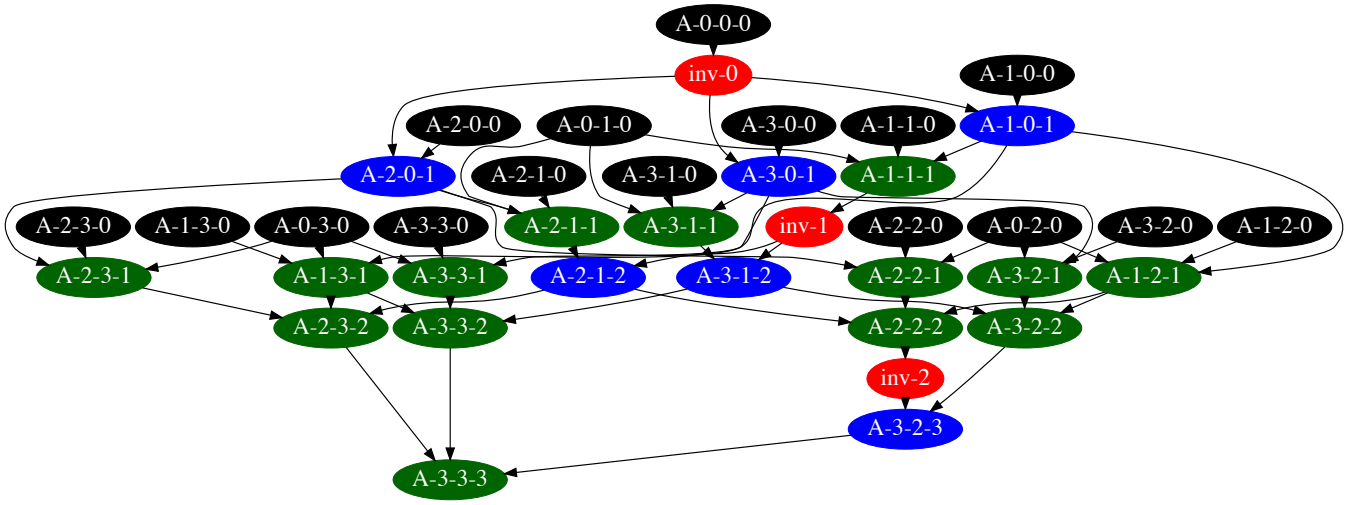


Figure 5.4: Block-based LU factorization graph for p = 4

5.1.4 Block-Based Resolution of Block Triangular Systems

In Algorithm 19, colors are used to represent the different operations performed in Algorithm 3. These colors are also used in Figure 5.5 to represent the task associated to the operations in the dependency graph In this algorithm, the solution of the dense linear system $A = LU = b$ is shown. It expects a matrix A in which the LU factorization of the input matrix is stored. This algorithm can be appended to the algorithm to perform the LU factorization.

Algorithm 19: Backward and forward substitution to solve a triangular linear system with a block-based LU factorization

For i **from** *0* **to** *p-2* **do**
  For j **from** *i+1* **to** p **do**
    (5) $b_j^{(i+1)} = b_j^{(i)} - A_{j,i}^{(i+1)} \cdot b_i^{(p)}$
  End for
End for

For k **from** *p-1* **to** *0* **step** *-1* **do**
  (6) solve $b_k^{(p)} = A_{k,k}^{(k)} \cdot b_k^{(p-1)}$
  For i **from** *1* **to** *k-1* **do**
    (7) $b_i^{(p-k+i)} = b_i^{(p-k+i-1)} - A_{i,k}^{(i)} \cdot b_k^{(p)}$
  End for
End for

In Figure 5.5, we added the backward and forward substitution to the LU factorization. This allows to solve a dense linear system by using a LU factorization and reuse the factorization if necessary.

The dependency graphs of three methods to solve dense linear systems were introduced. They are used to implement task based applications in which the tasks execute the operations on the blocks in the following

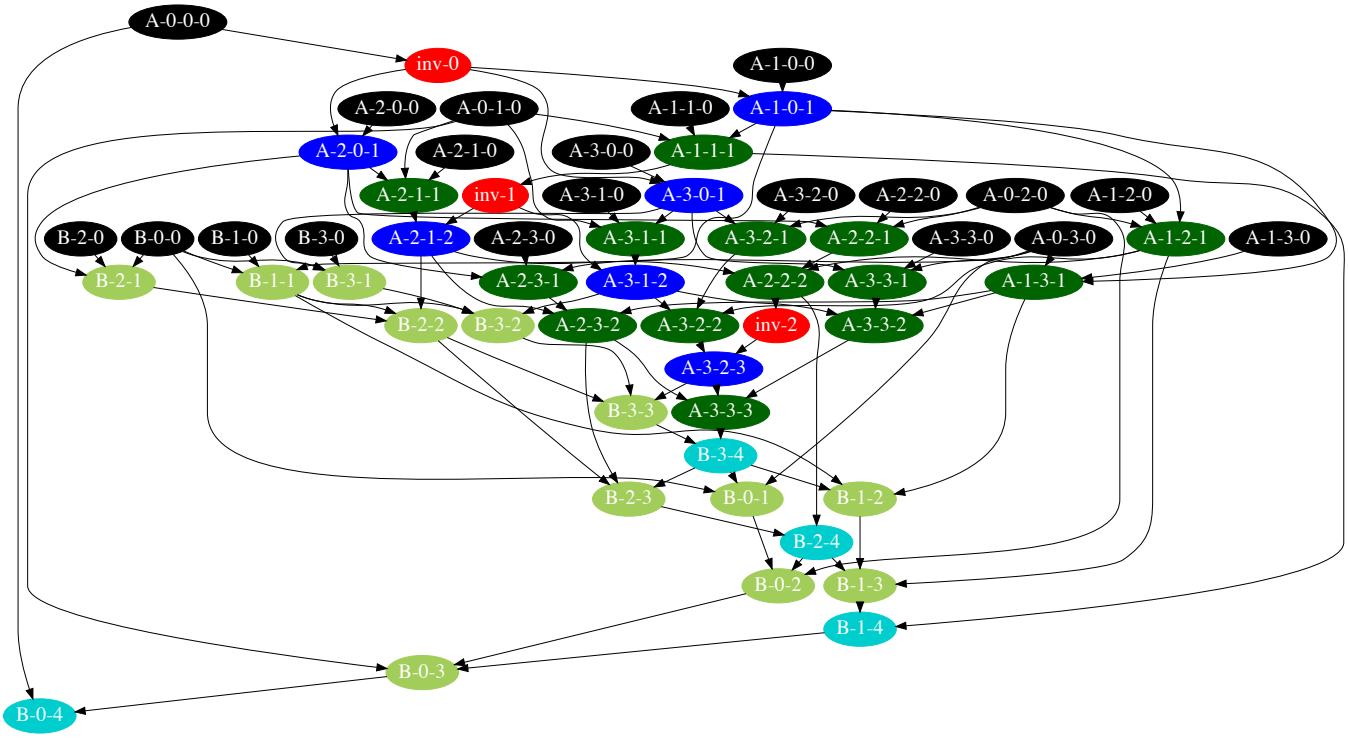


Figure 5.5: Block-based LU factorization graph with forward and backward substitutions for p = 4

sections. The operations themselves are parallelized in YML+XMP tasks.

5.2 Usage of YML+XMP and Experimentations

In the first place, the multi-level programming YML/XMP is compared to XMP on the K computer where OmniRPC and XMP are installed. Then, YML/XMP and XMP is compared to ScaLAPACK and the implementation of the resolution of a dense linear in MPI on Poincare.

5.2.1 Experiments on the K computer

5.2.1.1 The supercomputer

In this section, the tests were performed on the K Computer as shown on the left [1] of Figure 5.6 from Riken AICS in Kobe, Japan. This supercomputer was manufactured by Fujitsu. There is 88,128 compute nodes containing an eight-core SPARC64 VIIIfx processor and 16 Go of memory, for a total of 705,024 cores. The network is based on torus fusion (Tofu) interconnect, a six-dimensional mesh topology. It performed at 10.51 PFlop/s for Linpack and 602.736 TFlop/s for HPCG.

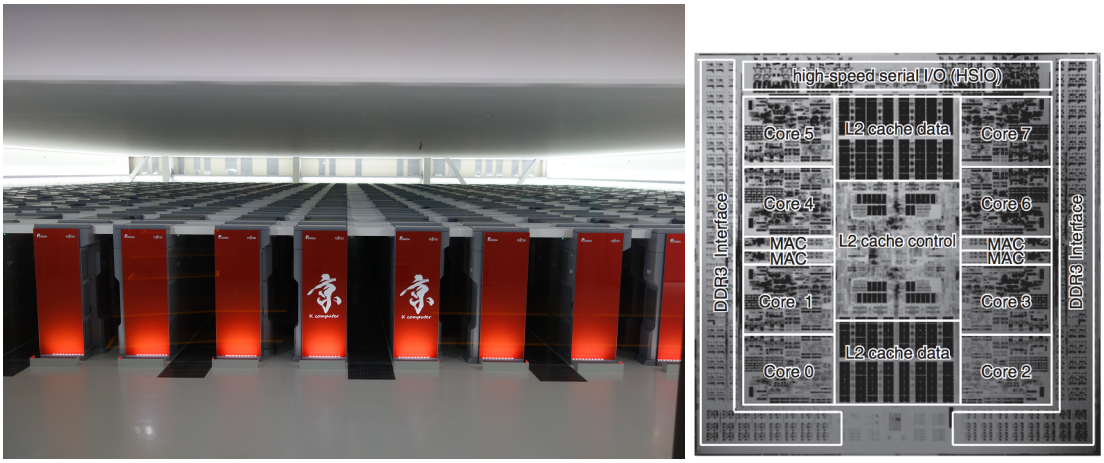


Figure 5.6: K Computer (left) and its SPARC64 VIIIfx processor (right) [120]

5.2.1.2 The Block Based Gaussian Elimination in YML

The Gaussian elimination in Algorithm 16 solves the system by doing a block triangularization of the block matrix A, and by solving the triangular system. In this algorithm, the number of assignments is $\frac{p^3+3p^2+2p}{3} \sim \frac{p^3}{3}$. The block-wise Gaussian elimination and back substitution have been implemented using YML/XMP.

For this experiments, the Algorithm 16 is expressed as YvetteML graph in Listing 5.1 and the operations on blocks are implemented using XMP. Each operation is performed on a matrix of a vector which are a subdivision of the global matrices and vectors. In YvetteML, there are two means to express the parallelism between components : the *par* loop and the *par* statement. To implement the block-wise Gaussian elimination, we expressed all the call to the components in parallel and we used the event management system of YvetteML to express the dependencies between the tasks. In the Algorithm 16, we observe that each block i,j at step k

[1] Toshihiro Matsui, https://www.flickr.com/photos/44861631@N02/32588659510, Creative Commons Attribution 2.0 Generic license

of the Gaussian Elimination is updated only once. Then, it is possible to associated the corresponding tasks to the triplets (i,j,k). Each task may be launched only when some tasks of the previous steps are completed and when the data are migrated to the allocated computing resource. Therefore, the dependency graph is equivalent in this case of the precedence between triplets: for example (i,j,k) will always depend of (i,j,l<k), for the adequate value of i,j, and k. If we associate each triplet to a Yvette array of events, each assignment of the block (i,j) at step k in the Algorithm 16 have to be preceded by a "wait" expressing the dependence between (i,j,k) and previous triplets as shown in Listing 5.1. For the block i of a given step of the back substitution, we use the same properties.

Listing 5.1: YML Graph for the Gaussian elimination

```
1  par (k:= 0; blockcount-2)
2  do
3      par
4          wait(Aev[k][k][k]);
5          compute XMP_inversion(A[k][k],AInv[k]);
6          notify(Aev[k][k][k+1]);
7      //
8          par(i:= k+1; blockcount-1)
9          do
10             #A[k,i] = AInv[k] * A[k,i]
11             wait(Aev[k][k][k+1] and Aev[k][i][k]);
12             compute XMP_prodMat(AInv[k],A[k][i]);
13             notify(Aev[k][i][k+1]);
14         enddo
15     //
16         wait(Aev[k][k][k+1] and Bev[k][k]);
17         compute XMP_prodMV(AInv[k],B[k]);
18         notify(Bev[k][k+1]);
19     //
20         par (i:= k+1; blockcount-1)
21         do
22             par
23                 par (j:= k+1; blockcount-1)
24                 do
25                     #A[i,j] = A[i,j] - A[i,k] * A[k,j]
26                     wait(Aev[i][k][k] and Aev[i][j][k] and Aev[k][j][k+1]);
27                     compute XMP_prodDiff(A[i][k],A[k][j],A[i][j]);
28                     notify(Aev[i][j][k+1]);
29                 enddo
30             //
31                 #B[i] = B[i] - A[i,k] * B[k]
```

```
32                      wait(Bev[i][k] and Bev[k][k+1] and Aev[i][k][k]);
33                      compute XMP_prodDiffMV(A[i][k],B[k],B[i]);
34                      notify(Bev[i][k+1]);
35                  endpar
36              enddo
37          endpar
38  enddo
39
40  compute XMP_inversion(A[blockcount-1][blockcount-1],AInv[blockcount-1]);
41  compute XMP_prodMV(AInv[blockcount-1],B[blockcount-1]);
42  notify(Bev[blockcount-1][blockcount]);
43
44  par(k:= 1; blockcount-1)
45  do
46      par (i:= 0; blockcount-k-1)
47      do
48          wait(Bev[i][k+i] and Bev[blockcount-k][blockcount]
49                           and Aev[i][blockcount-k][i+1]);
50          compute XMP_prodDiffMV(A[i][blockcount-k],B[blockcount-k],B[i]);
51          notify(Bev[i][k+i+1]);
52      enddo
53  enddo
```

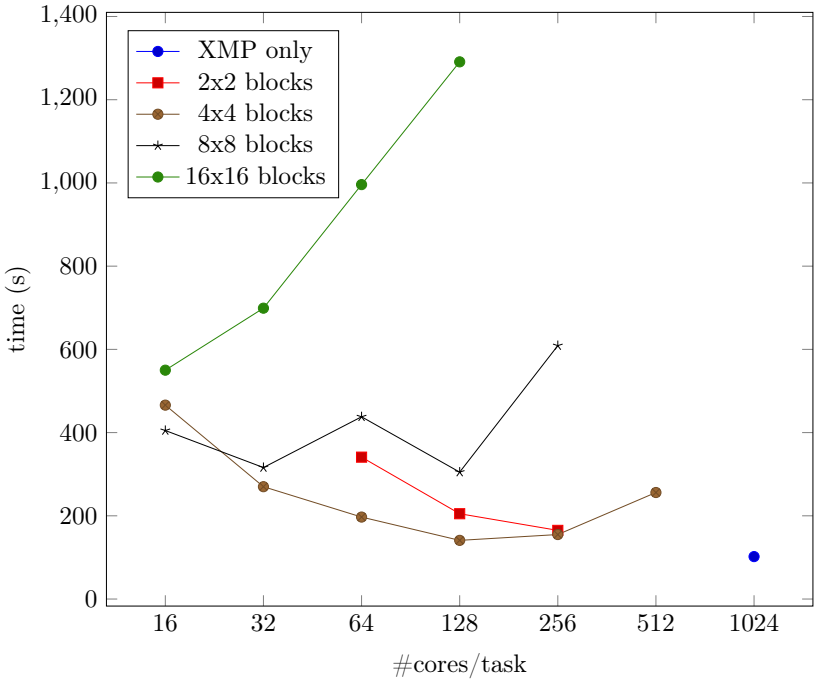
The XMP components that make the different operations are also written to take advantage of the XMP directives and use the processors allocated by YML to efficiently run the operations on the block matrices. The tasks perform operations on matrices like matrix product and inversion as well as operations on vectors like matrix vector product. Tasks on matrix products and matrix inversions have a comparable complexity compared to matrix vector products which have less operations. The *XMP_prodMat* component was implemented in Chapter 4. Listing 4.15 shows *XMP_prodMat* implementation component with XMP code and Listing 4.16 shows *XMP_prodMat* abstract component.

5.2.1.3 Resolution of a linear system of size 16384 on 1024 cores

This experiment consists in solving a dense linear system of size 16384 with the block-wise Gaussian elimination with YML/XMP. The matrix is already generated and each YML task has to load its data from the file system, makes the computations on the data then saves its results to the file system. The matrix is stored by blocks and each task makes an operation on blocks of the global matrix. YML expresses the operations on blocks while XMP is used to implement them. We experiment different numbers of blocks and numbers of cores per task. Only one process runs on each core. We used from 1×1 block to 16×16 blocks with tasks from 16 to 512 cores out of the 1024 cores of the K Computer allocated to each run. The time needed to solve the linear system without the generation of the matrix is evaluated in this study.

In the Figure 5.7, we observe the impact of the number of blocks and the number of cores per task on

Figure 5.7: Resolution of linear system using Gaussian elimination + back substitution (size of 16384) on the K computer

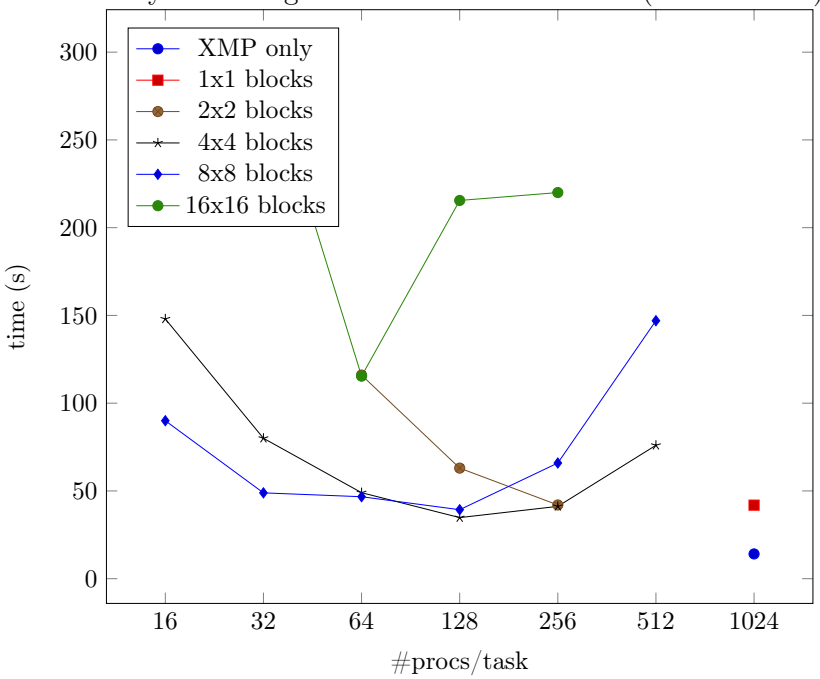


the execution time for the Gaussian elimination. We also observe the same kind of behavior for the two other applications using different methods to solve the same linear system : the Gauss-Jordan elimination in Figure 5.8 and the LU factorization in Figure 5.9. We also compare the YML/XMP application to a XMP implementation of the resolution of a linear system through the non-block Gaussian elimination. We reached the best time in YML/XMP with 4×4 blocks and 128 cores per task for 141s while the XMP code took 102s.

The number of blocks determines the number of tasks in the application. For $p \times p$ blocks, there are $\frac{p^3+3p^2+2p}{3}$ tasks in the Gaussian elimination application. Thus, the application needs to have enough tasks to contain enough parallelism without decreasing to significantly the execution time of each task. The number of blocks also influences the size of the block since the size of a block is $16384/p \times 16384/p$ values for the matrices and $16384/p$ values for the vectors in the case of a linear system of size 16384. If the number of blocks is large then the size of the block is small and the tasks on the blocks will be quick. In the opposite, a small number of blocks implies a large size of blocks so the task will be longer since it will need more operations. Moreover, if the application has too many tasks, the tasks execution will not compensate the overhead from YML.

The number of cores per task sets the number of YML workers since all the tasks use the same amount of cores and the total number of cores is fixed. Each worker launches a task at a time so there is the same number of parallel tasks as there is number of workers. For instance, we use 1024 cores in total and 128 cores per task so there are 8 parallel tasks maximum. The number of blocks needs to be high enough to use all the workers most of the time or there is unused compute power and the application will take more time. The number of cores per task also influences the speed of the execution of a task since a large number

Figure 5.8: Resolution of linear system using Gauss-Jordan elimination (size of 16384) on the K computer



of cores will have more compute power but will also introduce more communications between cores. On the opposite, a small number of cores will induce less communications but the task will take more time.

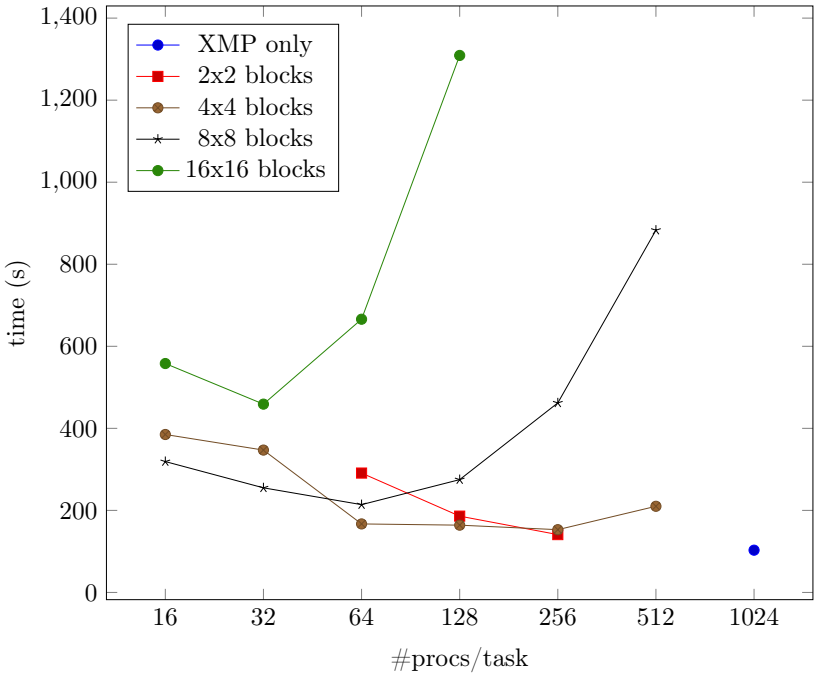
A compromise between the number of blocks and the number of cores per task is mandatory to obtain good performances. The number of blocks gives enough tasks so there is a great number of those tasks that can be run in parallel and determines the size of the data in the task. The number of cores per task gives the number of parallel workers and the execution time of each task. The good compromise gives enough parallel tasks so the workers are busy most of the time and the execution time of the tasks is balanced with the size of the data that need to be treated. It results in applications that use efficiently the available resources.

Table 5.1: Execution time (s) to solve a linear system on the K computer with 1024 cores

	YML/XMP			XMP
	blocks	cores/task	best time	
Gaussian elimination	4×4	512	141.02	102.82
Gauss-Jordan elimination	4×4	512	138.931	229.74
LU factorization	4×4	512	141.221	103.07

As shown in Table 5.1, XMP is faster than YML+XMP in solving the linear system without considering the method used. However, the Gauss-Jordan elimination elimination performs better in YML+XMP than in XMP whereas the Gaussian elimination and the LU factorization are faster in XMP than their counterpart in YML+XMP. Moreover, the Gauss-Jordan elimination is more compute and communication intensive compared to the LU factorization and the Gaussian elimination and thus, takes more time than the other two in XMP. Indeed, the operations of the Gauss-Jordan elimination are a bit different from the Gaussian elimination since the operations above the diagonal in the Gaussian elimination are also performed below

Figure 5.9: Resolution of linear system using LU factorization + forward and backward substitution (size of 16384) on the K computer



the diagonal in the Gauss-Jordan elimination. Hence, there is no resolution of triangular system and is directly solved. On the other hand, the Gauss-Jordan elimination in YML+XMP has a similar execution time compared to the Gaussian elimination and the LU factorization in YML+XMP. We think that the number of cores is too small thus the overhead from YML doesn't compensate for direct communications across all the cores in the Gaussian elimination and the LU factorization although it works well for the Gauss-Jordan elimination. Then, we tried to solve a bigger system on a higher number of cores.

5.2.1.4 Resolution of a linear system of size 32768 and 65536 on 8096 cores

We also made experiments on larger systems with more cores : 32768 and 65536 on 8096 cores. The Table 5.2 summarize the execution times obtained.

Table 5.2: Execution time (s) to solve a linear system on the K computer with 8096 cores

	Size	YML/XMP			XMP
		blocks	cores/task	best time	
Gaussian elimination	32768	4×4	512	276.8	508.5
	65536	8×8	512	690	2512
Gauss-Jordan elimination	32768	4×4	512	285.12	615.868
	65536	8×8	512	792.737	2970.393
LU factorization	32768	4×4	512	332.35	505.412
	65536	8×8	512	881.032	2306.163

In this experiment, we increased the size of the system and the number of cores. In this case, YML/XMP

performed better than XMP alone for the two different size for each application. This is mainly due to the fact that YML doesn't make any global communications like broadcast over all the cores allocated to the application. Indeed, each task runs on a subset of the resources and the communications between tasks are made through the data server. Although, the overhead of YML is noticeable (this will be discussed in the next section), YML+XMP runs faster than XMP alone on 8096 cores of the K Computer. Thus YML+XMP and the task programming languages may be a good solution to develop and execute complex applications on huge numbers of cores on large super-computers.

In this case, where the size of the problem and the number of computational resources has increased, the Gaussian elimination in YML+XMP is noticeably faster than the LU factorization and the Gauss-Jordan elimination in YML+XMP whereas the three of them were equivalent in the previous case. Moreover, the Gaussian elimination and the LU factorization have almost the same number of tasks but the Gaussian elimination executes faster. Indeed, the critical path of the Gaussian elimination is shorter than the critical path of the LU factorization due to the fact that, in the LU factorization, there is two triangular systems to solve while there is only one in the Gaussian elimination. The Gauss-Jordan elimination also performs better than the LU factorization although there is more computation but the critical path is even shorter than the critical path of the Gaussian elimination. We think that with greater sizes of problem and number of computational resources, the discrepancy between the LU factorization and the Gaussian elimination will increase whereas it will decrease between the Gauss-Jordan elimination and the Gaussian elimination. The Gauss-Jordan elimination may even execute faster than the Gaussian elimination at some point.

We compared YML/XMP to XMP for several cases on the K computer to evaluate the multi-level distributed/parallel programming associated to the studied programming paradigm on such supercomputer, even if the system of these machines does not yet propose smart scheduling. As we only used YML+XMP and XMP on the K Computer, we have to evaluate such programming on a more general cluster and compare it to more classic end-user programming. Then, in the next section, we compare YML/XMP and XMP to ScaLAPACK and our MPI implementation of the resolution of a dense linear system on Poincare, an IBM cluster. As MPI and ScaLAPACK are more optimized than our YML/XMP implementations, it is certain that they will obtain better performances. However, MPI is the current and most commonly used library to implement parallel and distributed applications. Therefore, we have to compare to it even if the purpose is to compare and analyze and not to be faster than MPI.

5.2.2 Experiments on Poincare, a cluster from La Maison de la Simulation

5.2.2.1 The cluster

The tests were performed on Poincare, the cluster of *La Maison de la Simulation* in France. It is an IBM cluster mainly composed of iDataPlex dx360 M4 servers, hosted at the CNRS supercomputer centre in Saclay, France. There is 92 compute nodes with 2 Sandy Bridge E5-2670 processors (8 cores each, so 16 cores per nodes) and 32 Go of memory. The file system is constituted of two parts : a replicated file system with the homes of the users and a scratch file system with a faster access from the nodes. The network is based on QLogic QDR InfiniBand.

5.2.2.2 The experiments

We performed the same tests on YML/XMP as on the K Computer. We also compared those results to ScaLAPACK, to our custom MPI implementations and 1×1 block in YML/XMP.

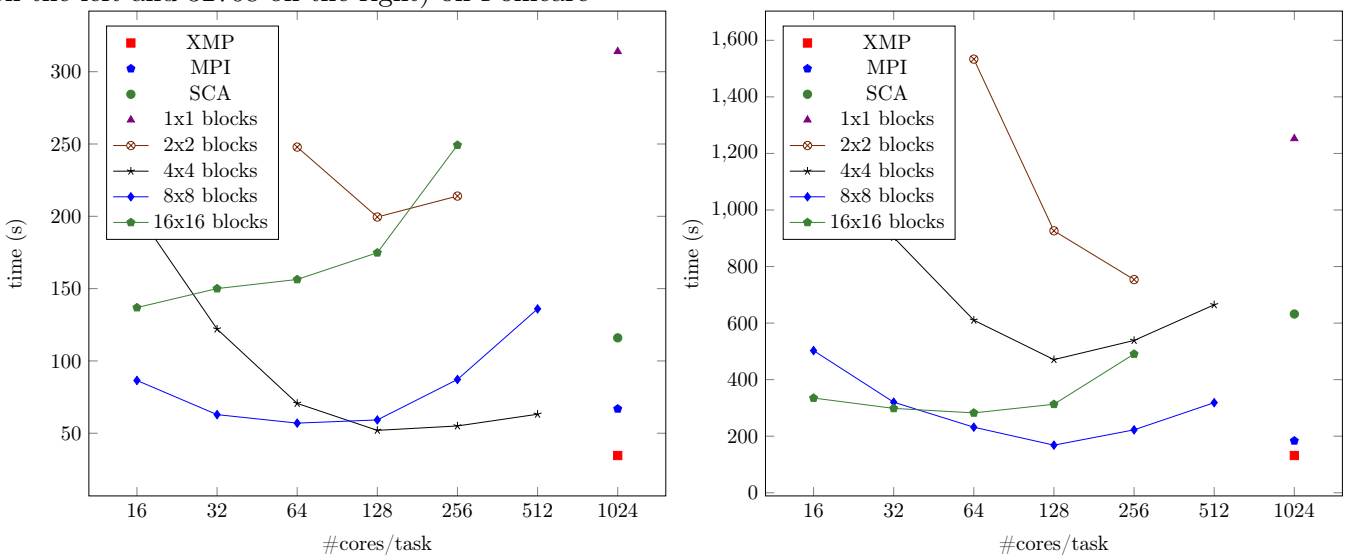
In the case of 1×1 block, there is only one assignment performed on a dense linear system with a XMP component. The component which runs the resolution of linear system is developed using XMP so we can evaluate the time that YML need to schedule the data, import it, run the XMP application then export the data. The data is loaded from the file system. This is the worst case for YML since it does not use the advantages of YML like scheduling and high grain parallelism.

We also compare the YML/XMP implementation of the Gaussian elimination, the Gauss-Jordan elimination and the Lu factorization to a XMP and a MPI implementations. Finally, we also compare to a ScaLAPACK implementation of the resolution of a dense linear system through LU factorization. In each case, we load the data from disk using MPIIO then we solve the linear system and save the result vector on disk using MPIIO.

In the MPI case, we used a cyclic distribution of the columns to keep a good load balancing in the cores. Moreover, we created a distributed cyclic array type in MPI in order to use it with MPIIO. This results in a matrix well stored by columns in the file. The cyclic distribution takes more time since it is easier to store the data without reordering them during the import/export.

5.2.2.3 The results

Figure 5.10: Resolution of linear system using Gaussian elimination + backward substitution (size of 16384 on the left and 32768 on the right) on Poincare



The Figure 5.10 shows the performances to solve a linear system of size 16384 and 32768 by the block Gaussian elimination and the block resolution of the resulting triangular system using YML/XMP. The best execution time for YML/XMP is obtained with 4×4 blocks and 128 cores per tasks for 51.9s for the size of 16768. The 1×1 block case with YML/XMP takes more time than the XMP only program (34.5s vs 313s).

We also obtain similar results with the LU factorization (see Figure 5.11) and the Gauss-Jordan elimination (see Figure 5.12).

The Table 5.3 gives a summary of the results obtained with the different programming paradigms. We have the values for two sizes of system 16384 and 32768. Moreover, for the MPI, XMP and ScaLAPACK cases, we also have the time spent in computation between parenthesis. The rest of the time is spent in IO to load and save the data as it is done for each component in YML/XMP.

Figure 5.11: Resolution of linear system using LU factorization + backward and forward substitution (size of 16384 on the left and 32768 on the right) on Poincare

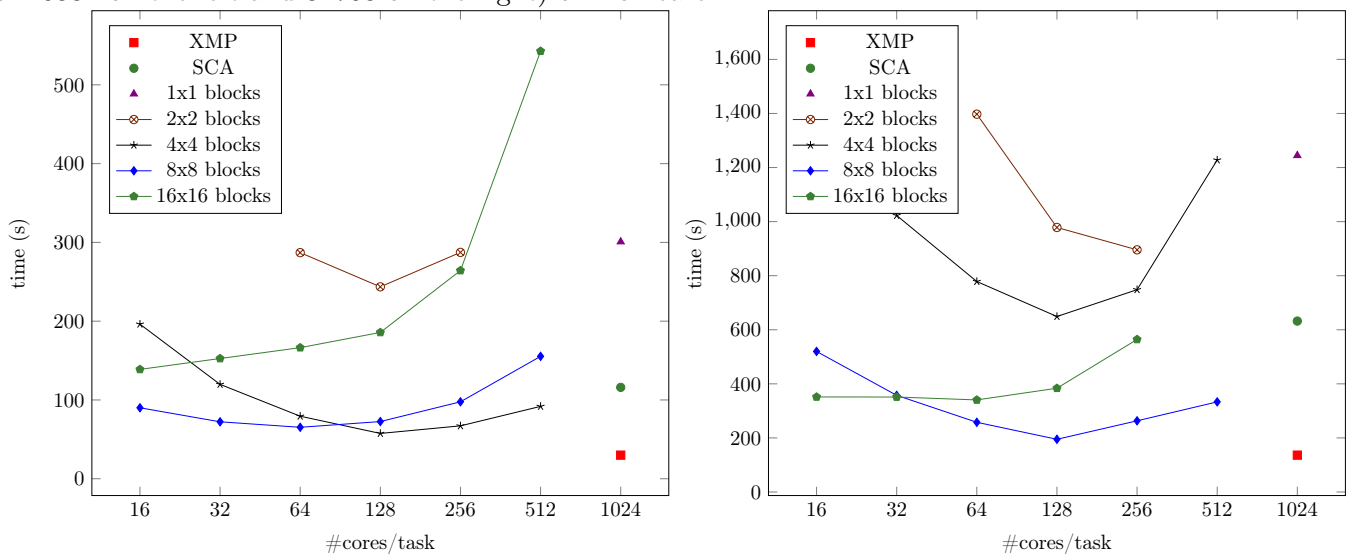
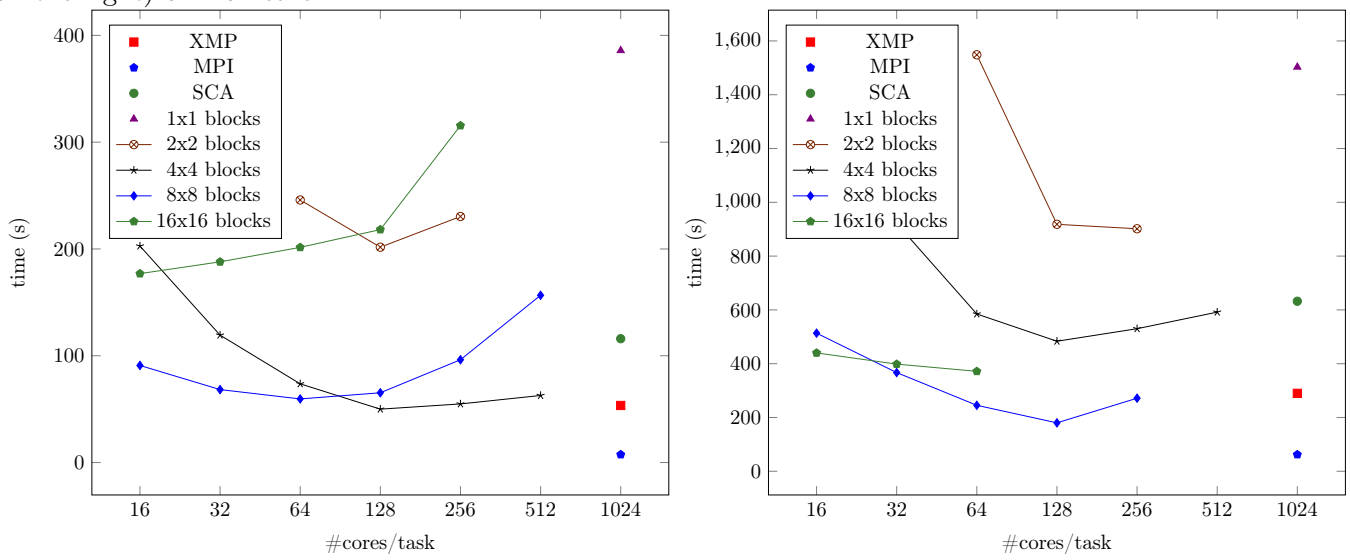


Figure 5.12: Resolution of linear system using Gauss-Jordan elimination (size of 16384 on the left and 32768 on the right) on Poincare



With the Gauss-Jordan method, YML/XMP runs faster than XMP on Poincare for the two sizes of systems and the difference increases with the size of the system. We observe the same effect of balance

between the number of cores per task and the number of blocks as we seen previously. The parallelism inside the Gauss-Jordan elimination is greater so the task parallel programming paradigm take advantage of this parallelism to execute the application. Moreover, there is no global communications in YML/XMP but they are widely used in XMP so the increase of operations from the Gaussian elimination to the Gauss-Jordan elimination is more impacting in XMP. Our MPI implementation is unexpectedly quite fast on a system of size 163824 but ScaLAPACK is more efficient in term of computation for the 32768 system. Finally, the Gaussian elimination performed almost two times better on XMP than the Gauss-Jordan elimination but the two methods got very close results on YML/XMP.

Table 5.3: Execution time (s) to solve a linear system

	Language	16384	32768
Gaussian elimination	YML/XMP (best case)	51.9	168.3
	YML/XMP (1×1 block)	313	1252
	MPI	66.9 (60.6)	183.7 (161.9)
	XMP	34.5 (32.7)	131.4 (129.6)
	ScaLAPACK	115.9 (18.5)	632.1 (55.6)
LU factorization	YML/XMP (best case)	57.508	194.8
	YML/XMP (1×1 block)	300.75	1244
	XMP	29.9 (29.4)	136.264 (135.6)
	ScaLAPACK	115.9 (18.5)	632.1 (55.6)
Gauss-Jordan elimination	YML/XMP (best case)	49.9	180
	YML/XMP (1×1 block)	385	1502
	MPI	7.4 (4.5)	62.1 (59)
	XMP	53.4 (52.8)	289.7 (289)
	ScaLAPACK	115.9 (18.5)	632.1 (55.6)

In this case, we globally see the same results although, the Gauss-Jordan elimination has more operations and uses more communications than the Gaussian elimination. We run faster with YML/XMP than XMP due the parallelism of the method being highly exploited by the parallel task paradigm and not using any global communications. The efficiency of the YML/XMP depends on the method implemented since the Gaussian elimination and the Gauss-Jordan elimination performed quite close on the two sizes of system while the Gauss-Jordan elimination has a significantly higher number of operations. The methods with a high parallelism are very suited to the parallel task programming paradigm and perform very well compared to XMP and ScaLAPACK. The format of the data stored in files by the data server (row-wise) wasn't suited for the distribution of the data used by ScaLAPACK so it didn't performed well as an application. The use of a better distribution of the data in ScaLAPACK and a data server making IO with the file system in YML/XMP may have shown better performances but YML/XMP and the parallel task programming paradigm show that, even with a high overhead for each task, we can get good performances.

5.2.2.4 A comparison between YML/XMP, XMP, MPI and ScaLAPACK

We can see a huge overhead in YML/XMP for 1×1 blocks compared to XMP alone in the Table 5.3. This is the case for the two sizes of system and the three applications. Thus, we can expect a pretty huge overhead from YML for each component. But, the most favorable case in YML/XMP solve the linear system in 51.9s whereas XMP took 34.5, MPI took 66.9s and ScaLAPACK took 115.9s. Although, from the 66.9s only 60.6s is spent on computing. The rest is spent in MPIIO to load and save the data. In the MPI case, we used a distributed cyclic array so the data in memory and the data on disk are not mapped the same way. This is why the IO take a bit more time in MPI than XMP since, the data are stored as they are in memory without reordering them.

Moreover, we didn't optimize our MPI implementation to the maximum. There is a margin of amelioration to get faster results with our MPI implementation. This implementation is close to the algorithm and use some properties of MPI while trying to reduce the communications between the cores. In this case, we observe that YML/XMP is slightly faster than our simple MPI implementation and the XMP is relatively close to YML/XMP (It takes 1.58 times more time in the case 16384 and 1.3 in the case 32768). We can expect that with larger size of systems, YML/XMP will be closer to XMP.

Finally, ScaLAPACK provides the best results if we only consider the computation time as shown in the Table 5.3. Indeed, most of the time is spent in the import of the matrix. This is due to the distribution of the data in memory. In this application, ScaLAPACK uses a cyclic distribution of the matrix in a two dimensional grid of cores while the matrix is stored as huge vector representing a row-wise matrix. Then the values that have to be put in a core are scattered trough the file without any value close to another one. Thus the import of the matrix is very costly. ScaLAPACK is the fastest in the of computation time but the import of the matrix from the file system is very costly. The distribution of the data in this application using ScaLAPACK is not suited for IO.

With a good compromise between the number of blocks and the number of cores per task, we got results relatively close to XMP and MPI. Moreover, we can expect to get closer results with greater sizes of systems and even get better results at some point. YML allows more local communications than XMP and MPI. In YML, the communications are only made on a subset of cores while, in XMP, communications are made across all the cores. Furthermore, YML has a scheduler that manage the tasks and the data migrations between the tasks in order to optimize them. They can optimized further with YML asynchronous communications between tasks. Finally, MPI applications are very well adapted to the current systems so the performances of these applications are great if the application is well implemented.

5.2.3 Prediction of the optimal parameters

The prediction of the optimal values for the number of blocks (size of the block) and the number of processes per task (number of parallel tasks) is not a deterministic problem since it depends on the machine and the available resources. On a given machine, the job manager will allow the unused resources which differ each time a job is submitted. Thus, the communications time between two cores will depend on their distance. The execution time of a task also depends on the loading of the machine since each user can use the network. Moreover, the computation time has to be higher than the scheduling time so it is worth scheduling the tasks. The execution time of the application will also depend on the scheduling strategies since it will change

the order of execution of the tasks.

The parameters are highly related. Indeed, the size of a block depends on the number of blocks which influences the total number of tasks. Furthermore, the size of the block will define the number operation in the task while the number of communications will depend on the number of processes and the size of the block.

Hence, the prediction of the optimal values is difficult but, for a given machine, the machine learning may be able to give an estimation of the value of the parameters. Besides, in the future, middlewares and schedulers will improve and integrate more efficiently with task based programming models and tools. Therefore, the prediction of these parameters will be easier, more efficient and accurate.

5.2.4 Results summary

We experimented YML/XMP, a programming paradigm based on a graph of parallel and distributed tasks, on the cluster of the *Maison de la Simulation* in France and the K computer in Japan. We solved a dense linear system of different size with 1024 and 8096 cores. We obtained results relatively close to XMP and MPI even though that YML/XMP on Poincare uses the file system to implement the asynchronous data migrations, the YML scheduler is not very efficient at this point that the overhead induced by YML is substantial. On the K computer, with a larger number of cores and and size of the system, YML/XMP ran faster than XMP. ScaLAPACK is slower because of the special distribution of the data used within it. In this application, we need to change the definition of the distribution of the data in order to reduce the import time. We can expect the same outcome on the K Computer if we run the ScaLAPACK application on it; the computation time would be excellent but the IO to import the data will slow down the application. As we are using ScaLAPACK, a YML task which make a call to the ScaLAPACK functions will not be efficient since YML load the data from disk and the distribution of the data in ScaLAPACK makes the IO difficult. Finally, we compared the Gaussian elimination to the Gauss-Jordan elimination and we find out that the Gauss-Jordan elimination has execution time close to the Gaussian elimination although the Gauss-Jordan elimination has more operations. So the parallel task programming paradigm seems well suited to execute methods with high parallelism. Thus a programming paradigm using a graph of parallel tasks where each task can also be parallel may be interesting to create efficient parallel and distributed applications on exascale super-computers.

We can outline the fact that a programming paradigm based on a graph of parallel and distributed tasks can obtain better performances on a huge amount of cores. Future supercomputers and post-petascale platforms would propose middle-ware and systems with smarter schedulers [121] and dedicated I/O systems [122] which will allow some efficient data persistence and anticipation of data migrations. In those machines, programming paradigms such as YML/XMP would be well-adapted and efficient. As the increasing number of nodes on such distributed and parallel architectures, with large latency for communications between farthest nodes, will penalize global communications, our approach which avoid global operations along all the nodes but generate operations just inside subsets of nodes, would be well-adapted.

5.3 Several Graph Based Language to Compute the LU factorization

5.3.1 Experiments details

We performed performance tests on up to 64 nodes of Poincare with the LU factorization implemented via MPI, ScaLAPACK, XMP, YML+XMP, HPX, PaRSEC and Regent. We used several sizes of matrices : 16384×16384, 32768×32768 and 49512×49512. 16384×16384 is the largest size we can use to perform the tests on one node since YML+XMP cannot perform the LU factorization with greater sizes of matrices on one node.

In HPX, PaRSEC and Regent, the performances depends on the number of blocks in each dimension (thus, the size of the blocks). We used several values for the number of blocks. Table 5.4, Table 5.5 and Table 5.6 show the block parameters which obtained the fastest execution time for each size of matrix. The execution times shown here are the case in which we obtained the fastest time for each number of nodes. We performed those test several times and computed the execution times mean of the same case. We compare the results of the task-based programming languages to those obtained with ScaLAPACK. We also compare them to our MPI and XMP implementations. Tests were run on several number of nodes in order to extract strong scaling information which are discussed in Section 5.3.3. We used 1, 2, 4, 8, 16, 32 and 64 nodes to factorize the 16384×16384 values matrix. Then, we used 4, 8, 16, 32 and 64 nodes for the 32768×32768 values matrix. And finally, we used 8, 16, 32 and 64 nodes for the 49512×49512 values matrix.

Table 5.4: Number of blocks for the fastest case on a 16384×16384 matrix with number of processes per tasks between parenthesis

	1	2	4	8	16	32	64
HPX	$90^2(1)$	$45^2(1)$	$80^2(1)$	$45^2(1)$	$45^2(1)$	$55^2(1)$	$55^2(1)$
PaRSEC	$150^2(1)$	$200^2(1)$	$70^2(1)$	$120^2(1)$	$210^2(1)$	$240^2(1)$	$250^2(1)$
Regent	$50^2(1)$	$50^2(1)$	$50^2(1)$	$35^2(1)$	$40^2(1)$	$35^2(1)$	$30^2(1)$
YML+XMP	$4^2(8)$	$8^2(8)$	$8^2(16)$	$8^2(32)$	$4^2(128)$	$4^2(128)$	$4^2(128)$

Table 5.5: Number of blocks for the fastest case on a 32768×32768 matrix with number of processes per tasks between parenthesis

	4	8	16	32	64
HPX	$90^2(1)$	$90^2(1)$	$90^2(1)$	$75^2(1)$	$81^2(1)$
PaRSEC	$70^2(1)$	$120^2(1)$	$270^2(1)$	$380^2(1)$	$420^2(1)$
Regent	$70^2(1)$	$70^2(1)$	$60^2(1)$	$50^2(1)$	$50^2(1)$
YML+XMP	$1^2(64)$	$4^2(64)$	$8^2(32)$	$8^2(128)$	$8^2(128)$

Table 5.6: Number of blocks for the fastest case on a 49512×49512 matrix with number of processes per tasks between parenthesis

	8	16	32	64
HPX	$148^2(1)$	$148^2(1)$	$148^2(1)$	$145^2(1)$
PaRSEC	$250^2(1)$	$250^2(1)$	$400^2(1)$	$420^2(1)$
Regent	$70^2(1)$	$70^2(1)$	$70^2(1)$	$70^2(1)$
YML+XMP	$1^2(128)$	$2^2(128)$	$4^2(128)$	$8^2(128)$

Our MPI application is MPI-only so we used MPI support for shared memory and used one MPI rank per core i.e. 16 processes per core.

ScaLAPACK has a MPI only distributed implementation so it is run with one MPI rank process per core.

Our XMP implementation only uses pure XMP directives which are converted to MPI calls. It is launched as a MPI only application with one MPI rank process per core.

Regent is a compiler that translates a Lua based code into Legion. Regent applications are launched by passing the MPI command to Regent launcher which will compile and run the application. It creates a Legion worker on each node. Each one of them spawns a process to manage the local tasks, a process to manage data and a process to execute the tasks by default. Then, the user has to specify the number of processes on which the tasks will be executed. We used 14 processes on each node to execute the tasks.

To launch our HPX application, we used *mpirun* to execute one instance of HPX runtime on each node. Then, HPX is able to infer the node configuration. It spawns a worker process on each core of the node and tasks are run as light-weight threads on those processes. HPX is able to detect that there is two sockets on the node and manages them internally.

PaRSEC applications were launched with one MPI rank per core.

YML scheduler is launched with MPI on one core (the first one in the machine file) which launch XMP tasks with *MPI_Comm_spawn* routine on the leftover cores available.

5.3.2 Performances

Fig. 5.13 shows the performances obtained for the LU factorization with HPX, MPI, PaRSEC, Regent, ScaLAPACK, XMP and YML+XMP on three sizes of matrices 16384×16384 (top), 32768×32768 (middle) and 49512×49512 (bottom).

On a 16384×16384 matrix, MPI is close to XMP on a small amount of nodes. When the number of node increases, MPI becomes significantly faster than XMP. Indeed, Fig. 5.13 middle and bottom charts show that MPI is significantly faster than XMP for each number of node. MPI and XMP applications share the same algorithm and a similar implementation but expressed with two different models. This may be due to an overhead from the PGAS description and access of the data in XMP compared to MPI.

Regent, HPX and PaRSEC are relatively close to one another on a small number of cores. However, we can outline tendencies. PaRSEC is faster than HPX on the lower number of node then HPX becomes

faster when the number of node increases. It also seems that when the size of the matrix increases, HPX and PaRSEC performances are becoming closer and that HPX becomes faster than PaRSEC on the larger number of nodes. Indeed, HPX becomes faster than PaRSEC after 4 nodes for a matrix of size 16384×16384, after 16 nodes for a matrix of size 32768×32768 and after 64 nodes 49512×49512. For the later value, the difference between the two is very small (330s vs 331) so we expect HPX to become significantly faster for this size of matrix with a greater number of nodes.

Regent is a little bit behind HPX and PaRSEC on each number of nodes and size of matrices except for 2 and 4 nodes on a 16384×16384 matrix where Regent is very efficient. We can also notice that Regent is taking more time on 64 nodes than on 32. This may be related to the fact that Regent does not seem to be able to manage a large number of tasks on a large number of nodes since the number of sub-matrices is decreasing when the number of cores is increasing as Table 5.4, Table 5.5 and Table 5.6 are showing. However, other task based languages obtain better results when the number of sub-matrices they process increase with the number of cores. It creates more task and parallelism so that the runtime can use the resources most efficiently.

The YML+XMP applications are the slowest compared to the the applications implemented with the other models. However, YML+XMP is the only model where tasks are also parallel and distributed. Moreover, it also uses the file system to perform the communications between the tasks so the communications between tasks are not efficient.

Our last application uses the ScaLAPACK library to compute the LU factorization. It performs very well on large number of nodes but HPX, PaRSEC and Regent are faster on lower number of cores for each size of matrix. They are not using the same block based algorithm but ScaLAPACK is using a tiled algorithm that makes computations on rows and columns of the matrix [68]. Therefore, it is an interesting comparison to our block-based algorithms where the operations on the blocks are implemented with tasks. For a 16384×16384 matrix ScaLAPACK and HPX are close on 64 nodes but ScaLAPACK is faster for greater size of matrices. This may be due to the cyclic distribution of data in ScaLAPACK which induces a different communication pattern very efficient on this kind of machine and algorithm.

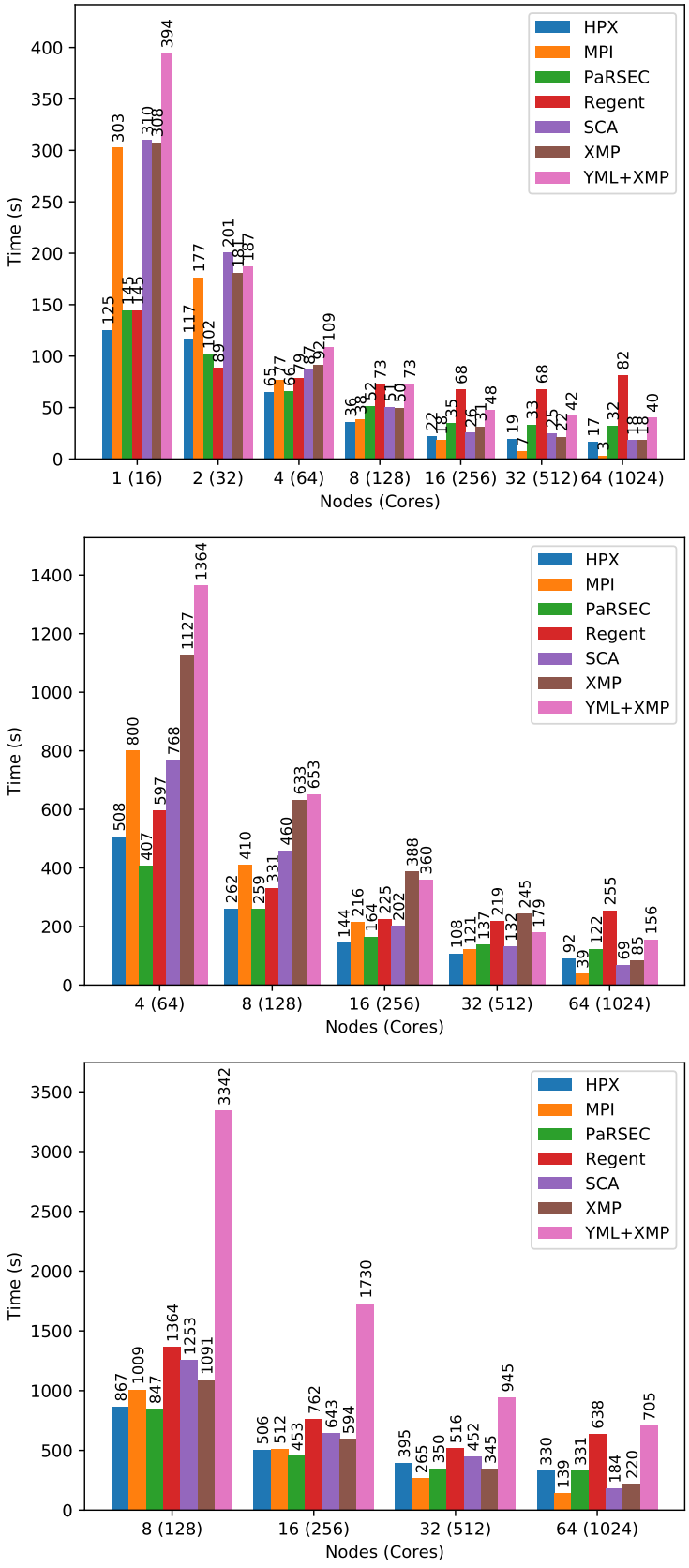


Figure 5.13: Execution times obtained with the block-based LU factorization implemented with several task-based programming models on a 16384 × 16384 matrix (top), a 32768 × 32768 matrix (middle) and a 49512 × 49512 matrix (bottom)

5.3.3 Strong scaling

Fig. 5.14 shows the speed-up extracted from the performances values from Fig. 5.13 for HPX, MPI, PaRSEC, Regent, ScaLAPACK, XMP and YML+XMP on three sizes of matrices 16384×16384 (top), 32768×32768 (middle) and 49512×49512 (bottom). It corresponds to the ratio t_S/t_N where t_N is the execution time for N nodes and t_F is the execution time of the first number of nodes considered in the test. In the top chart of Fig. 5.14, t_F is t_1 since the experiments start with 1 node. In the middle (bottom) chart, t_F corresponds to 4 (8). It translates how efficiently we are managing the addition of more resources to solve the same problem.

Our MPI regular LU factorization is scaling very well as we can see on the charts. These results are expected since the current systems are well optimized to work with MPI applications. It even exceeds the ideal speed-up with matrices of size 16384×16384 (Fig. 5.14 top chart) and 32768×32768 (Fig. 5.14 middle chart). We think that it may be due to processes not having enough computations to do on 32 and 64 nodes matrices of size 16384×16384. Indeed, when increasing the size of the matrix to 32768×32768, the strong scalability for our MPI application seems more reasonable. The same situation occurs for 64 nodes when increasing the size of the matrix from 32768×32768 to 49512×49512.

Our task based applications obtain better scalability with the increase of the data size and the number of tasks processed by the applications. Table 5.4, Table 5.5 and Table 5.6 show that the number of tasks for a given number of nodes increases with the size of the matrix for each task based programming model. It produces more parallelism and opportunities to optimize the scheduling of the tasks and improve the use of the computing resources.

Regent strong scalability decreases from 32 to 64 nodes for each size of matrix. We expect its strong scalability to decrease even more with the increase of the number of cores.

Our HPX application is scaling better than our PaRSEC application with matrices of size 16384×16384 and 32768×32768. It corresponds to the results we obtained in the previous section. We can also see that PaRSEC and HPX are very close with matrices of size 49512×49512 and that HPX is exceeding PaRSEC after 32 nodes. It seems that HPX may have a better scalability than PaRSEC on more than 64 nodes with matrices of size 49512×49512 if more nodes were available.

Finally, our YML+XMP application has the best strong scalability compared to the other task-based programming models. As YML+XMP rely on the file system to pass data from one task to another, the performances depends on the efficiency of the file system and its capacity to manage IOs. Therefore, we think that this programming model will be well adapted to larger machines with a distributed system and integrated schedulers.

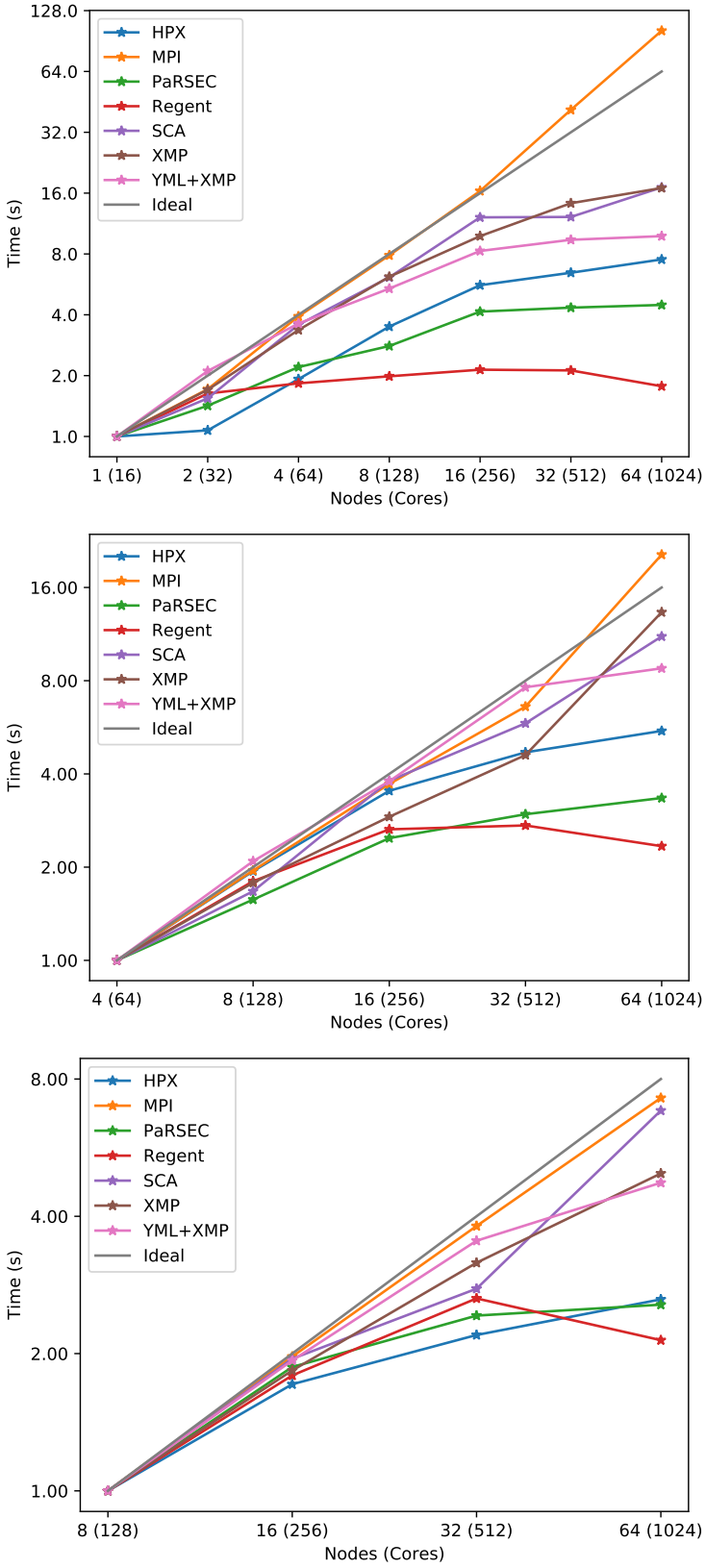


Figure 5.14: Speed-ups obtained with the block-based LU factorization implemented with several task-based programming models on a 16384×16384 matrix (top), a 32768×32768 matrix (middle) and a 49512×49512 matrix (bottom) - log_2 scale for the y-axis

5.3.4 Results summary

To conclude, MPI has the best results and scalability on 64 nodes but the application does not use partial pivoting so it is not comparable to ScaLAPACK. It is also faster than XMP since MPI routines are highly optimized. Even though, XMP translates its directives into MPI code, the PGAS model used in XMP is not as efficient as using directly MPI.

ScaLAPACK is faster than the applications implemented with the task based programming models but it uses very efficient kernel routines to perform operations internally whereas we are using unoptimized routines.

In term of task based programming models where we implemented everything (task dependencies and tasks themselves) with the programming model, HPX is the most efficient on 64 nodes. However, PaRSEC also shows interesting performances in specific circumstances. Furthermore, Regent applications are not faster while increasing the number of nodes from 32 to 64. We think that the difference of performances between those programming models comes from their ability to manage the number of tasks, the dependencies between the tasks, tasks workload and the data migrations between nodes. Indeed, Regent performs best with a smaller amount of tasks than HPX and PaRSEC but its performances are behind them (see Table 5.4, Table 5.5 and Table 5.6). HPX and PaRSEC are performing better with a higher number of smaller tasks. They seems to distribute very efficiently the tasks on the resources and optimize the data migrations between the nodes. Moreover, we used the default mapper for data and tasks provided by Regent and Legion. They may not be efficient enough in our case. Implementing a new mapper may improve the performances of our Regent application.

Finally, YML+XMP is the only programming model using tasks which are also distributed but the communications through the file system decrease its efficiency. YML+XMP performances may not be impressive on this number of nodes but with the increase of the number of nodes and its strong scalability higher than the other task-based programming languages, YML+XMP could be able to perform better than the other programming models on a very large scale as already experienced on the K computer [114]. Moreover, changing the use of the file system to make the communications between the tasks to in-memory communications could improve the performances even more.

5.4 Synthesis and Perspectives

In this chapter, we introduced the dependency graph of three block based algorithms to solve linear systems : the block Gaussian elimination, the block Gauss-Jordan elimination and the block LU factorization. Then we used the dependency graphs to implement corresponding task based applications with YML+XMP in which dependencies between tasks are expressed with a graph. Each task is parallel and distributed and correspond to a operation on sub-matrices in the block algorithms. We also implemented a XMP, a MPI and a ScaLAPACK version of these applications. Then, we compared the efficiency of the the YML+XMP and XMP applications on the 1024 cores of the petascale K Computer. We showed that XMP is faster than YML+XMP in this case since the overhead from YML does not compensate for direct communications across all the cores in the Gaussian elimination and the LU factorization although it works well for the Gauss-Jordan elimination. Therefore, we executed our applications on a larger amount of the K Computer resources. We ran the applications with 8096 cores and larger system sizes. We showed that the XMP applications are running

slower than the YML+XMP application for 32768 × 32768 and 65536 × 65536 dense linear systems on 8096 cores of the K Computer. We also showed that the Gaussian elimination in YML+XMP is noticeably faster than the LU factorization and the Gauss-Jordan elimination in YML+XMP whereas the three of them were equivalent with smaller sizes and resources. Moreover, the Gaussian elimination and the LU factorization have almost the same number of tasks but the Gaussian elimination executes faster since its critical path is shorter than the critical path of the LU factorization. This is due to the fact that, in the LU factorization, there is two triangular systems to solve while there is only one in the Gaussian elimination. This can also be seen on the dependency graphs introduced in the start of the chapter.

Furthermore, we implemented these applications with MPI and ScaLAPACK since MPI is the commonly used library to implement parallel and distributed applications and ScaLAPACK is an efficient implementation of the common dense linear operations. We showed that YML+XMP for 1×1 block has a large overhead when compared to the same application with XMP alone. Thus, YML induces a large overhead for each task. We showed that, with a good compromise between the number of blocks and the number of cores per task, we can get results relatively close to XMP and MPI with YML+XMP and even be faster in some cases. This compromise is difficult to obtain as it can only be found though trials and errors with the current schedulers. Moreover, we can expect to get closer results with greater sizes of systems and even get better results with more efficient storage systems since YML+XMP heavily rely on it. In YML, the communications are only made on a subset of cores while, in XMP, communications are made across all the cores. Furthermore, YML has a scheduler that manage the tasks and the data migrations between the tasks in order to optimize them. They can optimized further with YML asynchronous communications between tasks. MPI applications are very well adapted to the current systems so applications well implemented with MPI can expect good performances with a relatively low development cost. We also showed that, due to the cyclic data distribution in ScaLAPACK, the IOs involved in the reading and writing of the matrix is not as efficient as with MPI and most of the execution time is spent in IOs.

Finally, we focused on the block based LU factorization and implemented it with several task based programming models. We made task based applications with YML+XMP, HPX, PaRSEC and Regent. We also implemented MPI, XMP and ScaLAPACK applications as comparison. We performed strong scaling experiments with each application up to 64 nodes for 16384 × 16384, 32768 × 32768 and 49512 × 49512 matrices. We showed that MPI has the fastest execution times and the best scalability on 64 nodes and that, although, XMP translates its directives into MPI code, the PGAS model used in XMP is not as efficient as using directly MPI whereas HPX is the most efficient task based programming model on 64 nodes. However, PaRSEC also shows interesting performances in some cases and Regent applications execution time increase while increasing the number of nodes from 32 to 64 which should not be the case. This is due to their ability to manage the number of tasks, the dependencies between the tasks, tasks workload and the data migrations between nodes. Regent poor performances may also be due to Legion default mapper used by Regent since the performances of this mapper are not very efficient. We showed that HPX and PaRSEC are performing better with a higher number of smaller tasks. YML+XMP execution times are higher than the other task based programming models due to the use of the file system but its strong scaling speedups are higher so YML+XMP could obtain better performances on a larger number of resources as we showed on the K Computer.

Besides, the installation of the different task based programming models is not trivial. They require the

knowledge of the cluster or supercomputer on which the programming models will be installed as well as the knowledge of the installation process of the programming models themselves. For these experiments, we installed ScaLAPACK, XMP, YML, HPX, PaRSEC and Regent as well as their missing dependencies. Although, the installation process was successful on the K Computer and Poincare, there were issues. Indeed, Regent and YML+XMP have multiple dependencies that are not easy to install and do not provide enough installation help in order to install the softwares without help from the developer teams.

Moreover, integrating these task based programming models into an application also arises issues during the applications development and may limit what is possible to achieve. For instance, Legion default mapper used by Regent may reduce the performances obtain with the applications. Thus, the user has to use another one or implements his/her own. Another issue is the YML+XMP type that holds dense matrices does not support non divisible values and greatly limits the number of blocks that can be used.

Due to Regent difficulty of installation and the poor performances observed while scaling from 32 nodes to 64, we choose to stop using Regent to implement our applications. We still use HPX and PaRSEC as task based programming models due to their interesting properties and their easier use. We also use YML+XMP in our following applications with new data types to hold data so it bypasses the limitations of the dense matrix type.

This work could be completed with applications implemented with other task based programming models such as X10, Chapel, Uintah or Charm++. Improving the YML+XMP dense matrix type can also be considered since it would allow more flexibility for YML+XMP dense linear algebra applications and allow the use of parameter values that could lead to better results. These experiments could also be reproduced on other cluster or supercomputers. Furthermore, we introduced sparse linear algebra methods in Chapter 3. In the following chapter, the sparse matrix vector product will be implemented with the selected task based programming models. Then, we will perform numerical experiments on several clusters and petascale supercomputers as well as discuss the results we obtained.

Chapter 6

Task-Based, Parallel and Distributed Sparse Linear Algebra Applications

The sparse matrix vector product with several matrix storage format and matrix distribution across the computing resources is the basic algorithm considered in this chapter. Sequences of sparse matrix products are important and largely used in several applications such as iterative methods and neural network training. However, two executions of the sparse matrix vector are enough to outline the algorithmic issues without having to perform too much computations. Therefore, the sparse operation $A(Ax+x)$ is considered as it uses two times the sparse matrix vector product. We implement the sparse matrix vector algorithms previously introduced in Chapter 3 with the selected task based programming models and perform numerical experiments on several clusters and supercomputers. Finally, we discuss the results obtained in the experiments.

6.1 Task-Based Methods for Parallel and Distributed Algorithms

The sparse matrix vector product is executed in a distributed and parallel environment which means that computations can be run in parallel and data may not be accessed directly since they can be stored on a different resource than where they are needed. The data can be moved through the network but it can be costly especially in very large networks. The goal is to implement a parallel, distributed and task based sparse matrix vector product taking advantage of this set up while getting the best execution time and reducing the communications. In the sparse matrix vector product case, the distributed and parallel algorithms (both regular and task based) depends on the storage formats used and the distribution of the sparse matrix on the distributed resources. There is several way of distributing the matrix : either by splitting the columns or the rows. A combination of the two previous way of distribution, the block distribution is also considered here. They are the main way of distributing a matrix although other ways exist like a cyclic distribution where the consecutive values are put alternatively on a distributed resource. The sub-matrices are stored in memory with a sparse storage format.

6.1.1 Data Distribution

We consider three ways of distributing the data : a distribution by rows, by columns and by blocks. The distribution by rows and by columns are special cases of the distribution by blocks where the is only distri-

bution across one row for the distribution by columns and where the is only distribution across one column for the distribution by rows. The blocks are sub-matrices kept on memory with a sparse storage format. In this section, we are considering different ways of distributing sparse matrices. However, we are not looking into the load balancing issues that may arise if some blocks are more populated than others.

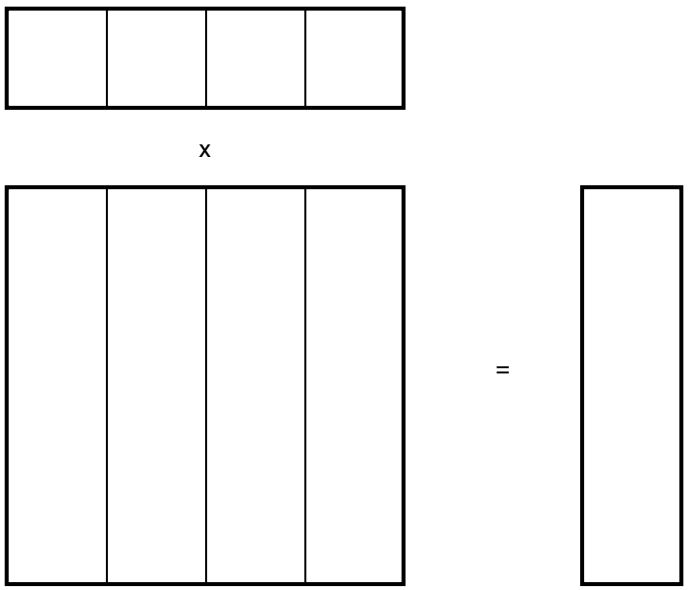


Figure 6.1: Distribution of the matrix by columns in which the sub-columns are compressed

The distribution by columns consists in splitting the rows and keeping the values of the same column in the same computing resource. These columns are stored in a sparse storage format locally. The input vector can also be split since only the rows of the vector corresponding to the columns stored in a computing resource are necessary for the matrix vector product. The Figure 6.1 shows a matrix distributed by rows to make a matrix product vector as well as the necessary input vector and the result vector. The result vector has the same size as the number of rows in the sub-matrices. In this case, it corresponds to the full matrix number of rows. However, each computing resource contains a part of the global solution of the matrix vector product. To obtain the global result, all the distributed results have to be summed.

The distribution by rows consists in splitting the columns and keeping the values of the same row in the same computing resource. These rows are stored in a sparse storage format locally. In this case, the input vector cannot be split since a complete row of the matrix may be stored in each computing resource. In the sparse case, the rows may not be complete on each process but it can not be known in advance. The input vector is duplicated in each computing resource. The result vector has the same size as the number of rows in the sub-matrices. Here, it corresponds to the number of rows in each corresponding sub-matrices. However, each computing resource contains a sub-vector. To obtain the global result, all the distributed results have to be gathered (which means that the global vector has just to be reconstructed without any computations).

The block distribution consists in both a distribution by rows and distribution by columns. The matrix in split in a 2D grid way. The $Nx \times Ny$ matrix is split in $Ngx \times Ngy$ sub-matrices. The sub-matrices are stored in a sparse storage format locally. In this case, the input vector is split across the columns of the matrix and the sub-vectors are duplicated on the sub-rows of the same column. The Figure 6.2 shows a

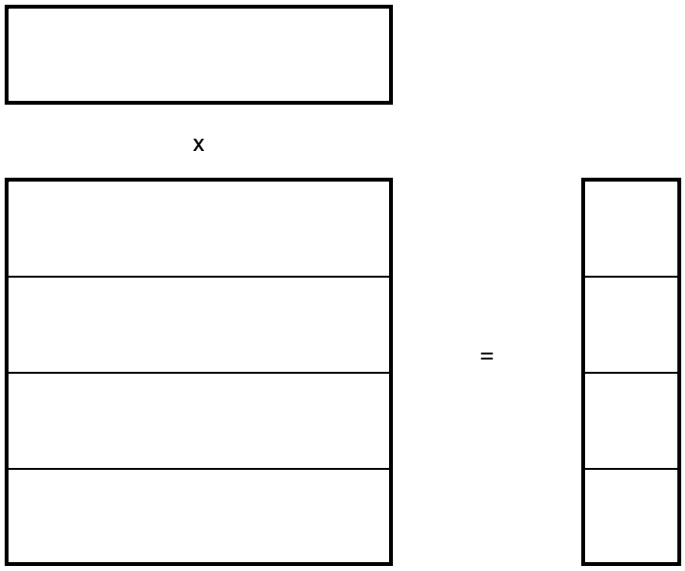


Figure 6.2: Distribution of the matrix by rows in which the sub-rows are compressed

matrix distributed by blocks to perform a matrix product vector as well as the necessary input vector and the result vector with $Ngx = 4$ and $Ngy = 4$. The result vector has the same size as the number of rows in the sub-matrices of the corresponding row. However, each computing resource contains a part of a sub-vector. To obtain the global result, all the distributed results of the same row have to be reduced then each rows have to be gathered if the full result vector is needed in one place.

The data distribution determines which part of the global matrix iq stored on a given computing resource. The Section 6.1.2 points out how these data is used to perform the matrix vector product.

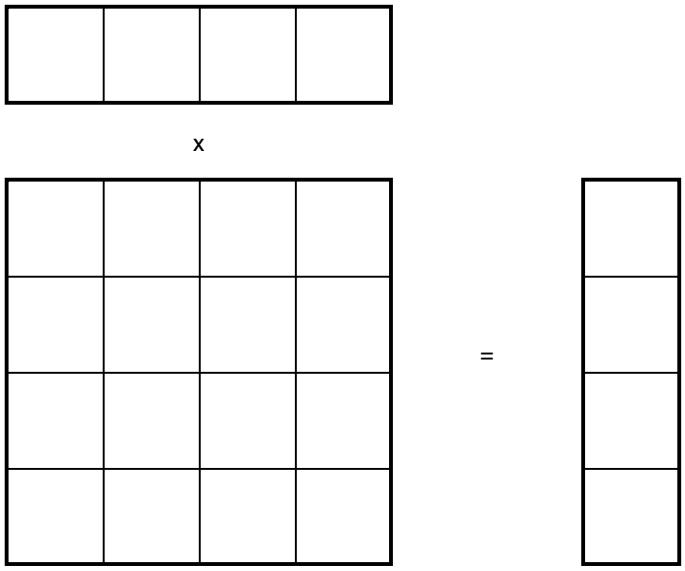


Figure 6.3: Distribution of the matrix by rows and columns in which the blocks are compressed matrices

6.1.2 Tasks Definition

In this section, we consider that the matrix is divided according to one of the distribution described previously (except for the COO storage format). Each sub-matrix is used as input of the matrix vector product tasks defined in the following algorithms. We define data structures to store the matrices and use them in the tasks. Finally, the algorithms for the tasks is designed.

Algorithm 20: COO format data structure and matrix vector product

```
struct {
    Array row, col, val
} MatrixCOO

Task spmv_coo()
    Data: m : MatrixCOO, v : Array
    Result: r : Array
    For i from 0 to m.val.size() - 1 do
        r[m.row[i]] += m.val[i] * v[m.col[i]]
```

Algorithm 20 defines a data structure to store the sub-matrices stored in the COO format. It also shows the algorithm to make a matrix vector product on the sub-matrix given the proper input vector. The data structure contains the three arrays necessary to store the matrix in the COO storage format. In this implementation of the matrix vector product with the COO format, the sub-matrix does not have to be sorted e.g. this implementation does not expect the coordinates of the values of the matrix to be in a given range. Therefore, this implementation can process any value in any position in any sub-matrix (which is not the case with the other storage formats). However, we cannot deduce the range of position in the full input

vector which is effectively used in the task as well as the range of position for the output vector since there is no restrictions on the range of coordinates of the values in the matrix. Thus, this task also expect the full vector as input for the tasks (independently of the data distribution described) and output a full vector.

It can allow a better load balancing at the cost of a full vector in input and output which change how the output vector is processed to construct the full output vector. This is discussed in Section 6.1.3.

Algorithm 21: SCOO format data structure and matrix vector product

```
struct {
    Array row, col, val
    Integer fr, fc
} MatrixSCOO

Task spmv_scoo()
    Data: m : MatrixCOO, v : Array
    Result: r : Array
    For i from 0 to m.val.size() - 1 do
        r[m.row[i] - m.fr] += m.val[i] * v[m.col[i] - m.fc]
```

Algorithm 21 defines a data structure to store the sub-matrices stored in the SCOO format in which we expect a given range of coordinates for the values contrary to the COO format. It also shows the algorithm to make a matrix vector product on the sub-matrix given the proper input vector. The data structure contains the three arrays necessary to store the matrix in the SCOO storage format as well as useful information about the position of the sub-matrix in the global matrix. This information is the first row (fr) and the first column (fc) possibles in the sub-matrix (It may not correspond to the actual values stored in the sub-matrix since it corresponds to the lower boundaries of the possible coordinates in the sub-matrix for the algorithm to work properly). The algorithm for the SCOO storage format is similar to the algorithm for COO whereas the supplementary information is used in the algorithm to properly position the output vector. In this case, the range of position for the input and output vectors can be computed since the range of coordinates for the values in the SCOO storage format are restricted to the data distributions discussed in Section 6.1.1. Therefore, depending of the data distribution, the input and output vector shapes change.

In this case, the construction of the full output vector depends on the division of the matrix as specified in Section 6.1.1. This is detailed further in Section 6.1.3.

Algorithm 22 defines a data structure to store the sub-matrices stored in the CSR format in which we expect a given range of coordinates for the values. It also shows the algorithm to make a matrix vector product on the sub-matrix given the proper input vector. The data structure contains the three arrays necessary to store the matrix in the CSR storage format as well as useful information about the position of the sub-matrix in the global matrix. This information is the first column (fc) possibles in the sub-matrix. As for the SCOO storage format, the ranges of position for the input and output vectors can be computed.

Algorithm 23 defines a data structure to store the sub-matrices stored in the ELL format in which we expect a given range of coordinates for the values. It also shows the algorithm to make a matrix vector product on the sub-matrix given the proper input vector. The data structure contains the number of columns needed

Algorithm 22: CSR format data structure and matrix vector product

```
struct {
    Array idx, col, val
    Integer fc
} MatrixCSR

Task spmv_ csr()
    Data: m : MatrixCSR, v : Array
    Result: r : Array
    For i from 0 to m.idx.size() - 1 do
        For j from m.idx[i] to m.idx[i+1] - 1 do
            r[i] += m.val[j] * v[m.col[j] - m.fc]
```

Algorithm 23: ELL format data structure and matrix vector product

```
struct {
    Array col, val
    Integer fc, max_col
} MatrixELL

Task spmv_ ell()
    Data: m : MatrixELL, v : Array
    Result: r : Array
    For i from 0 to m.lrs - 1 do
        For j from 0 to m.max_ col - 1 do
            r[i + m.rpos] += m.val[i * m.max_col + j] * v[m.col[i * m.max_col + j]- m.fc]
```

in the arrays and the two arrays necessary to store the matrix in the ELL storage format as well the first column (fc) possibles in the sub-matrix which is used in this algorithm. As for the SCOO and CSR storage formats, the ranges of position for the input and output vectors can be computed.

In this section, we presented the algorithms used to perform the matrix vector product on the sub-matrices in the tasks with the COO, SCOO, ELL and CSR sparse storage formats. They are used to perform the matrix vector product on the global matrix. The Section 6.1.3 introduces the task-based algorithms to perform the sparse matrix vector product and construct the complete output vector.

6.1.3 Task-Based Parallel Algorithms

In this section, we introduce the task-based algorithms for the high level sparse matrix vector product. They use the tasks designed in Section 6.1.2 to perform the matrix vector product on the global matrix. In the algorithms the *SPMV* tasks corresponds to one of the *spmv_ ** defined in the previous section except for

spmv_coo which needs a different data distribution. These high level task-based algorithms are independent of the matrix storage format.

Algorithm 24: Parallel and Distributed Task Based Algorithm for the Sparse Matrix Vector Product with Distributed Rows

For i **from** *0* **to** $Ngr - 1$ **do in parallel**
  M[i] = GenMatrix(i, Ngr, Nr, Nc)
End parallel for
V = GenVector(Ngc)

For i **from** *0* **to** $Ngr - 1$ **do in parallel**
  R[i] = SPMV(M[i], V)
End parallel for

`/* Necessary data migrations to relocate data for a new SPMV */`
If *dataRelocation* **then**
  V = MERGE(R[0 : $Ngr - 1$])
End if

Algorithm 24 performs the matrix vector product on the global matrix when it is distributed by rows as shown in Figure 6.2. First, we need to generate or load the initial sub-matrices and sub-vectors to use them in the tasks. With this data distribution, each task needs the full vector and output a piece of vector. Then, the *SPMV* tasks can be run on each sub-matrix. Finally, the output vector has to be reconstructed. In this case, we need to combine each of the output pieces of vector and create a full vector that can be reused later on.

Algorithm 25 performs the matrix vector product on the global matrix when it is distributed by columns as shown in Figure 6.1. With this data distribution, each task needs the only the piece of vector required to make the matrix vector product and output a full vector. Then, the *SPMV* tasks can be run on each sub-matrix. In this case, the reconstruction of the full output vector is made by summing all the vectors which are results of the *SPMV* task.

Algorithm 26 performs the matrix vector product on the global matrix when it is distributed as shown in Figure 6.3. With the 2D data distribution, each column of sub-matrices needs the corresponding sub-part of the input vector. Then, the *SPMV* tasks can be run on each sub-matrix. In this case, the reconstruction of the full output vector is made by summing all the vectors which are results of the *SPMV* task from the same row of sub-matrices then gathering the result of the sum in a complete vector.

Algorithm 27 performs the matrix vector product on the global matrix when it is distributed as a non restricted and split COO sparse storage format. The matrix is split so that there is the same amount of values in each sub-matrix. This improves the load balancing of the computations in the tasks (each task has the same number of computations). With this data distribution, the shape of the input and output vectors cannot be anticipated so the matrix vector product task expect the full vector as input and return a full vector. Then, to compute the output of the global matrix vector product, the output vector of each task has

Algorithm 25: Parallel and Distributed General Task Based Algorithm for the Sparse Matrix Vector Product with Distributed Columns

For j **from** 0 **to** $Ngc-1$ **do in parallel**
  M[j] = GenMatrix(j, Ngc, Nr, Nc)
  V[j] = GenVector(j, Ngc, Nc)
End parallel for

For j **from** 0 **to** $Ngc-1$ **do in parallel**
  Rl[j] = SPMV(M[j], V[j])
End parallel for

`/* Necessary data migrations to relocate data for a new SPMV */`
If *dataRelocation* **then**
  For j **from** 0 **to** $Ngc-1$ **do in parallel**
    V[j] = SUM(Rl[$0 : Ngc-1$])
  End parallel for
Else
  Rg = SUM(Rl[$0 : Ngc-1$])
End if

to be summed.

This section introduced the data distribution considered in this study. They are a division by blocks of rows, blocks of columns, rectangle blocks in the rows and columns and a distribution with the same number of value in each task. We also introduced the algorithms to implement the tasks for the sub-matrices with the different sparse matrix storage formats (COO, SCOO, ELL and CSR) as well as the task-based algorithms to perform the matrix vector product on the global matrix. These algorithms are used to implement a task-based sparse matrix vector application and compare the performances obtained across different programming models, matrix storage format and data distribution in Section 6.2.

Algorithm 26: Parallel and Distributed Task Based Algorithm for the Sparse Matrix Vector Product with 2D Distributed Matrices

For j **from** *0* **to** $Ngc - 1$ **do in parallel**
  For i **from** *0* **to** $Ngr - 1$ **do in parallel**
    M[i, j] = GenMatrix(i, j, Ngr, Ngc, Nr, Nc)
  End parallel for
  V[j] = GenVector(j, Ngc, Nc)
End parallel for

For i **from** *0* **to** $Ngr - 1$ **do in parallel**
  For j **from** *0* **to** $Ngc - 1$ **do in parallel**
    Rl[i, j] = SPMV(M[i, j], V[j])
  End parallel for
End parallel for
For i **from** *0* **to** $Ngr - 1$ **do in parallel**
  Rs[i] = SUM(Rl[i, $0 : Ngc - 1$])
End parallel for

If *dataRelocation* **then**
  Rg = MERGE(Rs[$0 : Ngr - 1$])
End if

Algorithm 27: Parallel and Distributed Task Based Algorithm for the Sparse Matrix Vector Product with COO matrices

For j **from** *0* **to** $N - 1$ **do in parallel**
  M[i] = GenMatrixCOO(i, N, Nr, Nc)
End parallel for
V = GenVector(Nc)

For j **from** *0* **to** $Ngc - 1$ **do in parallel**
  Rl[i, j] = SPMV(M[i, j], V[j])
End parallel for

Rg = SUM(Rl[$0 : N - 1$])

6.2 Numerical Experiments

In this section, a sparse matrix vector product using task based, distributed and parallel programming models as well as the numerical experiments performed. Three classes of programming models were chosen to implement the sparse matrix vector product. First, MPI, a message passing library was chosen since it is the main stream library used to perform communications and global operation across multiple nodes on supercomputers. Secondly, we chose to use a fine grain task based programming model in which dependencies between tasks are data oriented. HPX, Legion and PaRSEC are such programming models and we chose to use only HPX since they cover the same kind of properties. Finally, the last class is a parallel and distributed task based programming model in which tasks themselves are also parallel and distributed on a subset of the resources allocated to the application. In this case, YML+XMP was chosen over Pegasus and Swift since YML+XMP has been more used than the other two.

6.2.1 Application Description

The sparse matrix vector product and the sparse $A(Ax + x)$ has been implemented with MPI, HPX, and YML+XMP. The sparse operation $A(Ax + x)$ is the combination of two sparse matrix vector product Ax with additional sum with the x vector. Therefore, a kernel performing the sparse Ax has been implemented with the different storage format and is used as the base in MPI, HPX and YML+XMP to perform the sparse matrix vector product on the sub-matrices distributed on the computing resources. This kernel is called in the HPX and YML+XMP tasks and the MPI processes to perform the sparse matrix vector product on the local data. Then the programming model is used to make the appropriate data migrations in order to combine and form the output vector according to the algorithms introduced in Section 6.1.3.

The kernel is designed to perform the sparse matrix vector product on the sub-matrices obtained from the split of the global matrix. The input matrix can be split over a 2D grid with particular cases in which there is only one column (split only across rows) or only one row (split only across columns) as introduced in Section 6.1.1. The $N_r \times N_c$ matrix is divided into an $N_{gr} \times N_{gc}$ grid. This kernel is called in the MPI processes to perform the sparse matrix vector product on the local matrices. In MPI, $N_{gr} \times N_{gc}$ is equal to the number of avail processes in the application. This allow each process to manage its sub-matrix. The kernel is also called in HPX tasks to perform the sparse matrix vector product on the sub-matrix managed into a tasks. For HPX, there is no particular restrictions on N_{gr} and N_{gc} since there is no obligations to execute tasks on each worker process, even though, these workers could be idling due to the lack of tasks. Finally, the kernel is also used in YML+XMP components (tasks) to perform the sparse matrix vector product in each processes allocated to each sparse matrix vector product task. Afterwards, the output vector is combined inside the tasks then combined at the level of the graph of tasks. Each YML task has an allocated number of processes. Therefore, in order to make sure that each process works on a sub-matrix, the number of sub-matrices ($N_{tr} \times N_{tc}$) in the grid of sub-matrices in the tasks have to match the number of processes allocated to the tasks. In our application, each task has the same number of processes allocated, which means that, in order to fill each processes of the tasks, the number of rows (resp. columns) in the division of the global matrix has to match the number of rows (resp. columns) in the division of the matrix in the tasks multiplied by the number of rows (resp. columns) in the division of the matrix at the task level (see Figure 6.4).

The kernel has been implemented to support the CSR, ELL, COO, SCOO and dense storage formats. The

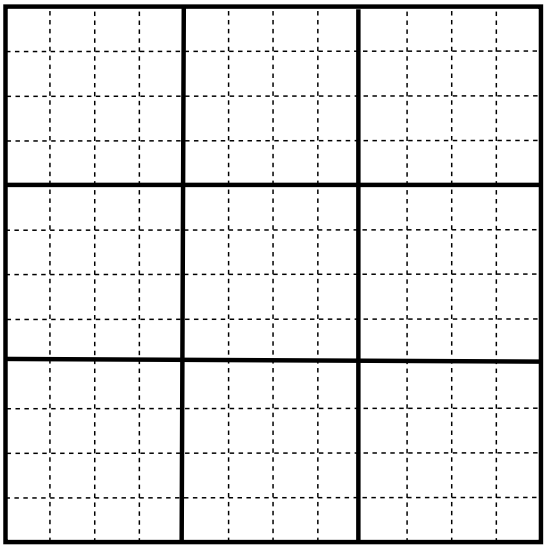


Figure 6.4: Two level of division in matrices managed by YML+XMP with 3×3 task level matrices in which there is 4×4 sub-matrices

sparse matrix vector products for the different storage formats were implemented according to the algorithms introduced in Section 6.1.2. Then, for each storage format and each programming model considered, the distributed and parallel version of the sparse matrix vector has been implemented based on the algorithms described in Section 6.1.3.

Moreover, the sparse operation $A(Ax + x)$ has also been implemented on top the sparse matrix vector product. Therefore, it also supports the CSR, ELL, COO, SCOO and dense storage formats as well as the different matrix distribution possibilities. This operation has been implemented in MPI, HPX and YML+XMP. It consists in a call to the sparse matrix vector product followed by the sum of the output vector with x then another sparse matrix vector product. The communications, the building of the output vector depending on the distribution of the matrix and the sum are managed with the programming models in which the application is implemented.

Finally, separate libraries containing the kernel as well as the MPI and HPX extensions of the kernel has been created. YML+XMP does not support the creation of libraries therefore, the sparse matrix vector product and the sparse operation $A(Ax + x)$ were directly implemented as applications. The libraries also contain functions to generate distributed sparse matrices and the YML+XMP applications has tasks for this purpose. The sparse matrices can be generated in each one of the supported storage format and in the data distribution introduced in Section 6.1.1. These matrices are the C-diagonal Q-perturbed matrices introduced in Chapter 3. They are were used to perform numerical experiments which is presented in the next section.

6.2.2 Results and Analyses on Total Petascale Pangea II

In these experiments, the sparse operation $A(Ax + x)$ has been performed on a sparse matrix stored COO, SCOO, ELL and CSR where A is the sparse matrix and x a vector. This operation has been chosen as test due to the fact that the output vector has to be recombined after the local sparse matrix vector in order to reuse it in the next sparse matrix vector product. The C-diagonal Q-perturbed matrices introduced in

Chapter 3 are used to evaluate the performances of the different programming models and storage formats. To do so, we perform strong scaling experiments on 2 000 000 × 2 000 000 and 4 000 000 × 4 000 000 C-diagonal Q-perturbed matrices for C = 300 and several values of Q. Moreover, we also study the impact of the number of values by rows by trying multiple values of C. Finally, we perform weak scaling experiments with the C-diagonal Q-perturbed matrices for C=300 by increasing the number of values in the matrix depending on the number of cores used to run the application.

6.2.2.1 Pangea II

Pangea II is the supercomputer on which the experiments have been performed. It is owned by Total and located in Pau, France. This supercomputer is composed of nodes built with 2 Xeon E5-2680v3 12C 2.5GHz processors for a total of 220 800 cores. The nodes are connected with Infiniband FDR interconnects. It performs at 5.283 PFlop/s for Linpack and 162.692 TFlop/s for HPCG.

6.2.2.2 Best Results Selection

These experiments depend on several parameters. The programming model is the first of them. There are also the parameters related to the test matrices used, C and Q as well as the size of the matrix. Moreover, the data distribution which can be represented by the 2D grid used to divide the matrix adds two more parameters, N_{gr} and N_{gc}. Finally, YML+XMP also adds several parameters to tune the two levels of parallelism it offers. These parameters are the number of processes allocated per task, the grid used to divide the matrix in the tasks and the higher level grid used to determine which part of the matrix is used in tasks. Even if these parameters are related, which reduces the range of values to try, there is a large amount of possibilities to try.

Therefore, after running the applications, despite using only a subset of the possible cases to try, a large number of runs for the different parameters is produced. Then, we have to select the case that produces the best execution time depending on which case we want to display. For instance, if we want the best case for MPI for a given matrix size, storage format, C, Q and a 2 nodes, the results change depending on the data distribution. Thus, we have to select which data distribution has made the best performances. The application is run multiple time for a given data distribution so we take the median execution time as metric to compare the different data distributions. Then, we chose the case that has the best median (usually the smallest since we want the fastest execution time).

6.2.2.3 Strong scaling with a 2 000 000 × 2 000 000 sparse matrix

Strong scaling experiments on the operation $A(Ax + x)$ have been performed with a 2 000 000 × 2 000 000 sparse C-diagonal Q-perturbed matrices for C = 300 and several values of Q. The matrix was stored in 4 storage formats : CSR, COO, SCOO and ELL. The applications were implemented in MPI, HPX and YML+XMP. We use a dense vector with fixed values that is generated during the execution of the applications.

Figure 6.5 summarizes the results obtained from the experiments with HPX in which the operation $A(Ax + x)$ is applied on a 2 000 000 × 2 000 000 sparse C-diagonal Q-perturbed matrices for C = 300 and a vector. This figure shows the execution time for the different sparse storage formats depending on the number

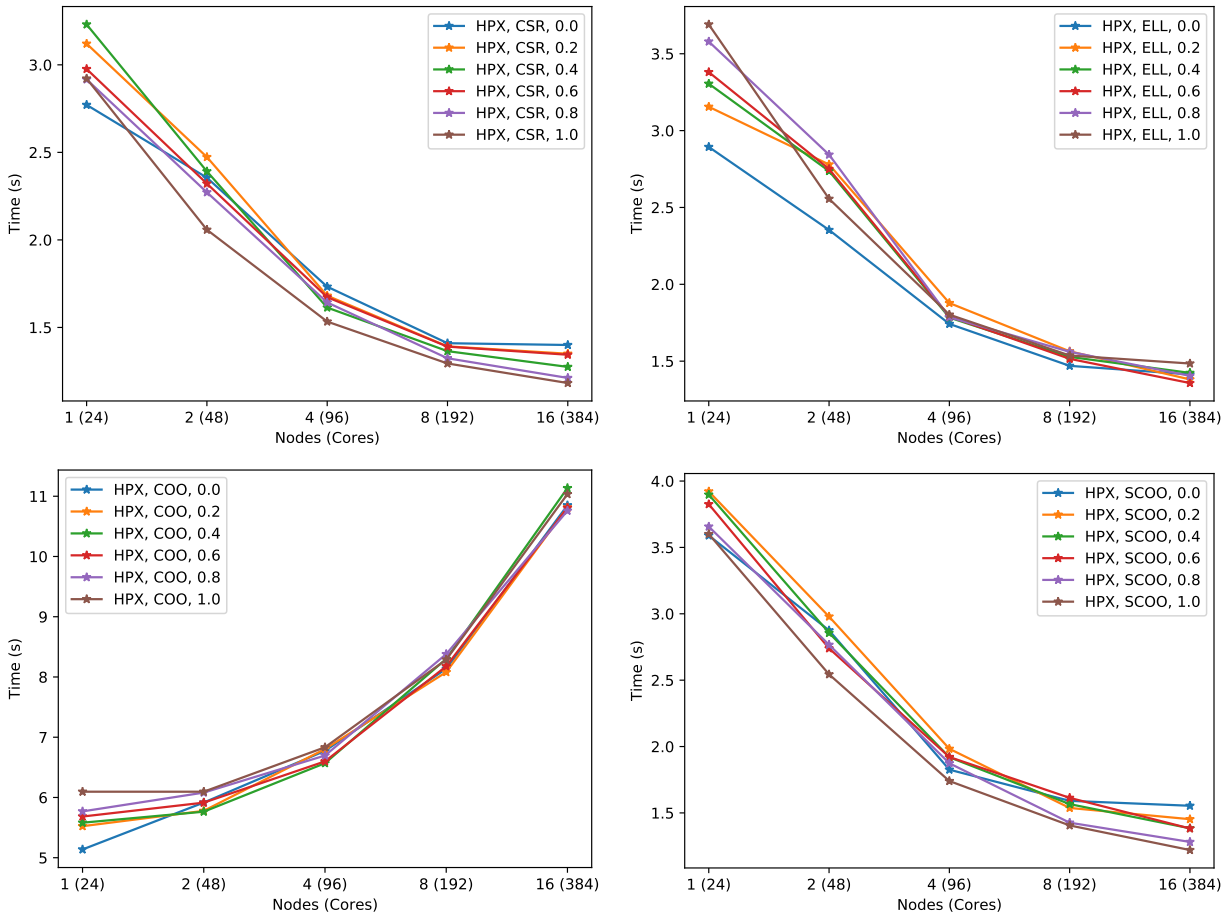


Figure 6.5: Strong scaling considering HPX and several storage formats for a 2 000 000 × 2 000 000 matrix with C = 300 on Pangea II. Legend is (model, format, Q).

of nodes (cores). For CSR, ELL and SCOO, HPX has a very good strong scalability up to 4 nodes then the performance improvement gained from the increase in computing resources diminishes. ELL and CSR have close performances on small number of nodes and CSR is better when the number of nodes increases. They perform better than the COO and SCOO. COO is the slowest within the HPX implementations. This format is a bit different since the values of the matrix are not restricted to a 2D grid like the others. Therefore, the output of each local sparse matrix vector product has to be summed to generate the complete output and this vector has the size of the rows of the full matrix since we do not restrict the positions of the values in the sub-matrices of this format. Furthermore, this sum is a very costly reduction since it concerns all the sub-matrices and the reduced vector is very large.

The best data distribution for SCOO, CSR and ELL is 3, 4 or 6 sub-columns until 8 nodes and it starts to increase to 8 or 12 for 16 nodes. For COO, there is no data distribution according to row or columns since it is divided according to a number of division. In practice, the COO are divided in $N_{gr} \times N_{gc}$ where $N_{gr} \times N_{gc}$ is the grid in which the other storage format divide their matrix. The best data distribution is similar for SCOO, CSR and ELL since they use the same pattern to combine the output vector.

Figure 6.6 summarizes the results obtained from the experiments with MPI in which the operation $A(Ax + x)$ is applied on a 2 000 000 × 2 000 000 sparse C-diagonal Q-perturbed matrices for C = 300 and a

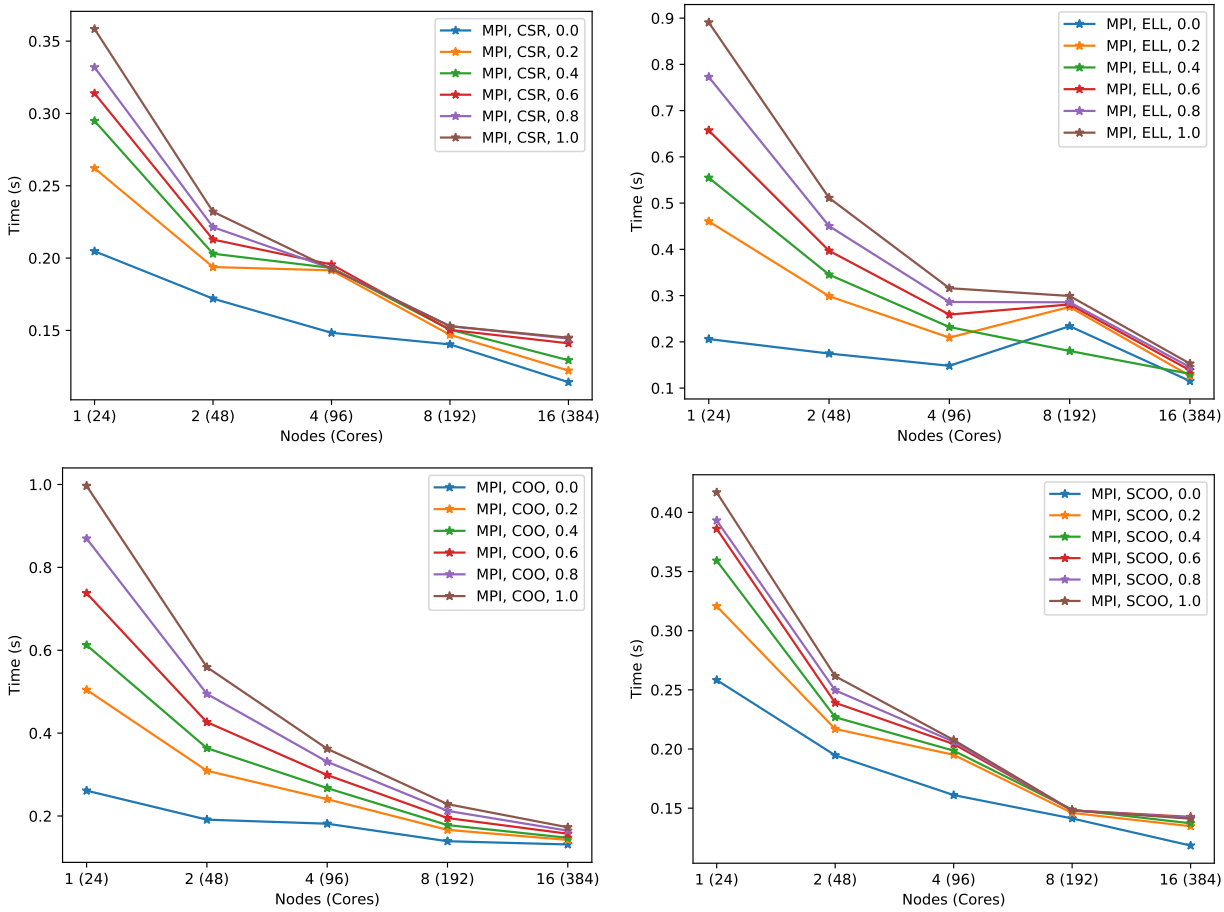


Figure 6.6: Strong scaling considering MPI and several storage formats for a 2 000 000 × 2 000 000 matrix with C = 300 on Pangea II. Legend is (model, format, Q).

vector. This figure shows the execution time for the different sparse storage formats depending on the number of nodes (cores). CSR obtains the vest performances with MPI. CSR and ELL have very close performances for Q = 0 and, for the other values of Q, it is SCOO who is very close to CSR. CSR and SCOO perform better than ELL and COO. COO is the slowest due to the fact that the output of each sub-matrix has to be summed and the output vector is of the size the global number of rows of the matrix. However it is very close to ELL when Q > 0 despite its costly global reduction.

The best data distribution for SCOO and CSR is 1 sub-row until 8 nodes. Then, starting at 16 nodes, there is cases in which the best performances are with 1 sub-column. For ELL, the best data distribution is with 1 sub-column except for 8 nodes where it is 4 sub-columns sometimes. As with HPX, for COO, there is no data distribution according to row or columns. It is interesting to notice that, SCOO and CSR obtain the best performances with the same data distribution and ELL best performances are obtained with another data distribution with MPI.

Figure 6.7 summarizes the results obtained from the experiments with YML in which the operation $A(Ax + x)$ is applied on a 2 000 000 × 2 000 000 sparse C-diagonal Q-perturbed matrices for C = 300 and a vector. This figure shows the execution time for the different sparse storage formats depending on the number of nodes (cores). In COO and SCOO, the performances are quite stable and we obtain execution

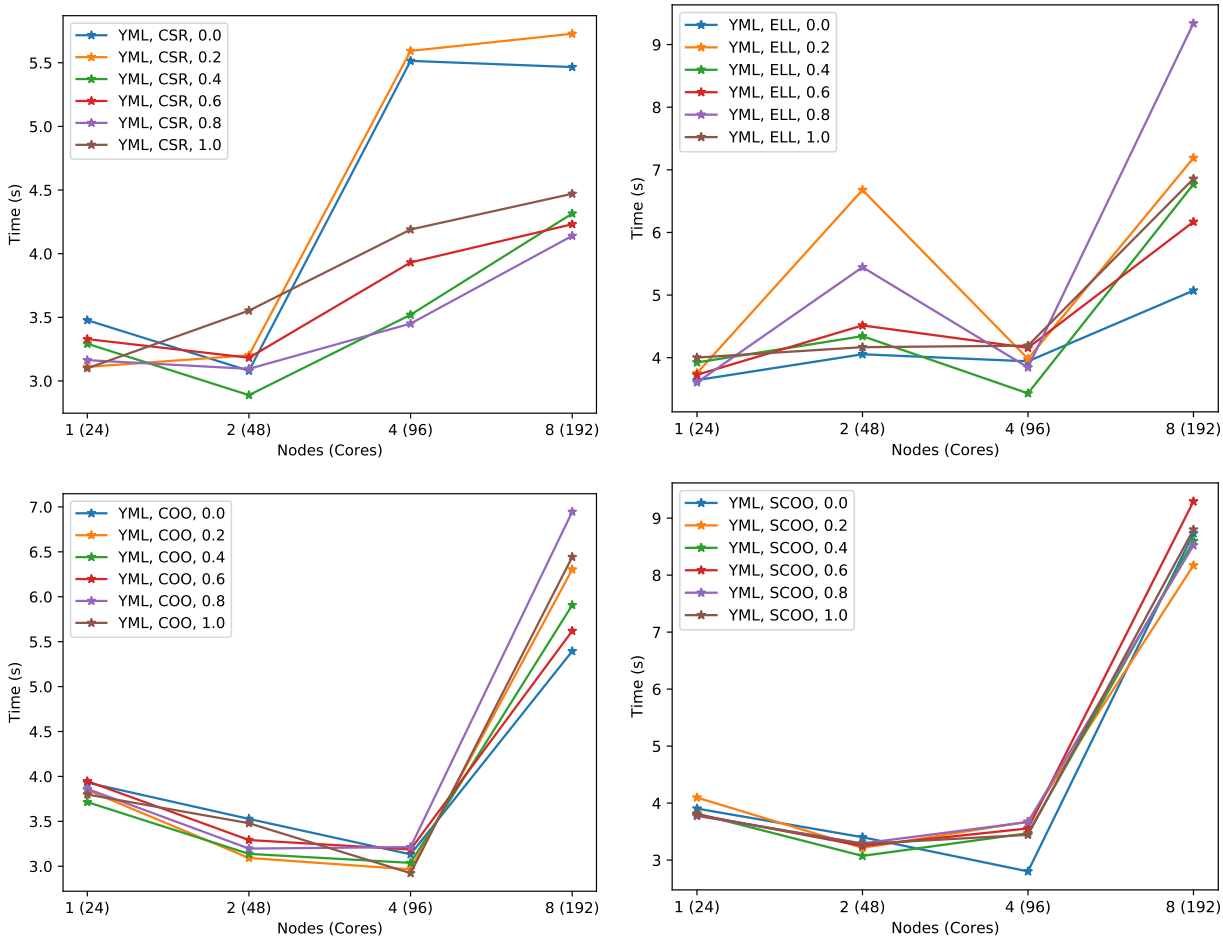


Figure 6.7: Strong scaling considering YML and several storage formats for a 2 000 000 × 2 000 000 matrix with C = 300 on Pangea II. Legend is (model, format, Q).

time close for the different values of Q. Until 4 nodes, there is a slight improvement of performances for COO and SCOO execution times stay almost constant. Then, for the increase from 4 nodes to 8 nodes, the performances plummet dramatically. We think it is due to the increasing number of processes making IOs (96 to 192) to communicate the data between the tasks since YML uses the file system to pass data. This induces a great variability between the results. On the other hand, CSR and ELL performances have more variability and it is difficult to find a logical pattern. CSR seems to have better performances overall but there is some cases where the other storage formats are prevalent.

Furthermore, YML+XMP introduces more parameters to describe the data distribution in the tasks and across the different tasks. This is an open set of parameters as the matrix can be divided into any shape. Even though there is relation between the parameters, it is still a large amount of parameters to try out. Moreover, due to technical issues, multiple YML+XMP applications cannot be run at same time while it is perfectly possible to run several MPI or HPX applications at the same time. This greatly limit the range of parameters that can be tested since it is not possible to take advantage of the large amount of computing resources available. The constraint to run only one application at a time and having a lot more parameters to try with YML+XMP lets the possibility of the existence of parameters that could produce

better performances than the ones used in the current experiments. We tried as much parameters we believed would lead to interesting results as we could.

In these strong scaling experiments, we executed the sparse operation $A(Ax + x)$ on a 2 000 000 $\times$ 2 000 000 sparse C-diagonal Q-perturbed matrices for C = 300 and a vector while increasing the number of nodes (cores) on which we ran the applications. We used several values for Q which represents the probability to increase the dispersion of the values in the matrix. The applications were also implemented with three programming models : MPI, HPX and YML+XMP. For MPI and HPX, CSR is the storage format that obtains the best performances. It also obtains good performances with YML+XMP but it is less clear due to the variability induced by the file system. In HPX, we see that SCOO, ELL and CSR have the same performance pattern (the curves are very similar) while the use of the different storage format explain the difference of performances between the storage formats. SCOO has a different communication pattern that needs to reduce the output of all the matrix vector products performed in the tasks. This reduction has been implemented for this application in HPX and is not very efficient whereas MPI reduction is heavily optimized and uses heuristics to perform a very efficient reduction. This is why there is no difference between ELL, SCOO and CSR on one side and COO on the other side with MPI since, in MPI, all the storage format have the same performance pattern. Finally, in YML+XMP the performances heavily rely on the capabilities of the file system due to the fact that the data going from one task to another are written then read from files. This appears when the application is executed on 8 nodes and the number of cores used increases from 96 to 192. Indeed, at this point, the application is taking a few more times to run on 8 nodes than to run on 4 nodes. Current file systems are not adapted to this kind of heavy use during the execution of an application and IOs duration varies with the number of IOs operations. Due to this, there is a lot of variability between the results. With the parameters tried in these experiments with YML+XMP, there is no clear data distribution that stands out and performs better than the others. CSR is leading in most of the case but the other storage formats also have cases where they perform better than CSR.

Q is also an interesting parameter that influences the performances. It is less noticeable on YML+XMP due to the complexity of the execution model and the variability of the interactions with the file system. However, its influences is clear for MPI and HPX since, most of the cases, the execution time is lower when Q is lower and it is higher when Q is higher. Moreover, each application uses the same kernel to execute the sparse matrix vector product on their smallest sub-matrices. Therefore, the differences in performances between the programming models is due to the management of the data and the implementation of the reduction to combine the output results of the sparse matrix vector product. As such, the influence of Q is the clearest for MPI since their communications are very efficient. For HPX and YML, the influence of Q is absorbed by our implementation of the reduction in HPX and YML+XMP as well as the IOs for YML+XMP. In C-diagonal Q-perturbed matrices, C represents the number of values per row above the diagonal and Q is the probability to perturb the fact that all the values are packed above the diagonal. When Q increases, the values on the row is more dispersed on this row instead of being grouped above the diagonal. Hence, the data access pattern of the input vector changes with the different values of Q. When Q is zero, C consecutive values starting from the diagonal of the matrix are multiplied with the input vector. Since the values are consecutive in memory and in term of columns that are accessed in the input vector, it only accesses a portion of size C of the input vector. This access is very regular and efficient. However, when Q increases, the values are more dispersed in the row and the accesses to the input vector when making the multiplication is not as

regular and changes from one row to another. This random access to the memory containing the input vector is less efficient. Therefore, Q influences the execution time of the sparse matrix vector kernel and explains the difference of performances with the different values of Q.

Then, the size of the sparse matrix is increased. The effect of the size of the matrix is addressed in the next section.

6.2.2.4 Strong scaling with a 4 000 000 × 4 000 000 sparse matrix

We continue our strong scaling experiments on the operation $A(Ax+x)$ with a 4 000 000 × 4 000 000 sparse C-diagonal Q-perturbed matrices for C = 300 and several values of Q. The CSR, COO, SCOO and ELL storage formats are used to in the experiments and the applications are implemented in MPI, HPX and YML+XMP.

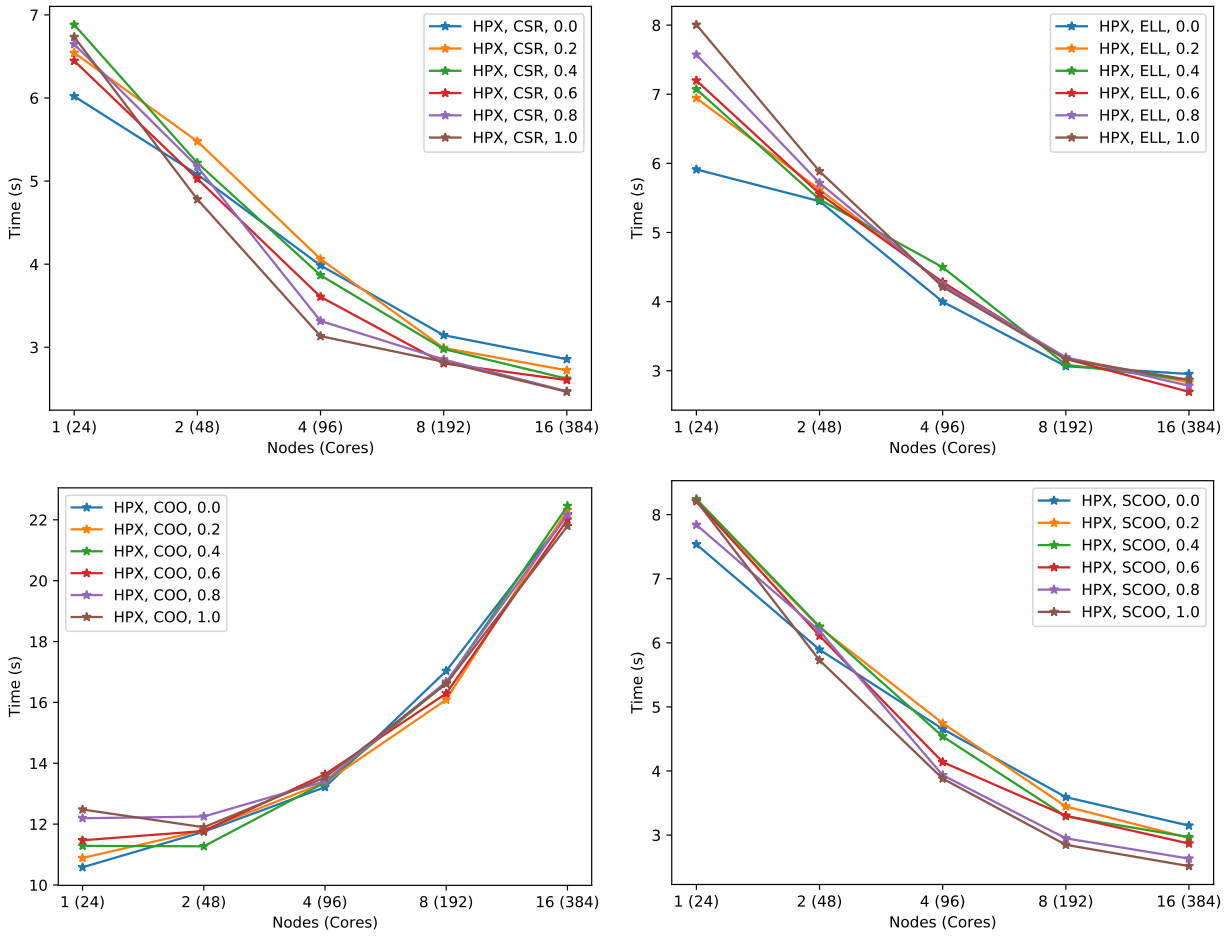


Figure 6.8: Strong scaling considering HPX and several storage formats for a 4 000 000 × 4 000 000 matrix with C = 300 on Pangea II. Legend is (model, format, Q).

Figure 6.8 summarizes the results obtained from the experiments with HPX in which the operation $A(Ax + x)$ is applied on a 4 000 000 × 4 000 000 sparse C-diagonal Q-perturbed matrices for C = 300 and a vector. This figure shows the execution time for the different sparse storage formats depending on the number of nodes (cores). For CSR, ELL and SCOO, HPX has a very good strong scalability up to 8

nodes then the performance improvement gained from the increase in computing resources diminishes. ELL and CSR have close performances for 1 node and Q equal to 0.2, 0.4 and 0.6. ELL has better performances than the other sparse storage formats for 1 node and Q = 0. Except for the case where ELL is better, CSR leads the performances. They perform better than the COO and SCOO. COO is the slowest within the HPX implementations due to its costly reduction. It is even more evident as the size of the reduced vector increases.

The best data distribution for SCOO, CSR and ELL is 3, 4 or 6 sub-columns until 8 nodes and it starts to increase to 8 or 12 for 16 nodes. This is the same parameters as the ones for the 2 000 000 × 2 000 000 matrix. Similarly, SCOO, CSR and ELL have the same kind of performance patterns and COO does not scale at all.

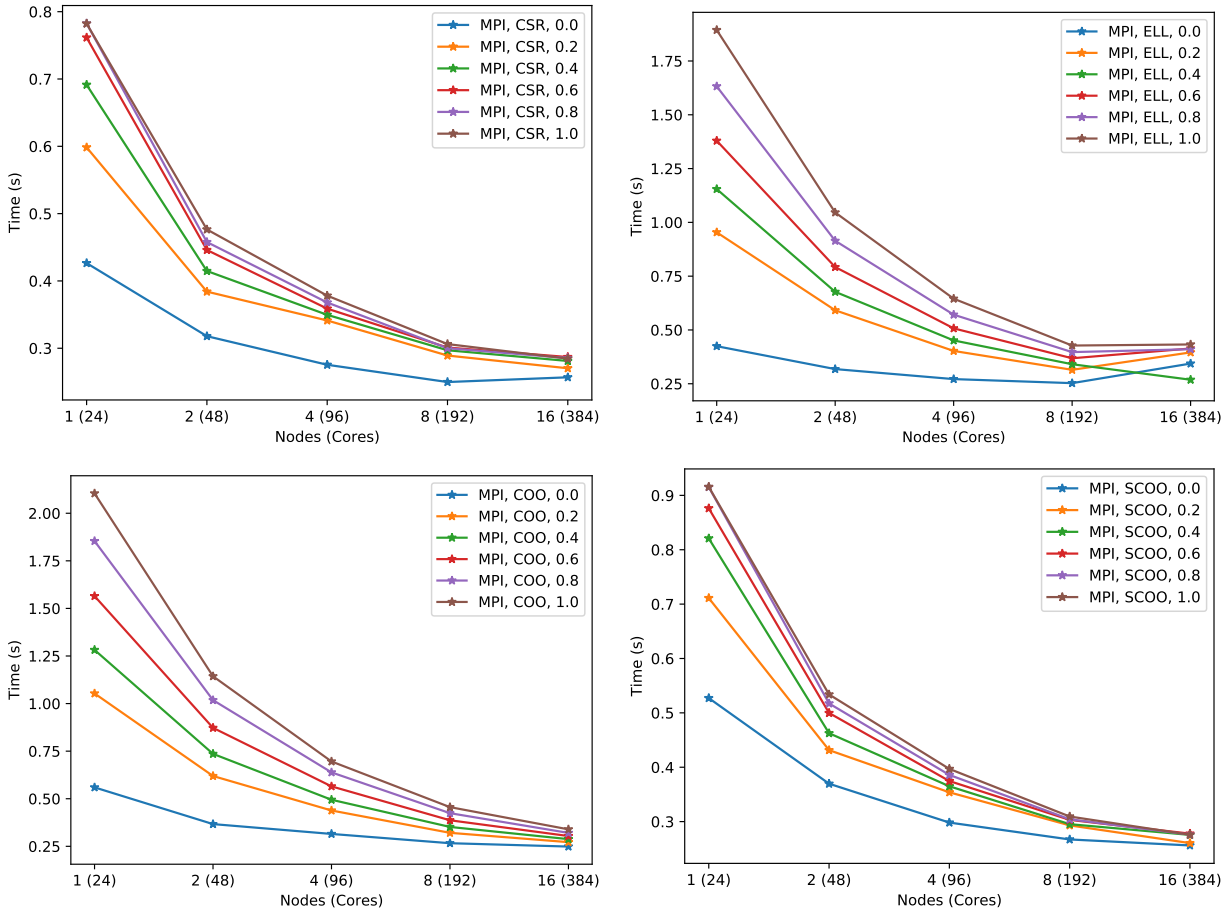


Figure 6.9: Strong scaling considering MPI and several storage formats for a 4 000 000 × 4 000 000 matrix with C = 300 on Pangea II. Legend is (model, format, Q).

Figure 6.9 summarizes the results obtained from the experiments with MPI in which the operation $A(Ax + x)$ is applied on a 4 000 000 × 4 000 000 sparse C-diagonal Q-perturbed matrices for C = 300 and a vector. This figure shows the execution time for the different sparse storage formats depending on the number of nodes (cores).

ELL with Q = 0 has very good performances compared to its performances with other values of Q. The impact of the variation of Q is greater on ELL than on the other storage formats as we can see on this

figure and on the figure for the 2 000 000 × 2 000 000 matrices. ELL also seems to have troubles scaling from 8 nodes to 16 nodes. With these bigger matrices, CSR is still the storage format that obtains the best performances. COO is, whithout surprises, the slowest due to the reduction on the vector of the size of the matrix. SCOO has performances close to CSR but is still a bit behind.

The best data distribution for SCOO and CSR is 1 sub-row except for Q = 0 where it is 1 sub-column. For ELL, the best data distribution is with 1 sub-column until 8 nodes. Then, starting from 16 nodes, it is 3 sub-columns. As with HPX, for COO, there is no data distribution according to row or columns. The best data distribution are similar to the ones obtained with a smaller matrix, even tough, the best data distribution is more evident with this size of matrix.

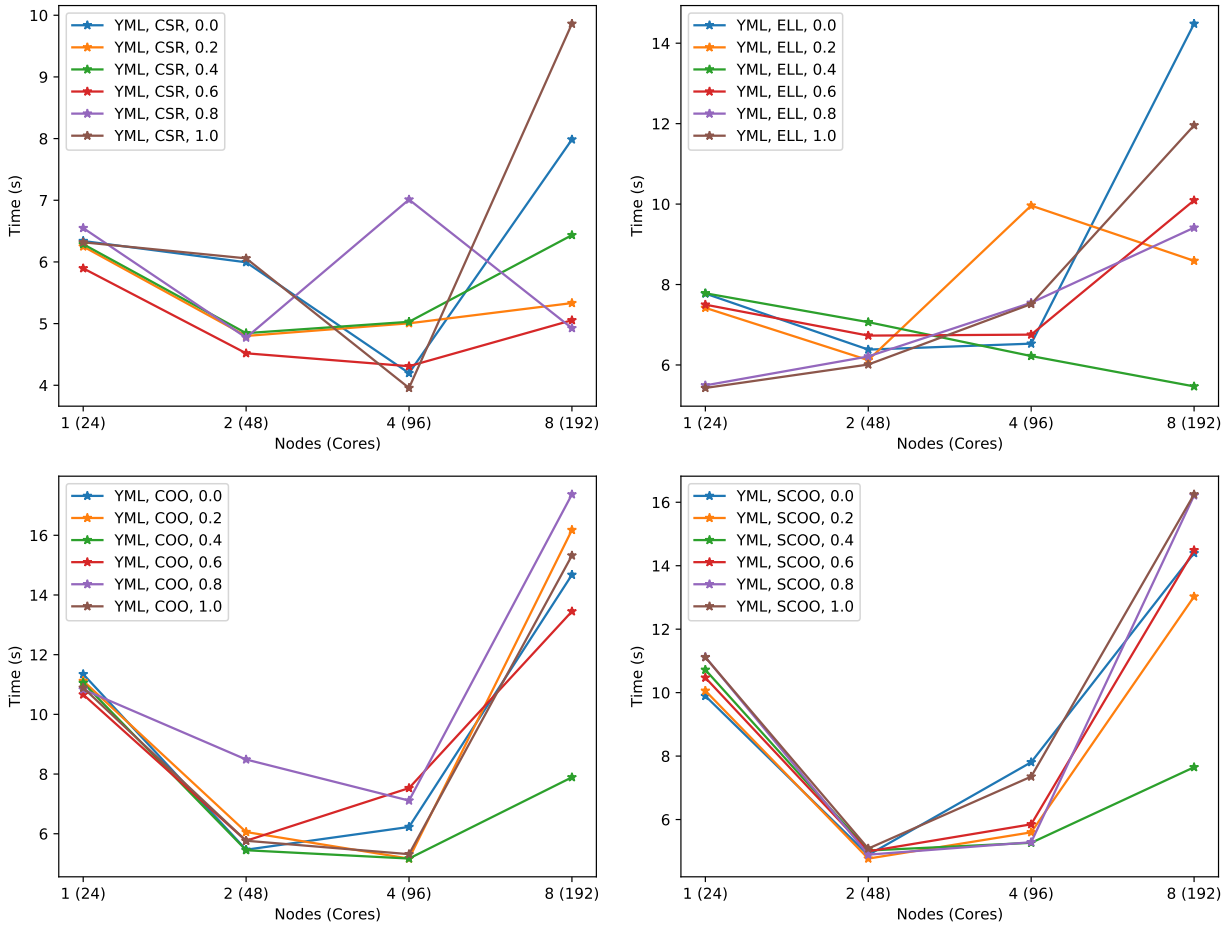


Figure 6.10: Strong scaling considering YML and several storage formats for a 4 000 000 × 4 000 000 matrix with C = 300 on Pangea II. Legend is (model, format, Q).

Figure 6.10 summarizes the results obtained from the experiments with YML in which the operation $A(Ax + x)$ is applied on a 4 000 000 × 4 000 000 sparse C-diagonal Q-perturbed matrices for C = 300 and a vector. This figure shows the execution time for the different sparse storage formats depending on the number of nodes (cores). As for the 2 000 000 × 2 000 000 sparse C-diagonal Q-perturbed matrices, we have the same kind of performances with COO and SCOO on one side and CSR and ELL on the other side. For COO and SCOO, the performances improve from 1 node to 2 then stay the same until 4 nodes and worsen

while going to 8 nodes. For CSR and ELL, the performances have more variations depending on the values of Q than for COO and SCOO. The execution times improve roughly until 4 nodes then also worsen from 4 nodes to 8 nodes. We believe that the use of the file system induces these variability in the performances.

We performed strong scaling experiments on the sparse operation $A(Ax+x)$ with a 4 000 000 $\times$ 4 000 000 sparse C-diagonal Q-perturbed matrices for C = 300 while increasing the number of nodes (cores) on which we ran the applications. We compared the performances obtained with the SCOO, COO, CSR and ELL storage formats for several values of Q. We implemented applications with MPI, HPX an YML+XMP. They are based on a common kernel performing the sparse matrix vector product. The performances obtained are similar to those obtained from our previous experiments with a 2 000 000 $\times$ 2 000 000 sparse C-diagonal Q-perturbed matrices for C = 300. Q induces variation for MPI in which the communications are very efficient and most of the execution time is spent running the kernel whereas the values of Q have less impact on YML+XMP and HPX in which more execution time is spent on the reduction and the data migration through the file system for YML. Finally, CSR is clearly the storage format that have the best performances for MPI while it CSR obtains good performances with the other programming models, the other storage formats are also having interesting performances. Similarly to what happens with Q, the time spent out of the kernel smooths the performance differences due to the storage formats in HPX and YML+XMP.

6.2.2.5 Weak Scaling 4 000 000 $\times$ 4 000 000 per node

Weak scaling experiments have been performed on a sparse C-diagonal Q-perturbed matrices for C = 300 which size has been increased with the number of nodes (cores) used to execute the applications. The applications can perform the sparse operation $A(Ax + x)$ with SCOO, COO, CSR and ELL storage formats and are implemented with MPI, HPX an YML+XMP. The base size of the matrix is 4 000 000 $\times$ 4 000 000 and each dimension is multiplied by the number of nodes in order to approximatively have a 4 000 000 $\times$ 4 000 000 per node. Since C is fixed to 300, there are 300 values per row and multiplying the two dimension only multiply the number of values by the number of nodes and not its square like it would if the matrix was dense. Moreover, the size of the input and output vectors increases depending on the number of nodes.

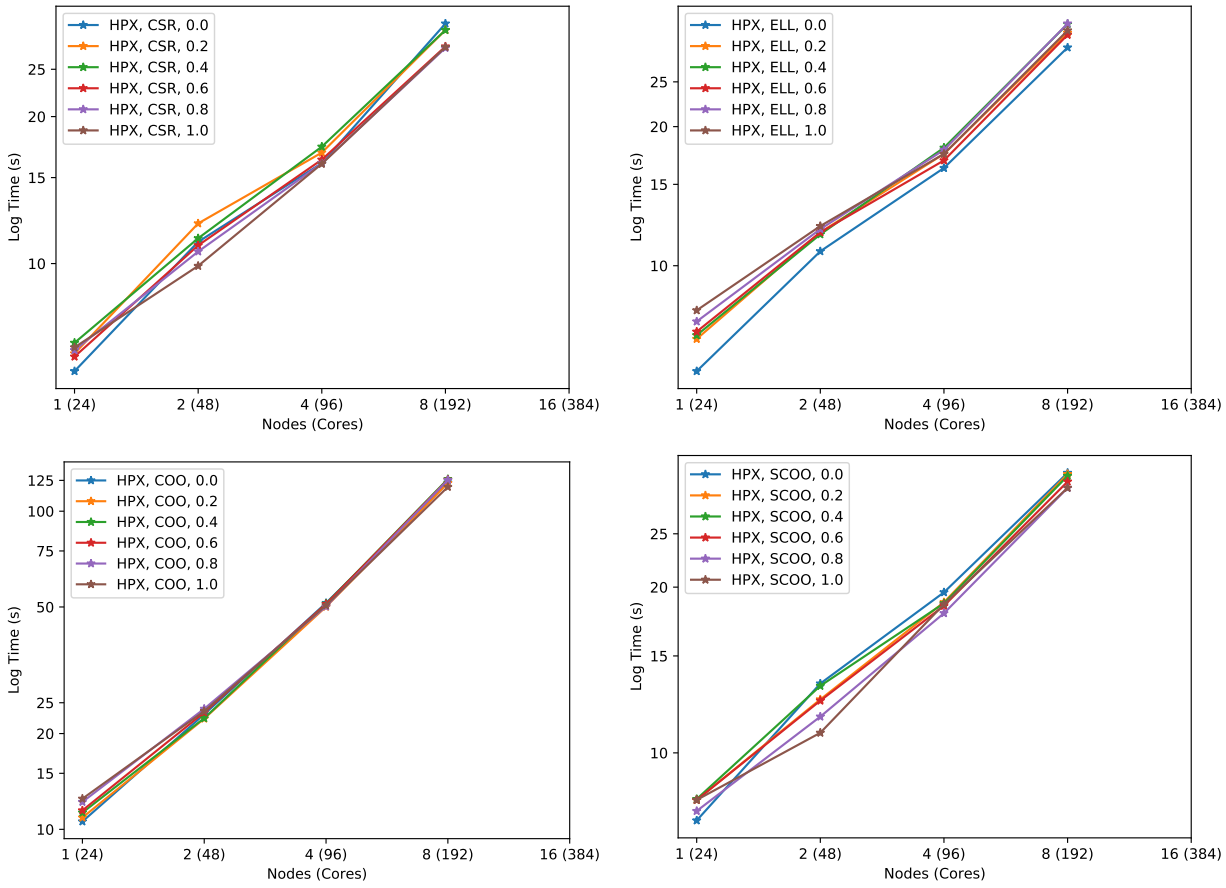


Figure 6.11: Weak scaling considering HPX and several storage formats for a 4 000 000 × 4 000 000 matrix per node with C = 300 on Pangea II. Legend is (model, format, Q).

Figure 6.11 summarizes the results obtained from the weak scaling experiments with HPX in which the operation $A(Ax+x)$ is applied on a sparse C-diagonal Q-perturbed matrices for C = 300 and a vector. This figure shows the execution time for the different sparse storage formats depending on the number of nodes (cores) as well as the size of the matrix that increase with the number of nodes. The ideal weak scaling would be obtaining same execution time while increasing the data size and the number of computing resources in the same proportions. The execution times increase while they should be staying constant. This shows the impact of the reduction and/or gather, depending on the data distribution, on the global execution time since the tasks process sub-matrices of an equivalent size. As for the strong scaling experiments, the application takes more time with the COO storage format due to the reductions on a larger vector.

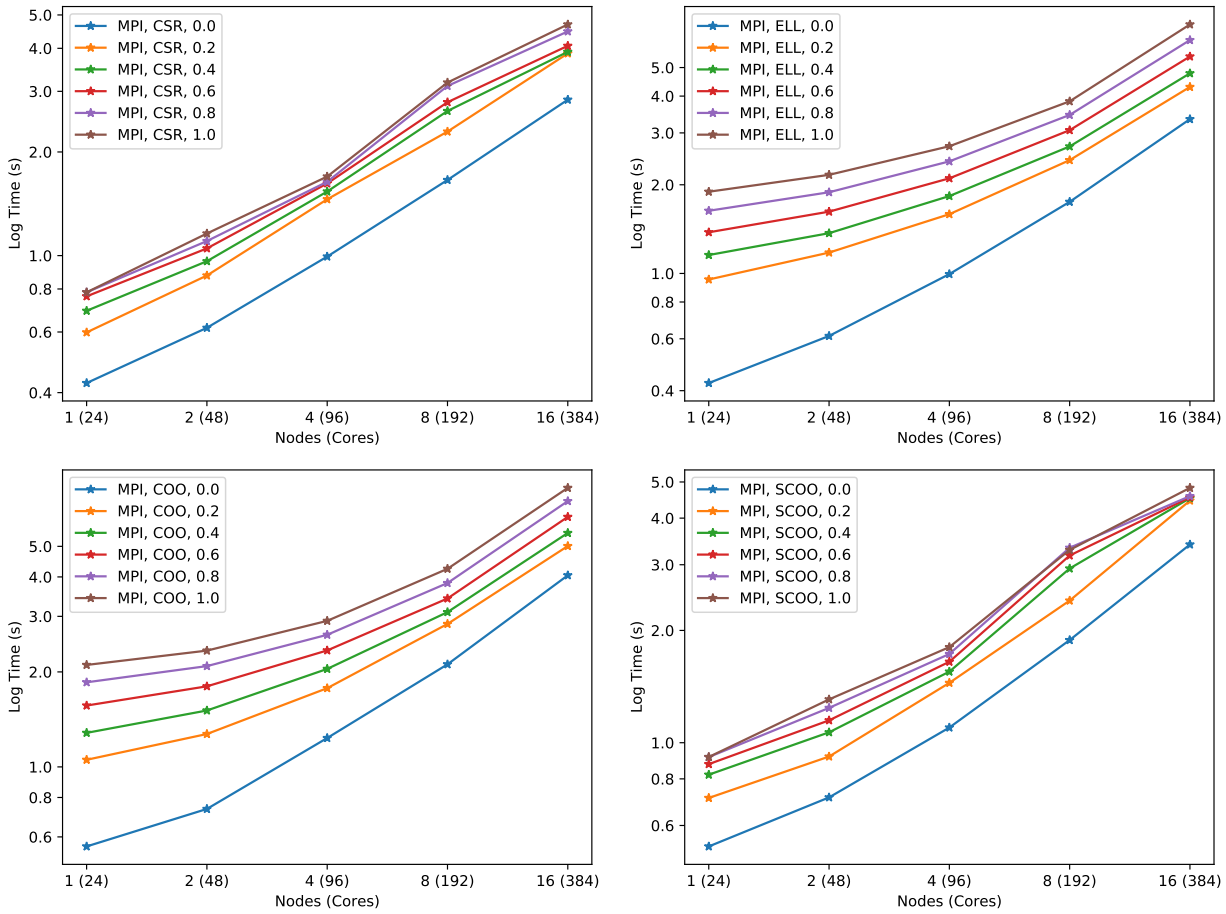


Figure 6.12: Weak scaling considering MPI and several storage formats for a 4 000 000 $\times$ 4 000 000 matrix per node with C = 300 on Pangea II. Legend is (model, format, Q).

Figure 6.12 summarizes the results obtained from the weak scaling experiments with MPI in which the operation $A(Ax + x)$ is applied on a sparse C-diagonal Q-perturbed matrices for C = 300 and a vector. This figure shows the execution time for the different sparse storage formats depending on the number of nodes (cores) as well as the size of the matrix that increase with the number of nodes. The execution times are also increasing with MPI although the execution times are not as high. This shows that the reduction and/or gather, depending on the data distribution, has less impact on the MPI application than on the HPX application.

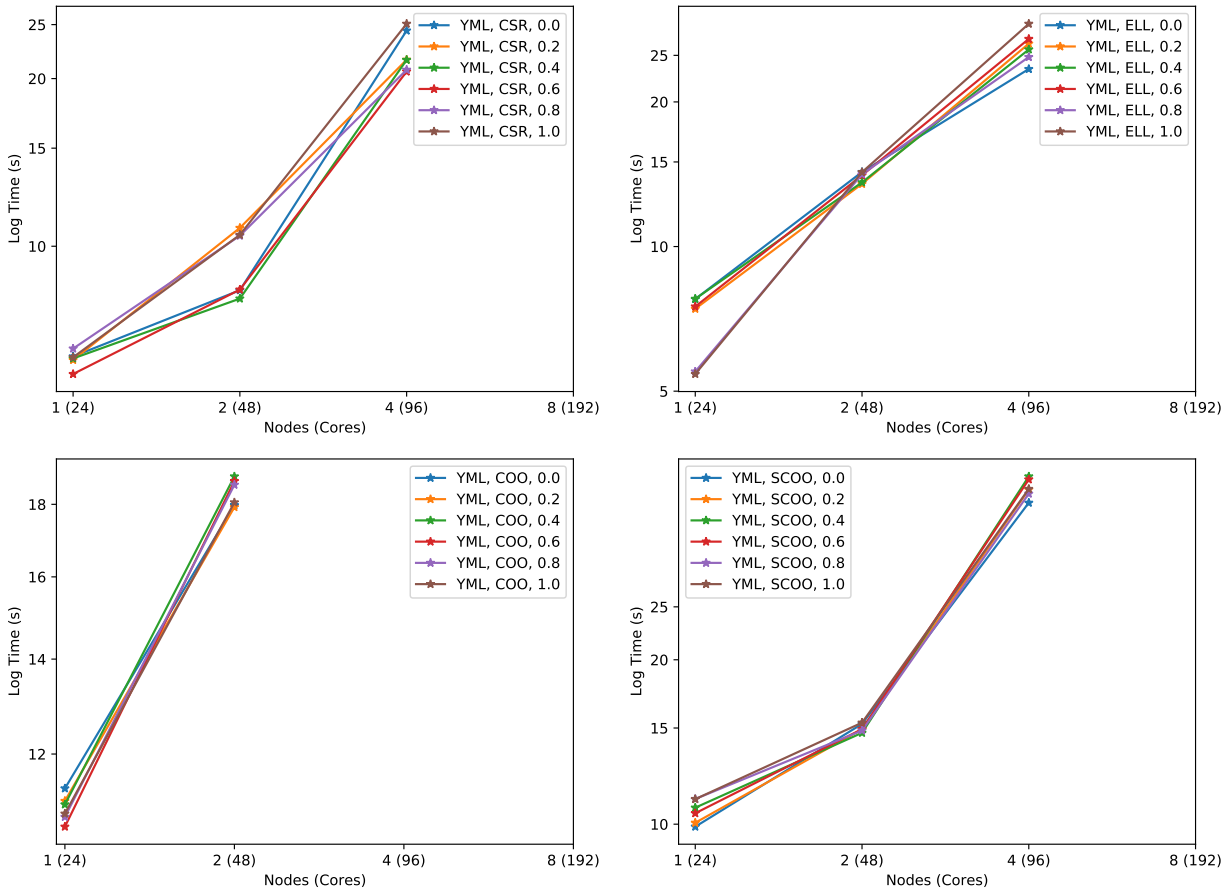


Figure 6.13: Weak scaling considering YML and several storage formats for a 4 000 000 × 4 000 000 matrix per node with C = 300 on Pangea II. Legend is (model, format, Q).

Figure 6.13 summarizes the results obtained from the weak scaling experiments with YML in which the operation $A(Ax+x)$ is applied on a sparse C-diagonal Q-perturbed matrices for C = 300 and a vector. This figure shows the execution time for the different sparse storage formats depending on the number of nodes (cores) as well as the size of the matrix that increase with the number of nodes. Similarly to HPX and MPI, the execution time are increasing. They are increasing faster for YML that for HPX and MPI since YML also makes IOs to transmit data between tasks in addition to the reduction and/or gather depending on the data distribution.

We performed weak scaling experiments on the operation $A(Ax+x)$ with a sparse C-diagonal Q-perturbed matrices of size 4 000 000 × 4 000 000 per node with C = 3. We multiplied the base size of the matrix (4 000 000 × 4 000 000) by the number of nodes that were used to execute the applications. The expected result is a constant execution time when the computing resources and the data increase in the same proportions. This the kind of experiments performed in this section. However, the results obtained differ from the ones expected. This is due to the communications (reductions and/or gathers depending on the data distribution) used to create the output vector and reuse it in the second sparse matrix vector. Moreover, the implementation of the sum / gather in the HPX and YML applications is less efficient since they take more time than the MPI application while executing the same kernel on the same data. Finally, YML also uses the file system to

transfer data between tasks which take more time.

6.2.2.6 Weak Scaling 3 000 000 × 3 000 000 per node

Weak scaling experiments have also been performed with a base matrix of size 3 000 000 × 3 000 000 per node with the same conditions as for 4 000 000 × 4 000 000. They were made to try to understand issues with the weak scaling experiments we did previously. It turned out to be an issue with *int* data type overflowing due to the large amount of values used on more than 2 nodes. With 8 000 000 rows on 2 nodes and C = 300 values per row, there was 2 400 000 000 values which exceeds 2 147 483 647, the maximum integer possible to store in an *int*. Once this issue was fixed, the weak scaling experiments were behaving more like what was expected. Therefore, these experiments were not strictly necessary anymore. We still performed them to confirm the results obtained with the 4 000 000 × 4 000 000 matrices as base.

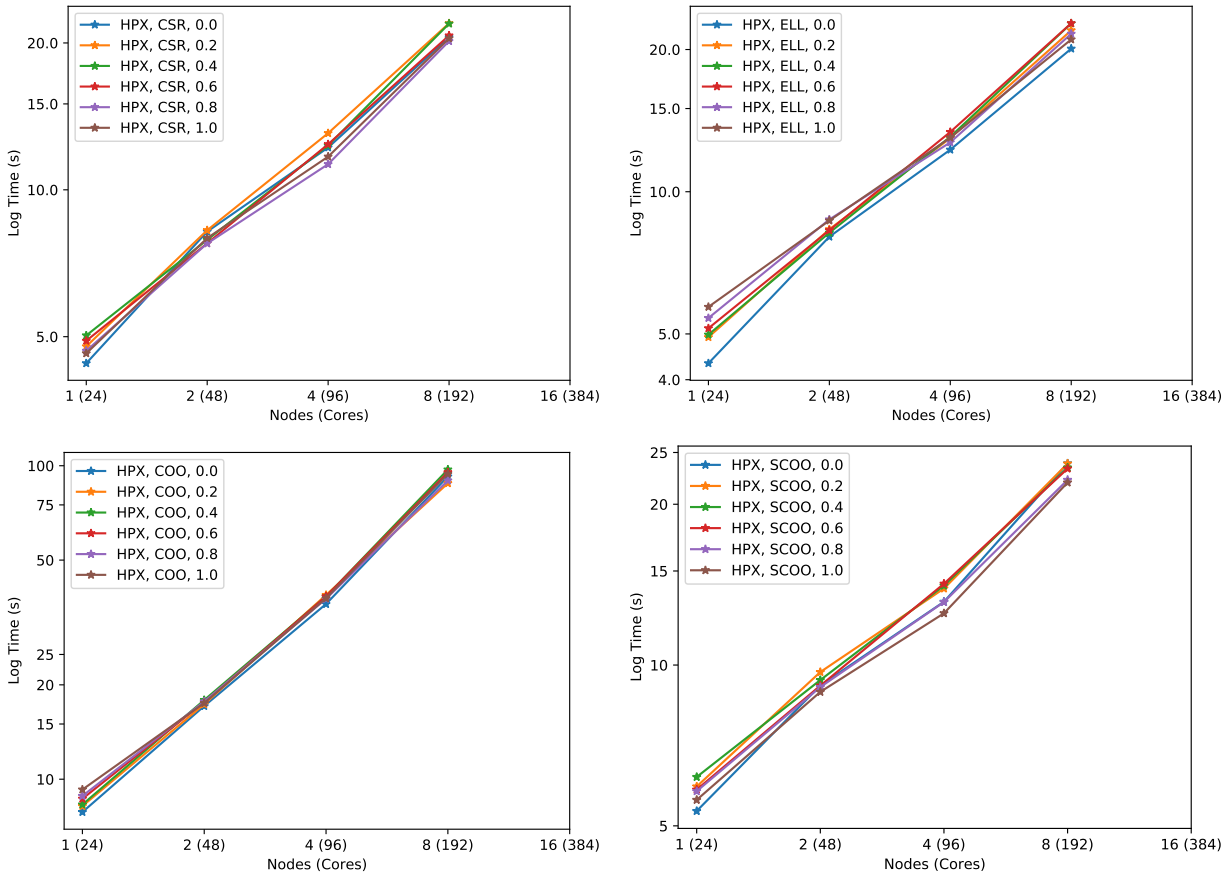


Figure 6.14: Weak scaling considering HPX and several storage formats for a 3 000 000 × 3 000 000 matrix per node with C = 300 on Pangea II. Legend is (model, format, Q).

Figure 6.14 summarizes the results obtained from the weak scaling experiments with HPX in which the operation $A(Ax+x)$ is applied on a sparse C-diagonal Q-perturbed matrices for C = 300 and a vector. This figure shows the execution time for the different sparse storage formats depending on the number of nodes (cores) as well as the size of the matrix that increase with the number of nodes. The execution times are similar to the ones obtain in Figure 6.11.

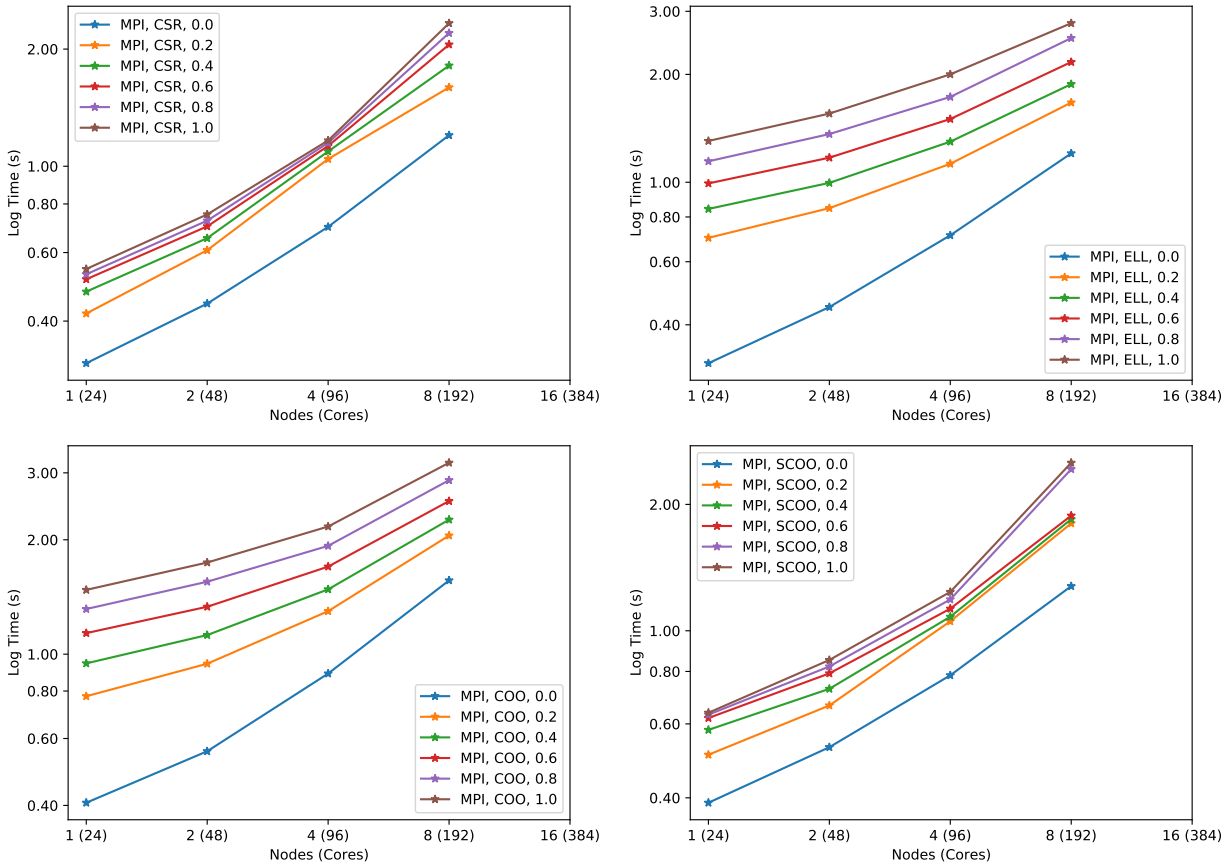


Figure 6.15: Weak scaling considering MPI and several storage formats for a 3 000 000 × 3 000 000 matrix per node with C = 300 on Pangea II. Legend is (model, format, Q).

Figure 6.15 summarizes the results obtained from the weak scaling experiments with MPI in which the operation $A(Ax+x)$ is applied on a sparse C-diagonal Q-perturbed matrices for C = 300 and a vector. This figure shows the execution time for the different sparse storage formats depending on the number of nodes (cores) as well as the size of the matrix that increase with the number of nodes. In this case, we obtain the same kind of results as obtained with the 4 000 000 × 4 000 000 in Figure 6.12 although the execution times are lower due to the smaller output vectors.

We performed weak scaling experiments on the operation $A(Ax+x)$ with a sparse C-diagonal Q-perturbed matrices of size 3 000 000 × 3 000 000 per node with C = 3. We multiplied the base size of the matrix (3 000 000 × 3 000 000) by the number of nodes that were used to execute the applications. These experiments were similar to the weak scaling experiments performed previously with a 4 000 000 × 4 000 000 base matrix. Similar execution times were obtained. This confirms the previous results and conclusions.

6.2.2.7 Influence of the Number of Values per Row

In this section, we perform strong scaling experiments on the operation $A(Ax+x)$ with a sparse C-diagonal Q-perturbed matrices of size 4 000 000 × 4 000 000 with several values of C. We try C equal to 75, 150, 225 and 300 with our MPI, HPX and YML+XMP applications for the COO, CSR, SCOO and ELL sparse storage formats.

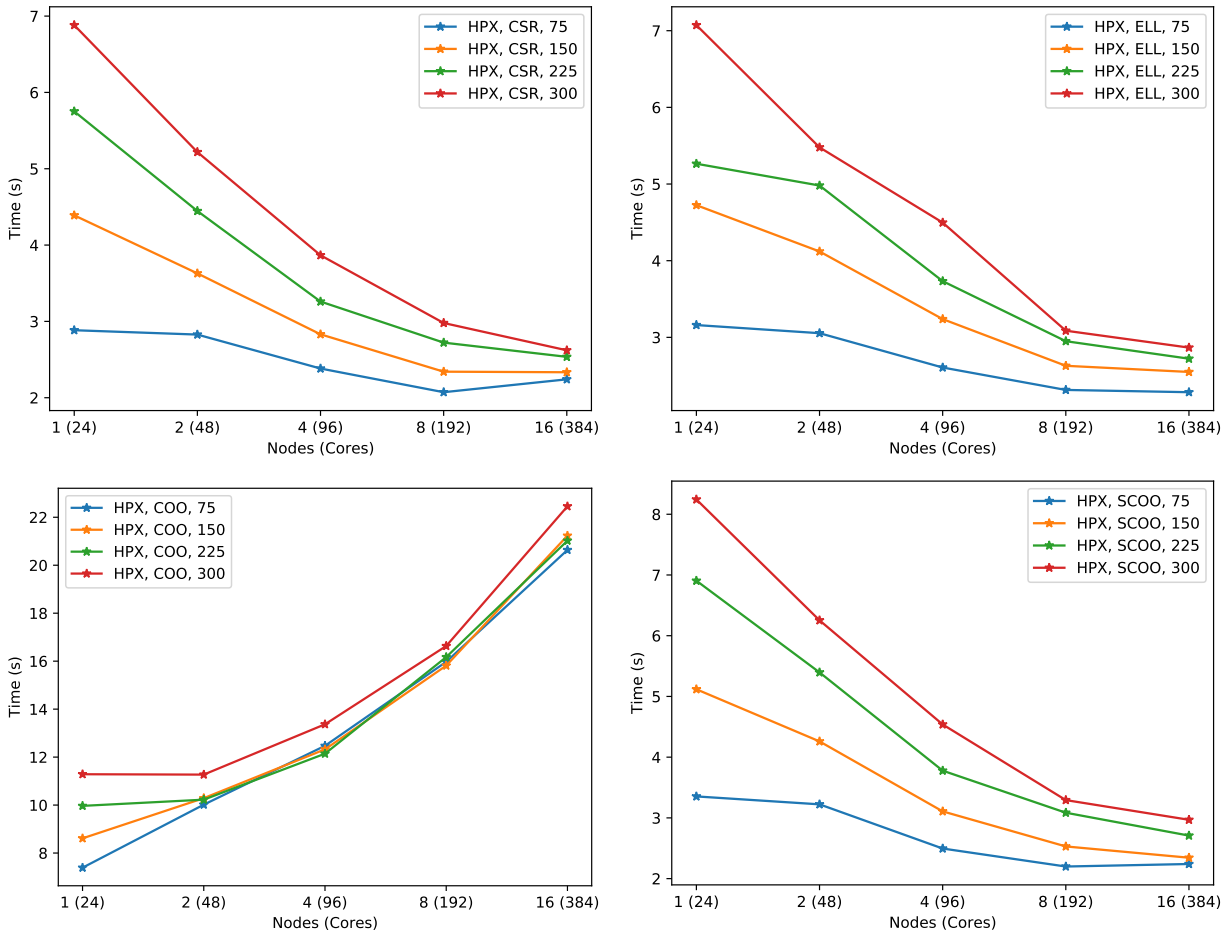


Figure 6.16: Variation of the number of values per row considering HPX and several storage formats for a 4 000 000 × 4 000 000 matrix per node with Q = 0.4 on Pangea II. Legend is (model, format, C).

Figure 6.16 summarizes the results obtained from the variation of the number of values per row (C) experiments with HPX in which the operation $A(Ax + x)$ is applied on a sparse C-diagonal Q-perturbed matrices of size 4 000 000 × 4 000 000 for Q = 0.4 and a vector. This figure shows the execution time for the different sparse storage formats depending on the number of nodes (cores). We find the same kind of performances patterns we get in the previous strong scaling experiments with HPX. The CSR, ELL and SCOO storage formats performances are improving when the number of nodes increases whereas it is not the case for COO due to the sum of the output vectors. The number of values per row only changes the time spent computing the local output vector in the kernel since these experiments use a constant matrix size. As such, for each value of C, the time spent out of the kernel is almost constant. Furthermore, the performances difference between the values of C is due to the different number of values to process in the kernel.

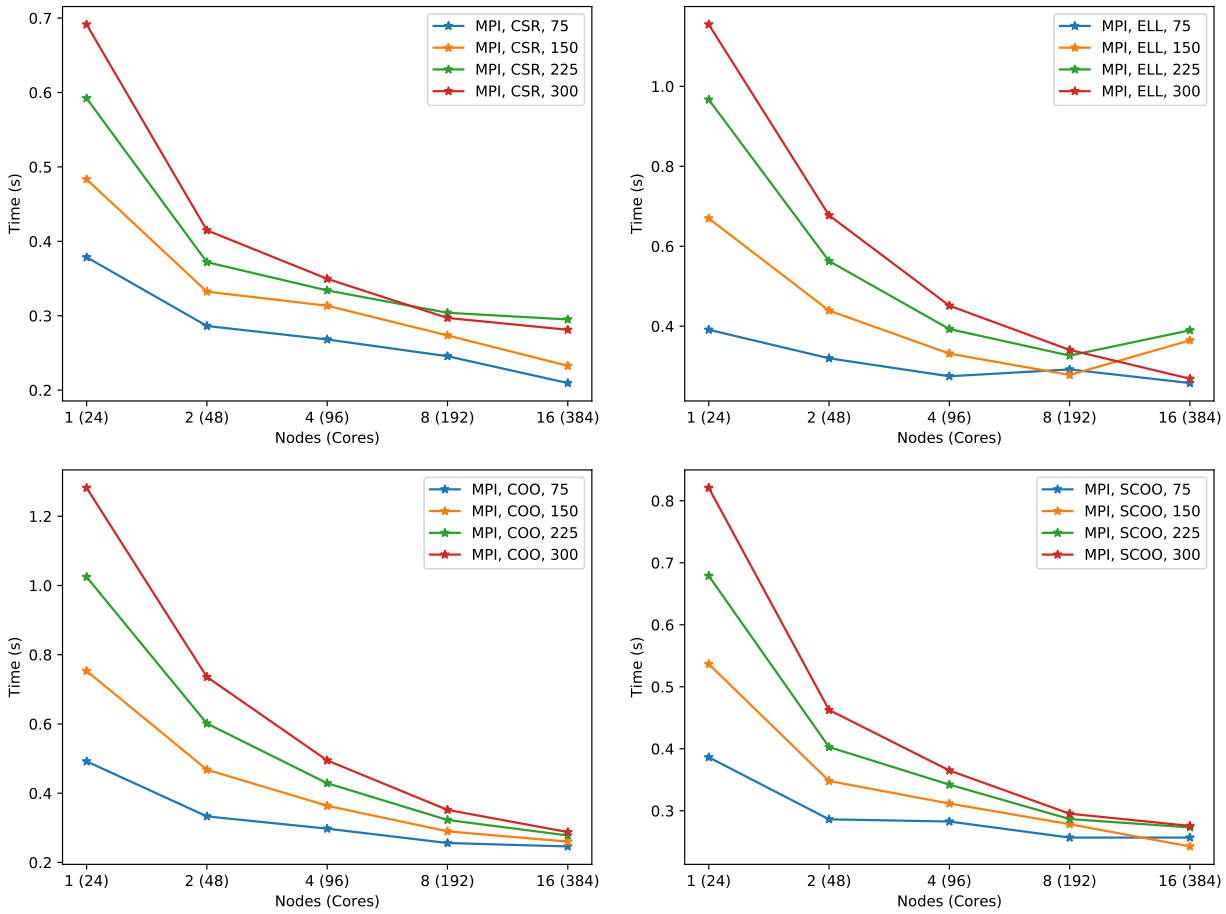


Figure 6.17: Variation of the number of values per row considering MPI and several storage formats for a 4 000 000 × 4 000 000 matrix per node with Q = 0.4 on Pangea II. Legend is (model, format, C).

Figure 6.17 summarizes the results obtained from the variation of the number of values per row (C) experiments with MPI in which the operation $A(Ax + x)$ is applied on a sparse C-diagonal Q-perturbed matrices of size 4 000 000 × 4 000 000 for Q = 0.4 and a vector. This figure shows the execution time for the different sparse storage formats depending on the number of nodes (cores). Since the matrix size stays constant, the input and output vectors size is also the same across the different values of C. Furthermore, the communications are performed on the same vector sizes and are taking the same amount on time independently of the value of C. We can see that the execution time increases with the value of C which is logical since the amount of data processed in the kernel increases.

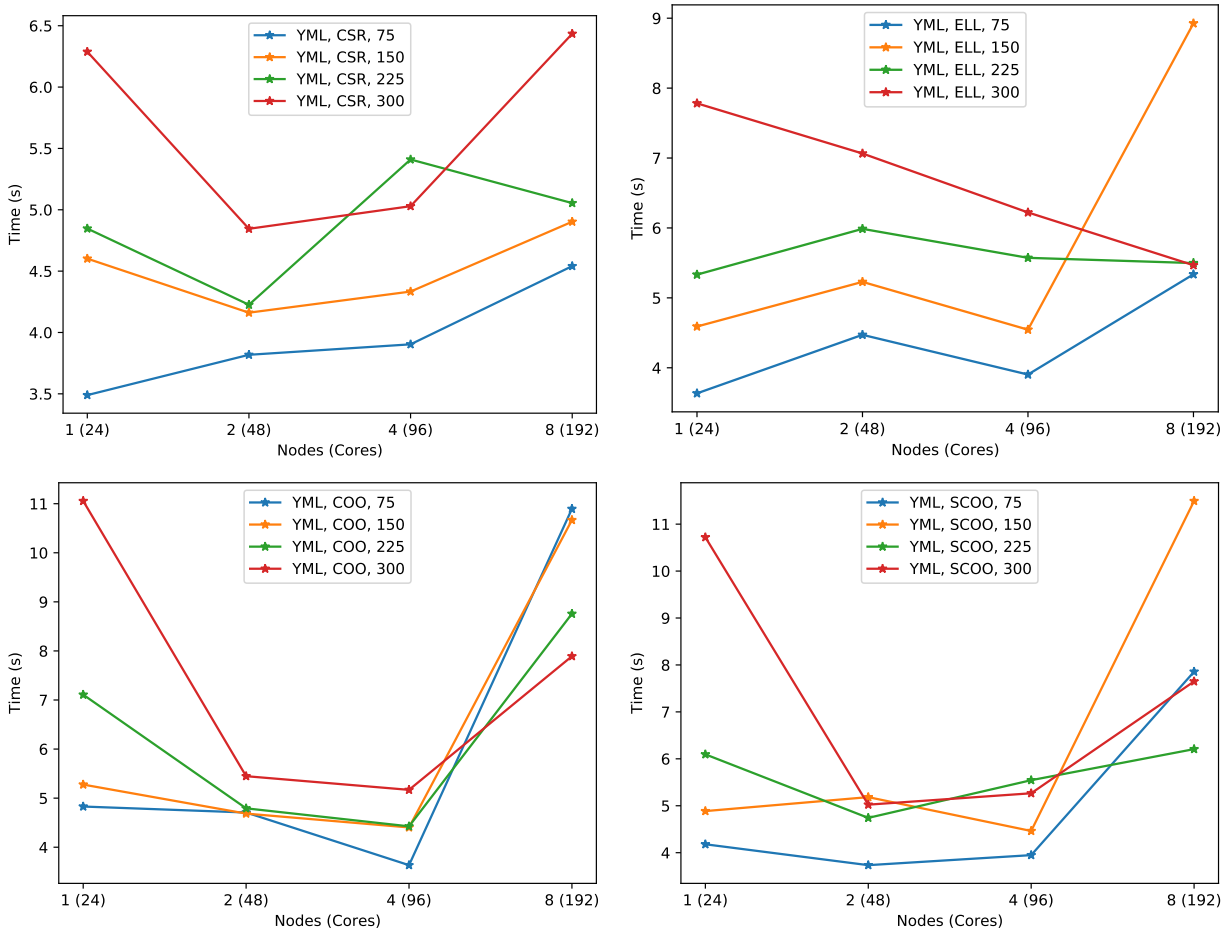


Figure 6.18: Variation of the number of values per row considering YML and several storage formats for a 4 000 000 × 4 000 000 matrix per node with Q = 0.4 on Pangea II. Legend is (model, format, C).

Figure 6.18 summarizes the results obtained from the variation of the number of values per row (C) experiments with YML in which the operation $A(Ax + x)$ is applied on a sparse C-diagonal Q-perturbed matrices of size 4 000 000 × 4 000 000 for Q = 0.4 and a vector. This figure shows the execution time for the different sparse storage formats depending on the number of nodes (cores). For the small number of nodes, the higher the value of C, the longer the application takes. However, when the number of nodes increases, it is less the case. As seen previously, YML has trouble improving its performances from 4 to 8 nodes in strong scaling like experiments. This may be due to the lack of experiments in YML+XMP like the existence of parameters that could produce better execution times. Moreover, there are cases where the application takes less time with larger matrices than with smaller matrices. This should not be the case but the YML+XMP communications between tasks through the file system induces large difference of performances.

In this subsection, we evaluated the influence of the number of values per row for the operation $A(Ax+x)$ in a sparse C-diagonal Q-perturbed matrices of size 4 000 000 × 4 000 000. We can see that HPX and MPI applications are becoming more efficient with the increase of the number of values per row since the applications are taking 2 more times to process 4 times more values. Indeed, the time spent in the kernel increases while the time spent in the communications stays constant when the number of values increases.

6.2.2.8 Experiments Conclusion

The sparse operation $A(Ax + x)$ has been implemented in three classes of programming models based on a sparse matrix vector kernel supporting the SCOO, COO, CSR and ELL storage formats. The first application is based on a message passing library, MPI which is used to execute the kernel on MPI processes and to make the necessary operations to create the complete output vector from the results of the kernel. The second application is based on a fine grain task based programming model in which dependencies between tasks are data oriented, HPX. HPX is used for the management of the data and the execution of the kernel in tasks. Finally, the third application is based on a parallel and distributed task based programming model in which tasks themselves are also parallel and distributed on a subset of the resources allocated to the application, YML+XMP. In YML+XMP, the tasks are parallel and distributed so there is multiple processes allocated per tasks.

The kernel also supports 2D grid data distribution of the matrix which makes the processing of the output vector from the kernel change depending on the data distribution. Thus, the performances of the applications depend on the data distribution. Hence, different data distribution are used in our tests. Furthermore, the number of processes allocated to YML tasks and the data distribution in and out of the tasks are parameters to consider. We tried as much parameters as we could for each programming model. However, there is a large amount of combination possible, especially for YML where there are more parameters to experiment with. Moreover, due to technical issues, YML+XMP applications cannot be executed with different parameters at the same time while it is possible to do so with MPI and HPX. Therefore, we were able to run enough experiments with different parameters for MPI and HPX to be reasonably confident that we found the best data distribution that executes the fastest for Pangea II. But this is not the case for YML+XMP since there is a lot more parameters to test and only one experiment can run at time whereas we were able to run up to 60 MPI and HPX applications at the same time. Hence, we may have missed parameters that may have been more efficient than the ones we have used.

YML+XMP also uses the file system to transmit data from one task to the other. The data are stored then loaded from files which is less efficient than in-memory communications on the current supercomputers. Programming models relying on the file system to transmit data between tasks are likely to run into the same issues where the IOs take most of the time and mask the influence of other parameters.

The dispersion of the values in the matrix also influences the performances of the kernel. The higher the dispersion, the more the output vector is randomly accessed which is very inefficient in term of memory accesses. This is especially visible with MPI in which there is less time spent outside of the kernel compared to the other programming models. Indeed, the performances differences due to the influence of the matrix dispersion is taken over by the reductions implemented in HPX and YML+XMP as well as the file system IOs for YML+XMP.

The output vector from the kernel is combined into a full vector that can be used to perform the second sparse matrix vector product in the sparse operation $A(Ax + x)$. This combination is performed with global sums and gathers. Theses operations are implemented through collective operations in MPI (reductions and gathers). However, these operations are not available in HPX and YML+XMP so they were implemented with the applications. They are not as efficient as the one included in MPI and it shows up in the performances. Although, they are not the only reason of the performance difference between the programming models.

6.3 Synthesis and Perspectives

In this chapter, we introduced the 2D block decomposition used to store our matrices in which each block is a sparse matrix compressed with a sparse storage format. Then, we introduced the task based algorithms that have been implemented with task based programming models as well as the algorithms implemented as task for each considered sparse storage format. We analyzed, evaluated and experimented the sequence of sparse operation $A(Ax+x)$ using HPX, a fine grain task based programming model with data-oriented dependencies, and YML+XMP, a task based programming model in which tasks are parallel and distributed, and MPI, a message passing library commonly used on supercomputers. The results are easy to extrapolate for larger sequences. We implemented the applications on top of a common kernel that performs the sparse matrix vector multiplication on matrices stored in CSR, ELL, COO and SCOO sparse classical storage formats. Our different experiments on a subset of a Petascale supercomputer with our C-diagonal Q-perturbed test matrices allow us to propose strong and weak scaling analysis, with respect to multiple values of C and Q, and several matrix distributions.

Our multi-level distributed and parallel experiments have shown that the dispersion of the matrix (Q) only influences the time spent in the kernel due to our data distribution and not the output vector construction which consists in collective operations. The CSR sparse format storage obtains the best performances in most of the cases. The COO storage format has better load balancing but requires larger output vector which increases the size of collective operations, therefore their execution time.

We have shown that collective operations are an issue in task-based programming models since most of them do not provide a highly optimized implementation and generate all-all communications. We also have shown that collective communications in MPI are taking more and more time as the number of computing resources used increases, especially for weak scaling experiments. Therefore, task-based programming models, that can avoid large scale collective communications by turning them in collective operations that can be efficiently scheduled or by running them on a subset of the allocated resources, is an interesting alternative to MPI.

However, current systems are more suitable for message passing libraries since both are designed and optimized to work well together. Moreover, file systems are not suitable either for programming models that use IOs to transfer data between tasks, it is the main reason that YML+XMP does not scale well above 4 nodes. The current task-based programming model schedulers lack the capability to completely manage the memory it uses and are not efficient enough to reduce the data migrations.

We did not implement a task based sparse matrix vector product with PaRSEC due to issues with the definition of new data types. PaRSEC provides dense matrix and vector types that can be used in applications as well as the possibility to register new types if needed. Nevertheless, we did not succeed in implementing types suitable to store the compressed matrices in the different sparse storage formats.

This work could be completed with applications implemented with other task based programming models such as X10, Chapel, Uintah or Charm++. These experiments could also be reproduced on other cluster or supercomputers. A possible amelioration is to improve the capabilities of the schedulers in terms of memory management and task scheduling in order to avoid the data migrations and fit data in the available memory of the nodes. Implementing efficient task based collective operations will also help to improve performances of task based programming models. Sequence of sparse matrix-vector multiplications on post-

petascale supercomputers would benefit of task-based programming paradigms when the supercomputer middleware and the IO file systems will be adapted.

The selected task based programming models are also used to implement a parallel and distributed task based Kirchhoff seismic pre-stack depth migration application in the following chapter. Then, we also perform numerical experiments on supercomputers as well as discuss the results obtained from these experiments.

Chapter 7

Task-Based, Parallel and Distributed Kirchhoff Seismic Pre-Stack Depth Migration Application

The Kirchhoff seismic pre-stack depth migration, previously introduced in Chapter 3, is a technique used to visualize the underground. It is often used in geoscience and is a large application developed in TOTAL. In this chapter the Kirchhoff seismic pre-stack depth migration is implemented with several task based programming models. The definition of the tasks is also introduced in this chapter. Moreover, we also perform numerical experiments on supercomputers as well as discuss the results obtained from these experiments.

7.1 Algorithms

In this section, the algorithms for the Kirchhoff seismic pre-stack depth migration is introduced. To begin with, the basic algorithm of the method is presented. Then, multi-grid algorithms are introduced. Finally, parallelism in the method is discussed and task based algorithms are shown.

7.1.1 Basic Algorithm

Algorithm 28 is the basic and most simple algorithm of the Kirchhoff seismic pre-stack depth migration. It is a basis on which the other algorithms are founded.

Algorithm 28 is computationally bound since the computation of the travel time between S and R through P takes a long time. Indeed, the travel time is computed by using Green functions which are heavy to compute from the initial velocity model. In practice two optimisations are used : multi-grids and pre-computation of the Green functions.

Algorithm 29 illustrates the use of two grids. In the multi-grids approach, the travel time is computed for the coarse grid then interpolated for the fine grid. The travel time for each point of the fine grid is interpolated from the travel time computed for the coarse grid.

Algorithm 30 shows the use of the two optimizations. The two grids are used and the Green functions are loaded from the file system.

Algorithm 28: Kirchhoff Migration

foreach *trace* $t_{S,R}$ **do**
- Read the source S and the receiver R from $t_{S,R}$.
- **foreach** *point P(x,y,z) of the image* **do**
  - Compute the travel time Ts between S and P (using Green functions)
  - Compute the travel time Tr between P and R (using Green functions)
  - Compute the amplitude $A_{S,R}(x, y, z)$.
  - $im(x, y, z) = im(x, y, z) + A_{S,R}(x, y, z) \cdot t_{S,R}(Ts + Tr)$

Algorithm 29: Kirchhoff Migration with coarse and fine grids

foreach *trace t* **do**
- **foreach** *(x, y, z) in the coarse grid CG* **do**
  - r = extractR(t)
  - s = extractS(t)
  - CG(x, y, z).TS(s) = computeTravelTimeFromVelocityModel(s, x, y, z)
  - CG(x, y, z).TR(r) = computeTravelTimeFromVelocityModel(r, x, y, z)
  - CG(x, y, z).A(s,r) = computeAmplitude(s, r, x, y, z)
  - **foreach** *(i, j, k) in the fine grid FG* **do**
    - CG(x, y, z).FG(i, j, k).TS(s) = interpolateTime(i, j, k, CG(x, y, z).TS(s), CG(x + 1, y, z).TS(s), CG(x - 1, y, z).TS(s), CG(x, y + 1, z).TS(s), CG(x, y - 1, z).TS(s), CG(x, y, z + 1).TS(s), CG(x, y, z - 1).TS(s))
    - CG(x, y, z).FG(i, j, k).TR(r) = interpolateTime(i, j, k, CG(x, y, z).TR(r), CG(x + 1, y, z).TR(r), CG(x - 1, y, z).TR(r), CG(x, y + 1, z).TR(r), CG(x, y - 1, z).TR(r), CG(x, y, z + 1).TR(r), CG(x, y, z - 1).TR(r))
    - CG(x, y, z).FG(i, j, k).A(s,r) = interpolateAmplitude(i, j, k, CG(x, y, z).A(s,r), CG(x + 1, y, z).A(s,r), CG(x - 1, y, z).A(s,r), CG(x, y + 1, z).A(s,r), CG(x, y - 1, z). A(s,r), CG(x, y, z + 1).A(s,r), CG(x, y, z - 1).A(s,r))
    - CG(x, y, z).FG(i, j, k).v += CG(x, y, z).FG(i, j, k).A(s, r) x ts,r(CG(x, y, z).FG(i, j, k).TS(s) + CG(x, y, z).FG(i, j, k).TR(r))

Algorithm 30: Kirchhoff Migration with multi-grids and loading precomputed Green functions

```
foreach trace t do
    r = extractR(t)
    s = extractS(t)
    Gs = loadPrecomputedGreenFunc(s)
    Gr = loadPrecomputedGreenFunc(r)
    foreach (x, y, z) in the coarse grid CG do
        CG(x, y, z).TS(s) = computeTravelTimeFromGreenFunc(Gs, x, y, z)
        CG(x, y, z).TR(r) = computeTravelTimeFromGreenFunc(Gr, x, y, z)
        CG(x, y, z).A(s,r) = computeAmplitude(s, r, x, y, z)
        foreach (i, j, k) in the fine grid FG do
            CG(x, y, z).FG(i, j, k).TS(s) = interpolateTime(i, j, k, CG(x, y, z).TS(s), CG(x + 1, y,
              z).TS(s), CG(x - 1, y, z).TS(s), CG(x, y + 1, z).TS(s), CG(x, y - 1, z).TS(s), CG(x, y, z +
              1).TS(s), CG(x, y, z - 1).TS(s))
            CG(x, y, z).FG(i, j, k).TR(r) = interpolateTime(i, j, k, CG(x, y, z).TR(r), CG(x + 1, y,
              z).TR(r), CG(x - 1, y, z).TR(r), CG(x, y + 1, z).TR(r), CG(x, y - 1, z).TR(r), CG(x, y, z
              + 1).TR(r), CG(x, y, z - 1).TR(r))
            CG(x, y, z).FG(i, j, k).A(s,r) = interpolateAmplitude(i, j, k, CG(x, y, z).A(s,r), CG(x + 1,
              y, z).A(s,r), CG(x - 1, y, z).A(s,r), CG(x, y + 1, z).A(s,r), CG(x, y - 1, z). A(s,r), CG(x, y,
              z + 1).A(s,r), CG(x, y, z - 1).A(s,r))
            CG(x, y, z).FG(i, j, k).v += CG(x, y, z).FG(i, j, k).A(s, r) x ts,r(CG(x, y, z).FG(i, j,
              k).TS(s) + CG(x, y, z).FG(i, j, k).TR(r))
```

7.1.2 Parallelism

In the Kirchhoff Migration, only the output image is modified. The traces and the Green functions are accessed only for reading. Therefore, the update of the image by the contribution of a trace conditions the parallelism available. Moreover, the traces are independent and the update of a point of the image depends only on the contribution of the trace.

It leans that **all the points can be updated at the same time by a trace**. It also means that there is concurrency problems if a point is updated by two traces. Actually, since the update is a sum, it is worth considering to create images for different sets of traces independently then reduce (sum) the different images into a final image.

From a computational point of view, traces can be treated in any order and points can be computed independently. Several traces can be treated at the same time with reductions on the output images. So there is a lot of available computational parallelism but there is also data to transfer between compute units and from the file system. Moreover, creating several images to reduce them later also produces communications to reunite them into one. In the case where the Green functions are precomputed, loading them into the memory can be an issue. They take a lot of space on disk so it is expensive to load them several times.

Thus, scheduling those data movements can be a good alternative to find an efficient way to manage the data without having to communicate too much. In order to better express the dependencies, the method is described as a graph of tasks.

7.1.3 Task algorithm

The algorithms are rewritten to allow the use of tasks. The task algorithm is the Algorithm 31 and the function associated can be found in the Algorithm 32.

Algorithm 31: Task Kirchhoff Migration

```
foreach trace t do
    r = extractR(t)
    s = extractS(t)
    Gs = loadPrecomputedGreenFunc(s)
    Gr = loadPrecomputedGreenFunc(r)
    foreach (x, y, z) sub-block of the coarse grid CG do
        CG(x, y, z).TS(s) = computeTravelTimeFromGreenFunc(Gs, x, y, z)
        CG(x, y, z).TR(r) = computeTravelTimeFromGreenFunc(Gr, x, y, z)
        CG(x, y, z).A(s,r) = computeAmplitude(s, r, x, y, z)
        updateFineGrid(CG(x, y, z), CG(x + 1, y, z).TS(s), CG(x - 1, y, z).TS(s), CG(x, y + 1,
         z).TS(s), CG(x, y - 1, z).TS(s), CG(x, y, z + 1).TS(s), CG(x, y, z - 1).TS(s), CG(x + 1, y,
         z).TR(r), CG(x - 1, y, z).TR(r), CG(x, y + 1, z).TR(r), CG(x, y - 1, z).TR(r), CG(x, y, z +
         1).TR(r), CG(x, y, z - 1).TR(r), CG(x + 1, y, z).A(s,r), CG(x - 1, y, z).A(s,r), CG(x, y + 1,
         z).A(s,r), CG(x, y - 1, z). A(s,r), CG(x, y, z + 1).A(s,r), CG(x, y, z - 1).A(s,r))
```

In Algorithm 31, the tasks on the blocks of the coarse grid for a given trace can be executed in parallel.

Algorithm 32: Task definition for Kirchhoff Migration

Task *updateFineGrid(B:current block:inout;*
TSu, TSb, TSr, TSl, TSf, TSh:TravelTime;in
TRu, TRb, TRr, TRl, TRf, TRh:TravelTime;in
*TSu, TSb, TSr, TSl, TSf, TSh:Amplitude;in)***is**
  foreach *(i, j, k) in the fine grid FG* **do**
    B.FG(i, j, k).TS(s) = interpolateTime(i, j, k, B.TS(s), TSu, TSb, TSr, TSl, TSf, TSh)
    B.FG(i, j, k).TR(r) = interpolateTime(i, j, k, B.TR(r), TRu, TRb, TRr, TRl, TRf, TRh)
    B.FG(i, j, k).A(s,r) = interpolateAmplitude(i, j, k, B.A(s,r), TSu, TSb, TSr, TSl, TSf, TSh)
    B.FG(i, j, k).v += B.FG(i, j, k).A(s, r) × ts,r(B.FG(i, j, k).TS(s) + B.FG(i, j, k).TR(r))

In Algorithm 32, the loop on the elements of the fine grid can also be executed in parallel. There is at least two level of parallelism.

7.1.4 Use of GPUs

This section aims to add the use of GPUs in the Kirchhoff seismic pre-stack depth migration. GPUs are slower than CPUs but there a lot more computing unit in GPUs. They are used to process huge amount of parallel operations and they are very useful to perform simple operations on huge amount of data. Therefore, most of the operations inside the fine grid can be performed on GPU since the operations are the same on all the data. Indeed, the operation is a linear combination of several arrays at the same point.

7.1.5 Pre-fetching and spreading the data

Since the main issue is the communications between the storage system where the Green functions are and the computing units inside the super-computer, a solution would be to pre-fetch the data. The Green functions would be stored in a storage system closer of the computing unit and faster. It is possible to efficiently predict which Green function is needed by the program when the source and the receiver of the trace are loaded.

Moreover, two block side by side share the Green functions at their frontier. That means the same Green functions are loaded several times in a short amount of time from the file system. The bandwidth between the file system and the computing unit is generally lower than the one between two nodes of the super-computer so use them will be more efficient. To do so, the pre-fetcher will be modified to become aware of the tasks that uses the Green functions. Then, if the Green function a task may want is still loaded in another task, the task using the Green function will send it to the task that also need it. It will reduce the communications to the file system containing the Green functions and so, make the whole process faster.

To go further, a computing node of the super-computer may be able to store some of the Green functions even if it does not use them in order to provide them to the other nodes as long as possible.

7.2 2D Implementation

For this implementation, we simplified the Kirchhoff seismic pre-stack depth migration by considering the absorption to be equal to 1 and by only studying the 2D case. This implementation is a very basic and simplified version of the Kirchhoff seismic pre-stack depth migration which keeps a similar high level algorithm while simplifying the computations needed to perform the migration on the image points. This application is available at https://github.com/jgurhem/KirchhoffMigration.

7.2.1 C Kernel Description

The kernel performing the main operations for this method is implemented in C and provide functions to manage traces, Green functions and images. There is three kinds of functions implemented in this kernel. The first type is IOs to read and write traces, images and Green functions from files. The second type of function is used to associate propagation time precomputed with Green functions to the trace that are used to create the image. Then, the last type is the actual migration.

The migration function expect a fine grain image to update and a trace with the propagation times to make the round trip from the source of the trace to the receiver passing by every point of the large grain image. It performs an interpolation to compute the propagation time needed to travel from the source to the receiver for each point of the fine grain image by using the propagation times shipped with the trace. So, with the knowledge of the time it would take to make the round trip to the considered image point, we scale it to the trace time scale in order to retrieve the amplitude of the sound wave at this time in the trace. Then, the amplitude is added to the value at the corresponding point in the fine grain image. This amplitude could be scaled with the absorption at this point but we set the absorption to 1 for this implementation so this is not necessary here. The migration function has OpenMP directives to parallelize the work on the image. They can be activated by compiling the library with OpenMP.

For our experiments, we use a use constant velocity model to generate the initial propagation times. Therefore, we implemented a function to interpolate propagation times on a large grain grid from the constant velocity model. A function to generate simple traces has also been implemented to use in our experiments.

Figure 7.1 gives an example of a generated trace used to test the 2D application with these simplifications. This trace is a sample used in our experiments.

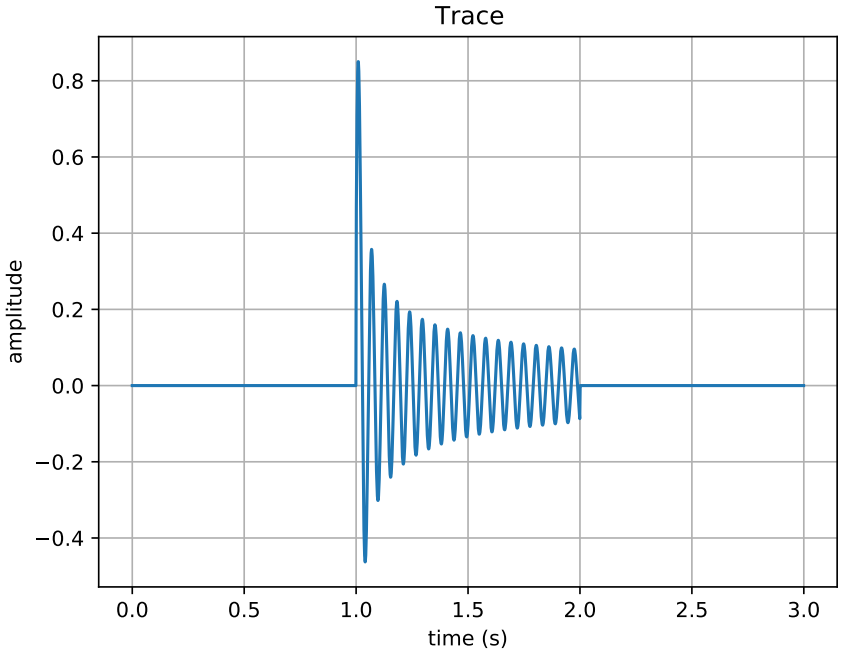


Figure 7.1: Trace example

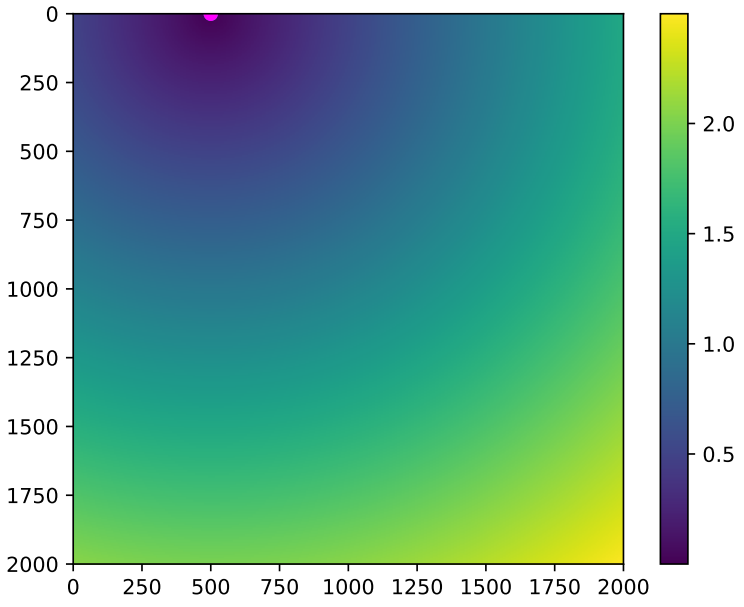


Figure 7.2: Propagation times for a source and receiver at the same place

Figure 7.2 is a plot of the propagation times to make the round trip from the source point located at (500,0), each point of the figure and the receiver located at (500, 0) in a uniform middle.

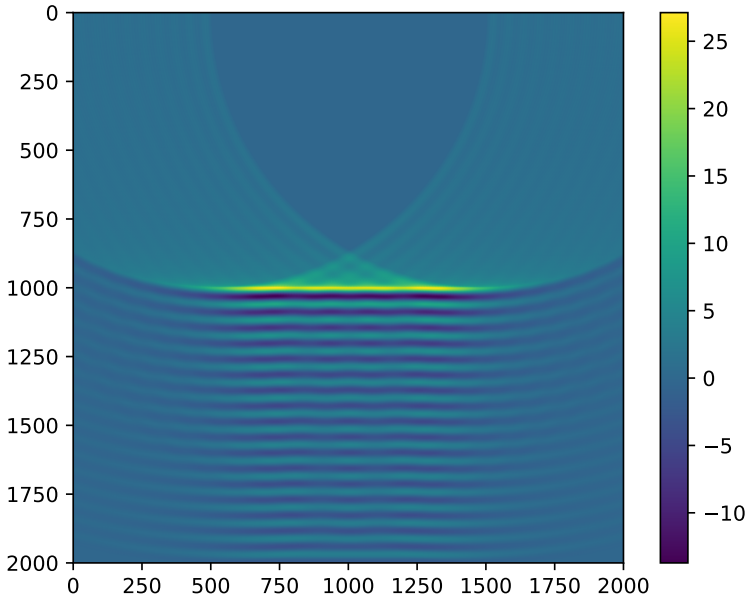


Figure 7.3: Image obtained from a migrated trace

Figure 7.3 is the migrated image produced by the Kirchhoff migration with the trace of Figure 7.1 taken at several different source locations (the source and the receiver are located at the same place).

7.2.2 Distributed and Parallel Applications

Distributed and parallel applications were implemented on top of the C kernel in MPI and HPX.

The MPI application uses the C functions to generate the traces and propagation time associated on the distributed processes. The image is distributed across the processes allocated to the application. The traces are duplicated on each process since they are accessed depending on the travel time extracted from the propagation time shipped with the trace. The propagation times are generated with the trace and only the part that is used in the migration is selected. Then, the migration can be executed on the local image with the local trace and propagation times.

The HPX application is based on same principles. The image is split and managed by tasks. A task calling the migration available in the kernel has been implemented as well as tasks to generate the traces and the propagation times. There is also tasks to associate the trace with the appropriate propagation times.

These applications support the generation and processing of several traces. They were designed to perform scaling experiments on the migration with a distributed image.

7.3 2D Numerical Experiments

In this section, we perform numerical experiments on the Kirchhoff seismic pre-stack depth migration implemented in MPI, MPI+OpenMP and HPX on Pangea II. We perform strong scaling and weak scaling

experiments as well as we study the influence of the number of OpenMP threads on our kernel.

7.3.1 Strong Scaling

Strong scaling experiments were performed on a 15000 × 15000 points image. We executed our Kirchhoff seismic pre-stack depth migration MPI, MPI+OpenMP and HPX applications on 10 generated traces with propagation times. We study the performances improvement of the migration while the number of nodes (cores) increases and the size of the image is kept constant.

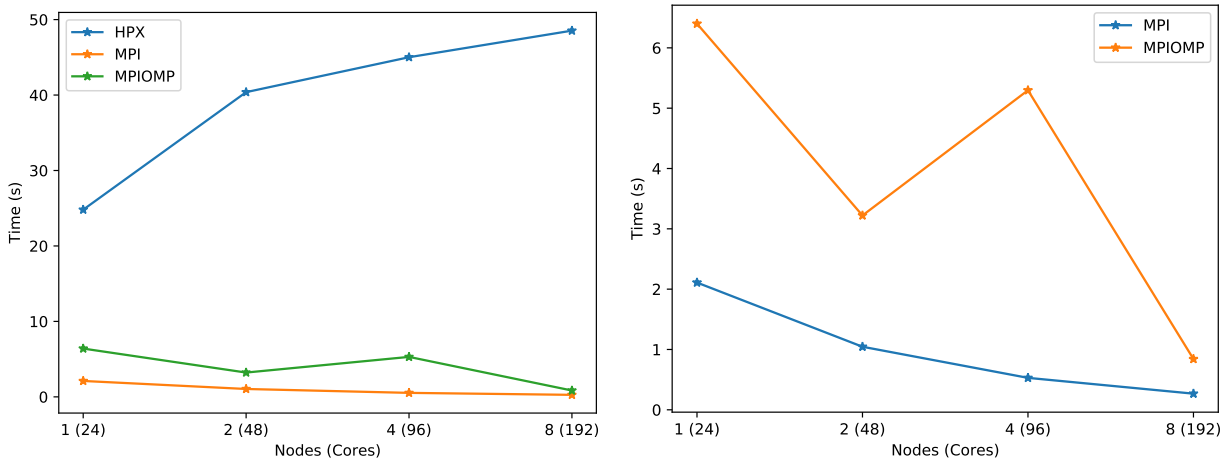


Figure 7.4: Strong scaling considering HPX, MPI and MPI+OpenMP for a 15000 × 15000 points image on Pangea II. Legend is (model).

Figure 7.4 shows the results of our strong scaling experiments on the Kirchhoff seismic pre-stack depth migration for a image of size of 15000 × 15000 and 10 traces. HPX execution times are increasing with the increase of computing resources which should not be the case.

The MPI and HPX applications use the same kernel on the same division of the data to perform the computations but the performances are different than MPI's. This may be due to the internal management of the memory in HPX as we have to convert the C data structures in C++ data structures that HPX can work with. Indeed, HPX can only migrate data structures that satisfies its requirement in order to be able to send them between nodes. However, the plain C pointers and data structures used to hold the data for our C kernel are not recognized by HPX. Therefore, we had to wrap our C data structures into C++ data structures that HPX can migrate. While doing so, we may have also introduced memory management issues that arise with the new C++ data structures and HPX data management system.

Moreover, HPX may induce image migration between nodes that are not necessary. HPX assigns data to a locality which represents a physical node. A trace may be assigned to a different location than the one of the image on which the trace will be migrated. At this moment, either the image or the trace has to be migrated to compute the new image. The image is larger than the trace so if the image is transfered between the nodes, it is not efficient. Besides, the traces can be used on several sub-images at the same time. This one-to-many dependency may not be properly understood by HPX, especially in the case where the data could be duplicated to run tasks on multiple instances of the data at the same time. In this case,

the one-to-many dependencies between the trace and the sub-image could be transformed by HPX into a sequence of tasks processing sub-images one by one instead of processing them at the same time. It could restrict the execution flow of the tasks and reduce the performances.

MPI+OpenMP has issues when scaling from 2 to 4 nodes but scales well from 4 to 8. It has performances close to MPI, especially for 8 nodes. MPI has the best performances and also scales well. The applications use the same kernel but the management of the memory is up to the programming model used to implement the application. This explain the difference of performances between MPI and HPX. As for the differences between MPI and MPI+OpenMP, the OpenMP directives introduced in the kernel does not seem to take advantage of the cores available on the nodes since we use 4 processes per nodes and 6 OpenMP threads per process.

7.3.2 Weak Scaling

Weak scaling experiments were performed on a image which number of points increases with the number of nodes used. The base image size is 15000 × 15000 and the first dimension is multiplied by the number of nodes. For instance, the image used for 4 nodes is 60000 × 15000. We executed our Kirchhoff seismic pre-stack depth migration MPI, MPI+OpenMP and HPX applications on 10 generated traces with propagation times. We study the performances improvement of the migration while the number of nodes (cores) and the size of the image increases.

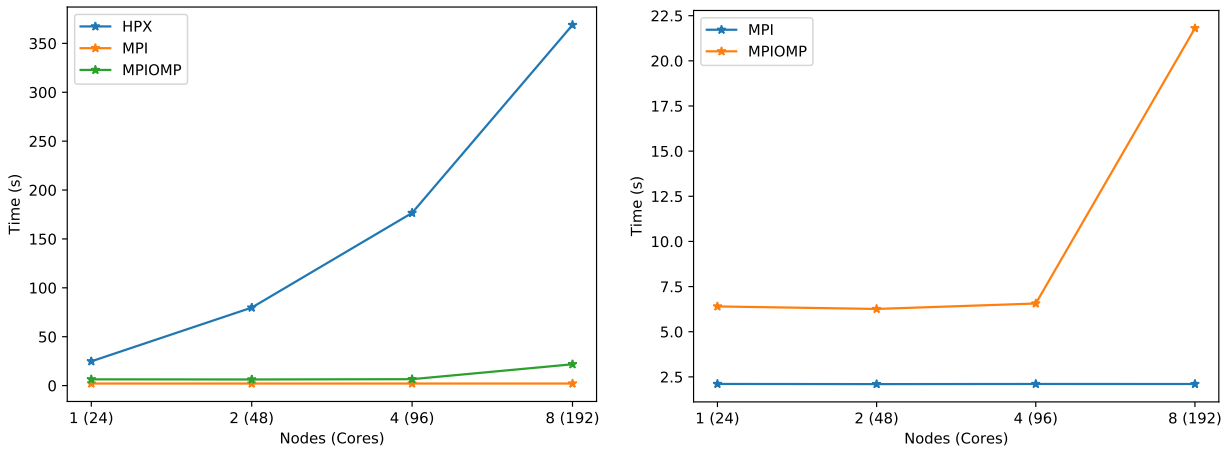


Figure 7.5: Weak scaling considering HPX, MPI and MPI+OpenMP for a 15000 × 15000 points image on Pangea II. Legend is (model).

Figure 7.5 shows the results of our weak scaling experiments on the Kirchhoff seismic pre-stack depth migration for a image of base size of 15000 × 15000 which grows with the number of nodes (cores) and 10 traces. The ideal weak scaling is a constant execution time when the computing resources allocated to the application are increasing in the same proportions as the data. This is not the case for HPX. This may be due to HPX memory management and unexpected image and trace migrations between nodes which we are not able to control. MPI+OpenMP weak scaling the almost perfect until 8 nodes where there is a issue and the execution time is abnormally high. The MPI application has a weak scaling almost perfect.

7.3.3 Variation of the Number of OpenMP Threads

Experimentations on the MPI+OpenMP Kirchhoff seismic pre-stack depth migration application with different number of OpenMP threads allocated to the application have been performed for an image of 15000 × 15000 points. On one hand, we used only one process and changed the number of OpenMP threads allocated to the application until all the cores of the node are used. On the other hand, we used all the cores of the node but we changed the number of processes allocated to the MPI+OpenMP as well as the number of OpenMP threads allocated per process in order to keep one thread per core. We test the performances of our migration kernel with the addition of the OpenMP directives.

Threads	Cores	Median
2	24	63.383
4	24	65.3781
6	24	71.5322
12	24	76.7288
24	24	254.1039

Table 7.1: Execution time for a pure OpenMP application while increasing the number of OpenMP threads allocated to the application for a 15000 × 15000 point image on Pangea II.

Table 7.1 shows the execution times of the kernel depending on the number of threads allocated to the pure OpenMP application on one node of Pangea II. We can see that the execution time does not change much while increasing the number of threads except for 24 threads where it is very high. The execution time should decrease with the increase of parallel resources to solve the same problem. This shows that our OpenMP implementation of the migration kernel is not efficient enough to take advantage of the available cores.

Processes	Cores	Median	Threads
2	24	46.7809	12
4	24	6.3982	6
6	24	4.3045	4
12	24	8.7822	2
24	24	2.0901	1

Table 7.2: Execution time for a hybrid MPI+OpenMP application while increasing the number of MPI process while keeping 24 OpenMP threads allocated to the application for a 15000 × 15000 point image.

Table 7.2 shows the execution times for the MPI+OpenMP executed on one node of Pangea II where all the cores run an OpenMP thread and the number of MPI processes changes. The number of thread allocated for each MPI process is the number of cores (24) divided by the number of MPI processes allocated to the application. We can clearly see that the pure MPI application (24 processes and 1 thread per process) is the fastest case. There is also two unexpected values ; for 4 processes with 6 threads per process and for 12 processes with 2 threads per process. These values does not line up with the rest. Especially the value for 12 processes with 2 threads per process that should reasonably be between the value for 6 and 24 processes.

The migration kernel has to make interpolation of the wave travel time from the receiver and the source to reach the considered point. This interpolation accesses the array containing the propagation times to perform the interpolation at the coordinates of the point. These accesses may be difficult to predict for the compiler thus the OpenMP runtime is not able to efficiently divide and parallelize the loops during the iteration over the image points while computing the interpolation of the travel times at the point. Thus, our implementation of the kernel of the kernel may not be written in a suitable way for the compiler and the OpenMP runtime to efficiently parallelize it. Therefore, using a pure MPI implementation avoid this problem by splitting the image and the propagation times beforehand at the process level. Then, the kernel is executed on each sub-image without the necessity to try to efficiently parallelize the loops iterating over the points to perform the interpolations since they are already split before calling the kernel.

In conclusion of our scaling experiments on the implementations of the Kirchhoff seismic pre-stack depth migration, our HPX application does not scale very well both in term of weak and strong scaling compared to our MPI application. In the MPI application, the location of each piece of data can be exactly controlled and there are no communications during the migration. On the contrary, the data migrations in our HPX application are up to the runtime of the programming model. Therefore, there may be migration of the images and the traces across the nodes which reduce the performances. Besides, the one-to-many dependencies between a trace and the sub-images may be expressed sequentially in HPX instead of in parallel which could restrict the execution and reduce the performances. Moreover, there is a conversion between the C data structures used in the C kernel and the C++ data structure used in HPX that may also decrease the performances of the application since the data are not converted in the C based MPI application. Finally, the introduction of OpenMP directives in the kernel is not successful since the performances are not improved with an hybrid MPI+OpenMP implementation compared to a pure MPI implementation.

7.4 Synthesis and Perspectives

In this chapter, we introduced task based algorithms for the Kirchhoff seismic pre-stack depth migration and the parallelism intrinsic to the method. Then, we introduced and implemented a simplified 2D version of the Kirchhoff seismic pre-stack depth migration as a task based application with HPX. We also implemented a MPI and a MPI+OpenMP applications to compare with our task based implementation. Then, we performed strong scaling and weak scaling experiments as well as studied the influence of the number of OpenMP threads on our kernel. We showed that our HPX application does not scale very well both in term of weak and strong scaling compared to our MPI application. We deduced that there were migrations of the images and the traces across the nodes which reduced the performances. We also showed that our addition of OpenMP directives in the kernel did not bring as good performances as our pure MPI application.

Task based programming models are an interesting alternative to MPI due to their capacity to schedule computations and data migrations in order to reduce costly communications while optimizing the use of the computing resources. Therefore, the scheduler could anticipate data migrations and, in particular, IOs with the file system in order to load data in advance so that they are ready to be used. However, the current task-based programming model schedulers lack the capability to completely manage the memory it uses and are not efficient enough to reduce the data migrations. Moreover, current systems are more suitable for message passing libraries since both are designed and optimized to work well together.

Our task based Kirchhoff seismic pre-stack depth migration could be improved to be closer to what is done in practice as well as include IOs tasks to test the ability of the task based programming models to schedule such tasks. The application could also be implemented with other task based programming models to explore their capabilities to solve such problem.

The experiments and the studies made with the tasks based programming models allowed the selection of key features for the task based programming models. We resume these key features and how they are expressed in the task based programming models in the following chapter.

Chapter 8

Taxonomy of Task-Based Programming Models and Recommendations

In this chapter, the taxonomy and the properties extracted from the usage of several task based programming models is introduced. This taxonomy also presents how each property is expressed in the different task based programming models. This work was introduced in [123] and is extended in this dissertation. Then, we provide recommendations to improve capabilities and performances of task based programming models. Finally, we present a methodology for parallel and distributed programming based on graphs of tasks for post-petascale supercomputers.

8.1 Taxonomy

In this section, we introduce the taxonomy of the task based programming models that is deduced from the studies and experiments. We divided the features of the taxonomy into three categories. *Task Capabilities* is the first category which contains the features related to the abilities of the tasks such as the granularity of the tasks and the architectures they can be executed on. The second category is *Task and Data Management* which contains the features related to the scheduling properties of the tasks and data migrations. The last category is *Programming Model Features* in which we present capabilities of the programming model itself such as data persistence and fault tolerance.

8.1.1 Task Capabilities

In this section, the features related to the category *Task Capabilities* are presented. They include task features that we believe interesting in the choice of a task based programming model to implement the tasks in a task based application. The *Task Granularity* can help to understand the scale and the amount of computations of the tasks. The memory *Architecture* is also a metric that help to understand the scale at which the task based programming models can be used to implement an application. The *Heterogeneity* and the *Portability Accelerator* help to understand if the task can be executed on different accelerators with the same implementation and on which type of accelerators. This information is critical depending on the cluster or supercomputers architectures targeted. The *Task Implementation* represents the interface available to user to implement the tasks and the kind of effort needed to port an existing kernel application into a

task. Finally, the *Data Handling* shows how the data are accessed from the tasks and can help to understand the effort necessary to use and perform operations on the data in the tasks.

8.1.1.1 Task Granularity

This property represents the amount of resources on which a task can be executed. For this study, we consider that either *sequential*, *parallel* or *parallel and distributed* code can be executed as task. Usually, a sequential task is run as a lightweight thread, a thread or a single process without multi-threading. A parallel task can also use multiple threads or processes up to one node like OpenMP or MPI on shared memory. Finally, distributed and parallel resources like multiple nodes from a cluster can be allocated to parallel and distributed tasks. These tasks could be able to execute MPI or PGAS based code on their allocated resources.

As seen in Chapter 5, a good balance between the number of tasks and the work load given to the task is required to express enough parallelism so that the computing resources can execute tasks and to make sure there is not too much overhead from having too many tasks. Therefore, sequential tasks are more likely to be fine grain tasks that process a relatively small amount of data while distributed and parallel tasks are more likely to process more data and have a larger grain. For instance, YML+XMP tasks are distributed and parallel while HPX, PaRSEC and Regent tasks are sequential tasks that run in threads. In Table 5.4, we explore the number of blocks used in YML+XMP, HPX, PaRSEC and Regent to get the best results for our task based LU factorization for a 16384 $\times$ 16384 matrix. The size of the block can be computed by dividing the size of the matrix by the number of blocks. Therefore, the higher the number of blocks, the smaller the size of the block is. In this case, there is up to 8 $\times$ 8 blocks for YML+XMP whereas there is at least 30 $\times$ 30 blocks for Regent, HPX and PaRSEC. Thus, YML+XMP distributed and parallel tasks managed around 16 times ($\sim (30/8)^2$) more data than the largest task executed by Regent, HPX and PaRSEC. Moreover, these tasks are matrix products and matrix inversions which correspond to $\sim n^3$ operations where n is the dimension of the matrix. So, YML+XMP tasks performed around 64 times ($\sim (30/8)^3$) more operations.

In Table 8.1, the granularity of the tasks of the considered task based programming models is presented. Most of the task based programming models use sequential tasks. For our studies and experiments, we used task based programming models for distributed memory with sequential tasks and with parallel and distributed tasks.

Table 8.1: Task Granularity property for each task based programming model

Programming model	Task Granularity	Programming model	Task Granularity
Chapel	Sequential	PaRSEC (PTG)	Sequential
Charm++	Sequential	Pegasus	Parallel and Distributed
Cilk	Sequential	PyGion	Sequential
ClusterSs	Sequential	QUARK	Parallel
CnC	Sequential	Regent	Sequential
Cpp-TaskFlow	Sequential	StarPU	Sequential
HPX	Sequential	Swift	Parallel and Distributed
HTCondor DAGMan	Parallel and Distributed	TBB	Sequential
Kokkos	Sequential	TensorFlow	Sequential
Legion	Sequential	Uintah	Sequential
OmpSs	Sequential	X10	Sequential
OpenMP	Sequential	YML	Sequential
PaRSEC (DTD)	Sequential	YML+XMP	Parallel and Distributed

8.1.1.2 Architecture

The architecture supported by the task based programming models is an important property to take into consideration since it will impact the scale at which the task based programming models can be used. There is two main scale used in high performance computing : the shared memory and the distributed memory. Shared memory involves the usage of limited resources since these task based programming models can only be used on one cluster or supercomputer since they do not provide means to transfer data between several nodes. They can be used with data transfer libraries to address distributed memory. One of such case is the use of OpenMP on the nodes and MPI to manage distributed memory. They can also be used in a task that could run on a complete node. On the other hand, with distributed memory, it is possible to address multiple nodes, thus, as much resources as available. In this case, the programming models that can manage distributed memory are able to migrate data between cluster nodes. Therefore, the two values considered for this property are the *shared memory* and the *distributed memory* architectures.

Shared memory task based programming models cannot be used to address exascale computing without being used with another technology to manage distributed memory.

In Table 8.2, the architecture which is supported by the considered task based programming models is introduced. Task based programming models that manage shared memory can only execute sequential tasks expect for QUARK that can execute multi-threaded functions. On the other hand, some of the task based programming models that manage distributed can execute parallel and distributed tasks.

Table 8.2: Architecture property for each task based programming model

Programming model	Architecture
Chapel	Distributed Memory
Charm++	Distributed Memory
Cilk	Shared Memory
ClusterSs	Distributed Memory
CnC	Shared Memory
Cpp-TaskFlow	Shared Memory
HPX	Distributed Memory
HTCondor DAGMan	Distributed Memory
Kokkos	Shared Memory
Legion	Distributed Memory
OmpSs	Shared Memory
OpenMP	Shared Memory
PaRSEC (DTD)	Distributed Memory
PaRSEC (PTG)	Distributed Memory
Pegasus	Distributed Memory
PyGion	Distributed Memory
QUARK	Shared Memory
Regent	Distributed Memory
StarPU	Distributed Memory
Swift	Distributed Memory
TBB	Shared Memory
TensorFlow	Shared Memory
Uintah	Distributed Memory
X10	Distributed Memory
YML	Distributed Memory
YML+XMP	Distributed Memory

8.1.1.3 Heterogeneity

This property indicates if the programming model supports accelerators (for instance, GPUs). *Explicit* support means that the user has to provide the implementation of the task that will be run on the accelerator. *Implicit* support means that the tasks can be run on different devices while the user has to provide only one implementation. In this case, the programming models also manage data transfer between the CPUs and the GPUs memory. For instance, tasks implemented with OpenMP can be run both on CPUs and GPUs while OpenMP manages the data offloads to the GPU.

In Table 8.3, it is shown if the accelerator support is explicit or implicit for the considered task based programming models. Most of the task based programming models do not support a direct implementation of the tasks for accelerators. Indeed, the users have to provide such implementation if they want to execute tasks addressing accelerator computations. Our experiments were mostly performed on CPUs so this feature was not used during our experiments.

Table 8.3: Heterogeneity property for each task based programming model

Programming model	Heterogeneity
Chapel	Implicit
Charm++	Implicit or Explicit
Cilk	Explicit
ClusterSs	Explicit
CnC	Explicit
Cpp-TaskFlow	Explicit
HPX	Explicit
HTCondor DAGMan	Explicit
Kokkos	Implicit
Legion	Explicit
OmpSs	Implicit
OpenMP	Implicit
PaRSEC (DTD)	Explicit
PaRSEC (PTG)	Explicit
Pegasus	Explicit
PyGion	Implicit
QUARK	Explicit
Regent	Implicit
StarPU	Explicit
Swift	Explicit
TBB	Explicit
TensorFlow	Implicit
Uintah	Implicit
X10	Implicit
YML	Explicit
YML+XMP	Explicit

8.1.1.4 Data Handling

This property describes how the data are accessed in a task. The data can be accessed *directly*. For instance, the data can be accessed through function parameters. The data can also be requested or retrieved from a *container*. For instance, the data can be retrieved through a future in HPX.

This represents the current (as experienced during this dissertation) implementations of the task based programming models and can evolve with time.

In Table 8.4, how to access the data from the tasks is described. Most of the programming models provide a direct access to the input and output data for tasks whereas only a few of them require the user to request for the data they use in their tasks. During our experiments, we used HPX that uses futures to retrieve data in tasks and YML+XMP that provide a direct access to the data.

Table 8.4: Data Handling property for each task based programming model

Programming model	Data Handling	Programming model	Data Handling
Chapel	Direct Access	PaRSEC (PTG)	Direct Access
Charm++	Direct Access	Pegasus	Access through Container
Cilk	Direct Access	PyGion	Direct Access
ClusterSs	Direct Access	QUARK	Direct Access
CnC	Direct Access	Regent	Access through Container
Cpp-TaskFlow	Direct Access	StarPU	Direct Access
HPX	Access through Container	Swift	Access through Container
HTCondor DAGMan	Direct Access	TBB	Direct Access
Kokkos	Access through Container	TensorFlow	Access through Container
Legion	Access through Container	Uintah	Direct Access
OmpSs	Direct Access	X10	Direct Access
OpenMP	Direct Access	YML	Direct Access
PaRSEC (DTD)	Direct Access	YML+XMP	Direct Access

8.1.1.5 Task Implementation

This property indicates what kind of interface the user has to fill in to create a task which will be executed during the execution of the application. The tasks can use a *program* where the user has to provide the parameters through the API such as Pegasus. It can also use a *function pointer* which the user has to pass to the API as well as its parameters. The programing paradigm can also be based on *pragmas* which are used to delimit and describe the task. Another possibility is a custom interface like a *function with specific parameters*. The tasks can also be implemented by *encapsulating* them such as in YML+XMP where the task code is encapsulated in an XML file. Moreover, the tasks can be implemented using the dedicated language of the task based programming model. For instance, Regent task have to be implemented with Regent which allows the conversion of the tasks into CPU and GPU code.

Task encapsulation or function calls with existing softwares is an important feature and useful for task based programming models since it allows to reuse already implemented functions in tasks. The programming model should be able to provide a way to easily access data from a task to call an existing library e.g. existing code already written in C/C++/Fortran. This allows faster developments since existing code can be reused.

In Table 8.5, how the tasks are implemented is detailed. During our experiments, we used Regent with its dedicated language based on Lua and Terra to implement applications and tasks. We also used HPX that uses function pointers to implement tasks as well as YML+XMP and PaRSEC.

Table 8.5: Task Implementation property for each task based programming model

Programming model	Task Implementation	Programming model	Task Implementation
Chapel	Dedicated Language	PaRSEC (PTG)	Code Encapsulation
Charm++	Dedicated Language	Pegasus	Program
Cilk	Dedicated Language	PyGion	Function Pointer
ClusterSs	Dedicated Language	QUARK	Function Pointer
CnC	Function Pointer	Regent	Dedicated Language
Cpp-TaskFlow	Function Pointer	StarPU	Function Pointer
HPX	Function Pointer	Swift	Function Pointer
HTCondor DAGMan	Program	TBB	Function Pointer
Kokkos	Function Pointer	TensorFlow	Function with Specific Parameters
Legion	Function with Specific Parameters	Uintah	Function Pointer
OmpSs	Pragma	X10	Dedicated Language
OpenMP	Pragma	YML	Code Encapsulation
PaRSEC (DTD)	Function Pointer	YML+XMP	Code Encapsulation

8.1.1.6 Portability Accelerators

This property indicates what kind of accelerators are supported by the task based programming models. They can support or generate *CUDA* code and execute it on NVIDIA GPUs. There is also the possibility of supporting *multiple* accelerator architectures. This may be achieved by having a suitable backend for each supported accelerator architecture. Another possibility is to let the *user* chose how and which type of accelerator to use by interfacing with an accelerator programming language.

It depends on the existing technologies and can evolve over time or with apparition of new technologies.

Furthermore, the multiplication of supercomputer architectures makes the portability between supercomputers difficult especially if the support of multiple architectures is necessary. Therefore, the portability of an application is mandatory if there is a large amount of users. Indeed, in a large user base, there is a high probability that the users will have access to different computer architectures. At least, the architecture of their local computers should be different than the supercomputer they have access to. Portability to a large set of architectures allows the users to be able to familiarize with the application locally and the developers to easily make local modifications and tests then experiment them on larger scale on supercomputers.

In Table 8.6, the accelerators supported by the different task based programming models are shown. Our applications do not support accelerators so we were not able to try this feature. Nvidia cards are the most used GPUs so CUDA that can address Nvidia GPU programming is supported by most of the task programming models that can execute tasks on GPUs. However, other constructors are producing GPUs such as Intel and AMD.

Table 8.6: Portability Accelerators property for each task based programming model

Programming model	Portability Accelerators	Programming model	Portability Accelerators
Chapel	CUDA	PaRSEC (PTG)	CUDA
Charm++	CUDA	Pegasus	User Choice
Cilk	User Choice	PyGion	CUDA
ClusterSs	User Choice	QUARK	User Choice
CnC	User Choice	Regent	CUDA
Cpp-TaskFlow	CUDA	StarPU	CUDA
HPX	CUDA	Swift	User Choice
HTCondor DAGMan	User Choice	TBB	OpenCL
Kokkos	Multiple Accelerators	TensorFlow	CUDA
Legion	CUDA	Uintah	Multiple Accelerator Languages
OmpSs	Multiple Accelerators	X10	CUDA
OpenMP	Multiple Accelerators	YML	User Choice
PaRSEC (DTD)	CUDA	YML+XMP	User Choice

8.1.2 Task and Data Management

In this section, the features related to the category *Task and Data Management* are introduced. They include task and data scheduling properties that we believe interesting in the choice of a task based programming model to implement a task based application. The *Dependency Type* is an interesting feature to know which type of dependency the user has to extract from his/her algorithm and to provide during the implementation of the task based application. The *Worker Management* indicated the user investment in term of application management. The *Data Distribution* describes how the scheduler manage the placement of the data on the computing resources. This placement can be made by the scheduler or directly by the user which demands more effort. The *Task Binding* represents the if the user has to provide the binding between tasks and computing resources. The *Task Insertion* indicates if the scheduler is capable of adding new tasks to schedule during the execution of already scheduled tasks.

8.1.2.1 Dependency Type

This property describes how the dependencies between the tasks are provided by the user of the task based programming models. This means that depending on the provided information (in task description, in particular), the scheduler can deduce other dependency types. For instance, the control dependency graph combined with the information on the use of the data parameters provided to the task could allow the scheduler to deduce the data dependency graph. Therefore, the user has to provide the required information so that the scheduler is able to work properly.

The user may have to provide *control* dependencies between the tasks. They describe in which order the tasks can be executed and which tasks can be executed at the time. *Data* dependencies can also be provided. This approach is used to infer the dependencies and the parallelism between the tasks by studying how the

data are used by the tasks and how they flow from one task to the other. The user may have to provide *both* dependency type although it is possible to convert a dependency graph type into the other with the appropriate information.

In Table 8.7, the dependency type used to describe the dependencies between the tasks in task based programming models is introduced. In our experiments, we used data oriented such as HPX and control oriented dependency programming models such as YML+XMP.

Table 8.7: Dependency Type property for each task based programming model

Programming model	Dependency Type	Programming model	Dependency Type
Chapel	Both	PaRSEC (PTG)	Data
Charm++	Control	Pegasus	Control
Cilk	Data	PyGion	Data
ClusterSs	Data	QUARK	Data
CnC	Both	Regent	Data
Cpp-TaskFlow	Control	StarPU	Both
HPX	Data	Swift	Data
HTCondor DAGMan	Control	TBB	Both
Kokkos	Data	TensorFlow	Data
Legion	Data	Uintah	Data
OmpSs	Both	X10	Data
OpenMP	Both	YML	Control
PaRSEC (DTD)	Control	YML+XMP	Control

8.1.2.2 Worker Management

This property indicates whether the worker thread or process which hosts the tasks in the task based programming models has to be started and maintained by the user (*explicit*) or is provided by the runtime (*implicit*). This means the user has to put more efforts to launch its application in the explicit case.

In Table 8.8, the worker management is shown for each task based programming model. Most of the task based programming models manage their worker placement to execute the tasks without user interference. Only a few need user intervention while some of them allow hints from the user.

Table 8.8: Worker Management property for each task based programming model

Programming model	Worker Management
Chapel	Implicit
Charm++	Implicit
Cilk	Implicit
ClusterSs	Implicit or Explicit
CnC	Implicit
Cpp-TaskFlow	Implicit
HPX	Implicit or Explicit
HTCondor DAGMan	Implicit
Kokkos	Implicit
Legion	Implicit
OmpSs	Implicit
OpenMP	Implicit
PaRSEC (DTD)	Implicit
PaRSEC (PTG)	Implicit
Pegasus	Implicit
PyGion	Implicit
QUARK	Implicit
Regent	Implicit
StarPU	Implicit
Swift	Implicit
TBB	Implicit
TensorFlow	Explicit
Uintah	Explicit
X10	Implicit
YML	Implicit
YML+XMP	Implicit

8.1.2.3 Data Distribution

This property describes how the data distribution is handled by the programming model scheduler. The scheduler can be able to migrate data between computing resources. *Implicit* data distribution means that the runtime system decides where to place the data on the nodes whereas *explicit* data distribution means that the user has to specify the distribution of the data across the nodes. In the implicit case, the scheduler has more liberty to optimize the computations and the data migrations when the scheduler is able to do so.

When the schedulers will be more efficient and perform better, the implicit management of the data will be very interesting to manage data. However, it is still not the case as we showed in Chapter 7.

In Table 8.9, the data distribution management in the task based programming models is introduced. In our YML+XMP applications, we had implicit data distributions for the tasks but not in the parallel and distributed tasks in which we had to place our data. For our HPX and PaRSEC applications, we had to explicitly map our data with process ranks representing cores or nodes from a cluster or supercomputer.

Table 8.9: Data Distribution property for each task based programming model

Programming model	Data Distribution
Chapel	Implicit
Charm++	Implicit or Explicit
Cilk	Not distributed
ClusterSs	Implicit
CnC	Not distributed
Cpp-TaskFlow	Not distributed
HPX	Implicit or Explicit
HTCondor DAGMan	Implicit
Kokkos	Not distributed
Legion	Implicit
OmpSs	Not distributed
OpenMP	Not distributed
PaRSEC (DTD)	Explicit
PaRSEC (PTG)	Explicit
Pegasus	Implicit
PyGion	Implicit
QUARK	Not distributed
Regent	Implicit
StarPU	Implicit
Swift	Explicit
TBB	Not distributed
TensorFlow	Not distributed
Uintah	Implicit
X10	Implicit
YML	Implicit
YML+XMP	Implicit

8.1.2.4 Task Binding

This property describes how the tasks are bound to the allocated hardware resources. It demands more effort to the scheduler in order to place the tasks on the resources but with efficient enough schedulers, it will allow to increase the performances of the task based programming models. The binding can be *implicit* when automatically determined by the task based programming runtime or *explicit* when the user has to provide a binding map.

In Table 8.10, the task binding management of the task based programming models is highlighted. All the programming models propose a implicit task binding with some of them allowing the user to pin tasks on computing resources. For our experiments we did not try to optimize the task binding by giving hints to PaRSEC and HPX that support explicit bindings.

Table 8.10: Task Binding property for each task based programming model

Programming model	Task Binding
Chapel	Implicit
Charm++	Implicit or Explicit
Cilk	Implicit
ClusterSs	Implicit
CnC	Implicit
Cpp-TaskFlow	Implicit
HPX	Implicit or Explicit
HTCondor DAGMan	Implicit
Kokkos	Implicit
Legion	Implicit
OmpSs	Implicit
OpenMP	Implicit
PaRSEC (DTD)	Implicit or Explicit
PaRSEC (PTG)	Implicit or Explicit
Pegasus	Implicit
PyGion	Implicit
QUARK	Implicit or Explicit
Regent	Implicit
StarPU	Implicit
Swift	Implicit
TBB	Implicit
TensorFlow	Implicit
Uintah	Implicit
X10	Implicit or Explicit
YML	Implicit
YML+XMP	Implicit

8.1.2.5 Task Insertion

This property indicates if new tasks can be added to the task pool during the execution of the already scheduled tasks. This feature was not required for the studied applications but it is an important feature for certain applications such as iterative methods. This feature can allow them to start a new iteration or stop iterating according to a given metrics that can be the output of a task. For instance, HPX provides such features.

In Table 8.11, the insertion of new tasks during the execution of the tasks is described. This feature is very useful for iterative methods but we did not implement such methods so we did not use this feature in our experiments.

Table 8.11: Task Insertion property for each task based programming model

Programming model	Task Insertion
Chapel	Yes
Charm++	Yes
Cilk	Yes
ClusterSs	No
CnC	No
Cpp-TaskFlow	Yes
HPX	Yes
HTCondor DAGMan	Yes
Kokkos	Yes
Legion	Yes
OmpSs	Yes
OpenMP	Yes
PaRSEC (DTD)	Yes
PaRSEC (PTG)	No
Pegasus	No
PyGion	Yes
QUARK	Yes
Regent	No
StarPU	No
Swift	Yes
TBB	Yes
TensorFlow	Yes
Uintah	No
X10	Yes
YML	No
YML+XMP	No

8.1.3 Programming Model Features

In this section, the features related to the category *Programming Model Features* are described. They include programming model properties that we believe interesting in the choice of a task based programming model to implement a task based application. The *Dependency Expression* property describes the type of structures that hold the dependencies between the tasks and can help the user understand how he/she has to express the dependencies during the implementation. The *Communication Model* property can help the user chose the communication model the most appropriate for the application and the cluster or supercomputer on which the application will be executed. The *Fault Tolerance* property indicates if the programming model supports fault tolerance. The *Implementation Type* property refers to how the programming model is implemented and can help the user to understand how the programming model will be integrated in the development and the deployment of the application. The *Data Persistence* property indicates if the task based programming model support data persistence which can be a mandatory feature needed for an application. Finally, the *Scheduler Location* property helps the user to understand the scheduling policy of the task based programming model schedulers.

8.1.3.1 Dependency Expression

This property describes how the dependencies between the tasks are represented in the considered model. The possibilities include a *graph*, a *directed acyclic graph* or DAG, a *tree* and a *Petri Net*.

In Table 8.12, how the dependency are expressed in each task based programming model is detailed. The difference between graph and DAG is not always clear in the implementations of the task based programming models. In this table, we recorded the information we found in papers introducing the task based programming models.

Table 8.12: Dependency Expression property for each task based programming model

Programming model	Dependency Expression
Chapel	Directed Acyclic Graph
Charm++	Directed Acyclic Graph
Cilk	Directed Acyclic Graph
ClusterSs	Directed Acyclic Graph
CnC	Petri Network
Cpp-TaskFlow	Graph
HPX	Directed Acyclic Graph
HTCondor DAGMan	Directed Acyclic Graph
Kokkos	Directed Acyclic Graph
Legion	Tree
OmpSs	Graph
OpenMP	Graph
PaRSEC (DTD)	Directed Acyclic Graph
PaRSEC (PTG)	Directed Acyclic Graph
Pegasus	Directed Acyclic Graph
PyGion	Tree
QUARK	Directed Acyclic Graph
Regent	Tree
StarPU	Queue
Swift	Graph
TBB	Graph
TensorFlow	Graph
Uintah	Directed Acyclic Graph
X10	Directed Acyclic Graph
YML	Directed Acyclic Graph
YML+XMP	Directed Acyclic Graph

8.1.3.2 Communication Model

This property describes how data are sent from a task to another. The runtime system can use *message passing* (msg), *global address space* (gas) or the *file system* (fs).

This feature can provide many interesting optimizations such as data migration anticipation or data pre-loading. However, these features are not implemented yet. They also induces complex choices for the scheduler that may not improve performances at the end.

In Table 8.13, the communication model used in each task based programming model is shown. Task based programming models that only work in shared memory do not have a communication model since they do not manage distributed memory. In our applications, we used programming models that use each communication model.

Table 8.13: Communication Model property for each task based programming model

Programming model	Communication Model
Chapel	Global Address Space
Charm++	Global Address Space
Cilk	Local Shared Memory
ClusterSs	Global Address Space
CnC	Local Shared Memory
Cpp-TaskFlow	Local Shared Memory
HPX	Global Address Space
HTCondor DAGMan	File system (I/Os)
Kokkos	Local Shared Memory
Legion	Global Address Space
OmpSs	Local Shared Memory
OpenMP	Local Shared Memory
PaRSEC (DTD)	Message Passing
PaRSEC (PTG)	Message Passing
Pegasus	File system (I/Os)
PyGion	Global Address Space
QUARK	Local Shared Memory
Regent	Global Address Space
StarPU	Message Passing
Swift	Message Passing
TBB	Local Shared Memory
TensorFlow	Local Shared Memory
Uintah	Message Passing
X10	Global Address Space
YML	File system (I/Os)
YML+XMP	File system (I/Os)

8.1.3.3 Fault Tolerance

This property indicates if the task based programming models support fault tolerance. Fault tolerance allows applications to recover from errors during execution. If errors still appear, the application can be stopped properly.

In Table 8.14, the fault tolerance support is given for each task based programming model. Task based programming models that support fault tolerance are mainly the ones that can execute distributed and parallel tasks such as YML+XMP we used to implement our applications.

Table 8.14: Fault Tolerance property for each task based programming model

Programming model	Fault Tolerance
Chapel	No
Charm++	Yes
Cilk	No
ClusterSs	No
CnC	No
Cpp-TaskFlow	No
HPX	No
HTCondor DAGMan	Yes
Kokkos	No
Legion	No
OmpSs	No
OpenMP	No
PaRSEC (DTD)	No
PaRSEC (PTG)	No
Pegasus	Yes
PyGion	No
QUARK	No
Regent	No
StarPU	No
Swift	No
TBB	No
TensorFlow	No
Uintah	Yes
X10	Yes
YML	Yes
YML+XMP	Yes

8.1.3.4 Implementation Type

This property describes how the programming model API is included in an application. It can be done through a *library*, a *language extension* or a *language*. This property shows how the programming models are used to implement an application with them.

In Table 8.15, the API access is introduced. Most of the task based programming models are available as libraries or languages. During our experiments, we used one of each implementation type.

Table 8.15: Implementation Type property for each task based programming model

Programming model	Implementation Type
Chapel	Language
Charm++	Language Extension
Cilk	Runtime System
ClusterSs	Language Extension
CnC	Language
Cpp-TaskFlow	Library
HPX	Library
HTCondor DAGMan	Language
Kokkos	Library
Legion	Library
OmpSs	Language Extension
OpenMP	Language Extension
PaRSEC (DTD)	Library
PaRSEC (PTG)	Library
Pegasus	Language
PyGion	Language Extension
QUARK	Runtime System
Regent	Language
StarPU	Runtime System
Swift	Language
TBB	Library
TensorFlow	Library
Uintah	Language
X10	Language
YML	Language
YML+XMP	Language

8.1.3.5 Data Persistence

This property indicates if the task based programming model supports data persistence.

The support for data persistence is given in Table 8.16. The task based programming models predominantly do not support data persistence. Task based programming models using the file system to transfer data between the tasks can be considered as using data persistence since data stays if it is not deleted by the programming models after errors or applications terminations.

Table 8.16: Data Persistence property for each task based programming model

Programming model	Data Persistence
Chapel	No
Charm++	Yes
Cilk	No
ClusterSs	No
CnC	No
Cpp-TaskFlow	No
HPX	No
HTCondor DAGMan	Yes
Kokkos	No
Legion	No
OmpSs	No
OpenMP	No
PaRSEC (DTD)	No
PaRSEC (PTG)	No
Pegasus	Yes
PyGion	No
QUARK	No
Regent	No
StarPU	No
Swift	Limited
TBB	No
TensorFlow	No
Uintah	No
X10	Yes
YML	Yes
YML+XMP	Yes

8.1.3.6 Scheduler Location

This property describes where the scheduler instances of the task based programming models are located. It can be *centralized* where there only one scheduler instance that manages all the tasks. The tasks can also be managed at the local level in the workers in a *distributed* way.

In Table 8.17, the location of the scheduler is shown for each task based programming model. Most of the task based programming models use centralized schedulers. In our experiments, we used task based programming models that use both type of schedulers.

Table 8.17: Scheduler Location property for each task based programming model

Programming model	Scheduler Location
Chapel	Centralized
Charm++	Distributed
Cilk	Centralized
ClusterSs	Centralized
CnC	Centralized
Cpp-TaskFlow	Centralized
HPX	Distributed
HTCondor DAGMan	Centralized
Kokkos	Centralized
Legion	Distributed
OmpSs	Centralized
OpenMP	Centralized
PaRSEC (DTD)	Distributed
PaRSEC (PTG)	Distributed
Pegasus	External
PyGion	Distributed
QUARK	Centralized
Regent	Distributed
StarPU	Distributed
Swift	Centralized
TBB	Centralized
TensorFlow	Centralized
Uintah	Distributed
X10	Distributed
YML	Centralized
YML+XMP	Centralized

8.2 Taxonomy Summary

In this section, we present a summary for the features presented in each category as well as the expression of these features so that it is easier to compare the features of the different task based programming models.

Table 8.18 summarizes the *Task Capabilities* category. In this category, we introduced the *Task Granularity* which can take the values Sequential tasks (s) as well as parallel and distributed tasks (p). We also indicated if the programming models support *Nested Tasks*. The values for the property *Task Implementation* are Dedicated Language (dl), Function Pointer (fp), Program (pgm), Pragma (p) and Code Encapsulation (ce). We detailed if the *Heterogeneity* is implicit (i) or explicit (e). We explained the memory *Architecture* values which are Distributed Memory (d) and Shared Memory (s). We introduced the *Data Handling* that can be a direct access (d) or made through a request to the data management system (c). Withe the *Portability Accelerators* property, we gave the technology that can be used to address accelerators with the task based programming models.

Table 8.19 summarizes the *Task and Data Management* category. We gave the *Dependency Type* used to express the dependencies between the tasks which are data (d), control (c) or both (b). We also indicated if the task based programming models support *Task Insertion*. We described *Data Distribution*, *Worker Management* and *Task Binding* properties which values are implicit (i) or explicit (e).

Table 8.20 summarizes the *Programming Model Features* category. We indicated if the task based programming models support *Fault Tolerance* and *Data Persistence*. We explained which structure holds the dependencies between the tasks in *Dependency Expression*. We detailed which *Communication Model* is used. The values can be Global Address Space (gas), Shared Memory (sm), File System (fs) and Message Passing (msg). We also introduced the *Implementation Type* of the task based programming models which can be a language (lang), a runtime system (rt), a library (lib) and a language extension (ext). We gave the *Scheduler*

Location which can be external (e), distributed (d) or centralized (c).

Table 8.18: Task Capabilities Summary

	Task Granularity	Nested Tasks	Task Implementation	Heterogeneity	Architecture	Data Handling	Portability Accelerators
Chapel	s	yes	dl	i	d	d	CUDA
Charm++	s	yes	dl	i/e	d	d	CUDA
Cilk	s	yes	dl	e	s	d	user
ClusterSs	s	no	dl	e	d	d	user
CnC	s	no	fp	e	s	d	user
Cpp-TaskFlow	s	yes	fp	e	s	d	CUDA
HPX	s	yes	fp	e	d	c	CUDA
HTCondor DAGMan	p	no	pgm	e	d	d	user
Kokkos	s	yes	fp	i	s	c	s
Legion	s	yes	sf	e	d	c	CUDA
OmpSs	s	yes	p	i	s	d	s
OpenMP	s	yes	p	i	s	d	s
PaRSEC (DTD)	s	no	fp	e	d	d	CUDA
PaRSEC (PTG)	s	no	ce	e	d	d	CUDA
Pegasus	p	no	pgm	e	d	c	user
PyGion	s	yes	fp	i	d	d	CUDA
QUARK	p	no	fp	e	s	d	user
Regent	s	yes	dl	i	d	c	CUDA
StarPU	s	no	fp	e	d	d	CUDA
Swift	p	no	fp	e	d	c	user
TBB	s	yes	fp	e	s	d	OpenCL
TensorFlow	s	no	sf	i	s	c	CUDA
Uintah	s	no	fp	i	d	d	m
X10	s	yes	dl	i	d	d	CUDA
YML	s	no	ce	e	d	d	user
YML+XMP	p	no	ce	e	d	d	user

Table 8.19: Task and Data Management Summary

	Dependency Type	Task Insertion	Data Distribution	Worker Management	Task Binding
Chapel	b	yes	i	i	i
Charm++	c	yes	i/e	i	i/e
Cilk	d	yes	n	i	i
ClusterSs	d	no	i	i/e	i
CnC	b	no	n	i	i
Cpp-TaskFlow	c	yes	n	i	i
HPX	d	yes	i/e	i/e	i/e
HTCondor DAGMan	c	yes	i	i	i
Kokkos	d	yes	n	i	i
Legion	d	yes	i	i	i
OmpSs	b	yes	n	i	i
OpenMP	b	yes	n	i	i
PaRSEC (DTD)	c	yes	e	i	i/e
PaRSEC (PTG)	d	no	e	i	i/e
Pegasus	c	no	i	i	i
PyGion	d	yes	i	i	i
QUARK	d	yes	n	i	i/e
Regent	d	no	i	i	i
StarPU	b	no	i	i	i
Swift	d	yes	e	i	i
TBB	b	yes	n	i	i
TensorFlow	d	yes	n	e	i
Uintah	d	no	i	e	i
X10	d	yes	i	i	i/e
YML	c	no	i	i	i
YML+XMP	c	no	i	i	i

Table 8.20: Programming Model Features Summary

	Dependency Expression	Communication Model	Fault Tolerance	Implementation Type	Data Persistence	Scheduler Location
Chapel	DAG	gas	no	lang	no	c
Charm++	SDAG	gas	yes	ext	yes	d
Cilk	DAG	sm	no	rt	no	c
ClusterSs	DAG	gas	no	ext	no	c
CnC	Petri	sm	no	lang	no	c
Cpp-TaskFlow	Graph	sm	no	lib	no	c
HPX	DAG	gas	no	lib	no	d
HTCondor DAGMan	DAG	fs	yes	lang	yes	c
Kokkos	DAG	sm	no	lib	no	c
Legion	Tree	gas	no	lib	no	d
OmpSs	Graph	sm	no	ext	no	c
OpenMP	Graph	sm	no	ext	no	c
PaRSEC (DTD)	DAG	msg	no	lib	no	d
PaRSEC (PTG)	DAG	msg	no	lib	no	d
Pegasus	DAG	fs	yes	lang	yes	e
PyGion	Tree	gas	no	ext	no	d
QUARK	DAG	sm	no	rt	no	c
Regent	Tree	gas	no	lang	no	d
StarPU	Queue	msg	no	rt	no	d
Swift	Graph	msg	no	lang	limited	c
TBB	Graph	sm	no	lib	no	c
TensorFlow	Graph	sm	no	lib	no	c
Uintah	DAG	msg	yes	lang	no	d
X10	DAG	gas	yes	lang	yes	d
YML	DAG	fs	yes	lang	yes	c
YML+XMP	DAG	fs	yes	lang	yes	c

We summarized the different features and values we introduced in this taxonomy. Now, we can propose recommendations for implementing task based applications and improving task based programming models.

8.3 Analyze and Recommendations

In this section, we will provide recommendations for implementing task based applications and improving task based programming models. We introduce them by importance order. We start with those we believe are the more important.

8.3.1 Adapted Programming Model to Algorithm Granularity

Choosing the right granularity for tasks is an important parameter to obtain optimal performances with task based programming models. Moreover, the number of tasks is also important to get optimal performances since too much tasks induces a large scheduling and task launching overhead whereas not enough tasks does not expose enough parallelism nor computations to take advantage of all the computing resources available which is sub-optimal. Thus, an compromise between the number of tasks executed and the granularity of the tasks is mandatory as we experienced on the K Computer in Chapter 5. Besides, the granularity also depends on the cluster or supercomputer used to execute the applications due to the difference of core performances. Thus, the compromise has to be found for each different architecture.

8.3.2 Data Migrations

As data migrations are costing more time and energy due to the larger networks, it is mandatory to favor programming models optimizing them in order to reduce their usage while still obtaining good performances. In order to do so, tasks based programming models require the data and control dependencies between the tasks to be able to optimize the computations placement on the computing resources as well as the data migrations. Data migrations comprise communications between nodes, data copy between caches, offloading to accelerators and IOs with the file system. An efficient scheduler could favor data locality and cache reuse depending on the granularity of the tasks.

8.3.3 Encapsulated Tasks

As we try to reduce the communications between the nodes of the clusters and supercomputers in order to improve performances, it is not advised to make let the tasks make communications during their executions. The only data transfer that should happen in tasks are during their initialization with their input parameters and during their finalization with the output parameters. Note that parameters could be inputs and outputs. In this case, the are modified by the tasks and the programming model should be able to manage such cases. This way, the communications and the parameters of the tasks can be managed efficiently by the task based programming model schedulers.

8.3.4 Dependencies Expression

The dependencies should be expressed as a graph or an equivalent. It is not mandatory to explicitly be a graph. However, under the format of a graph, it may be possible to use graph exploration methods in order to optimize the data distribution and the computations placement so that the communications are minimal and the maximum of computations can run at the same time. For instance, YML multidimensional arrays of event used to represent dependencies can be transformed into graphs then used to optimize execution of the tasks and the data migrations. With graphs, even irregular dependencies can be expressed which may no be the case with other dependencies representations.

8.3.5 Dynamical Task Scheduling

With dynamical scheduling, it is possible to introduce conditional branchings in the graph of dependencies. For instance, this could be used to stop iterative methods expressed with graphs of tasks when they reach their stopping conditions. Dynamical graphs for tasks execution in YML+XMP were introduced in [124].

8.3.6 High Level Languages

Although pragmas are a fast solution to implement task based applications, in practice, only task based programming models working on shared memory choose this approach such as OpenMP which is based on directives and OmPSs which is an extension of OpenMP. Higher level programming models are suitable to express complex dependencies and give task related information. For instance, Kokkos is a high level library that also provides a tasking interface.

8.3.7 Fault Tolerance

With the increase of the usage of computing resources for the execution of an application, the probability of a failure appearing has increased. Indeed, even if the probability of a failure on a given computing resource is low, the large amount of computing resources used means that the probability of a failure occurring on one of the computing resource is higher. Therefore, fault tolerance is an important feature to manage the increasing probability of a failure occurring. Moreover, fault tolerance can be integrated in schedulers since most of the failures should happen during tasks execution.

If a task fails, it can be re-scheduled and executed on another resource. If it still fails, it is probably an error in the task then the application should stop. If tasks are regularly failing on a computing resource and are working fine on other computing resources, it means that there is a problem with this computing resource and it should be excluded from the list of available resources.

In case of missing data, the scheduler could re-execute previously executed task in order to rebuild the missing data from the data still available to the scheduler or stop the application if the missing data cannot be recovered. To do so, the scheduler has to keep the input data of a task until the task is finished in order to be able to relaunch the task to recompute the missing data. Check-pointing can be a solution to save critical data regularly and rebuild missing data if necessary.

8.3.8 Check-Pointing

There is only few programming models that provide check-pointing but extending task based programming models to support check-pointing is possible. Usually, it is done via external and dedicated tools. A scheduler could be implemented to regularly save the output of tasks since the expression of tasks and their data input and output is a very helpful base to define and implement check-pointing for task based programming models. Then the scheduler could re-start the application and continue executing task from the last saved task parameters. The user could also provide information about which data has to be saved and when through the task based programming model API since no task based programming model provide an API that lets the user chose the check-pointing operations. In conjunction with fault tolerance, check-pointing could allow the recovery of data lost during computing resources failures and the success of the application even with critical resource failures.

8.3.9 Multi-Level Programming

Multi-level programming can be an interesting solution to express very efficient applications. Indeed, it could be possible to express large grain tasks with high level task based programming models and use parallel and distributed tasks. Then these tasks could be implemented with lower level programming models that are very efficient with medium sized applications. For instance, YML+XMP+StarPU, which was used on T2K at Tsukuba, is such an example. YML was used to express large grain task and dependencies. Parallel and distributed tasks were implemented with XMP and small grain computations at the level of the processes were managed by StarPU. Another possibility is to implement large grain tasks as a smaller grain task based application.

8.3.10 Collective Operations

Collective operations are the equivalent to collective communications defined in MPI such as all-to-all, one-to-all or all-to-one communications for task based applications. Collective communications include reductions, gathers and scatters. They exist in an one-to-all version as well as an all-to-all version depending on where the results are expected to be sent. Such operations on task outputs do not exist natively in most of the task based programming models. Therefore, efficiently implementing such task operations can help to obtain better performances when these operations are needed, for instance, in the task based sparse matrix vector product implemented in Chapter 6.

A possible implementation for a reduction could be implemented as high level operation that takes an array of task output dependencies and a reduction operation that will be performed on the data such as a sum, a multiplication or a maximum. The reduction operation has to be provided by the user for custom types defined by the user. It corresponds to the task performed on two data to reduce it. It also should return the result of the reduction operation. Then, the results can be reused as input for other tasks. This high level reduction could create a dependency graph to efficiently perform the reduction operation with every data dependency provided. Afterwards, the scheduler can schedule and execute tasks according to the generated dependencies and the provided reduction operation.

Furthermore, other collective operations could be implemented following this process. They could be reused efficiently multiple times in an application. Moreover, they could be efficiently implemented and even

improved when possible.

8.4 Preliminary Methodology for Post-Petascale Programming

The recommendations provided in the previous section allow to make a preliminary methodology for task based parallel and distributed programming which optimizes data migrations and computations for post-petascale supercomputers. The programming introduced in this methodology is based on efficient schedulers and file systems. To begin with, the user has to chose an adapted granularity for the tasks. The tasks are executed by the scheduler while optimizing computations placement and data migrations including file system IOs.

The tasks have to perform enough computations in order to recoup the scheduling cost of the tasks while expressing enough parallelism and tasks to execute in order to keep busy the avail computing resources. Tasks could also be implemented with multi-level programming in which large grain parallel and distributed tasks could be implemented with graphs of finer grain tasks. Task based programming models working only on shared memory cannot be directly used to implement applications for post-petascale supercomputers since they are built with a large amount of node with distributed memory. However, they can be used to implement large scale tasks that can be executed on one node.

This would require a high level and multi-level task based programming model to implement such applications. These programming models would also be used to express tasks and data dependencies as well as provide task implementations. They could also be used to allow the user to provide semantic information that would help the scheduler to improve its performances. This information could also include data that the users want to see regularly saved through check-pointing. Furthermore, task based programming models should provide fault tolerance and check-pointing features in order to ensure data safety and complete execution of the application. These are important features for post-petascale supercomputers since the probability of a failure occurring is increasing due the large amount of resources used to execute an application.

Moreover, graph of task dependencies should be dynamic to support conditional branchings in dependencies in order to increase the expressiveness of the task based programming models. Tasks should be encapsulated and should not be allowed to perform communications by themselves. The data transfers between the tasks should be efficiently scheduled by the task based programming model schedulers in order to minimize data migrations between nodes. Then, the tasks should only receive data through their input parameters and return their results through their output parameters. The encapsulation of tasks could also avoid blocking communications performed by the user during the execution of the task that would make the task waiting for data that will only be available when the other task concerned by the communication is executed.

It is always difficult to propose a methodology to solve such complex problems. Nonetheless, with our experiments, taxonomy and recommendations, we can list the main points by order of importance in the following list.

- Use a programming model that provides an adapted scheduler that can optimize data migrations, IOs management and computation placement in order perform the minimum amount of data migrations while using the available computing resources to execute the maximum amount of tasks at the same

time. If the selected task based programming model does not provide one, it may be interesting to integrate a more efficient external scheduler or improve the existing one. This can also mean that the selected task based programming model is not suitable enough and that it should be replaced. This scheduler should also support fault tolerance and check-pointing.

- Find an appropriate granularity for tasks. It can be done by finding a compromise between the size of the task and the number of tasks to execute. Indeed, tasks without enough computations will not recover the overhead for schedule them while tasks that are too large means that there is not enough parallelism at the level of the graph of task and that some of the computing resources will not be used.

- Several level of programming should be available, for instance, with a graph of parallel and distributed tasks. This provides at least three levels of programming. The higher level could be the graph of task that performs the high level algorithm. The middle level could be the parallel and distributed task that manages distributed memory or a most a node with shared memory. The low level could be the process or thread that runs computations at the level of the core. Furthermore, the parallel and distributed tasks that can be considered coarse grain could also be implemented with finer grain tasks. This could be made by using task based programming model with sequential tasks that can be executed on one thread or process.

- The task based programming model should allow the user to express data and control dependencies as well as task implementation in order to discover as much parallelism as possible. It also should allow to provide semantic information about tasks such as placement hints and check-pointing options.

8.5 Synthesis and Perspectives

In this chapter, we introduced the features that we believe interesting for task based programming models and how they are expressed in the different task based programming models studied in this dissertation. We divided the features into three categories : *Task Capabilities* which contains the features related to the abilities of the tasks, the *Task and Data Management* which contains the features related to the scheduling properties of the tasks and data migrations as well as *Programming Model Features* in which we present capabilities of the programming models. Then, we provided recommendations to improve the task based programming models according to the features they possess and the features we believe task based programming should have in order to increase their capabilities and performances. Finally, we proposed a preliminary methodology for parallel and distributed programming based on graphs of tasks which optimizes data migrations and computations scheduling.

To go further, more features and their values for each task based programming model could be added in the taxonomy. Furthermore, the recommendations could be integrated into some of the existing task based programming models and make experiments with the additions to show their efficiency. Another possibility is to implement a new task based programming model showcasing the recommendations as main features.

Chapter 9

Conclusion and Perspectives

We will start by giving a synthesis of the content of this dissertation then we will conclude and suggest perspectives for future researches.

9.1 Conclusion

First, we introduced the High Performance Computing (HPC) time line as well as most of the task based programming models used by the HPC community. We also discussed the upcoming challenges for exascale in HPC.

Afterwards, we explained the three kinds of methods we implemented with several programming models in this dissertation. The first algorithms we introduced are block-based direct methods to solve dense linear systems; the block Gaussian elimination, the block Gauss-Jordan elimination and the LU factorization with the forward and backward substitutions to solve the two triangular systems. The second algorithm is the sparse matrix vector product. In this part, we introduced several sparse matrix storage formats that are more efficient to store matrices with large amount of zeros in memory. In particular, we explained the compressed sparse row (CSR), Ellpack (ELL) and coordinates (COO) sparse storage formats as well as their corresponding algorithms to perform the sparse matrix vector product. The last method is the Kirchhoff seismic pre-stack depth migration which is commonly used in geoscience and by Total to study the underground.

Thereafter, we described several task based programming models in greater details as our first contribution. In this study, we are interested in task based programming models working on distributed memory. we split them into two categories : fine grain task-based distributed and parallel programming models in which tasks are executed on one thread or process and task-based distributed and parallel programming models in which tasks are executed on multiple processes or nodes. For the first category, we selected Regent, Legion, HPX, PaRSEC and TensorFlow. For the second category, we selected YML+XMP, Pegasus and Swift. We highlighted the main features of each programming model studied and how to use them to define task, the dependencies between them, how to register the data used in the tasks and how to execute the tasks to perform the intended computations.

Then, we introduced our task-based applications to solve dense linear systems and perform the LU factorization. These applications are part of our second contribution. We introduced three block based direct methods to solve dense linear systems : the block Gaussian elimination, the block Gauss-Jordan elimination

and the LU factorization. We implemented the solution to dense linear systems in YML+XMP, XMP, MPI and ScaLAPACK whereas the LU factorization application was also implemented with HPX, Regent and PaRSEC. We performed experiments on the petascale K Computer with YML+XMP and XMP. We showed that the YML+XMP applications were running faster than the XMP corresponding applications on 8096 cores. We also performed experiments on Poincare, *La Maison de la Simulation* cluster, with the LU factorization implemented with the task based programming models YML+XMP, HPX, PaRSEC and Regent as well as MPI, XMP and ScaLAPACK. We performed strong scaling experiments with each application up to 64 nodes for 16384×16384, 32768×32768 and 49512×49512 matrices. We showed that MPI has the fastest execution times and the best scalability on 64 nodes and that, although, XMP translates its directives into MPI code, the PGAS model used in XMP is not as efficient as using directly MPI. Moreover, we showed that HPX is the most efficient task based programming model on 64 nodes. However, PaRSEC also shows interesting performances in some cases and Regent applications execution times increase while increasing the number of nodes from 32 to 64 which should not be the case. We showed that HPX and PaRSEC are performing better with a higher number of smaller tasks. YML+XMP execution times are higher than the other task based programming models due to the use of the file system but its strong scaling speedups are higher so YML+XMP could obtain better performances on a larger number of resources as we showed on the K Computer.

Subsequently, we described our sparse matrix vector product based application with its 2D block decomposition used to store our matrices in which each block is a sparse matrix compressed with a sparse storage format. This application is also included in our second contribution. We analyzed, evaluated and experimented the sparse operation $A(Ax + x)$ using HPX, YML+XMP and MPI on PangeaII, a petascale supercomputer from Total. The applications are implemented on top of a common kernel that performs the sparse matrix vector multiplication on matrices stored in CSR, ELL, COO and SCOO sparse classical storage formats. We showed that the CSR sparse format storage obtains the best performances in most of the cases and that the COO storage format has better load balancing but requires larger output vector which increases the size of collective operations, therefore their execution time. We have shown that collective operations are an issue in task-based programming models since most of them do not provide a highly optimized implementation and generate all-all communications. We also have shown that collective communications in MPI are taking more and more time as the number of computing resources used increases, especially for weak scaling experiments. Therefore, task-based programming models, that can avoid large scale collective communications by turning them in collective operations that can be efficiently scheduled or by running them on a subset of the allocated resources, is an interesting alternative to MPI. However, current systems are more suitable for message passing libraries since both are designed and optimized to work well together. Moreover, file systems are not suitable either for programming models that use IOs to transfer data between tasks. The current task-based programming model schedulers lack the capability to completely manage the memory they use and are not efficient enough to reduce the data migrations.

Thereupon, we introduced task based algorithms for the Kirchhoff seismic pre-stack depth migration, a sub-surface visualization application used by Total and the parallelism intrinsic to the method. Then, we presented and implemented a simplified 2D version of the Kirchhoff seismic pre-stack depth migration with HPX, MPI and MPI+OpenMP. This application is the last part of our second contribution. We used these applications to perform strong scaling and weak scaling experiments as well as study the influence of the

number of OpenMP threads on our kernel. These experiments were also executed on PangeaII. We showed that our HPX application does not scale very well both in term of weak and strong scaling compared to our MPI application. We deduced that there were migrations of the images and the traces across the nodes which reduced the performances. We also showed that our addition of OpenMP directives in the kernel did not bring as good performances as our pure MPI application.

Finally, we summarized the key features that we were able to extract from the experiments and studies performed. We also showed how they are expressed in the different task based programming models. Then we used these data to form a taxonomy of the task based programming models. We also provided recommendations to improve the efficiency and performances of task based programming models and their usage. Finally, we presented a methodology for parallel and distributed programming based on graphs of tasks for post-petascale supercomputers. These are our third contribution.

The purpose of this dissertation was to study and experiment with task based programming models on post-petascale supercomputers. We introduced and explained the usage of several task based programming models. We also implemented several applications to solve dense linear system, an application to perform the sparse operation $A(Ax + x)$ based on a common kernel that performs the sparse matrix vector product with several sparse storage formats and a scientific application used by Total to explore the sub-surface, the Kirchhoff seismic pre-stack depth migration. Then, we performed experiments with our applications such as strong scaling and weak scaling experiments. We had to explore a large amount of parameters including seven different applications (three direct methods to solve linear systems, the LU factorization alone, the sparse matrix vector product with several sparse storage formats and the Kirchhoff seismic pre-stack depth migration) implemented with multiple programming models such as MPI, XMP and several task based programming models. For each application, we had to find optimal parameters especially the number of tasks and the size of the task for the dense linear applications as well as the data distribution for the sparse applications. We installed and performed experiments on two post-petascale computers, PangeII at Total and Jean Zay at the CNRS National Center of Computation (IDRIS). We did not have results on Jean Zay since we did not manage to install the task based programming models and fully deploy our applications before the end of the dissertation writing. Moreover, due to exploitation reasons, we were not able to use the complete PangeII supercomputer. Nonetheless, we were able to observe the capabilities and performances as well as the shortcomings and limitations of the task based programming models on post-petascale supercomputers. We encountered installation and deployment issues while proceeding to the installation of the task based programming models and the deployment of our applications such as missing dependencies that we had to install and incompatibilities with some versions and implementations of compilers. We also showed the behavior of these algorithms while implemented with task based programming models. The goal was not to make the most efficient parallelization of the algorithms but we tried to obtain the best performances we could.

We showed that task based programming models are very suitable to solve problems with irregular dependencies such as block dense linear algebra. However, to achieve good performances, the user has to find the optimal parameters in term of the number and the granularity of tasks in order to balance overhead from task scheduling and execution with enough parallelism and tasks to fully use the available resources. As for algorithm mainly based on a fork-join process, in which computations are performed on distributed data then the output of the computations has to be combined in order to be reused in a similar fashion,

the performances heavily depend on the ability of the programming model to perform the combination of the results. This is illustrated by our implementation of the sparse operation $A(Ax + x)$ which needs to combine the output of the first sparse matrix vector through a reduction and/or a gather depending on the decomposition of the sparse matrix. We showed that task based programming models do not provide efficient collective operations that can be executed on multiple data coming from tasks. It makes algorithm based on fork-join parallelization not suitable for task based programming models without collective operations on a given amount of task outputs. Lastly, the current task based programming models support interactions with the file system only by directly implementing IOs in tasks without having any hint that the task performs IOs. This means that the scheduler is not aware that the task performs IOs and cannot schedule it accordingly. Therefore, we were not able to make IOs task properly scheduled by the programming model for our Kirchhoff seismic pre-stack depth migration that heavily relies on the file system to load the input data and the Green functions used to approximate the travel time of the recorded sound waves. Thus, we generated most of our data during the execution which was not as close to the original application as we would like. Moreover, we also greatly simplified the application which resulted in less dependencies and a more regular application especially with the implementation of a 2D case instead of the 3D case used in practice. We implemented the Kirchhoff seismic pre-stack depth migration application from scratch which resulted in an application with less features implemented. It may have been more interesting to try to port an original and already implemented application to a task based application in order to have more realistic tasks and dependencies.

Besides, we highlighted that task based programming models have great potential in efficiently scheduling data migrations including file system IOs. The scheduler could also manage fault tolerance due to the possibility of restarting failed tasks on other resources if necessary. Moreover, the expression of tasks and data dependencies provides the necessary information to implement check-pointing. Indeed, the task output and the execution information such as the position in the execution graph could be regularly and efficiently saved during execution in order to let the scheduler relaunch the application at a previous save depending on the available data.

Therefore, task based programming models require a good balance between the number and the granularity of tasks to obtain the best performances but these crucial parameters can only be found by experimenting. They are very suitable for applications with irregular dependencies such as block linear algebra. However, they are less suitable for fork-join based applications which rely on same computations on split data on multiple computing resources and an operation that combines each output due to the lack of efficient to express and execute such operations (like reductions) with task based programming models. Furthermore, we outlined that task based programming models have great potential for optimizing computation placement and data migrations including file system IOs as well as managing fault tolerance and check-pointing with an efficient scheduler. With the appearance of more efficient schedulers that are adapted to the post-petascale supercomputer used and the integration of our recommendations, programming models based on a graph of tasks could become a very efficient tool to implement scientific applications on post-petascale supercomputers such as linear algebra applications or geoscience applications for Total.

9.2 Future Researches

This work could be completed with applications implemented and experiments performed with Pegasus since it is very suitable with our methodology. Moreover, applications could be implemented with other task based programming models such as X10, Chapel, Uintah or Charm++ in order to explore their capabilities to solve such problem. These experiments could also be reproduced on other clusters or supercomputers. Our task based Kirchhoff seismic pre-stack depth migration could be improved to be closer to what is done in practice as well as include IOs tasks to test the ability of the task based programming models to schedule such tasks.

Improving the YML+XMP dense matrix type can also be considered since it would allow more flexibility for YML+XMP dense linear algebra applications and allow the use of parameter values that could lead to better results.

A possible amelioration to task based programming models is to improve the capabilities of the schedulers in terms of memory management and task scheduling in order to avoid the data migrations and fit data in the available memory of the nodes. Implementing efficient task based collective operations will also help to improve performances of task based programming models.

Fault tolerance can also be integrated in task based programming models in order to restart failed tasks. The scheduler could try to determine if the failure comes from the task implementation by trying to restart the failed tasks on other resources or if the failure comes from the hardware by checking if the workers that execute tasks are still functioning.

Check-pointing is also an interesting feature that could be integrated into task based programming models. The scheduler could add tasks to save data regularly in oder to restart the application on a given snapshot.

References

[1] M. P. Forum, "Mpi: a message-passing interface standard", USA, Tech. Rep., 1994.

[2] L. Dagum and R. Menon, "Openmp: an industry-standard api for shared-memory programming", *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998, ISSN: 1070-9924. DOI: 10.1109/99.660313. [Online]. Available: https://doi.org/10.1109/99.660313.

[3] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*, 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012, ISBN: 9780124159334.

[4] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. DeBardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen, "Addressing failures in exascale computing", *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014. DOI: 10.1177/1094342014522573. [Online]. Available: https://doi.org/10.1177/1094342014522573.

[5] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda, "An evaluation of global address space languages: co-array fortran and unified parallel c", Jan. 2005, pp. 36–47. DOI: 10.1145/1065944.1065950.

[6] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "Dague: a generic distributed dag engine for high performance computing", in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 2011, pp. 1151–1158. DOI: 10.1109/IPDPS.2011.281.

[7] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: expressing locality and independence with logical regions", in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, Nov. 2012, pp. 1–11. DOI: 10.1109/SC.2012.71.

[8] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: a high-productivity programming language for hpc with logical regions", in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12. DOI: 10.1145/2807591.2807629.

[9] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J.

Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: large-scale machine learning on heterogeneous distributed systems", *CoRR*, vol. abs/1603.04467, 2016. arXiv: 1603.04467. [Online]. Available: http://arxiv.org/abs/1603.04467.

[10] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: a task based programming model in a global address space", in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14, Eugene, OR, USA: ACM, 2014, 6:1–6:11, ISBN: 978-1-4503-3247-7. DOI: 10.1145/2676870.2676883. [Online]. Available: http://doi.acm.org/10.1145/2676870.2676883.

[11] O. Delannoy and S. Petiton, "A peer to peer computing framework: design and performance evaluation of yml", in *Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Jul. 2004, pp. 362–369. DOI: 10.1109/ISPDC.2004.7.

[12] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: a framework for mapping complex scientific workflows onto distributed systems", *Scientific Programming Journal*, vol. 13, no. 3, pp. 219–237, 2005. [Online]. Available: http://pegasus.isi.edu/publications/Sci.pdf.

[13] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde, "Swift: fast, reliable, loosely coupled parallel computation", in *2007 IEEE Congress on Services (Services 2007)*, Jul. 2007, pp. 199–206. DOI: 10.1109/SERVICES.2007.63.

[14] D. Kirk and W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, ser. Applications of GPU Computing Series. Morgan Kaufmann Publishers, 2010, ISBN: 9780123814722. [Online]. Available: https://books.google.fr/books?id=x8oNlQEACAAJ.

[15] W. Hillis, *The Connection Machine*, ser. ACM distinguished dissertations. Cambridge, 1989, ISBN: 9780262580977. [Online]. Available: https://books.google.fr/books?id=xg_yaoC6CNEC.

[16] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, "Pvm 3 user's guide and reference manual", Nov. 1995.

[17] D. Gelernter and N. Carriero, "Coordination languages and their significance", *Commun. ACM*, vol. 35, no. 2, 97–107, Feb. 1992, ISSN: 0001-0782. DOI: 10.1145/129630.129635. [Online]. Available: https://doi.org/10.1145/129630.129635.

[18] N. J. Carriero, D. Gelernter, T. G. Mattson, and A. H. Sherman, "The linda alternative to message-passing systems", *Parallel Computing*, vol. 20, no. 4, pp. 633 –655, 1994, Message Passing Interfaces, ISSN: 0167-8191. DOI: https://doi.org/10.1016/0167-8191(94)90032-9. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0167819194900329.

[19] R. Butler and E. Lusk, "User's guide to the p4 parallel programming system", Technical Report ANL-92/17, Argonne National Laboratory, Tech. Rep., 1992.

[20] A. YarKhan, J. Kurzak, and J. Dongarra, "Quark users' guide: queueing and runtime for kernels", 2011.

[21] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: an imperative style, high-performance deep learning library", in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[22] C.-X. Lin, T.-W. Huang, G. Guo, and M. D. F. Wong, "A modern c++ parallel task programming library", in *Proceedings of the 27th ACM International Conference on Multimedia*, ser. MM '19, Nice, France: Association for Computing Machinery, 2019, 2284–2287, ISBN: 9781450368896. DOI: 10.1145/3343031.3350537. [Online]. Available: https://doi.org/10.1145/3343031.3350537.

[23] T. Huang, C. Lin, G. Guo, and M. Wong, "Cpp-taskflow: fast task-based parallel programming using modern c++", in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 974–983.

[24] C. Lin, T. Huang, G. Guo, and M. D. F. Wong, "An efficient and composable parallel task programming library", in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, 2019, pp. 1–7.

[25] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of openmp tasks", *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, 2009.

[26] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: enabling manycore performance portability through polymorphic memory access patterns", *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 –3216, 2014, Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2014.07.003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257.

[27] H. C. Edwards and D. A. Ibanez, "Kokkos' task dag capabilities.", Sep. 2017. DOI: 10.2172/1398234.

[28] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system", in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '95, Santa Barbara, California, USA: Association for Computing Machinery, 1995, 207–216, ISBN: 0897917006. DOI: 10.1145/209936.209958. [Online]. Available: https://doi.org/10.1145/209936.209958.

[29] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing", *J. ACM*, vol. 46, no. 5, 720–748, Sep. 1999, ISSN: 0004-5411. DOI: 10.1145/324133.324234. [Online]. Available: https://doi.org/10.1145/324133.324234.

[30] C. E. Leiserson, "The cilk++ concurrency platform", in *Proceedings of the 46th Annual Design Automation Conference*, ser. DAC '09, San Francisco, California: Association for Computing Machinery, 2009, 522–527, ISBN: 9781605584973. DOI: 10.1145/1629911.1630048. [Online]. Available: https://doi.org/10.1145/1629911.1630048.

[31] J. Reinders, *Intel Threading Building Blocks*, First. USA: O'Reilly and Associates, Inc., 2007, ISBN: 9780596514808. DOI: 10.5555/1461409.

[32] C. Pheatt, "Intel threading building blocks", vol. 23, no. 4, p. 298, Apr. 2008, ISSN: 1937-4771. DOI: 10.5555/1352079.1352134.

[33] Dask Development Team, *Dask: library for dynamic task scheduling*, 2016. [Online]. Available: https://dask.org.

[34] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures", *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011. DOI: 10.1142/S0129626411000151.

[35] J. M. Pérez, R. M. Badia, and J. Labarta, "A flexible and portable programming model for smp and multi-cores bsc-upc computer sciences program", 2007.

[36] A. Chandramowlishwaran, K. Knobe, and R. Vuduc, "Performance evaluation of concurrent collections on high-performance multicore computing systems", in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, Apr. 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470404.

[37] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşirlar, "Concurrent collections", *Sci. Program.*, vol. 18, no. 3-4, pp. 203–217, Aug. 2010, ISSN: 1058-9244. DOI: 10.1155/2010/521797. [Online]. Available: http://dx.doi.org/10.1155/2010/521797.

[38] J. D.d. S. Germain, J. McCorquodale, S. G. Parker, and C. R. Johnson, "Uintah: a massively parallel problem solving environment", in *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing*, ser. HPDC '00, USA: IEEE Computer Society, 2000, p. 33, ISBN: 0769507832.

[39] A. Humphrey and M. Berzins, "An evaluation of an asynchronous task based dataflow approach for uintah", in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2, 2019, pp. 652–657. [Online]. Available: http://www.sci.utah.edu/publications/Hum2019b/dfm19_humphrey_berzins.pdf.

[40] D. Sahasrabudhe, M. Berzins, and J. Schmidt, "Node failure resiliency for uintah without checkpointing", *Concurrency and Computation: Practice and Experience*, e5340, 2019. [Online]. Available: http://www.sci.utah.edu/publications/Sah2019a/00_Uintah_Resiliency.pdf.

[41] J. K. Holmen, B. Peterson, A. Humphrey, D. Sunderland, O. H. Diaz-Ibarra, J. N. Thornock, and M. Berzins, "Portably improving uintah's readiness for exascale systems through the use of kokkos", SCI Institute, Tech. Rep., 2019. [Online]. Available: http://www.sci.utah.edu/publications/Hol2019a/UUSCI-2019-001.pdf.

[42] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++", in *Proceedings of OOPSLA'93*, A. Paepcke, Ed., ACM Press, 1993, pp. 91–108.

[43] M. P. Robson, R. Buch, and L. V. Kale, "Runtime coordinated heterogeneous tasks in charm++", in *Proceedings of the Second Internationsl Workshop on Extreme Scale Programming Models and Middleware*, ser. ESPM2, Salt Lake City, Utah: IEEE Press, 2016, pp. 40–43, ISBN: 978-1-5090-3858-9. DOI: 10.1109/ESPM2.2016.7. [Online]. Available: https://doi.org/10.1109/ESPM2.2016.7.

[44] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing", *SIGPLAN Not.*, vol. 40, no. 10, 519–538, Oct. 2005, ISSN: 0362-1340. DOI: 10.1145/1103845.1094852. [Online]. Available: https://doi.org/10.1145/1103845.1094852.

[45] S. S. Hamouda, B. Herta, J. Milthorpe, D. Grove, and O. Tardieu, "Resilient x10 over mpi user level failure mitigation", in *Proceedings of the 6th ACM SIGPLAN Workshop on X10*, ser. X10 2016, Santa Barbara, CA, USA: Association for Computing Machinery, 2016, 18–23, ISBN: 9781450343862. DOI: 10.1145/2931028.2931030. [Online]. Available: https://doi.org/10.1145/2931028.2931030.

[46] D. Grove, S. S. Hamouda, B. Herta, A. Iyengar, K. Kawachiya, J. Milthorpe, V. Saraswat, A. Shinnar, M. Takeuchi, and O. Tardieu, "Failure recovery in resilient x10", *ACM Trans. Program. Lang. Syst.*, vol. 41, no. 3, Jul. 2019, ISSN: 0164-0925. DOI: 10.1145/3332372. [Online]. Available: https://doi.org/10.1145/3332372.

[47] E. Slaughter and A. Aiken, "Pygion: flexible, scalable task-based parallelism with python", in *2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*, 2019, pp. 58–72.

[48] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "Parsec: exploiting heterogeneity to enhance scalability", *Computing in Science Engineering*, vol. 15, no. 6, pp. 36–45, Nov. 2013, ISSN: 1521-9615. DOI: 10.1109/MCSE.2013.98.

[49] A. Danalis, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra, "Ptg: an abstraction for unhindered parallelism", in *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, 2014, pp. 21–30. DOI: 10.1109/WOLFHPC.2014.8.

[50] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra, "Dynamic task discovery in parsec: a data-flow task-based runtime", in *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ser. ScalA '17, Denver, Colorado: ACM, 2017, 6:1–6:8, ISBN: 978-1-4503-5125-6. DOI: 10.1145/3148226.3148233. [Online]. Available: http://doi.acm.org/10.1145/3148226.3148233.

[51] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures", *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011. DOI: 10.1002/cpe.1631.

[52] E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi, and J. Labarta, "Clusterss: a task-based programming model for clusters", in *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, ser. HPDC '11, San Jose, California, USA: Association for Computing Machinery, 2011, 267–268, ISBN: 9781450305525. DOI: 10.1145/1996130.1996168. [Online]. Available: https://doi.org/10.1145/1996130.1996168.

[53] D. Callahan, B. L. Chamberlain, and H. P. Zima, "The cascade high productivity language", in *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.*, 2004, pp. 52–60. DOI: 10.1109/HIPS.2004.1299190.

[54] O. Delannoy, F. Emad, and S. Petiton, "Workflow global computing with yml", in *2006 7th IEEE/ACM International Conference on Grid Computing*, 2006, pp. 25–32. DOI: 10.1109/ICGRID.2006.310994.

[55] M. Tsuji, M. Sato, M. Hugues, and S. Petiton, "Multiple-spmd programming environment based on pgas and workflow toward post-petascale computing", in *2013 42nd International Conference on Parallel Processing*, Oct. 2013, pp. 480–485. DOI: 10.1109/ICPP.2013.58.

[56] M. Sato, M. Hirano, Y. Tanaka, and S. Sekiguchi, "Omnirpc: a grid rpc facility for cluster and global computing in openmp", in *International Workshop on OpenMP Applications and Tools*, Springer, 2001, pp. 130–136. DOI: 10.1007/3-540-44587-0_12.

[57] M. Tsuji, S. Petiton, and M. Sato, "Fault tolerance features of a new multi-spmd programming/execution environment", in *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware*, ser. ESPM '15, Austin, Texas: ACM, 2015, pp. 20–27, ISBN: 978-1-4503-3996-4. DOI: 10.1145/2832241.2832243. [Online]. Available: http://doi.acm.org/10.1145/2832241.2832243.

[58] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: a language for distributed parallel scripting", *Parallel Computing*, vol. 37, no. 9, pp. 633 –652, 2011, Emerging Programming Paradigms for Large-Scale Scientific Computing, ISSN: 0167-8191. DOI: 10.1016/j.parco.2011.05.005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111000524.

[59] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster, "Swift/t: large-scale application composition via distributed-memory dataflow processing", in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, May 2013, pp. 95–102. DOI: 10.1109/CCGrid.2013.99.

[60] J. M. Wozniak, T. G. Armstrong, K. Maheshwari, E. L. Lusk, D. S. Katz, M. Wilde, and I. T. Foster, "Turbine: a distributed-memory dataflow engine for extreme-scale many-task applications", in *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, ser. SWEET '12, Scottsdale, Arizona, USA: ACM, 2012, 5:1–5:12, ISBN: 978-1-4503-1876-1. DOI: 10.1145/2443416.2443421.

[61] E. L. Lusk, S. C. Pieper, R. M. Butler, and M. T. S. Univ., "More scalability, less pain : a simple programming model and its implementation for extreme computing.", *SciDAC Rev.*, vol. 17, no. 2010, Jan. 2010.

[62] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the condor experience", *Concurrency and Computation: Practice and Experience*, vol. 17, no. 24, pp. 323–356, 2005. DOI: 10.1002/cpe.938. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.938. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.938.

[63] ——, "Condor and the grid", in *Grid Computing: Making the Global Infrastructure a Reality*, F. Berman, G. Fox, and T. Hey, Eds., John Wiley & Sons Inc., Dec. 2002.

[64] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger, "Workflow management in condor", in *Workflows for e-Science: Scientific Workflows for Grids*, I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, Eds. London: Springer London, 2007, pp. 357–375, ISBN: 978-1-84628-757-2. DOI: 10.1007/978-1-84628-757-2_22. [Online]. Available: https://doi.org/10.1007/978-1-84628-757-2_22.

[65] B. N. Parlett, "Linpack users' guide (j. j. dongarra, j. r. bunch, c. b. moler and g. w. stewart)", *SIAM Review*, vol. 23, no. 1, pp. 126–128, 1981. DOI: 10.1137/1023033. eprint: https://doi.org/10.1137/1023033. [Online]. Available: https://doi.org/10.1137/1023033.

[66] J. J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: past, present and future", *Concurrency and Computation: Practice and Experience*, vol. 15, no. 9, pp. 803–820, 2003. DOI: 10.1002/cpe.728. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.728. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.728.

[67] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999, ISBN: 0-89871-447-8 (paperback).

[68] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "Scalapack: a scalable linear algebra library for distributed memory concurrent computers", in *[Proceedings 1992] The Fourth Symposium on the Frontiers of Massively Parallel Computation*, 1992, pp. 120–127. DOI: 10.1109/FMPC.1992.234898.

[69] L. S. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, "Scalapack: a portable linear algebra library for distributed memory computers - design issues and performance", in *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, 1996, pp. 5–5. DOI: 10.1109/SUPERC.1996.183513.

[70] J. W. Demmel, N. J. Higham, and R. S. Schreiber, "Stability of block lu factorization", *Numerical Linear Algebra with Applications*, vol. 2, no. 2, pp. 173–190, 1995. DOI: 10.1002/nla.1680020208. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/nla.1680020208. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/nla.1680020208.

[71] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley, "The design and implementation of the scalapack lu, qr, and cholesky factorization routines", *Sci. Program.*, vol. 5, no. 3, 173–184, Aug. 1996. DOI: 10.1155/1996/483083. [Online]. Available: https://doi.org/10.1155/1996/483083.

[72] G. H. Golub and C. F. Van Loan, "Matrix computations", *Johns Hopkins University, Baltimore*, 1983.

[73] J. R. Bunch, "Block methods for solving sparse linear systems", in *Sparse Matrix Computations*, J. R. BUNCH and D. J. ROSE, Eds., Academic Press, 1976, pp. 39 –58, ISBN: 978-0-12-141050-6. DOI: https://doi.org/10.1016/B978-0-12-141050-6.50008-8. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780121410506500088.

[74] R. H. Bisseling and J. G. G. van de Vorst, "Parallel lu decomposition on a transputer network", in *Parallel Computing 1988*, G. A. van Zee and J. G. G. van de Vorst, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 61–77, ISBN: 978-3-540-46689-5.

[75] E. F. van de Velde, "Experiments with multicomputer lu-decomposition", *Concurrency: Practice and Experience*, vol. 2, no. 1, pp. 1–26, 1990. DOI: 10.1002/cpe.4330020102. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4330020102. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4330020102.

[76] Y. Saad, "Communication complexity of the gaussian elimination algorithm on multiprocessors", *Linear Algebra and its Applications*, vol. 77, pp. 315 –340, 1986, ISSN: 0024-3795. DOI: https://doi.org/10.1016/0024-3795(86)90174-6. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0024379586901746.

[77] E. Chu and A. George, "Gaussian elimination with partial pivoting and load balancing on a multiprocessor", *Parallel Comput.*, vol. 5, pp. 65–74, 1987.

[78] S. G. Petiton, "Parallelization on an mimd computer with realtime scheduler, gauss-jordan example", in *Aspects of computation on asynchronous parallel processors*, North Holland, 1989.

[79] S. Attaway, "Chapter 12 - matrix representation of linear algebraic equations", in *Matlab (Second Edition)*, S. Attaway, Ed., Second Edition, Boston: Butterworth-Heinemann, 2012, pp. 367 –399, ISBN: 978-0-12-385081-2. DOI: https://doi.org/10.1016/B978-0-12-385081-2.00012-0. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780123850812000120.

[80] N. Melab, E. G. Talbi, and S. Petiton, "A parallel adaptive version of the block-based gauss-jordan algorithm", in *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999*, 1999, pp. 350–354. DOI: 10.1109/IPPS.1999.760499.

[81] Y. Saad, "Krylov subspace methods on supercomputers", *Siam Journal on Scientific and Statistical Computing*, vol. 10, pp. 1200–1232, 1989.

[82] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Second. Society for Industrial and Applied Mathematics, 2003. DOI: 10.1137/1.9780898718003. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898718003. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9780898718003.

[83] C. Ridings and M. Shishigin, "Pagerank uncovered", *Technical Paper for the Search Engine Optimization Online Community*, 2002.

[84] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*, J. Kepner and J. Gilbert, Eds. Society for Industrial and Applied Mathematics, 2011. DOI: 10.1137/1.9780898719918. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898719918. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9780898719918.

[85] W. S. Song, V. Gleyzer, A. Lomakin, and J. Kepner, "Novel graph processor architecture, prototype system, and results", *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, 2016. DOI: 10.1109/hpec.2016.7761635. [Online]. Available: http://dx.doi.org/10.1109/HPEC.2016.7761635.

[86] Y. Saad, "Sparskit: a basic tool kit for sparse matrix computations", Jun. 1990.

[87] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters", in *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation - Volume 6*, ser. OSDI'04, San Francisco, CA: USENIX Association, 2004, p. 10.

[88] G. Hachtel, R. Brayton, and F. Gustavson, "The sparse tableau approach to network analysis and design", *IEEE Transactions on Circuit Theory*, vol. 18, no. 1, pp. 101–113, Jan. 1971. DOI: 10.1109/TCT.1971.1083223.

[89] F. G. Gustavson, "Some basic techniques for solving sparse systems of linear equations", in *Sparse Matrices and their Applications: Proceedings of a Symposium on Sparse Matrices and Their Applications, held September 9–10, 1971, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, the National Science Foundation, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*. D. J. Rose and R. A. Willoughby, Eds. Boston, MA: Springer US, 1972, pp. 41–52, ISBN: 978-1-4615-8675-3. DOI: 10.1007/978-1-4615-8675-3_4. [Online]. Available: https://doi.org/10.1007/978-1-4615-8675-3_4.

[90] S. C. Eisenstat, H. C. Elman, M. H. Schultz, and A. H. Sherman, "The yale sparse matrix package", Office of Naval Research under contract N00014-82-K-0184 and National Science Foundation under grant MCS-81-04874, Tech. Rep., 1977.

[91] S. G. Petiton, "Massively Parallel Sparse Matrix Computation for Iterative Methods", Yale University, Department of Computer Science, Tech. Rep., 1991.

[92] S. G. Petiton and N. Emad, "A data parallel scientific computing introduction", in *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, Berlin, Heidelberg: Springer-Verlag, 1996, pp. 45–64, ISBN: 3-540-61736-1. [Online]. Available: http://dl.acm.org/citation.cfm?id=647429.723576.

[93] F. Ye, C. Calvin, and S. G. Petiton, "A study of spmv implementation using mpi and openmp on intel many-core architecture", in *High Performance Computing for Computational Science – VECPAR 2014*, M. Daydé, O. Marques, and K. Nakajima, Eds., Cham: Springer International Publishing, 2015, pp. 43–56, ISBN: 978-3-319-17353-5.

[94] B. A. Page and P. M. Kogge, "Scalability of hybrid sparse matrix dense vector (spmv) multiplication", in *2018 International Conference on High Performance Computing Simulation (HPCS)*, 2018, pp. 406–414. DOI: 10.1109/HPCS.2018.00072.

[95] W. Ferng, K. Wu, S. Petiton, and Y. Saad, "Basic sparse matrix computations on massively parallel computers", Mar. 1993.

[96] A. Pinar and M. T. Heath, "Improving performance of sparse matrix-vector multiplication", in *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, ser. SC '99, Portland, Oregon, USA: Association for Computing Machinery, 1999, 30–es, ISBN: 1581130910. DOI: 10.1145/331532.331562. [Online]. Available: https://doi.org/10.1145/331532.331562.

[97] S. Toledo, "Improving the memory-system performance of sparse-matrix vector multiplication", *IBM Journal of Research and Development*, vol. 41, no. 6, pp. 711–725, 1997.

[98] B. Bramas and P. Kus, "Computing the sparse matrix vector product using block-based kernels without zero padding on processors with avx-512 instructions", *PeerJ Computer Science*, vol. 4, e151, 2018.

[99] R. W. Vuduc, "Automatic performance tuning of sparse matrix kernels", PhD thesis, University of California, Berkeley, 2003.

[100] E.-J. Im, "Optimizing the performance of sparse matrix-vector multiplication", PhD thesis, EECS Department, University of California, Berkeley, 2000.

[101] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda", *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008, ISSN: 1542-7730. DOI: 10.1145/1365490.1365500. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500.

[102] M. R. Hugues and S. G. Petiton, "Sparse matrix formats evaluation and optimization on a gpu", in *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, 2010, pp. 122–129. DOI: 10.1109/HPCC.2010.85.

[103] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors", in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13, Eugene, Oregon, USA: ACM, 2013, pp. 273–282, ISBN: 978-1-4503-2130-3. DOI: 10.1145/2464996.2465013. [Online]. Available: http://doi.acm.org/10.1145/2464996.2465013.

[104] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix vector multiplication on emerging multicore platforms", *Parallel Computing*, vol. 35, no. 3, pp. 178 –194, 2009, Revolutionary Technologies for Acceleration of Emerging Petascale Applications, ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2008.12.006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819108001403.

[105] R. W. Vuduc and H.-J. Moon, "Fast sparse matrix-vector multiplication by exploiting variable block structure", in *High Performance Computing and Communications*, L. T. Yang, O. F. Rana, B. Di Martino, and J. Dongarra, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 807–816, ISBN: 978-3-540-32079-1.

[106] S. Petiton and C. Weill-Duflos, "Massively parallel preconditioners for the sparse conjugate gradient method", in *Parallel Processing: CONPAR 92—VAPP V*, L. Bougé, M. Cosnard, Y. Robert, and D. Trystram, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 373–378, ISBN: 978-3-540-47306-0.

[107] S. G. Petiton, "Expertise en calcul scientifique haute performance concernant la programmation par graphes de taches/composants et les nouveaux langages de programmation - programmation par graphe de tâches-composants et migration de kirchhoff", Laboratoire d'Informatique Fondamentale de Lille, Lille University, Tech. Rep., Jun. 2014.

[108] L. Ping and C. Yunhe, "Seismic 3d prestack time migration on parallel computers", *Journal of Systems Engineering and Electronics*, vol. 6, no. 3, pp. 49–55, Sep. 1995.

[109] J. Panetta, T. Teixeira, P. R. P. de Souza Filho, C. A. da Cunha Finho, D. Sotelo, F. M. R. da Motta, S. S. Pinheiro, I. P. Junior, A. L. R. Rosa, L. R. Monnerat, L. T. Carneiro, and C. H. B. de Albrecht, "Accelerating kirchhoff migration by cpu and gpu cooperation", in *2009 21st International Symposium on Computer Architecture and High Performance Computing*, Oct. 2009, pp. 26–32. DOI: 10.1109/SBAC-PAD.2009.29.

[110] R. Xu, M. Hugues, H. Calandra, S. Chandrasekaran, and B. Chapman, "Accelerating kirchhoff migration on gpu using directives", in *2014 First Workshop on Accelerator Programming using Directives*, Nov. 2014, pp. 37–46. DOI: 10.1109/WACCPD.2014.8.

[111] H. Kaiser, B. A. L. aka wash, T. Heller, A. Bergé, M. Simberg, J. Biddiscombe, A. Bikineev, G. Mercer, A. Schäfer, A. Serio, T. Kwon, K. Huck, J. Habraken, M. Anderson, M. Copik, S. R. Brandt, M. Stumpf, D. Bourgeois, D. Blank, S. Jakobovits, V. Amatya, L. Viklund, Z. Khatami, D. Bacharwar, S. Yang, E. Schnetter, P. Diehl, N. Gupta, B. Wagle, and Christopher, *STEllAR-GROUP/hpx: HPX V1.2.1: The C++ Standards Library for Parallelism and Concurrency*, Feb. 2019. DOI: 10.5281/zenodo.2573213. [Online]. Available: https://doi.org/10.5281/zenodo.2573213.

[112] M. Sato, T. Boku, and D. Takahashi, "Omnirpc: a grid rpc system for parallel programming in cluster and grid environment", in *CCGrid 2003. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2003. Proceedings.*, 2003, pp. 206–213. DOI: 10.1109/CCGRID.2003.1199370.

[113] J. Lee and M. Sato, "Implementation and performance evaluation of xcalablemp: a parallel programming language for distributed memory systems", in *2010 39th International Conference on Parallel Processing Workshops*, Sep. 2010, pp. 413–420. DOI: 10.1109/ICPPW.2010.62.

[114] J. Gurhem, M. Tsuji, S. G. Petiton, and M. Sato, "Distributed and parallel programming paradigms on the k computer and a cluster", in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, ser. HPC Asia 2019, Guangzhou, China: ACM, 2019, pp. 9–17, ISBN: 978-1-4503-6632-8. DOI: 10.1145/3293320.3293330. [Online]. Available: http://doi.acm.org/10.1145/3293320.3293330.

[115] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger, "Pegasus: a workflow management system for science automation", *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015, Funding Acknowledgements: NSF ACI SDCI 0722019, NSF ACI SI2-SSI 1148515 and NSF OCI-1053575. DOI: 10.1016/j.future.2014.10.008. [Online]. Available: http://pegasus.isi.edu/publications/2014/2014-fgcs-deelman.pdf.

[116] W. L. Poehlman, M. Rynge, C. Branton, D. Balamurugan, and F. A. Feltus, "Osg-gem: gene expression matrix construction using the open science grid", *Bioinformatics and Biology Insights*, vol. 10, BBI.S38193, 2016. DOI: 10.4137/BBI.S38193. eprint: https://doi.org/10.4137/BBI.S38193. [Online]. Available: https://doi.org/10.4137/BBI.S38193.

[117] A. N. Adhikari, J. Peng, M. Wilde, J. Xu, K. F. Freed, and T. R. Sosnick, "Modeling large regions in proteins: applications to loops, termini, and folding", *Protein Science*, vol. 21, no. 1, pp. 107–121, DOI: 10.1002/pro.767.

[118] M. Woitaszek, J. M. Dennis, and T. R. Sines, "Parallel high-resolution climate data analysis using swift", in *Proceedings of the 2011 ACM International Workshop on Many Task Computing on Grids and Supercomputers*, ser. MTAGS '11, Seattle, Washington, USA: ACM, 2011, pp. 5–14, ISBN: 978-1-4503-1145-8. DOI: 10.1145/2132876.2132882.

[119] J. Gurhem and S. G. Petiton, "A current task-based programming paradigms analysis", in *Computational Science – ICCS 2020*, V. V. Krzhizhanovskaya, G. Závodszky, M. H. Lees, J. J. Dongarra, P. M. A. Sloot, S. Brissos, and J. Teixeira, Eds., Cham: Springer International Publishing, 2020, pp. 203–216, ISBN: 978-3-030-50426-7.

[120] T. Yoshida, M. Hondo, R. Kan, and G. Sugizaki, "Sparc64 viiifx: cpu for the k computer", *Fujitsu scientific and technical journal*, vol. 48, pp. 274–279, Jul. 2012.

[121] D. Ellsworth, T. Patki, S. Perarnau, S. Seo, A. Amer, J. A. Zounmevo, R. Gupta, K. Yoshii, H. Hoffman, A. Malony, M. Schulz, and P. H. Beckman, "Systemwide power management with argo", in *Parallel and Distributed Processing Symposium Workshops*, IEEE, IEEE, 2016. DOI: 10.1109/IPDPSW.2016.81.

[122] M. R. Hugues, M. Moretti, S. G. Petiton, and H. Calandra, "Asiods - an asynchronous and smart i/o delegation system", *Procedia Computer Science*, vol. 4, pp. 471 –478, 2011, Proceedings of the International Conference on Computational Science, ICCS 2011, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2011.04.049. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050911001074.

[123] S. G. Petiton, J. Gurhem, and H. Calandra, "A taxonomy of distributed and parallel languages for high performance tasks-based multilevel computing", in *SIAM 2020 Conference on Parallel Processing for Scientific Computing*, Feb. 2020.

[124] X. Wu, "Contribution to the emergence of new intelligent parallel and distributed methods using a multi-level programming paradigm for extreme computing", PhD thesis, University of Lille, 2019.