Parallelization of iterative methods to solve sparse linear systems using task based runtime systems on multi and many-core architectures: application to Multi-Level Domain Decomposition methods

Adrien Roussel

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 25 mai 2016

Présentée par

Adrien ROUSSEL

Thèse dirigée par **Thierry Gautier**
et co-encadrée par **Jean-Marc Gratien**

préparée au sein de l' **INRIA Rhône-Alpes**
et de **IFP Energies Nouvelles**
et de **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Parallélisation sur un moteur exécutif à base de tâches des méthodes itératives pour la résolution de systèmes linéaires creux sur architecture multi et many cœurs : application aux méthodes de types décomposition de domaines multi-niveaux

Thèse soutenue publiquement le **06.02.2018**,
devant le Jury composé de :

M. Pierre Manneback
Professeur, Université de Mons, Rapporteur, Président du Jury
M. Joel Falcou
Maitre de conférence HDR, LRI, Université Paris Sud, Rapporteur
M. Bruno Raffin
Directeur de recherche INRIA, LIG, Université de Grenoble, Examinateur
M. Raymond Namyst
Professeur, INRIA, LaBRI, Université de Bordeaux, Examinateur
M. Thierry Gautier
Chargé de Recherche INRIA, LIP, ENS-Lyon, Directeur de thèse
M. Jean-Marc Gratien
Ingénieur de recherche, IFP Energies Nouvelles, Co-Encadrant de thèse

Remerciements

Enfin ! Cette page de remerciements représente l'aboutissement de près de 3 ans et demi de thèse. Toutes ces années m'ont apporté un grand enrichissement, aussi bien sur le plan professionnel que personnel. Avec ces quelques mots, je tiens à remercier tous ceux qui, de près ou de loin, m'ont permis de m'épanouir, de progresser et de cloturer cette expérience.

Tout d'abord, je tiens à remercier IFP Energies Nouvelles et INRIA Rhone-Alpes pour avoir soutenu les travaux de recherche de cette thèse. Je voudrais également remercier les membres du Jury de soutenance pour avoir accepter d'évaluer mes travaux. Tout d'abord, je tiens à remercier Joel Falcou et Pierre Manneback, qui m'a également fait l'honneur de président le jury de soutenance, pour leur rapport détaillé sur mon mémoire. Je voudrais également remercier chaleureusement Bruno Raffin et Raymond Namyst qui m'ont fait le plaisir d'accepter de participer à mon Jury de soutenance en tant qu'examinateurs. Enfin, je tiens à remercier Jean-Marc Gratien et Thierry Gautier de m'avoir encadré durant ces travaux, et de leur conseils sur toute la durée de ma thèse. Leur aide m'a beaucoup apporté durant ma thèse. Je tiens particulièrement à remercier Thierry qui, malgré la distance, m'a beaucoup conseillé et encouragé. Durant nos échanges, j'ai été particulièrement admiratif de ta grande culture scientifique et de la pertinence des échanges que l'on a pu avoir tout au long de cette thèse. Dorénavant, je ne vais plus surcharger ta boite mail !

Je voudrais remercier le Laboratoire d'Informatique du Parallélisme (LIP) de l'ENS-Lyon et l'équipe AVALON de l'INRIA pour leur accueil et l'accès aux machines qu'ils m'ont garanti pour faire certaines de mes expériences. Plus particulièrement, merci à Simon Delamare pour son aide dans la configuration du KNL et les installations logicielles !

Je n'aurais jamais pu commencer cette expérience de thèse sans Ani Anciaux et Thomas Guignon qui, alors que j'étais encore en Master à l'Université de Reims Champagne-Ardenne (URCA), m'ont permis de rencontrer Jean-Marc afin de postuler pour ce projet. A ceci, j'aimerais ajouter mes remerciements à Michael Krajecki, Arnaud Renard, Christophe Jaillet et Hervé Deleau de l'URCA pour m'avoir donné l'envie et la curiosité de poursuivre mes études en thèse, et également de m'avoir présenter à Ani et Thomas dans le cadre du stage de fin d'études de M2.

La thèse n'est pas qu'un travail académique, c'est aussi une expérience humaine. J'en profite donc pour remercier l'ensemble des membres du département d'Informatique Scientifique de l'IFPEN (R114), avec qui j'ai passé de très bons moments de rire et d'enrichissement professionnel : Ani, Thomas Guignon, Bruno, Christophe Cornet, Thomas Crabie, Laurent Astart, Christophe Delage, Jacques, Sylvie, Jean-Yves, Jean-Louis, Elsie, Vincent, Henry, Olivier et enfin Pierre Fery-Forgues.

Mention toute particulière aux thésards R114-R115 d'IFPEN. Je commence par les plus anciens, avec lesquels j'ai passé une première partie de thèse agréable en leur compagnie : Aboubacar et Huong. J'ai débuté cette expérience en même temps de Riad, et par la suite Mohamed, avec qui j'ai également de très bons moments. Bon courage à vous deux dans la poursuite professionnelle, nos chemins nous amèneront peut-être un jour à nous recroiser ! Et enfin, bon courage à ceux pour qui l'aventure continue encore un peu : Nicolas, Zakariae, Karine, Julien et Bastien. Je garderais encore pour longtemps en mémoire ces quelques moments de pauses passés avec vous tous. L'ambiance du groupe des thésards était vraiment top, et j'espère vous revoir très vite !

Enfin, je tiens à remercier les différentes personnes avec qui j'ai partagé mon environnement de travail, et avec qui j'ai eu l'occasion de beaucoup discuter et d'échangers sur divers sujets : Pierre, Youness, Jose Martin Lozano Aparicio (alias le péruvien chanteur), à nouveau Nicolas, et Olivier.

Je tiens à remercier Caroline, qui a réussi à me supporter au quotidien durant toute cette période de thèse. Je ne pourrais jamais exprimer suffisamment à quel point ta présence à mes côtés m'a permis de relever ce défi. Pour finir, je tiens également à remercier ma famille qui m'a apporté tout leur soutient inconditionnel dans l'élaboration de ce projet qu'a été ma thèse.

Contents

List of Figures

List of Tables

List of Algorithms

Résumé

Contexte et Motivation

La simulation numérique de réservoirs de pétrole conduit à la résolution d'équations aux dérivées partielles (EDP). Les schémas numériques aux Volumes Finis amènent à résoudre ces EDP via la résolution d'un système non-linéaire. La solution de ce système est approchée par une méthode de Newton, où à chaque itération ce dernier est linéarisé. Ce système linéaire, creux et de grande taille, doit être ensuite résolu grâce à un solveur linéaire. Cumulées, les phases de résolution de systèmes linéaires représentent jusqu'à 80% du temps total d'exécution des simulateurs de réservoirs à IFPEN.

Bien que robustes et précises, les méthodes de résolution directes ne conviennent pas aux systèmes de grandes tailles car elles nécessitent trop de mémoire. De plus, elles sont difficiles à paralléliser à cause de leur structure algorithmique. Dans ces conditions, les méthodes dites itératives sont donc privilégiées, bien que donnant une solution approchée et dépendent de la structure du système. Pour accélérer la vitesse de convergence de ces méthodes, on applique un préconditionneur sur le système. Un préconditionneur comme *Algebraic Multi-Grid* (AMG), bien que adapté numériquement à nos problèmes, se révèle cependant difficile à paralléliser à un niveau suffisamment fin pour les architectures parallèles possédant un grand nombre d'unités de calcul.

En revanche, les préconditionneurs de type Décomposition de Domaine sont eux adaptés aux architectures parallèles modernes. Se basant sur une stratégie « diviser pour mieux régner », la structure algébrique de ces méthodes est naturellement parallèle. En revanche, ces méthodes ne sont pas assez robustes numériquement pour des problèmes mal conditionnés comme ceux rencontrés en simulation réservoir. Néanmoins, avec l'ajout d'un second niveau, et notamment l'opérateur grossier GenEO introduit par les travaux de N. Spillane [Spillane et al., 2014] puis parallélisé par P. Jolivet [Jolivet et al., 2013] sur une machine à mémoire distribuée, ces méthodes regagnent d'intérêt dans notre domaine d'application. Pour autant, la construction de cet opérateur repose sur une discrétisation selon un schéma aux éléments finis, ce qui diffère d'un schéma aux volumes finis utilisés dans nos applications.

Actuellement, les architectures informatiques parallèles à mémoire partagée se basent sur le modèle « Non Uniform Memory Access » (NUMA). En plus d'une mémoire qui est segmentée, ces systèmes ont tendance à accroître leur nombre de cœurs de calculs. De plus, la hiérarchie mémoire qui règne dans ces machines est également importante. Ces caractéristiques font émerger des problèmes de latence d'accès à la mémoire et de la contention des bus mémoire. Le nombre d'unités de calculs par socket explose en passant par une simplification des cœurs. Ces processeurs many-cœurs apportent une complexité de programmation supplémentaire, comme la nécessité de vectorisation.

Une partie de la complexité de programmation engendrée par ces architectures migre de plus en plus d'une programmation bas niveau à un niveau supérieur. Des problématiques telles que la gestion des données ou l'ordonnancement des tâches parallèles sur les différentes ressources de calculs se retrouvent dorénavant gérées au travers d'un support exécutif. L'utilisation de nouveaux modèles de programmation parallèle a déjà été adopté en algèbre linéaire dense. En revanche, des problèmes épineux restent à surmonter lors de la parallélisation de méthodes algébriques telles que celles employées dans les simulateurs numériques.

Objectifs et contributions

L'objectif global de ces travaux est de fournir un préconditionneur de type Décomposition de Domaines qui exploite au mieux les capacités des architectures multi-cœurs. Cet objectif est en réalité double, il devra à la fois être robuste numériquement sur les cas de simulation réservoir et devra bénéficier également d'une implémentation parallèle efficace sur les architectures émergentes. Il est donc nécessaire de mettre en place les briques numériques et logicielles pour atteindre un tel niveau d'efficacité.

La première contribution de ce travail est d'évaluer une parallélisation de méthodes itératives via un modèle de programmation de type data-flow avec tâches. La parallélisation a été réalisée à travers une interface de programmation (i.e. API) qui permet à l'utilisateur de s'abstraire de l'implémentation sous-jacente et de se focaliser uniquement sur les aspects algorithmiques de la méthode. Cette API a été développée au-dessus de plusieurs supports d'exécution, tels que OpenMP 4.0, OmpSs, X-Kaapi et HARTS. L'interface de programmation s'articule autour de plusieurs principes, tels que la notion d'itération ou le découpage en tâches selon le graphe d'adjacence de la matrice des coefficients. Ces travaux ont fait l'objet d'une communication lors de la conférence nationale Compas [Roussel, 2016].

Après une première étude publiée dans [Virouleau et al., 2016b] sur la parallélisation de la multiplication d'une matrice creuse par un vecteur grâce au moteur exécutif XKaapi, nous avons intégré dans HARTS une manière de gérer l'exécution des tâches sur architectures NUMA en gérant de bout en bout la distribution des données jusqu'à l'exécution. Cette fonctionnalité est initialisée avant le début des sections parallèles car elle prend uniquement en compte le partitionnement des données, qui est connu qu début de l'exécution. A l'exécution, c'est l'ordonnanceur, basé sur un algorithme de type vol de tâche,

qui chargera alors le thread voleur de dérober une tâche opérant sur des données les plus proches possible.

Nous avons mis en place un système d'évaluation de performances à l'intérieur du support d'exécution HARTS. A l'aide de ces outils, nous avons la possibilité d'observer avec minutie le comportement de nos applications. Nous pouvons par exemple retracer le cours des évènements à travers la génération de diagramme de Gantt. A une échelle plus fine, nous pouvons faire des mesures au niveau d'une tâche, pour connaître aussi bien son temps d'exécution, sa granularité et même sur quelle unité de calculs elle s'est exécutée. Post-mortem, ces évènements sont utilisés pour aider à l'analyse des performances des exécutions. Par exemple par le calcul du chemin critique, du temps de travail, des temps d'inactivité par thread, etc.

Nous avons adapté nos méthodes pour les architectures de processeurs many-cœurs. Nous avons mis en évidence l'importance de la vectorisation des codes existants et l'utilisation de la mémoire à haute vitesse que nous propose le processeur Intel Xeon Phi Knights Landing. Bénéficiant du fait que nos données tiennent intégralement dans la mémoire à bande passante élevée, nous montrons la performance de nos applications sur ce type d'architecture.

Enfin, nous avons développé au dessus de l'API précédemment évoquée le préconditionneur de type Décomposition de Domaine 2-niveaux, basé sur l'opérateur grossier GenEO. Ainsi, nous avons pu attester de la performance numérique de ce préconditionneur sur des cas réservoirs qui, à ce niveau, permet d'être concurrent face à d'autres méthodes plus connues comme AMG. Nous avons aussi pu, à travers les travaux cités précédemment, mettre en place des implémentations multi-cœurs et many-cœurs efficaces du préconditionneur. Une étude préliminaire des performances du préconditionneur sur architecture multi-cœurs a été publiée dans la revue OGST [Roussel et al., 2016].

Résumé des chapitres

Dans le **premier chapitre**, nous introduisons le contexte de nos travaux et les objectifs attendus. Nous présentons également les contributions que nous avons apportées tout au long de cette thèse, avant d'introduire le plan du mémoire.

Dans le **second chapitre**, nous replaçons le contexte du problème qui nous a été posé ainsi que les difficultés rencontrées. Dans ce cadre, nous rappelons aussi bien les enjeux dans les domaines du numérique et de l'informatique dont découlent les problématiques alors posées tout au long de ces travaux.

Dans le **troisième chapitre**, nous introduisons une API d'algèbre linéaire creuse qui permet à ses utilisateurs de pouvoir écrire des algorithmes itératifs de haut niveau avec une sémantique séquentielle. La parallélisation est implicite, et repose sur le paradigme de programmation par tâches. L'API est construite autour du concept de *Sequence* qui représente une suite d'opérations. Une *Sequence* est une structure persistante, encapsulant les différentes opérations à effectuer sous forme de tâches. A l'exécution, les *Sequence*

peuvent être rejouées plusieurs fois, ce qui permet de les encapsuler à l'intérieur de boucles itératives. Les tâches sont construites à partir du partitionnement du graphe d'adjacence représentant la matrice donnée en entrée du solveur, *i.e.* le système linéaire à résoudre. Lors du déclenchement de l'exécution de la *Sequence*, les tâches ainsi que leurs dépendances sont alors traduites puis envoyées à un support executif. Jusqu'à présent, notre API supporte les supports d'exécution suivants : OpenMP 4.0, OmpSs, X-Kaapi et HARTS. Lors d'une étude préliminaire, nous avons fait un comparatif d'exécutions en utilisant différents supports exécutifs.
Constatant que la plupart des machines multi-cœurs repose sur une architecture mémoire de type NUMA, nous avons cherché un moyen de prendre en compte la localité des données lors de l'exécution de nos méthodes. Nous avons alors mis en place un système de tâches d'initialisation dans HARTS pour pouvoir bénéficier d'une politique de type *First-touch* et ainsi distribuer nos données sur l'ensemble des bancs NUMA disponibles. En contrôlant l'ordonnancement des tâches et grâce à l'utilisation d'une queue distribuée sur chaque thread, nous garantissons que les tâches d'initialisation possédant un même numéro de partition sont exécutées par la même unité de calcul. Par conséquent, des données différentes seront distribuées de la même manière selon le partitionnement du graphe de la matrice d'entrée. Nous remplissons la queue appropriée lors de l'instanciation des tâches. Lors de l'équilibrage de charge, garanti par un algorithme de type vol de travail et grâce à la connaissance de la distribution des données, un thread inactif va pouvoir voler une tâche à un autre thread, tout en essayant de minimiser la distance NUMA qui le sépare de sa victime. Grâce à cette stratégie, nous arrivons à diminuer les accès sur des données distantes. Afin d'avoir une meilleure analyse de nos performances, nous avons mis en place un système de monitoring à l'intérieur du support exécutif HARTS. Grâce à des métriques collectées tout au long de l'exécution, nous pouvons alors analyser les données une fois l'exécution parallèle terminée. Par ce biais, nous pouvons tracer aussi bien des courbes d'activités de threads, diagrammes de Gantt, ou des analyses sur le chemin critique. Pour finir, nous analysons les performances que nous obtenons sur des méthodes itératives implémentées au dessus de notre API sur les machines multi-cœurs.

En l'état, notre API ne nous permet pas de pouvoir tirer parti efficacement des processeurs many-cœurs car elle ne tient pas en compte d'optimisations spécifiques qui sont requises pour ce type d'architecture. Le propos du **quatrième chapitre** est donc de proposer des extensions à l'API afin de pouvoir gérer un autre type d'architecture que celle que nous avons étudié par avant, tout en gardant la portabilité des performances à notre approche. Pour ce faire, nous concentrons nos travaux sur les processeurs many-cœurs Knights Landing d'Intel. Ceux-ci proposent une architecture particulière, basée notamment sur un réseau d'interconnexion des cœurs sous forme de maillage 2D. Grâce à une simplification des cœurs de calcul, le processeur peut alors posséder jusqu'à 64 unités de calculs. De plus, il embarque une mémoire à bande passante élevée appelée *MCDRAM* qui a pour but de réduire le coût des communications mémoire. De là, il en découle deux challenges de programmation que sont la gestion de la vectorisation des codes de calculs et une gestion efficace de la MCDRAM. Par le biais de notre API et les solveurs précédemments développés au-dessus de celle-ci, nous avons pu montrer que les

performances de nos méthodes continuent d'augmenter malgré l'augmentation du nombre de coeurs de calculs. A travers différents préconditionneurs, nous avons pu établir une évaluation de performances montrant que nous arrivons à obtenir de bonnes performances parallèles pour nos méthodes itératives, sur une architecture basée sur le processeur many-cœurs Knights Landing. Pour arriver à ce niveau de performances, nous avons dû étendre notre API pour permettre de prendre en considération la vectorisation des méthodes, mais également de pouvoir tirer avantage de la mémoire MCDRAM embarquée sur le processeur KNL. Ces extensions ont été faites dans le respect de la portabilité des performances de l'API sur différentes architectures. Ces choix d'optimisations pour une architecture spécifique se font à la compilation, sans altérer la manière dont on écrit nos méthodes itératives. Par exemple, nous montrons dans ces travaux comment nous permettons la vectorisation d'une opération de type produit matrice creuse par un vecteur (SpMV) grâce au changement de structure de la matrice en entrée. A l'arrivée, cette opération nous permet alors d'accroître les performances de nos solveurs écrits au-dessus de notre API.

Bien que nous pouvons proposer dorénavant une parallélisation des méthodes numériques employées en simulation réservoir, aussi bien sur architectures multi-cœurs que many-cœurs, les méthodes développées jusqu'ici s'avèrent inefficaces sur le plan numérique face à des système mal conditionnés. Pour palier à ce problème, dans le **cinquième chapitre** nous proposons alors l'implémentation d'un préconditionneur de type décomposition de domaine multi-niveaux. Après une rapide présentation des méthodes de type *Additive Schwarz*, nous rappelons que l'ajout d'un opérateur grossier tel que GenEO permet de palier à la robustesse et l'extensibilité des préconditionneurs de type décomposition de domaine sur des cas provenant de simulations numériques réelles. Nous retrouvons ces résultats à travers les mesures expérimentales que nous avons effectuées sur notre implémentation. L'ajout d'un second niveau nous amène alors à la résolution d'un système linéaire de petite taille. Cette résolution, dite grossière, nous permet alors de synchroniser l'ensemble des sous-domaines pour favoriser la communication entre eux et accélérer la convergence de la méthode. Nous proposons dans un second temps une parallélisation par tâches de cette méthode à l'aide de notre API. Expérimentalement, nous montrons qu'il existe un compromis à faire entre deux tailles de systèmes à résoudre à chaque itération. En effet, en fonction du nombre de sous-domaines la taille des systèmes locaux tend à diminuer, alors qu'à l'inverse la taille du système grossier augmente. Un compromis est alors de mise pour trouver le meilleur équilibre en termes de performances, pour ne pas que chacune des opérations ne freine les performances globales. Nous évaluons par la suite notre implémentation sur des systèmes de diverses tailles sur une machine de type multi-cœurs. Notre implémentation présente de bonnes performances, indépendamment de la taille du système. Comparée à la référence AMG, notre application se montre un peu moins efficace, mais bien plus que le préconditonneur ILU(0). Sur architecture many-cœurs, les performances ne sont pas en reste et présente une bonne efficacité de calculs. De même que sur la précédente machine, notre implémentation présente un niveau de performance un peu en retrait par rapport au code de référence AMG. Nous expliquons cette différence par une analyse de la répartition des temps de calculs à l'exécution qui montre que des tâches restent très consommatrices en temps de calculs et nécessitent encore d'être vectorisées à court terme. En définitif, les

performances actuelles du préconditionneur DDML sont en deçà de ce qui pourrait être obtenu. Une fois ce problème d'optimisation résolu nous pouvons espérer de meilleures performances sur ces machines car nous avons mesuré que le facteur d'accélération de notre code est plus important par rapport à la version séquentielle du même code. Toute amélioration séquentielle sera bénéfique.

Enfin, le **sixième et dernier chapitre** permet de conclure ce mémoire de thèse. Nous rappelons alors l'ensemble des travaux effectués ainsi que les différents résultats obtenus. Nous profitons également de ce chapitre pour donner des perspectives pour la suite de ces travaux.

Chapter 1

Introduction

Contents

1.1 Context and motivation

Numerical simulations in reservoir engineering lead up to the resolution of Partial Differential Equations (PDE), which are then discretized with a finite volume scheme. We solve it thanks to a Newton solver, in which the system is linearized at each step. The given linear system is generally large and sparse, and has to be solved via a linear solver method. This part is time consuming because of the large amount of computations, and it represents up to 80% of the total simulation time.

Direct resolution methods are accurate and robust algorithms to solve linear systems. However, it is unsuitable for large system because of the memory requirements and the lack of parallelism. Iterative methods are thus favored, despite the fact that they give approximate solutions and depend on the matrix structure. Preconditioners are applied on the system in order to increase their convergence rates. The *Algebraic Multi-Grid* (AMG) preconditioner presents good numerical qualities for the problems encountered in reservoir simulations. However, an efficient parallel implementation of a such method requires a significant programming effort.

On the other hand, *Domain Decomposition methods* are well-suited for modern parallel architectures. Its design relies on the "divide and conquer" strategy. Hence, its algorithmic structure naturally fits on parallel computers and enables a parallelization at a fine granularity. This later advantage is particularly interesting with the increase in number of computational units per socket. However, these methods are not robust enough for our case study. Recently, the problem was fixed with the advent of 2-level decomposition domain methods and the GenEO coarse operator [Spillane et al., 2014]. This

method already benefits from an efficient parallel implementation on a distributed memory computer [Jolivet et al., 2013].

Parallel computer architectures based on a shared memory model actually rely on the Non Uniform Memory Access (NUMA) design. Such systems are characterized by a segmented physical memory and an increase in the number of cores per chip. In addition of that, memory hierarchy that rules memory transfers impacts the application performances. In such designs, data latency problems arises from memory bus contention. Nowadays, the number of cores per socket massively increases because of the simplification of the core design. Programming efficiently these many-core processors is an additional challenges. Performances enhancement often involves vector processing and a good memory management.

A part of programming complexity moves from a low-level programming to a higher layer. Runtime systems are tools that helps data management and work scheduling among computational resources. The use of emergent parallel programming models is already adopted in dense linear algebra. However, complex challenges still require much more investment in the parallel design of the methods used in the reservoir simulation field.

1.2 Objectives and contributions

The main objective of this work is to develop a Domain Decomposition preconditioner which efficiently exploits the capacities of multi-core and many-core systems. It can be viewed as the combination of two derived objectives. The first one is to design a robust preconditioner that can be used in oil reservoir simulation. The second is to provide a parallel implementation of numerical methods on emerging architectures. We thus need to prepare beforehand main concepts, both numerical and software, on which our work will rely on. This preconditioner has to be integrated in the MCGSolver library [Anciaux-Sedrakian et al., 2014], developed at IFPEN, which aims to provide efficient parallel linear solvers for numerical methods' developers.

The first contribution of this work is to propose an efficient parallel implementation of iterative methods. It relies on a data-flow programming model, and is packed within an abstract linear algebra programming interface (i.e. API). By this way, users do not see parallel implementation and can focus on the algorithm. This API has been developed over several runtime systems among OpenMP, OmpSs, X-Kaapi and HARTS. The interface was build around the iterative pattern of the methods while the tasks were created according to the adjacency graph of the matrix. This work has been presented in [Roussel, 2016].

Within the X-Kaapi's OpenMP interface, we also highlight the importance of data locality scheduling policy on NUMA architecture. It enables to collect information on data at run time to place tasks among computational units. We expose the case of a sparse matrix vector product on a large scale NUMA machine in [Virouleau et al., 2016b].
Thanks to HARTS, we implement a data locality aware scheduling policy based on graph partitioning information provided by the API. By this way, tasks are pushed in a distributed

queue according to partition information. It enables to distribute data among memory nodes thanks to *first-touch* NUMA policy and allocating tasks of the API. At run time, when a thread is idle, a work stealing scheduler selects a victim according to the initial data distribution.

Moreover, we enhance our understanding of application behavior with the implementation of performances analysis tools inside HARTS. This model allows users to collect information at run time time, which enable to build application analysis after the execution. For example, it enables to build a time line from events records (e.g. beginning and end of tasks) gathered in a Gantt chart. At a lower scale, measurements can be collected at a task level to know its granularity or which resource performs it. After an execution, we also can analyze critical path which can give us an hint to a parallelism default.

Afterward, we adapt our numerical methods on many-core processors. On Intel Knights Landing processor, we highlight the importance of code vectorization and effective usage of high bandwidth memory. However, the performances of the API have to be portable whatever the architecture in use. We thus need to extend the API while keeping the same semantic. In order to enhance performances by code vectorization we enable the developers to write kernels with multiple implementations and a multiple structures management. Moreover, we extend the API with memory management unit to enhance applications with high bandwidth memory when it is available (depending on the architecture). We illustrate the performances we obtained with this architecture with various numerical methods we implemented.

Eventually, we develop a 2-level Domain Decomposition preconditioner with the GenEO coarse operator. This work was achieved within the abstract linear algebra API. We validate the robustness of such a preconditioner on benchmarks coming from oil reservoir simulations. By this way, we evaluate its numerical performance that is close to other widespread methods as AMG. Thanks to the work previously described, we develop both multi-core and many-core efficient parallel implementations. A preliminary study of this preconditioner on multi-core architecture was published in [Roussel et al., 2016].

1.3 Outline

In the second chapter, we introduce the challenges that leads to high performance computing in the sparse linear solver context. From algorithmic to the parallel implementation, there are many issues to overcome. This chapter aims to review them from algorithmic to computer architectures, while discussing on the implementation between them.

In the third chapter, we describe a way to write efficient parallel numerical methods on multi-core architectures. We present the framework we set up, and the required optimization to maximize performances. This chapter is devoted to the evaluation of parallel performances of iterative methods.

Many-core architecture is one alternative to multi-core processors in order to increase computing performances. However, these architectures introduce new programming

challenges. We review these challenges in the fourth chapter, and propose solutions for sparse linear algebra methods.

We propose an efficient parallelization of a 2-level domain decomposition preconditioner in the fifth chapter. We gather the various points we raised up in the previous chapters. By this way, we develop a preconditioner which is robust and efficient on both multi-core and many-core systems.

We conclude this thesis with the last chapter, on which we review our contribution and the results we obtained all along this work.

Chapter 2

Background

Contents

In basin modelization or oil reservoir simulation, multi-phase flow in porous media models lead to solve complex *Partial Differential Equations* (PDEs) systems. These PDEs are discretized with a cell-centered Finite Volume scheme in space and an Euler implicit method in time, leading to a nonlinear system which is solved with an iterative Newton's method [Kelley, 2003]. At each Newton step, the system is linearized then solved using a linear solver.

The equation reads $Ax = b$, where x is the vector of unknowns, A the input matrix and b the right-hand side vector. The resolution of such systems is the most expensive part among the simulator's workflow. It may represent up to 80% of the simulation time. Moreover, the generated linear system is ill-conditioned, large and sparse. As it is a bottleneck for application performance, parallel computing is thus a necessity to enhance

simulator's performances. Hence, methods must be adapted, or sometimes re-designed, for efficient parallel computations. In addition of that, parallel computers become more complex and programming them efficiently is challenging.

This chapter aims to review the challenges that parallel linear algebra algorithms have to overcome at several levels of complexity.

At the top level, we have linear solver algorithmic which aims to find a feasible solution in a reasonable time. Different classes of methods exist, which are designed from various classes of problems. These methods are reviewed in the first section.

At the lowest level, there is the computer hardware on which we eventually address parallel numerical methods. Parallel architectures are complex, and their understanding is a first step to face challenges we encounter. In the second section, we thus review widespread parallel computer designs in the high performance computing field.

Between these two levels, programming sparse linear algebra methods on parallel architectures remains challenging. A way to address the work flow is given by a parallel programming model to abstract the underlying architecture. Then, runtime systems tools implement it to manage the computations between the processing units. We review programming challenges in the third section.

Eventually, we discuss on the HPC trends in parallel sparse linear algebra field.

2.1 Sparse linear algebra

2.1.1 Linear Solvers

There are two ways to solve linear systems. The most intuitive one is direct method to obtain an exact solution of the system. Another way is to use an iterative method which refines an initial guess within a loop, until it reaches convergence to an approximate solution. We here aim to review both methods.

2.1.1.1 Direct solver

A method which allows the computation of the unknowns vector x in a finite number of operations is named a direct method for solving the the linear system $Ax = b$.

One of the most famous direct method consist to process a Gaussian elimination method. It reduces any linear system to a triangular one. It is then trivial to obtain the exact solution x.
Other methods [Ciarlet, 1998] rely on the factorization of the matrix A as a product of two other matrices such that $A = BC$. The solution of the system will be replace by the solution of two easily invertible systems as the matrices B and C are triangular or orthogonal. Common factorization methods are LU or QR decomposition.

Direct methods provide a robust and accurate way to solve linear systems. Robustness

property come from the fact that they do not depend on the shape of the linear system. However, such methods are greedy in memory. Moreover, the computation of the exact solution is time consuming and highly depends on the shape of the system to solve. Hence, these methods are not used to solve large linear systems.

2.1.1.2 Iterative Methods

Given an initial guess, an iterative method will refine the approximate solution until convergence is reached. This method modifies one or a few components of the solution at each step, and stops when the convergence criterion is satisfied under a certain desired precision ε. It can be theoretically proved that these methods converge to the exact solution.

At the k-th iteration, error estimation is given by the formula:

$$e_k = x_k - x \tag{2.1}$$

The iterative method converges if and only if e_k converges to 0. As x is unknown we cannot compute it. It is pretty much easier to compute the residual vector at the k-th step, which is given by :

$$r_k = b - Ax_k \tag{2.2}$$

and then convergence criterion is evaluated to satisfy $\|b - Ax_k\| \leq \varepsilon$. In practice, we prefer to refer to a relative error to be sure that the criterion is not misleading, and it is given by:

$$\frac{\|b - Ax_k\|}{\|b - Ax_0\|} \leq \varepsilon \tag{2.3}$$

Iterative methods are less accurate than direct methods because of the computation of an approximate solution to the problem, but the error is controlled. As a lack of robustness, the distinction between structured and unstructured matrices may affect iterative methods.

Some of them are based of coordinates relaxation. These basic iterative methods are algorithms like *Jacobi*, *Gauss-Seidel* or *Successive Over Relaxation* (SOR) methods. But these techniques are rarely good alternatives for solving large and sparse systems.

Another way is the use of methods based on projection techniques. In this case, the most popular ones are Krylov subspace methods. Common algorithms used for unstructured systems are the *Generalized Minimum Residual Method* (GMRES) [Saad and Schultz, 1986] and the *Biconjugate Gradient Stabilized* (BiCGStab) [van der Vorst, 1992] algorithms (illustrated in Algorithm 3.1). In our work, we mainly focus on the BiCGStab algorithm because

of its efficiency on unstructured linear systems arising in oil reservoir simulations.

Algorithm 2.1: BiCGStab Algorithm

Compute $r_0 = b - Ax_0$;
r_0^* arbitrary;
$p_0 = r_0$;
$j = 0$;
do
  $\alpha_j = (r_j, r_0^*)/(Ap_j, r_0^*)$;
  $s_j = r_j - \alpha_j Ap_j$;
  $\omega_j = (As_j, s_j)/(As_j, As_j)$;
  $x_{j+1} = x_j + \alpha_j p_j + \omega_j s_j$;
  $r_{j+1} = s_j - \omega_j As_j$;
  $\beta_j = \frac{(r_{j+1}, r_0^*)}{(r_j, r_0^*)} \times \frac{\alpha_j}{\omega_j}$;
  $p_{j+1} = r_{j+1} + \beta_j(p_j - \omega_j Ap_j)$;
  $j = j + 1$;
while *! convergence*;

2.1.2 Preconditioners

2.1.2.1 Generalities

In scientific computing, there is no exact computations. Real numbers are represented in computer under a certain precision degree (representation with 32 bits for a single precision, or with 64 bits for a double precision). A slight modification of matrix entries may have a significant impact on the final computed solution. A numerical method which does not amplify errors is said to be *stable*. Because of their recursive nature, stability property for iterative methods is a way to not propagate and amplify error over iterations. Otherwise, computed solution may be far away from the expected result.

To measure how sensitive is the solution to the problem of rounding error, we now introduce the notion of *matrix conditioning*. It helps to measure the sensitivity of the solution x of the linear system $Ax = b$ to perturbations of the data A and b. The condition number $cond(A)$ quantify the conditioning or the sensitivity of the problem $Ax = b$ to perturbations in the data A or b. Condition number of a singular matrix (i.e invertible) A is given by

$$cond(A) = \|A\|.\|A^{-1}\|. \tag{2.4}$$

Even if the relative error of the data is small, the impact on the solution x can be huge if the condition number of A is large. In other words, the condition number $cond(A)$ measures the amplification of errors in A and b. A matrix A is said to be *well conditioned* if $cond(A) \simeq 1$, and conversely a matrix is said to be *ill conditioned* if $cond(A) \gg 1$.

Due to the complexity to compute the matrix A^{-1} when A is large, we generally

compute condition number by approximations which is enough to predict the amplification of errors in data.

Highly heterogeneous geological data and complex studied geometry lead to ill-conditioned linear systems solved with iterative methods. To prevent the propagation of errors, preconditioning methods transform a system to another while reducing its condition number. However, it does not change the solution of the problem. A left preconditioned system is given by

$$M^{-1}Ax = M^{-1}b, \tag{2.5}$$

in which A is an ill-conditioned matrix, and M a non-singular matrix named a *preconditioner*, or a preconditioning matrix. There is also a right preconditioned system, given by

$$AP^{-1}y = b$$

with

$$x = P^{-1}y, \tag{2.6}$$

such that P is a non singular and easily invertible matrix. An important property of a preconditioned system is that $cond(M^{-1}A) < cond(A)$. The problem is now to find a matrix M which is easily invertible so that $M-1A$ is close to the identity matrix (whose conditioning is equal to 1). Computing A^{-1} is a problem at least as difficult as to solve the linear system. So we have to find a matrix M which is close to A. Another important property of a preconditioner is that is easy to apply. Indeed, we will apply this preconditioner at each iteration of an iterative method, so it has to be not too expensive to apply it on A.

Ill-conditioned system fails to converge in a reasonable time because of the error propagation. Improving conditioning of a matrix with a preconditioner is so important to enhance linear solver's convergence rate. Indeed, it reduces number of iteration required to converge. Linear systems arising from oil reservoir simulation are generally ill-conditioned.

2.1.2.2 Polynomial

The idea of such a preconditioner is to define $M^{-1} = p(A)$, where p is a polynomial such that $cond(C^{-1}A) < cond(A)$. A good choice is to take for $p(x)$, a truncation of the expansion in power series of A^{-1}, given by $p(x) = 1 + \sum_{k\geq 1}^{d}(1-x)^k$. Although it is easy to built, this preconditioner is most of the time not efficient for complex problems arising from oil reservoir simulations.

2.1.2.3 ILU(0)

Given a factorization of a large sparse matrix A such that

$$A = LU, \tag{2.7}$$

where L is a lower triangular matrix, and U an upper triangular matrix. It is well known that usually in the factorization procedure, the matrices L and U have more non zero entries then A. These extra entries are called fill-in entries. The *Incomplete LU factorization* (**ILU**) [Varga, 1960] consists in dropping some of these elements. Incomplete factorization preconditioners consist in taking $M = \bar{L}\bar{U} \approx A$, where $\bar{L}$ and $\bar{U}$ stand for the incomplete LU factors of A. Let S be a subset of $n \times n$ positions of the original matrix generally including the main diagonal, and $\forall(i,j)$ such as $a_{i,j} \neq 0$. An incomplete LU factorization of A only allows fill-in positions which are in S, which is designated by the elements to drop at each step. S has to be specified in advance in a static way by defining a zero pattern which must exclude the main diagonal. Therefore, for any zero pattern P, such that

$$P \subset \{(i,j)|i \neq j; 1 \leq i,j \leq n\} \tag{2.8}$$

an Incomplete LU Factorization, ILU, can be computed as in Algorithm 2.

Algorithm 2.2: Incomplete LU Factorization Algorithm

```
for i = 2, ..., n do
    for k = 1, ..., i − 1 and (i, k) ∉ P do
        a_ik = a_ik / a_kk for j = k + 1, ..., n and (i, j) ∉ P do
            a_ij = a_ij − a_ik a_kj
        end
    end
end
```

For an incomplete factorization with no-fill, named ILU(0), we define the pattern P as the zero pattern of A. However, more accurate factorization can be obtained by allowing some fill-in, denoted by ILU(p) where p stands for the desired level of fill. This class of preconditioner has some difficulties to converge in a reasonably number of iterations on ill-conditioned systems.

2.1.2.4 AMG

Algebraic MultiGrid (**AMG**) [Brandt et al., 1984] method solves linear systems based on multigrid principles. It is a complex algorithm which is widely spread in simulation area because of its robustness property on large sparse and unstructured systems. The setup phase is non-negligible in the total solving time because of the determination, at each coarse grid level, of the coarse grid, interpolation and coarse grid operators. The setup time may be longer than the solving time as the solver requires only a few iterates to converge. The solving phase is composed of two complementary operations, also called the smoothing and the coarse grid correction steps. The first one attempt to reduce high-frequency error by the application of a smoother, also called relaxation method. Coarse grid correction eliminates low-frequency error. It performs a transfer of information to a coarser grid (also called restriction operator), then a coarse-grid operator is solved and at last the solution is send back to the fine grid (also called the interpolation). The major improvements of such

a method is that it operates on smaller problems, and the computational cost is therefore smaller. Moreover, it requires no information on the problem geometry. The number of iterations needed to converge only depends on the system size. In addition to that, coarsening and smoothing strategies may be tuned to affect convergence rates.

2.2 Parallel Computers

Moore's law [Moore, 1965] states that the number of transistors on a chip is multiplied by 2 every 18 months. This law shows its limits because of the downsizing of the hardware components is not enough to reach high performances regarding actual challenges of scientific computing. Since the rise of multi-core processors, parallel computing is an increasing need to take advantage of these architectures. Nowadays, processor's frequency tends to converge because of the heat dissipation induced by hardware limitations.

Since the multiplication of processors on a chip, the Admahl's law defines the theoretical speed-up obtained on an application. An application can be split in a part that is parallelizable, and another which is not. Let T_p be the execution time on p processors, and S the time to compute the serial part. The sequential time to compute the parallel part, denoted by B, is so deduced by $B = T_1 - S$. On p processors, a lower bound is so given by $T_p = S + (B/p)$. However, some overheads occur and factors such as data latency or memory bandwidth make the theoretical time slightly different.

The aim of this section is to describe parallelism from hardware to programmer's point of view. In the first sub-section, we describe a way for parallel architectures and parallelism classification. We then describe each significant hardware device which composes an actual architecture and its hierarchy. In the third section, we detail the programming models that help the developer to efficiently take advantage of parallel computers.

2.2.1 Generalities

The rise of multi-processors systems increases the complexity of programming. Several types of parallel computers exist, and there is a need to find a classification to describe any machine according to hardware features. There are several ways to classify parallel computers, and we first present the most used in parallel architectures: the Flynn taxonomy [Flynn, 1972]. This classification is based on instruction and data streams. In a second time, we present memory types encountered on parallel machines.

2.2.1.1 Classification

On a sequential computer, executing model is the same as the Von Neumann machine [Von Neumann, 1945]. There is a unique instructions stream in which one instruction is process per processor cycle on a single data item. In parallel computing area, an application can be viewed as one or several instructions streams acting on one or several data streams.

The classification proposed by Flynn [Flynn, 1972] is based on it. It deals with both instruction and data streams, and enables to store any computer in one of the four following categories:

— **Single Instruction stream, Single Data stream (*SISD*) –** It refers to the uniprocessor category. Developers see it as a sequential computer.
— **Single Instruction stream, Multiple Data stream (*SIMD*) –** The same instruction is processed by multiple processors on multiple data in parallel. Such computers own a single control unit. Exploiting this type of architectures needs sufficient amount of data. Vector architectures and data parallel architectures are placed in this computer class.
— **Multiple Instruction stream, Single Data stream (*MISD*) –** Until now, there is no computer of this class.
— **Multiple Instruction stream, Multiple Data stream (*MIMD*) –** Each processor executes its own instruction stream on its own data stream. Programming such architecture is more flexible, and imposes a significant grain size to efficiently exploit parallelism.

For example, a common architecture in clusters are nodes composed of two processors. Such a computational node can be classified in the MIMD category. A single single processor includes several computing units (cores) which have SIMD capabilities.

There are several other models which extend the classification presented above. For example, Single Program Multiple Data (SPMD) and Multiple Program Multiple Data (MPMD) take care of a running program which can be distributed on one or several computational nodes, and process local data at the same time.

2.2.1.2 Levels of parallelism

Parallelism is a meaning to divide a problem in smaller pieces processed by some computational units. There are several ways and different levels to express parallelism. In this part, we gradually review each level of parallelism.

Instruction Level Parallelism (*ILP*) in which instructions can be grouped or reorganized to be processed in parallel if dependencies between data allow it, without changing the result. Instruction level parallelism can be exploited by two major ways. The first one is at hardware level with a dedicated logic on a chip. The second one is at software level with the use of compiler.

Data Level Parallelism (*DLP*) is the ability to divide a problem by splitting processed data which are related to its resolution. Data are then distributed among working threads or processes which operate on it the same instructions. Efficient data parallelism requires a large amount of data to process. In this case, the amount of data processed is known as the work's granularity.

Task Level Parallelism (*TLP*) is the ability to divide the whole work in several tasks that can be performed simultaneously on different data. Tasks may be different from each other, and the completion of all of them lead to the final result of the problem.

2.2.2 Processors Architecture

The processor is the hardware unit responsible to perform computations. Nowadays, computing processors handle several execution units to enable parallel computing. Classical multi-processor chips are equipped with few but powerful execution units, also named as cores. On the other hand, compared to classical processors, computing accelerators generally handle much more cores which are less powerful. However, massive parallelism induced by these accelerators may increase performances.

Computational accelerators may take several forms, and we aim in this subsection to review some of the most popular ones. The first one, Graphical Processing Units (*GPUs*), refers to a widespread component in computer systems which are now used for achieving high performance computing. The second one is the Many Integrated Cores (MIC) introduced by Intel to provide huge number of simplified cores to make computations.

2.2.2.1 General purpose processors

Current processors in a computer refer to multi-processors chips or multi-core processors. A multi-core processor is composed of several processors, also called cores, on the same die. It means that these chips contain several independent processing units, which collaborate in parallel to achieve better performances than a single core processor. The set of instructions that a processor can process is called the Instruction Set Architecture (*ISA*). One of the most popular ISA is the x86 instruction set. Instructions represent data transfers, arithmetic, conditional, branches and logical operations. A register is the fastest memory location and is used to store temporary variables. There are some extensions to classical ISAs to SIMD instructions like Streaming SIMD Extension (*SSE*) instructions and also Advanced Vector Extension (*AVX*) support.

Current generation of multi-processors chips contains a cache memory hierarchy of three levels of cache, from the closest to the farthest. Cache memory avoids a processor to retrieve data each time from main memory, and so reduces latency by this means. At the top level, there is the L1 cache which is private to a core. This is the fastest and the closest cache level to the core but the size is highly limited. The L2 cache level is slightly bigger, and local to core but data access has higher latency. In some cases, L2 may also be shared (e.g. AMD Buldozer has shared L2 cache between 2 cores). The L3 cache level is the largest cache level, and is generally shared between all the cores. Nowadays, this is the last cache level to store data on current generation of processors. If after all, the data is not store in one of the three levels of cache, the core request data from main memory and the latency is therefore higher.

2.2.2.2 Graphic Processing Units (GPU)

GPUs are no longer exclusively used for graphical applications. General Purpose processing on GPU (*GPGPU*) refers to the fact that now GPUs are used as computational accelerators for general purpose computations. They provide a massively parallel architectures composed of a huge number of cores which have a simplified design. The most used architectures for GPGPU are provided by the Nvidia company, and are named CUDA-aware architectures. In Kepler architecture, cores are grouped in streaming multiprocessors (SMX), where each one shares a L2 cache memory. Each SMX owns its own memory hierarchy which is shared by all the cores of this SMX. For instance, each one shares a L1 cache between all the cores, and a constant cache memory. GPU computing introduces Single Instruction Multiple Threads (*SIMT*) execution model. Threads are grouped in a set called warp (generally equals to 32 threads). Each SMX also handles a quad warp scheduler, which allows to process the same instruction concurrently on four threads of different warps. Such data transfers between host (CPU side) and device (GPU device) are processed through the PCI Express bus. If an application processes too many data transfers, it could be a bottleneck for application performances. Designing a good scheduler that reduce data transfers and balance the work load is challenging. More details can be found in the Kepler architecture white paper [Nvidia Corporation, 2012].

2.2.2.3 Intel® Xeon Phi Coprocessor

The accelerator proposed by Intel® is based on a Many Integrated Core (MIC) architecture, named Intel® Xeon Phi [Jeffers and Reinders, 2013]. The coprocessor is known under the name Knights Corner (KNC) product. The board is composed of more or less 60 x86 cores, linked by a bi-directional ring interconnect. Each core owns its own memory hierarchy with L1 cache which is closer to the core, and L2 cache which is still closer but in direct link with the interconnect. There are 8 memory controllers on the ring interconnect to dispatch data around the ring from external memory device. The key feature of such an architecture is that each core contains a vector processing unit to process vector instructions. Difficulties come from the code optimization to take care of vector processing, hyper-threading and multi-level parallelism. The hardware architecture of the KNC coprocessor board is illustrated in Fig. 2.1.

2.2.3 Memory Systems

It is also important to distinguish which type of memory is used on parallel computers. The reduction of both latency and bus contention is a determinant factor in memory systems. The latency defines the time from the request to a data access to the answer. Bus contention refers to the fact that more than one device attempt to simultaneously access to memory.

Nowadays, multi-processors chips handle several layers of memory hierarchy to reduce latency. There are memory blocks, called caches, very closed to processing units to reduce time to data retrieval. On the other hand, there is a need to handle memory

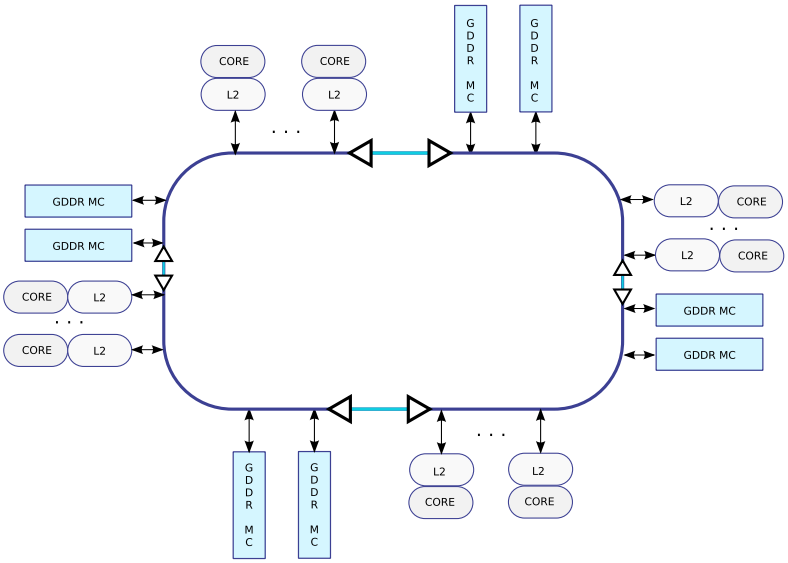


Figure 2.1 – *Intel®Xeon Phi architecture*

coherency between all the layers of the memory hierarchy. There are specific protocols to ensure the validity of the data between all the layers of the cache hierarchy of all the cores. Such a protocol is named a cache coherence protocol. An architecture which is not cache coherent is not easy to program.

We review here the two main categories of memory systems space. A shared memory system relies on a centralized memory shared between the processors. In the other hand, distributed memory is located at different locations such that each processor has not a direct hardware access to this memory bank.

2.2.3.1 Shared Memory

In shared memory systems, a set of processors is linked to a memory system through an interconnection network, and a processor can access to each memory location without any software support. The interconnect can either connect all the processors to the main memory, or each processor has its own local memory and can access to other memory locations. So the main categories of shared memory systems are the Uniform Memory Access (*UMA*) and Non-Uniform Memory Access (*NUMA*) architectures.

Uniform Memory Access (UMA)

UMA architectures are characterized by the fact that each processor has the same path length to any memory address. In this case, bus contention may be a bottleneck for application when several cores have to access the centralized memory at the same time. An example of an UMA architecture with 8 threads is given in Fig.2.2. Systems which use UMA architecture are Symmetric Multi-Processors (SMP) and the first multi-core processors.

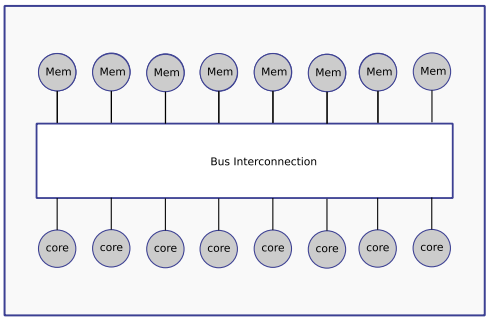


Figure 2.2 – *UMA system with bus interconnection*

Non Uniform Memory Access (NUMA)

In a NUMA architecture, data access time depends on its location in memory. Each processor or group of processors has its own local memory. In this kind of architecture, memory locality plays a significant role. Remote memory accesses increase the latency. Local memory accesses not only favor a lighter latency overhead but also reduce bus contention on the interconnect and other memory controllers. Nowadays, NUMA systems are encountered in multi-processors systems. These systems may own several NUMA nodes, where there is a local memory and a group of processors per NUMA node. Fig.2.3 illustrates a computation node based on a NUMA design which is composed of two NUMA nodes, where each contains a 4 cores processor and a dedicated memory. In such machines, we can thus define the notion of distance. Indeed, the distance between processors and memory banks can vary. We measure the distance in hops to join a memory bank to a processing unit, which depends on the network. Hence, a core and a memory bank located in the same NUMA node have a smaller distance than if they are on two distinct nodes. Memory latency depends on the distance distance between a core and the memory banks the threads attempt to reach.

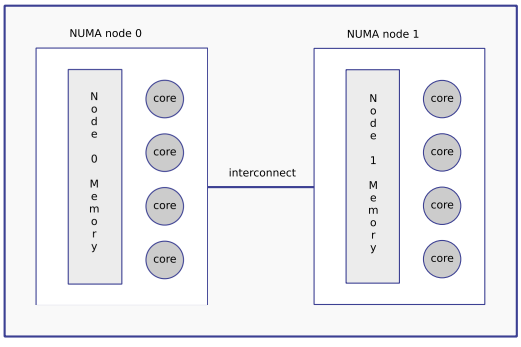


Figure 2.3 – *System with 2 NUMA nodes with 8 processors*

Most of NUMA systems are cache coherent, that means that the coherence of shared data inside processor caches is ensure by a specific protocol for both processors. Such systems are called *CC-NUMA*.

2.2.3.2 Distributed Memory

In such kind of system, each processor or group of processors is paired with its own private memory paired with an interconnection network. Distinct processors at various locations may explicitly communicate by sending messages to exchange data (see section 2.3.1.2). Well known examples are clusters, composed of multiple nodes (such as a system composed of one or two NUMA nodes) with their own local and private memory. Nodes are then linked by interconnection network, like ethernet, infiniband, ... This class of memory is out of the scope of this work.

2.2.4 Topology discovery

Most of current computer architecture designs rely on a hierarchical topology. Taking care of underlying hardware topology is a key point to reach high performance on parallel computing. Nowadays, clusters are composed of several nodes which contains several sockets in where we can found multi-core processors and accelerators. There it exists specific memory hierarchy such as cache memory which can be shared or not between cores. The way of binding cores is a determinant factor, specifically when it involves communication, synchronization and sharing between them. It means that two related computing tasks have to be placed on neighbor cores to optimize communication and/or synchronization. Moreover, the democratization of heterogeneous architectures highlights that locality importance grows and it is now applied to computational accelerators or network interfaces [Goglin, 2014]. Thus, affinities between computing tasks and hardware resources play a key role.

The *Hardware Locality* software **Hwloc** [Broquedis et al., 2010] aims to gather topology information of the underlying hardware. The generated hierarchical tree contains at each node some specific objects (among Machine, Node, Socket, Cache, Core, ...) and various attributes such as cache type and size or core identifier. The depth of a node represents the depth in the machine topology, and an edge between two nodes represents a physical link between these components. It thus offers the possibility to discover I/O devices to have a more detailed view of the architecture, such as some kind of accelerators. By browsing the tree, developers can use information to adapt the behaviour of application at run-time to the underlying hardware specificities. Fig.2.4 illustrates a topology scheme of a dual-socket node obtained with the *Hwloc* software.

2.3 Parallel Programming Models and Runtime Systems

A programming model is an abstraction of the underlying architecture. It is generally presented as a bridge between applications and hardware [Asanović et al., 2006]. It offers a way for users to express parallel algorithms and match applications with the underlying architecture. Programming models are still in constant evolution according to architecture's changes which provide new programming challenges.

Figure 2.4 – *Hwloc illustration on a dual-socket machine*

The implementation of a parallel programming model may be provided by a library invoked from a sequential language, an extension to an existing language or a new programming language. Runtime systems are tools which implement a parallel programming model while discharging users from the parallel execution.

2.3.1 Parallel Programming Models

In this subsection, we review three parallel programming models designed for multi-core architectures. We first remind fork-join programming model. We then present message passing model. We eventually present the data flow model.

2.3.1.1 Fork-Join Model

Fork-Join model is a model composed of two step. The work is initially divided in smaller and independent tasks that can be performed in parallel: it is the fork phase. All tasks are then reduced (i.e. the join phase), which generates a synchronization point at task exit point. This programming model imposes that the granularity of tasks have to be similar to not generate excessive inactivity time.

The POSIX threads library, pthread [Nichols et al., 1996], is a widespread library that allows to easily implement the former programming model. The programming standard for multi-threaded application, OpenMP [OpenMP Architecture Review Board, 2008], also implements it via "`parallel`" and "`parallel for`" construct directives.

2.3.1.2 Message Passing Model

In message passing model, each processor owns a private memory which is not reachable from other processor. The only way for processors to exchange data is to use explicit communication by sending messages. The two main functions are *send* and *receive*. There is two main type of communication: *synchronous* and *asynchronous*. Synchronous communications involve that two processors have to be synchronized to exchange data. In this situation, when a processor has to wait for a message in its queue before resuming its execution. At the opposite, asynchronous communications does not need that two processors are synchronized to communicate.

Most of programming tools which implement this programming model use a multi processes execution. The Message Passing Interface, MPI [Forum, 1994], is the most popular to implement this programming model. It provides both C and Fortran interfaces to implement message passing model.

2.3.1.3 Data flow Model

The data flow model described in [Johnston et al., 2004] enables a program to process work according to dependencies between instructions. Initial model only considers instructions, but its evolution upgrades to also consider macro instructions which now correspond to a function call. We consider here that this model is not limited to data flow machines [Davis, 1978]. There are two states of a data in this model: ready or not. A data become ready when all its previous contributions are computed. Following this model, we can associate access modes to a data (read, write or read/write). A Direct Acyclic Graph (DAG) of the execution can be deduced by data flow computations. The graph is processed following the dependency rules. Each node contains a function call, which corresponds to a piece of work entirely written with a sequential language. A task is characterized by work's granularity, predecessor tasks, and successors.

2.3.2 Runtime Systems

To fully exploit parallel computer performances, runtime systems are key elements. A runtime system is a tool which implements a programming model while balancing the workload on the available computational units. On the software stack, runtime systems are useful from users' point of view to describe an application without taking care of workload management. By this way, no knowledge about hardware is required to efficiently program complex parallel architectures.

We only focus on task-based runtime systems with on-line scheduling policy (i.e. dynamic). Fundamentally, such a runtime system handles a task pool [Korch and Rauber, 2004]. It can be shared or not between a set of "workers" (*i.e.* threads in our study case) which are responsible of performing computations. Runtime systems then define how a queue is accessed by a thread, and the associated functions to push or pop a task. In the case of

a centralized pool, all the threads share the access to it. In this case, concurrent accesses may degrade performances if work granularity is too small. On the opposite, the queue is distributed and can be operated by only one thread. The compromise between these two strategies is to design a distributed pool where each queue is operated by a group of threads. The scheduler conducts the execution. It assigns the work to do during execution time by pushing tasks into the queue. The existing task-based runtime systems differ from the way they manage tasks and scheduling policy they employ. We aim here to review some of the most famous task-based runtime systems designed for shared memory machines.

2.3.2.1 Cilk

Cilk [Blumofe et al., 1995] is a multi-threaded runtime system which extends the C programming language. It relies on the Fork-Join programming model with a notion of tasks. Users are in charge to explicitly expose parallelism, while tasks scheduling is abstracted by the runtime. In the Cilk-5 formalism [Frigo et al., 1998], a Cilk program can be expressed within three keywords: *cilk*, *spawn*, *sync*. The keyword *cilk* defines a Cilk procedure. The keyword *spawn* is used to execute a Cilk procedure in parallel. The scheduler is then responsible to assign threads the spawned tasks. A local barrier is generated when *sync* is encountered, which enforces a task to wait for the completion of its children.

The Cilk scheduler relies on a work-stealing policy [Blumofe and Leiserson, 1994]. A thread is represented as a worker that performs tasks from its own deque. A worker operates on its local pool with push and pop functions at the tail of its deque. When a thread runs out of work, it becomes a thief, and tries to steal work from the head of another worker's deque named the victim.

2.3.2.2 OpenMP

OpenMP [OpenMP Architecture Review Board, 2013] is the well-known standard in shared memory parallel programming. Code annotations with compiler directives avoid to rewrite the whole application from scratch. When entering the first parallel region a typical OpenMP runtime instantiates a thread pool which is reused for following parallel regions. Threads in pool wait for work on a semi active synchronization barrier. Each thread actively (thread in run state) spins waiting for work up to a maximum spin value and if a thread reach this maximum value without work the runtime puts the thread in an operating system wait queue (thread in sleep state) using mutex or futex like mechanisms. This approach allows low latency parallel job start without monopolizing processor time.

Now a more advanced tasking feature is available since the version 3.0 of the specification [OpenMP Architecture Review Board, 2008], and enables to explicitly define a task [Ayguade et al., 2009]. Tasks are created in the order defined by the (partial) order of `#pragma task` OpenMP keyword in the executing program. This means that in order to have concurrent tasks these have to be created inside a parallel section. Access modes of

Listing 2.1 – *OpenMP – Axpy task*

```
1
2     void omp_axpy(Value const& a,
3                   Vector const& x, Vector& y,
4                   const int size)
5     {
6       #pragma omp task firstprivate(a, x, y, size) \
7                        depend(in: x[:size], a[:1] ) \
8                        depend(out: y[:size] ) untied
9       {
10        for (int i = 0; i < size; ++i)
11          y[i] += a * x[i];
12      }
13    }
```

tasks' data for dataflow directed computation are now available since the version 4.0 of the standard [OpenMP Architecture Review Board, 2013], at the same time of work offloading on computational accelerators. Access modes allow creating task dependencies thus ready tasks pool is filled with tasks for which dependencies are satisfied. Work scheduling depend on the hidden underlying implementations offered by compiler which implement OpenMP (Gnu GCC and Intel ICC).

The standard offers a flat memory view of complex NUMA hardware. Recent extensions allows to specify mapping of threads among cores of a parallel architecture. An example of a Axpy task is thus illustrated in Listing 2.1. It simply shows the construction of a task via the `pragma omp task` directive, and the dependencies via the `depend` keyword.

2.3.2.3 OmpSs

OmpSs is a task based programming environment which covers both heterogeneous and homogeneous architectures used nowadays. Its target is the programming of multi-GPU, many-core or multi-core architectures and offers asynchronous parallelism in the execution of the tasks. OmpSs is build on top of Mercurium compiler and Nanos++ runtime system. OmpSs syntax offers a flexible way to express the given tasks to be executed on target architectures via the target construct. Then, Nanos++ is able to schedule and run these tasks, taking care of the required data transfers and synchronizations on the accurate resources. Resources can be used to bind a task to a certain node,socket, core or GPU.

Tasks in OmpSs are annotated with data directionality clauses that specify the use of data, and how it will be used (read, write or read&write). Dependencies are then deduced at run-time from user supplied annotations of data accesses which are translated into a format that can be exploited by Nanos++.

Nanos++ proposes several scheduling policies which defines how ready tasks are

Listing 2.2 – *OmpSs – Axpy task*

```
1  void ompss_axpy(Value const& a,
2                  Vector const& x, Vector& y,
3                  const int size)
4  {
5    #pragma omp task no_copy_deps label(Axpy)
6                     in( x, a )
7                     out( y )
8    {
9      for (int i = 0; i < size; ++i)
10       y[i] += a * x[i];
11   }
12 }
```

executed. We can list bf, dbf, socket, affinity, affinity-smartpriority or versioning. While the first two scheduling implements a mechanisms to execute one after the other tasks with a single or a queue per thread. The socket scheduling refers to a work stealing scheduler which take care of NUMA affinity. There are also mechanisms to manage work priority within scheduler. The most suitable scheduling policy can depend on the application and architecture used.

OmpSs uses a thread-pool execution model. There is a master thread that starts the execution and several other threads that cooperate executing the work it creates from work sharing or task constructs [Ayguadé et al., 2010].

Nanos++ provides also a support for instrumentation which allows to obtain traces, for performance analysis, and graph of dependencies, to better understand the application characteristics.

OmpSs relies on the same principles as OpenMP, and does not provide execution hints for task placement. The task construct is quite similar to the OpenMP formalism, as it is illustrated in Listing 2.2. However, it is designed to run on heterogeneous architectures with possibly several memory address spaces. In the case of we do not need to copy data in several memory address space, the `no_copy_deps` statement is used.

2.3.2.4 X-Kaapi

X-Kaapi [Gautier et al., 2007] is a C library for supporting task-based programming model with data flow dependencies for heterogeneous architecture [Ferreira Lima et al., 2015]. The data flow model enables non-blocking task creation: the caller creates the task and proceeds with the program execution. Parallelism is explicit while the detection of synchronizations is implicit: dependencies between tasks and memory transfers are automatically managed by the runtime thanks to user annotation on data usage (annotation if data is read, written or written concurrently). Target machines are homogeneous or heterogeneous

such as machines with several GPUs [Gautier et al., 2013]. X-Kaapi offers several APIs (C, C++, Fortran) and it provides a OpenMP runtime with binary compatibility with GCC's libGOMP runtime or it can be targeted by the K'STAR OpenMP compiler [1].

X-Kaapi runtime is structured around the notion of worker: it is the internal representation of kernel thread. It executes the code of the tasks and it takes local scheduling decisions. On hierarchical machine, several level of the hierarchy hold a queue of ready tasks. At lower level the core manages tasks into stack for fast task's creations. Depending of the scheduling algorithm, the data flow graph is entirely built or only ready tasks are discovered when inactive thread try to steal work. At a first glance, the schedulers in X-Kaapi are list-based algorithms. They are composed of three operations that act on queues of tasks: pop, push and steal. Specific schedulers have been developed to better exploit NUMA machines: A locality-aware work stealing for multi-GPUs [Gautier et al., 2013] or a Distributed Affinity Dual Approximation [Bleuse et al., 2014] that tries to better compute a compromise between balancing the workload and reducing communication. Recently, NUMA-aware scheduling policy was implemented in [Virouleau et al., 2016a]. At run time, a task can be placed by the scheduler on a specific processor according to task affinity. This affinity can be chosen according to data placement, or by a user-defined numa node placement. It was implemented through the OpenMP task descriptor and evaluated with several benchmarks in [Virouleau et al., 2016c].

2.3.2.5 StarPU

StarPU [Augonnet et al., 2011] runtime systems is a library designed for heterogeneous architectures while relying on a unified executing model. A parallel application can be performed on multiple architectures without rewriting it from scratch. Its programming model relies on task programming paradigm with data dependencies. At run time, a StarPU application may address computations on various computing hardware technologies: multicore processors or accelerators. This feature thus requires some explicit data movements. Hence, StarPU provides a high level library [Augonnet and Namyst, 2008] to manipulate data between disjoint address spaces. The library helps to minimize data transfers, to ensure data coherency and to overcome the limited amount of memory available on accelerators.

Within the use of *codelet* structure a task may have various implementations, each one targeting a specific technology. A codelet handles input and output data, and their corresponding access modes (Read, Write, Read/Write). Various callbacks are also part of this structure. At run time, the scheduler thus has the choice to perform one among the task version on the appropriate computing resources.

When a task is ready (i.e. all its dependencies are fulfilled), it is directly pushed in one of the queues. StarPU scheduling policies are based on list schedulers. The runtime system offers a collection schedulers among them work stealing and Heterogeneous Earliest Finish Time (HEFT) [Topcuouglu et al., 2002]. However, StarPU offers developers the capacity to write portable scheduling policies within a high level interface. Programmers can also

1. See http://kstar.gforge.inria.fr

specify scheduling hints in codelet structure. By this way, developer may declare prioritized tasks, or also try to balance the task graph with weighted tasks.

2.3.2.6 HARTS

The runtime system HARTS [Gratien, 2013] relies on abstract concepts to describe and manage the layer between application and hardware that help to distribute and manage work between computation units and the associated data movements between the different memories. The library is based on a hardware model, a task model, a data model and an executing model. Its hardware model is based on hardware discovery with the *Hwloc* library [Broquedis et al., 2010]. It enables to describe various heterogeneous architectures with different kinds of compute units, different levels of memories and different kind of connections between each units. The Data model enables to encapsulate the data manipulated by task objects in *DataHandler* objects, managed and organized in a centralized data manager. The data model provides also tools to split data with partitioner and partial views of each sub part of the data.

The task model is responsible of tasks objects creation and management within the runtime system. A task may have multiple representations for the target devices on which they may be executed. At its creation, a task is directly pushed in a task pool. HARTS provides a centralized task pool that is shared between all the threads. Tasks are also organized in a DAG that can be saved to be replayed as tasks are persistent (i.e. not deleted after its execution).

Eventually, the executing model models the way that the tasks are performed on the target machine. A thread pool attempts to operates on the task queue following the scheduling policy. At execution time, roots of different DAGs are given to the scheduler to dispatch tasks and balance them between computation units. Two main schedulers are available in HARTS. The first one operates on centralized task pool and only fairly distribute the tasks between the threads. The second one is a work-stealing scheduler which is used with the centralized task queue. This model enables a thread to steal task from another one following a user-defined policy.

2.4 Discussion on HPC Trend

2.4.1 Hardware

The main problem on heterogeneous architectures is the fact that computation speed is faster than memory communication. So data transfers between host and accelerator(s) can represent a bottleneck for application's performances. An emergent solution is to give a direct access from accelerator to main memory or to offer an access to a high-speed memory bank. A possible way is to integrate GPU in processor chips. It has been experimented with AMD (with AMD Fusion product family) and Intel (on some Intel Core products) on x86 processors for laptops. Recently, Nvidia launched the Tegra X1 processor which

Listing 2.3 – *HARTS – Axpy task*

```
void compute(DataArgsType& args)
{
  typedef vector<double> VectorType;

  // Get arguments encapsulated in "args"
  VectorType & X = * (args.get("X")->get<VectorType>()) ;
  VectorType & Y = * (args.get("Y")->get<VectorType>()) ;
  double& a = * (args.get("ALPHA")->get<double>()) ;

  // Prepare vectors to be split
  SplitConstVectorType sX(X,this->m_partition) ;
  SplitVectorType sY(Y,this->m_partition) ;

  // get a data view of "X"
  typename SplitConstVectorType::ConstViewType x;
  x = sX.view(this->m_partition_id) ;

  // get a data view of "Y"
  typename SplitVectorType::ViewType y;
  y = sY.view(this->m_partition_id) ;

  // Compute
  for(int i = 0; i < size; ++i)
    y[i] += a * x[i]
}
```

offers a similar way but the chip is composed of 8 ARM 64-bits cores, and 256 GPU's cores based on Maxwell architecture. Both of these architectures are currently designed for portable devices such as laptops, tablets or smartphones. However, it become an interesting way to explore because of the processors reduce energy consumption while keeping good performances.

Another way to reduce data retrieval cost is to put memory banks closer to computational units. This approach was first experimented by AMD and Hynx which build a stacked memory on the processor die. By this way, the memory bandwidth increases but the size of this memory is limited. Recently, a stacked memory called "*High Bandwidth Memory* (HBM)" is integrated onto the Nvidia's Pascal GPUs but also onto Intel Knights Landing processors.

2.4.2 Runtime Systems

In [Olivier and Prins, 2010] the authors compare several OpenMP-3.0 runtimes on unbalanced task graphs against standard library or language such as Cilk and IntelTBB. Considering several implementations of OpenMP 3.0 (Intel, Sun, GCC and the Nanos), the now outdated results show important variations in the performances for the different software. Moreover, the considered benchmark is compute-bound application with few impact due to hierarchical NUMA architecture.

Closer to our problem, in [Kurzak et al., 2010], the authors compare several parallel programming environments for dense linear operations on multi-core processors. The work includes comparison between depend task environment (SMPss, precursor of OMPSs) and independent task based language such as Cilk. Conclusions promote the expression of parallel computation using dependent tasks due to finer possible synchronisation between task among different iteration steps.

Recently, [Lacoste et al., 2014] reports experiments on using two general purpose runtimes, StarPU [Augonnet et al., 2011] and PARSEC [Bosilca et al., 2012] to replace the internal highly specialized scheduler of the PaStiX [Henon et al., 2002] parallel sparse direct solver [2]. While the authors conclude that general purpose runtimes can the replace specific internal dynamic scheduler of PaStiX, they also reports differences in the performances due to different design decisions.

2.4.3 Parallel Linear algebra

Many work has been done before to address sparse linear algebra application on complex parallel architectures. However, each one differ from the way they address computations on a given parallel architecture and the manner data are managed.

Efficient preconditioners are the keystone of many researches. According to the evolution of HPC systems, finding a highly parallelizable method while solving quickly the

2. http://pastix.gforge.inria.fr

problem is challenging. Some developers integrate parallel implementation of numerical methods in linear algebra frameworks. By this way, users can take advantage of robust methods regardless of the parallel implementation and its understanding. Each framework differs from the architectures they target, or the programming model they use.

Liszt [DeVito et al., 2011] is a domain specific language that enables to write parallel PDE solvers on various architectures. It provides the essential mesh components to users, such as vertex, edge, face and cell. The framework split the work according to two distinct execution strategies, graph partitioning or graph coloring. Then, they can address parallelism to hardware by different ways such as pthreads for shared-memory architectures, MPI for distributed systems and CUDA to target Nvidia's GPUs. Within the results they obtained, they prove that they are competitive to classical standards implementation while hiding parallel features.

Developers of PETSc [Balay et al., 1997] provide software package to solve large sparse linear systems. It offers many data structures to write efficient and portable code on parallel machines. It enables to solve sparse linear systems coming from numerical simulations. It also provide efficient preconditioners to speed up the convergence rates of the solvers. The model relies on objected oriented concepts, and provides efficient parallelism using distributed memory model with MPI library.

PSBLAS [Filippone and Colajanni, 2000] is a library designed for sparse linear algebra computations on parallel machines. The interface is based on the Fortran 90 language. It abstracts both sparse iterative linear solvers and data structures used by users. The library aims to maximize load balancing while reducing data communications in a distributed memory model. They implement it on top of the MPI library.

Hypre [Falgout et al., 2006] is a widespread library which provides efficient parallel preconditioners and solvers for sparse linear systems. It relies on a object-oriented conception, but is written in C language. The library mainly focus on multigrid preconditioners. It takes advantage of large distributed memory cluster with the MPI library. However, the developers currently implement multi-threaded execution with the OpenMP standard for shared-memory architectures.

2.5 Conclusion

From the complexity induced by both sparse linear solvers and parallel computers, we aim to design parallel linear algebra methods in a convenient way. Our purpose is to provide efficient parallel methods independently from the underlying architectures. However, this objective often impose developers to maintain several versions of the code, each targeting a specific architecture. This solution does not offer code portability. Hence, we propose to design a portable sparse linear algebra framework which supports both multi-core and many-core architectures. It relies on a task programming model, in which the operations are transparently translated to graphs of tasks which are then executed by a given runtime system. Within this strategy, we can write portable application with only

one semantic and take advantage of different computer architectures.

Chapter 3

Design and Evaluation of an Abstract Linear Algebra API on top of a runtime system

Contents

The development of complex algorithms well adapted for parallel computers is not so easy. Parallel architectures are complex with deep memory hierarchy, limited latency and huge number of cores, which make effective programming increasingly challenging.

In the numerical simulation field, most of the developers are not computer scientists. They need some abstractions of the parallelism to only focus on the numerical robustness of the methods they implement.

To overcome the programming challenge on complex architectures, we design an abstract linear algebra API which targets parallel sparse iterative methods. Such methods are widespread in linear solver software and numerical simulations, especially in oil reservoir

simulators. This framework enables implicit parallelism at end-users level. The users does not deal with parallelism and hardware specific implementations. The API provides users a collection of functions which generates parallelism automatically. By hiding parallelism, we can address the execution flow to various runtime systems without rewriting the application from scratch. We detail the design and the various implementations of the API in the first section of this chapter.

The API we develop targets several objectives. It has to keep its utilization simple for users who are not from computer science area. It also enables the portability of performances on high level algorithms for various computer architectures. Thanks to the separation of concern pattern, management of parallelism is hidden to the user application. A runtime system maps the parallelism described by API implementations to the underlying hardware. Several implementation of the API have been implemented, each one targeting a specific runtime system.

The API is first designed for multi-core architectures. Shared memory and hierarchical NUMA memory designs are widespread in such systems. The memory is segmented in several locations. Latency highly depends on the distance between a core and the location of the data in memory. Thanks to our API, we develop a multi-level strategy to increase data locality consideration at scheduling time. Our approach is detailed in the second section.

In order to enhance our understanding of the parallel applications behavior, we extended HARTS with performance counters and monitoring tools support. Data are collected all along the various parallel regions, and gathered when the application ends. By this way, we can analyze and then optimize performances. We give an overview of what it is possible to extract with our tools in the third section.

All these developments have been experimented on real hardware. We report the performance evaluations of some parallel preconditioners written thanks to the API. Indeed, we implement two well-known preconditioners: polynomial and ILU(0). After describing them, we make comparison on a multi-core machine. We analyze the results obtained from monitoring tools. This is the purpose of the fourth and last section.

3.1 An Abstract Sparse Linear Algebra API

Experts on numerical simulation require to enhance performances of their application thanks to parallel computing on complex architectures. However, these architectures are mostly programmed by scientific experts of other application domains. Hence, we develop a sparse linear algebra API which provides to users a collection of functionalities to write efficient parallel methods. The framework aims to hide the parallel programming complexity. One key feature is its sequential semantic presented in the next section.

Until the execution, the API includes several steps. To take advantage of loop pattern from iterative methods, a separation of concern between initialization and execution is operated. At the initialization phase, the API is responsible to prepare the work which

has to be performed later (*i.e.* during parallel region). The next section is devoted to tasks decomposition.
Once the work is prepared, the API is thus responsible to send the entire work to the distinct execution units. Tasks are eventually performed by a specific runtime system that could be chosen by user. The challenge is to keep a unique semantic for various runtime systems with different features. The third subsection is devoted to the understanding of the bridge that links API to a given runtime system.

3.1.1 A sequential semantic to describe numerical methods

The API aims to design efficient parallel algorithms while hiding programming complexity to users. Linear solver algorithms, in particular iterative Krylov methods, can be viewed as the execution of some successive sequences of linear algebra operations which are repeated several times up to reach a user-defined convergence criteria. These operations are mostly level 1 or level 2 BLAS (*Basic Linear Algebra Subprograms*) vector operations, some sparse matrix-vector products (SpMV) or some specific sparse matrix preconditioning operations. For example, the BiCGStab [van der Vorst, 1992] algorithm is sketched in Algorithm 3.1.

Algorithm 3.1: BiCGStab Algorithm

```
1  Matrix A;
2  Vector b, pp, p, r, v;
3  Preconditioner P;
4  Scalar a;
5  do
6  |  pp = inv(P).p;
7  |  v = A.pp ;
8  |  r += v;
9  |  a = dot(p,r);
10 |  if(a==0) break;
11 |  ...;
12 while (|r| < tol * |b|);
```

We implement these steps with our abstract algebraic API aiming to parallelize in a transparent way linear algebra algorithms. The available list of kernels allows developers to write a large collection of parallel linear solvers. The whole interface and its kernels are illustrated in Listing 3.1. Each function takes some matrix, vector or scalar number as arguments. By convention, we store results in the last argument of the function. The API allows developers to describe compositions of kernels with a sequential semantic. The result of a parallel execution remains the same as if performed sequentially, independently of the degree of parallelism used. Thanks to the API, the user keeps a high level syntax and could express the BiCGStab algorithm by the program of Listing 3.2.

The key feature is the capability to describe iterations around the concept of sequence. A sequence represents a list of operations that can be replayed several times inside a loop.

Listing 3.1 – *Linear Algebra API*

```
class AlgebraKernel
{
   // allocates 'size' bits and returns it in 'v' ;
   virtual void allocate(size_t size,
                         Vector& v) ;

   // Out: y = op() ;
   virtual void assign(LambdaT op, Vector& y) ;

   // Out: y = x ;
   virtual void copy(Vector const& x,
                     Vector& y) ;

   // Out: y = y * a ;
   virtual void scal(Value const &a,
                     Vector& y) ;

   // Out: y += a * x ;
   virtual void axpy(Value const &a,
                     Vector const& x,
                     Vector& y) ;

   // Out: a = ( x . y ) ;
   virtual void dot(Vector const& x,
                    Vector const& y,
                    Value& a) ;

   // Out: y = A * x ;
   virtual void mult(Matrix const& A,
                     Vector const& x,
                     Vector& y) ;

   // Out: preconditioner 'P' applied to 'x' and stored in 'y' ;
   virtual void exec(Precond const& P,
                     Vector const& x,
                     Vector& y) ;

   // Break if value is null ;
   virtual void assertNull(double const& value) ;

   // Performs the Sequence objects referred by "id" ;
   virtual void process(SequenceType id) ;
} ;
```

Listing 3.2 – *BiCGStab sequence*

```
1  Matrix A;
2  PartitionerType partition(A, nb_partitions);
3  AlgebraKernelType alg(partition);
4  Vector p,pp,r,v;
5  Value alpha;
6  Value tol = 1E-6;
7  int max_iter = 1000;
8  Iteration iter(tol, max_iter);
9  SequenceType seq = alg.newSequence();
10 alg.exec(precond,p,pp,seq) ;
11 alg.mult(A,pp,v,seq);
12 alg.axpy(1.,r,v,seq);
13 alg.dot(p,r,alpha,seq);
14 alg.assertNull(alpha,seq);
15 while(!iter.stop())
16 {
17   alg.process(seq);
18 }
```

Parallel iterative methods are built from sequence object concept. The API's kernels' calls take a sequence identifier in arguments to feed it with work to do. When it is completed, a `Sequence` execution can be invoked via the `process` function. Thereby, a parallel region begins when the work is transferred from the API to a given runtime system through the use of a `Sequence` object. The parallel region is triggered when the `process()` function of `Sequence` object is called. In the example, a parallel execution begins at line 17, and ends at line 18. `Sequence` objects are encapsulated within a loop and are performed several times until the convergence criteria is reached (line 15).

Convergence criteria is user-defined to stop the solver under a reasonable degree of precision for the desired solution. As we can see in the Listing 3.2, the Sequence (referred by `seq`) processing is trapped within a `while` loop. The list of operations that are contained in `seq` are replayed until the condition described in `iter` is fulfilled. In this case, the solver will stop if the solution reaches a precision of 10^{-6} or the method iterates more than 10^3 times (lines 6-8).

3.1.2 Internal creation and representation of parallelism

The purpose of the API is to generate parallelism, and then the parallel execution is offloaded to runtime systems. The API is not restricted to one parallel programming model, however our work is devoted to dataflow programming paradigm as described in Section 2.3.1.3. In our model, the developers of API's functions are thus responsible to split the work in several tasks until their execution. Tasks decomposition, and thus the parallelism, come from data partitioning.

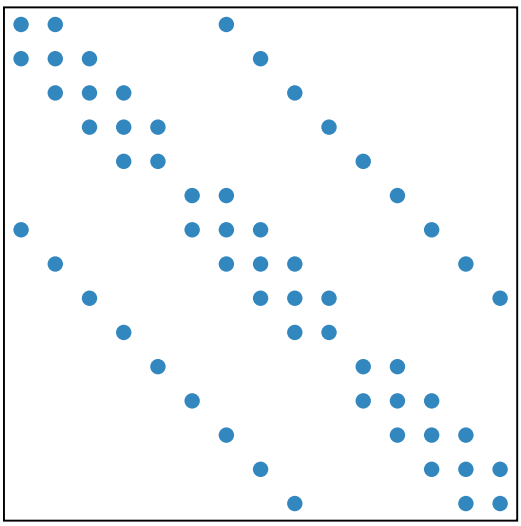


Figure 3.1 – *Sparsity pattern*

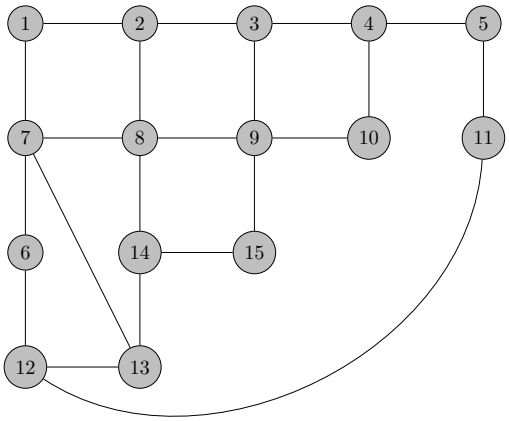


Figure 3.2 – *Graph representation*

3.1.2.1 Data Partitioning using Graph partitioning

A `Partitioner` object computes a data distribution thanks to a given number of partitions and load balancing consideration. The number of partitions may be chosen according to the hardware architecture we use or according to the granularity needed by the method. By default, if not specified, we chose to fix the number of partitions to the number of cores we use.
The partitioning is built from the partition of the coefficient matrix which represents the linear system to solve. Thanks to the graph partition techniques, we thus split it in several sub-parts. Except the input matrix, most of the data we operate are vectors. The vector structures are also split thanks to the matrix's partitioning. Dependencies between tasks are then built according to the dependencies between the nodes of the graph.

Let consider a matrix A and its coefficients $(a_{i,j})$ with $0 \leq i < n$ and $0 \leq j < n$. Let $G_A = (V, S)$ be the undirected graph generated from adjacency matrix of A where $V = (v_i)$ is the set of vertices representing the N_{rows} rows of A, and $S = (s_{i,j})$ the set of edges connecting vertices v_i and v_j such that the matrix entry $a_{i,j} \neq 0$.

We define a partition P of G_A in N_{part} subparts as the set $(V^k)_{0 \leq k < p}$ of subsets of V where $V^k \subset V$ and $V^{k_1} \cap V^{k_2} = \emptyset$ if $k_1 \neq k_2$ and $V = \cup (V^k)_{0 \leq k < p}$.

For each V^k, we define two sets. Vi^k is the set of interior vertices $v_i \in V^k$ such as if $a_{i,j} \neq 0$, $v_i \in V_k$ and $v_j \notin V^{k_1}$, where $k_1 \neq k$, then $j \in Vi_k$. Vb^k is the set of boundary vertices $v_i \in V^k$ such as there is at least one $k_2 \neq k$ and one $v_j \in V^{k_2}$ such as $a_{i,j} \neq 0$.
In other words, the set Vi_k represents elements that are local to the given partition. On the opposite, Vb_k represents the set of remote elements that require data exchange with another partition.

Let's consider a vector x and its components (x_i) with $0 \leq i < n$. The vector

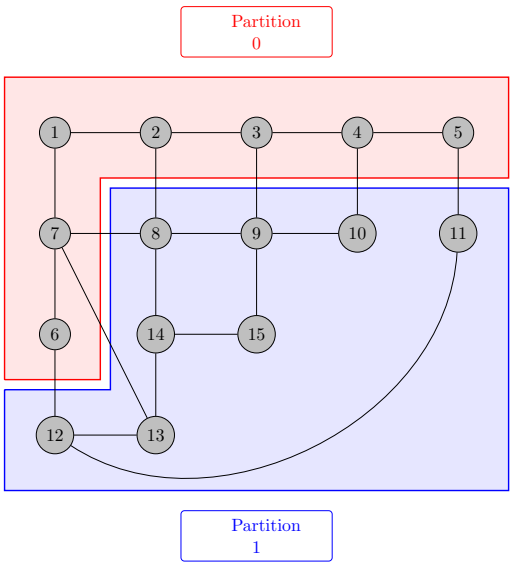


Figure 3.3 – *Row partitioning*

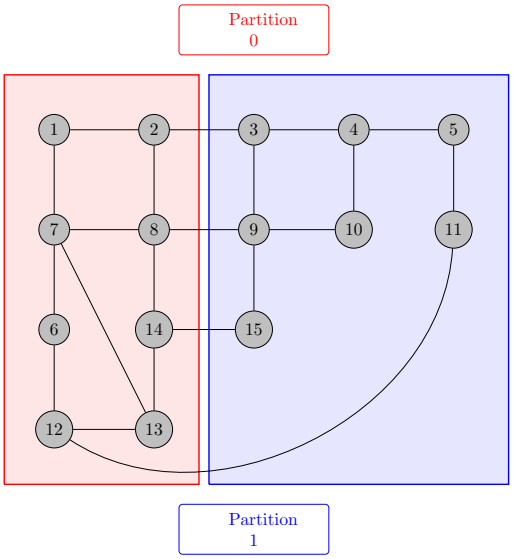


Figure 3.4 – *Custom partitioning*

components x_i are associated to the vertices v_i of G_A while the nonzero entries $a_{i,j}$ of A are associated to the edges $s_{i,j}$ of G_A.

We can take the example of a 15×15 sparse matrix A, whose nonzero elements are illustrated in the sparsity pattern in Figure 3.1. Its graph representation is then deduced from adjacency matrix and illustrated in Figure 3.2.

Constraint graph partition problem is a NP-complete problem [Garey and Johnson, 1990]. In our case of study, a graph is said to be well partitioned if:

1. it limits the number of edges running between two separate subgraphs. By this way, we intend to limit the amount of communications between distinct graph partitions.
2. it balances the number of internal edges and vertices of all the subgraphs to fairly balance the work load between partitions.

Given the graph illustrated in Figure 3.2, we can create several distinct combinations of partitions. In the case where the number of partitions is equals to two, the combinations proposed in Figure 3.3 and in Figure 3.4 satisfy the previously defined conditions.

Figure 3.3 illustrates a row partitioning of the matrix, without taking care of communications or load balancing. Each partition intends to be responsible of N_{rows}/N_{part}, where N_{part} stands for the number of desired partitions. However, this solution is rarely optimal. Instead of this classic strategy used in dense linear algebra, the graph can be partitioned under given criteria. The Figure 3.4 shows the previous graph which is partitioned in two partitions with more consideration about communications than the previous splitting. Indeed, edges that links the two subgraphs are reduced to 4, instead of 7 with row partitioning. If we also weight the graph with nonzero elements per node, we also can take care of load balancing between the subgraphs.

As the graph partition problem is complex, exact solution computation needs an inten-

sive computing effort. Thus, heuristics are used to find a reasonable solution close to the optimal one in a feasible time. Softwares and libraries such as Metis [Karypis and Kumar, 1995] or Scotch [Pellegrini and Roman, 1996] are tools that provide to developers a good candidate for graph partition problem. Those tools can be thereafter integrated to our API to partition data (i.e. matrices and vectors). Task distribution is then built thanks to the graph partition. The graph partitioning is initialized, and then encapsulated in a `AlgebraKernel` structure as illustrated in the Listing 3.2 at line 3.

In practice, row partitioning is not successful for unstructured systems. Hence, we favor a graph partitioning coming from the use of the Metis library.

3.1.2.2 Task Decomposition

A `Partitioner` object is responsible to partition and reorder the input matrix according to graph partitioning. From this partition, vectors are also partitioned. Internal nodes set of a partition k represents the vertex ids of its Vi^k set (see section 3.1.2.1 for the notation). These nodes' values correspond to a set of row numbers of the coefficient matrix A which is partitioned. From these rows, we can also reorder vectors to then partition them in the same way as matrix A. Before the execution, the right hand side vector b is reordered and partitioned. In our current tasking model, parallelism comes from data partitioning. According to a given operation, data related to a partition are encapsulated in a task descriptor provided by the API.

Let's take the example of the task decomposition of an *axpy* operation which is sketched in the Listing 3.3. In this example, an instantiation of the `Partitioner`'s structure is named `partitioner`. This structure is the same as the argument's constructor of the `AlgebraKernel` class previously illustrated in Listing 3.2. From coefficient matrix's graph partitioning, vectors are split in N_{part} parts. The `Partitioner` object computes a sub-set of entries per partition, which corresponds to the internal nodes of the graph partition. As the matrix is reordered, vectors need it too. Before starting parallel execution of the linear solver, we only need to reorder right-hand side vector b, and the operation is performed only once. When this operation is done, we only need to shift the vector to beginning of the partition's offset. It is illustrated in the listing at line 17 on which we obtain the beginning of the current partition's offset. At lines 24 and 25, we thus shift the vector to the beginning's partition position and operate on `size` elements. Vector's partitions are disjoint. By this way, tasks only operate on its own sub-set of values, without taking care of others partitions. At line 22, we encapsulate data in a task descriptor object which describe the operation performed at run time.

3.1.2.3 Dependencies between tasks

When a task is created, it is necessary to express its dependencies between previously created tasks. The tasks are stored in an instance of the current `Sequence` object.

BLAS operations compose a large part of the iterative methods' operations. They

Listing 3.3 – *Axpy decomposition*

```
1  void
2  AlgebraKernel::axpy (Value const& a, Vector const& x,
3                       Vector& y, Sequence seq)
4  {
5    size_t begin = 0, end = 0;
6    // bring back the number of partitions
7    // in the variable 'psize'
8    int psize = partitioner->getNbPartitions();
9
10   // Shifting to the partition range
11   // is given by the Partitioner object
12   int const* offset = partitioner->getOffset();
13
14   for (int ipart = 0; ipart < psize; ++ipart)
15   {
16     /* Range */
17     begin = offset[ipart];
18     end   = offset[ipart + 1];
19     size  = end - begin;
20
21     // y[begin:end] += a * x[begin:end]
22     seq.push_back (
23         new axpy_Descriptor(ipart, size, a,
24                                          x[begin],
25                                          y[begin],
26                                          ));
27   }
28 }
```

generally only need dependencies on the last created task which update the same partition. The last argument is the output while the others are input parameters.

However, a few of them need particular attention as they need synchronization from all the partitions. Let's take the example of the dot product. Each partition computes its own local result from its partition's vector entries. Then a reduction from all the local results computes the final result. The translation of this reduction is given by a dependencies from all previous tasks on the various partitions to a reduction task.

Some other operations need dependencies from various number of partitions. It is the case of the SpMV or specific preconditioner operations. Given a SpMV operation given by the operation $Ab = y$, where b is the vector to be multiplied by the matrix A, and y the result vector. The computation over a partition of the result vector require some external contributions of the input vector. Hence, we express dependencies from external partition of the current one, to the SpMV local operation.

3.1.3 Translation from the API to runtime system

The API we developed targets multiple runtime systems following the concept "*write once, run everywhere*". Runtime systems provide different functionality. The challenge is to keep the API's sequential semantic true whatever the runtime used. We propose various implementations of the API based on the inheritance principle offered by object oriented programming. Each implementation has to inherit from the interface given in Listing 3.1, and has to redefines functions which enable to create tasks to the targeted runtime system. In accordance with the factory pattern, the kernel provides all the function needed by the user who does not have to care about the underlying implementation of the API. To add support for another runtime system, the developer just needs to implement all the functions required by the API interface. The figure 3.5 illustrates the diagram class on which the API relies. It illustrates `AlgebraKernel`'s inheritance feature to create new API's implementation.

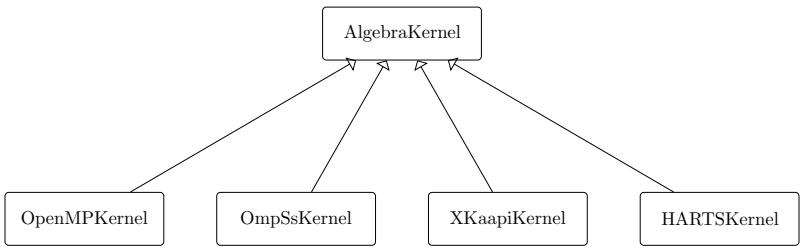


Figure 3.5 – *AlgebraKernel diagram class*

Next sections focus on the management of iterative computations and on dataflow expressions.

3.1.3.1 From the API to the runtime system

The API is a bridge that links a given algorithm to a runtime system which is then responsible of the parallel execution.

The main objective here is to keep the same semantic for all the runtime systems we target. However, the API is based on iterative pattern employed in sparse linear algebra methods. The iterative pattern is thus captured within `Sequence` objects, replayed several times. Most of the runtime systems create tasks on the fly inside a parallel region. Tasks are then performed when their dependencies are fulfilled, and eventually deleted. Among the selected runtime systems, only HARTS provides support for making tasks persistent over iterations.

With OpenMP [OpenMP Architecture Review Board, 2013], OmpSs [Ayguadé et al., 2010] and X-Kaapi [Ferreira Lima et al., 2015] runtime systems tasks are executed once. Even if graph is complex into one iteration, it has to be built at each iteration. The API's developer has to save the required information to build on the fly the DAG in the `Sequence` object. In this case, we store the data with their access modes and a function pointer in a **task descriptor** object. When a sequence execution is invoked, the API's tasks' descriptors will be translated into to tasks of the runtime system. API's tasks descriptors are stored in a FIFO (*First In First Out*) queue to respect sequential semantic. One queue represents the work to do in a sequence. Tasks descriptors are not deleted after their utilization because they could be reused over iterations to rebuild the DAG later. The sequence identifier refers to the queue to operate. When starting the execution of a sequence, tasks' descriptors are thus activated one after another, following the insertion order from the latest to the newest descriptor. The activation of such a structure means that we translate a task's descriptor to a runtime system's task in a FIFO order: we thus instantiate a new task in the runtime system tool. Tasks' descriptor are not removed from the queue to ensure persistence. A sequence's queue is deleted only when the sequence is deleted from the API.

In HARTS, the tasks are not deleted after their execution: tasks are said to be persistent. HARTS provides high level features which simplify the design of the API implementation. At the API level, developer handles a collection of HARTS' tasks belonging to the current sequences. Hence, there is no need to impose intermediate structures between the API and HARTS. Tasks are not deleted after their completion, and can be replayed several times. At the `Sequence` object level, developer needs to save the roots of the DAG he enriches at each API's function call. In this case, a `Sequence` is more than a list of tasks to perform: it already represents the DAG to be performed later. When the processing of the `Sequence` object is triggered, the roots of the `Sequence`'s DAG are send to HARTS.

3.1.3.2 Declaration of dependencies

Most of the runtime systems we selected rely on the data flow programming model, but there are some differences between each others to express task's dependencies. In the

OpenMP task construct, the `depend` clause describes dependencies and their access modes (read, write, read/write). We thus point to the piece of vector or matrix the task needs for computations. Dependencies are then used by the runtime system to compute data flow dependencies between tasks. The DAG is thus computed at execution time. X-Kaapi and OmpSs benefit from the same kind of data flow description with respect to data accesses made by tasks.

However, HARTS does not fully support this data-flow computations. It is managed by the API developer at tasks' creation. The HARTS's task object implements a `addChild()` method, which enables to explicitly build the dependencies between two tasks. When a task is created, we need to search the last task using the same data as described in section 3.1.2.3. When all tasks are created, the DAG is completed and ready to be sent to HARTS.

Let's take the example of the dot product task decomposition using HARTS in Listing 3.4. In the first loop at line 14, we thus operate the task decomposition of the dot product and save the result in a temporary array. In this loop, we also save pointers to the *dot HARTS tasks* (line 29). By this way we are able to add dependencies from these tasks to the reduction one in a second loop at line 35.

The two previous models enable to express dataflow dependencies in different ways. Except for HARTS, we encounter some issues to successfully express all the dependencies between tasks we need. The dot product is one of the example we encounter to perform a true data flow directed execution. The problem comes from the reduction task which needs to synchronize all the threads before its execution. In OpenMP, it is impossible to express this kind of dependencies as it requires to know the number of partitions we will have at compile time. If it was possible, the reduction task should require input dependencies from all the sub-vectors we generate from graph partitioning. However, the number of sub-vectors is variable from one execution to another (depending on N_{part} parameter).

X-Kaapi offers a way to fix this issue with the *Concurrent Write* (CW) access mode. Thanks to atomic operations, a data can be operated by several threads at a time without making it incoherent because of data race. By this way, we do not need a reduction task for dot product operation. The `res` is declared with the CW access mode. OpenMP enables atomic operations too with the `#pragma omp atomic` construct. However, this clause do not affect the DAG construction.

However, this model only works to produce a data thanks to concurrent threads. Let's take the example of a task which needs a variable number of input data to produce a result. The *concurrent write* model is insufficient as it does not guarantee that a data is ready before computations. To overcome this difficulty, a solution was proposed by OmpSs and X-Kaapi thanks to the libKomp library. Developer provides a vector of data pointers as input dependencies to task construct. By this way, the number of dependencies can be dynamic and our issue is fixed.
This problem was encountered in SpMV operation. Indeed, the operation computes the operation $A * x = b$, where x is the vector to multiply by the matrix A, and b is the result vector. Each task operates on independent piece of vector b, however the vector x need to

Listing 3.4 – *HARTS Dot decomposition*

```
1  // res = x . y
2  void AlgebraKernel::dot (Vector const& x,
3                           Vector const& y,
4                           Value& res, Sequence seq)
5  {
6    size_t begin = 0, end = 0;
7    int psize = partitioner->getNbPartitions();
8    int const* offset = partitioner->getOffset();
9
10
11   Value* tmp = (Value*) malloc(sizeof(Value) * psize);
12   Task** last_inserted = (Task**)
13                          malloc(sizeof(Task*) * psize);
14
15   for (int ipart = 0; ipart < psize; ++ipart)
16   {
17     /* Range */
18     begin = offset[ipart];
19     end   = offset[ipart + 1];
20     size  = end - begin;
21
22     // tmp[ipart] = x[begin:end] . y[begin:end]
23     seq.push_back (
24         new dot_Descriptor(ipart, size,
25                                       x[begin],
26                                       y[begin],
27                                       tmp
28                                         ));
29     // Save dot task pointers
30     last_inserted[i] = seq.end();
31   }
32   // pushing the reduction task, res = sum(tmp[i]);
33   seq.push_back(
34       new reduction_Descriptor(psize, tmp, res) );
35
36   for (int ipart = 0; ipart < psize; ++ipart)
37   {
38    // Add dependencies from Dot tasks to Reduction
39    last_inserted[i]->addChild(seq.end());
40   }
41 }
```

be global. In fact, a task will only read x values related to the interior domain but also on values belonging to the neighborhood of the partition. We thus need extra dependencies to express it. Until now, we fix this issue on OpenMP-like models – which does not support dynamic dependency declaration – with the addition of a global synchronization (*i.e.* `taskwait` construct) before performing a SpMV task. It is not the best solution because it may break the parallelism and impact performances. However, it is the only way we have to respect tasks dependencies.

3.1.4 DAG execution

At the end of these steps, the threads instantiated by the runtime system will be responsible of the execution of a data-flow directed DAG. This graph will be built during the execution, according to the specified data dependencies. Figure 3.6 illustrates the DAG related to the sequence of the BiCGStab algorithm, first illustrated in Listing 3.2. The graph represents a task decomposition of the sequence in four partitions. This DAG is built at `Sequence` initialization with HARTS, and at execution time for the other runtime systems. Ready tasks are submitted to runtime system which is then responsible to its execution. Runtime system's scheduler dispatch tasks between processing units while taking care of load balancing.

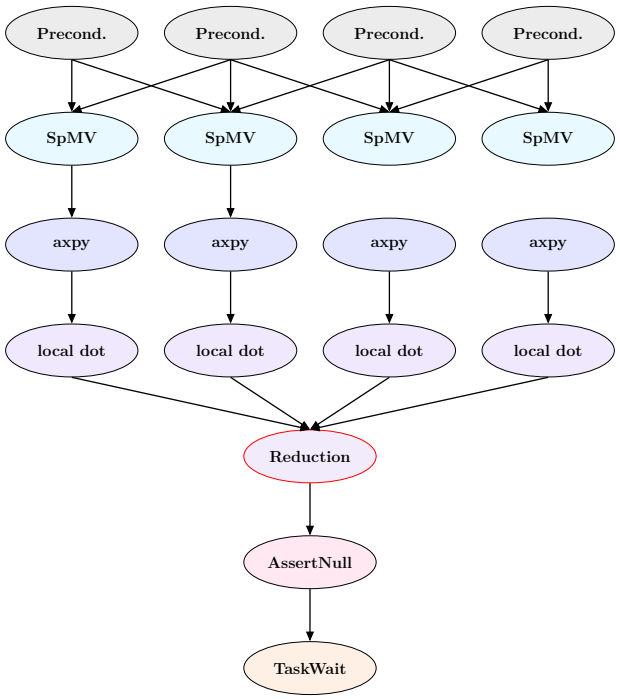


Figure 3.6 – *Direct Acyclic Graph of BiCGStab sequence (see Listing 3.2)*

3.1.5 Preliminary evaluations

In this section, we present a comparative study of the API with various runtime systems. We benchmark the preconditioned BiCGStab algorithm presented on a collection M_R of matrices extracted from real petroleum reservoir simulations. We also pick up matrices from a collection M_{Lp} of matrices coming from a Finite Volume discretization of a 2D Laplace problem on a unit cube with regular meshes with different sizes. For each system, we denote $N = N_{Rows}$ the number of rows, N_{nz} the number of nonzero entries of the matrix.

We first study the overhead due to the use of each runtime system regarding the equivalent sequential hand written code. By this way, we can evaluate the impact on the overall performances from the number of iterations and the number of generated tasks. We then evaluate the efficiency of the parallelization of the solver algorithm regarding the runtime system used to implement the API and the size N of the linear system.

Our experiments are run on a dual socket machine, linked by a Quick Path Interconnect (QPI). Each one is composed of an octo-core Sandy Bridge processor clocked at 2.60GHz with 32Gb of memory.

We evaluate the runtime systems of OpenMP 4.0 implemented in the Gnu GCC 4.9.0 compiler, of X-Kaapi release version 3.1.0 rc10, and of OmpSs with Nanos++ version 0.10a and Mercurium version 1.99.9.

In the presented results, we denote T_{seq} the sequential time to compute a sequence of operations without any runtime system, and T_p refers to the time to compute the same sequence on p cores.

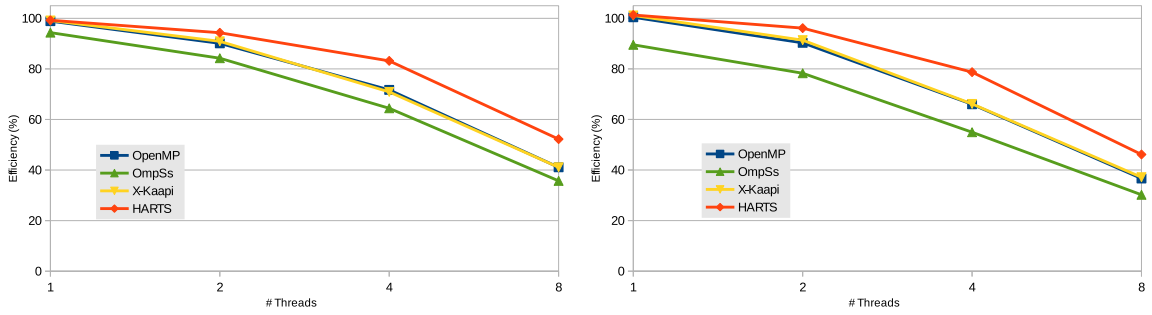
3.1.5.1 Instantiation and management of tasks

To evaluate the runtime system task management costs in the implementation of our iterative algorithm, we run for each matrix of size N, the solver on a fixed computed number of iterations N_{it} with 1 thread. We preconditioned our system at each iterate with a diagonal preconditioner, which is composed of a single copy task. We compare the execution time T_1 to the execution time T_{seq} of the hand written code for the following values of N_{it}: 10, 100, 500 and 1000. We compute the ratio $R = ((T_1 - T_{seq})/T_{seq})/N_{iter}$ to measure the overhead of the task usage per iteration compared to an implementation without any runtime systems. The results are gathered in Table 3.1 where we can compare the ratio R regarding the matrices sizes, each runtime system and the values of N_{it}.

We notice that each runtime system hides additional costs. All of them, except for OmpSs, show a lower cost per iteration when N_{iter} is equal to 10, and the gap between parallel and sequential versions decreases. OmpSs has a higher cost first when number of iterations is small, but decreases over iterations. X-Kaapi offers the lowest cost for both task insertion and execution. Moreover, it successes to catch up HARTS which provides task persistence feature. Indeed, X-Kaapi enables a constant size to task, so pushing one has a

N_{iter}	OpenMP	OmpSs	X-Kaapi	HARTS
10	-0,0277	1,1022	-0,1641	-0,1258
100	-0,0036	0,1164	-0,0157	-0,0126
500	-0,0008	0,02331	-0,0032	-0,0026
1000	-0,0004	0,0117	-0,0015	-0,0013

Table 3.1 – *Cost Evaluation of tasking model (%) – N = 2 188 842*



N = 556 494 *N = 2 188 842*

Figure 3.7 – *Cost Evaluation of tasking model*

constant time. A dynamic memory management may also explain it. Task execution order changes because it is led by dataflow computation. Produced data are cached and next task may use it again. As sequential version does not follow this strategy, the cache effect does not operate. We therefore obtain better performances with runtime systems than with sequential versions.

3.1.5.2 Efficiency on a single NUMA node

The second experience consists in running a multi-threaded execution of the previous benchmark, with a fixed the number of iterations, $N_{iter} = 1000$. We first compare the sequential time T_{seq} to the parallel time T_p with $p = \{1, 2, 4, 8\}$. We then analyze the parallel efficiency on p processors, $\text{Eff}(p) = ((T_{seq}/T_p)/p)$, for each implementation. Results are gathered in Figure 3.7 which illustrates the parallel efficiency for systems of size N according to the number of threads and the runtime system used.

By analyzing the graphic, we can notice that the curves have all the same trend regarding the number of threads. The efficiency between two runtime increases with the system size (greater for $N = 2188842$ than for $N = 556494$). In particular, X-Kaapi has a similar behaviour than OpenMP for 1 and 2 threads. All these observations can be explained by the specificities of the internal task management of each runtime system.

3.1.5.3 Impact of over decomposition

Task decomposition comes from the data partitioning through graph partitioning techniques. We now attempt to evaluate the impact of the task decomposition on the working time, i.e. the sum of the duration of all the tasks denoted by W. This time does not include runtime system's task management cost.

We thus benchmark a BiCGStab algorithm on matrices of various sizes coming from the finite volume discretization of a 2D Laplace problem. We run it on a machine equipped with 2 Broadwell 14 cores processors clocked at 2.40Ghz and each is paired with 64Gb of memory. We only use one core but we vary the the granularity of tasks by modifying the number of partitions we use. In Figure 3.8, we report working time in function of the number of partitions we use for task decomposition. The execution was performed by the HARTS runtime system.

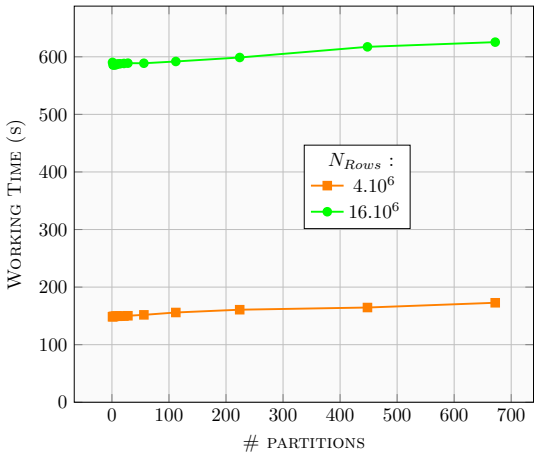


Figure 3.8 – *Work inflation over partitioning*

We can observe a slight work inflation while increasing the number of partitions for both input matrices. From 1 partition to 672, the effective working time inflation does not exceed 0.2%. It means that we add some extra operations induced by task decomposition. However, it has not a significant impact on performances as it does not degrade it. Hence, we can decompose the problem in many subparts regardless the generated overhead.

3.2 Managing data locality computations

NUMA designs are widespread in shared-memory systems. Computations' efficiency thus depends on data locality and how work is scheduled while limiting remote memory accesses. Indeed, a data located in a memory bank close to the running thread will reduce memory latency and bus contention. At run time, a thread generally picks a task from a queue without taking care of data locality. Threads' queue is fed by scheduler, which can make the decision on which task is given to a thread.

3.2.1 NUMA aware policy and runtime systems

The authors of [Drebes et al., 2014] describe scheduling techniques to control both data placement and task placement through runtime system for the OpenStream language. This strategy exploits data locality and information about data dependencies. At the execution time, it evaluates the best candidate to perform a task in order to avoid intensive data communications between NUMA nodes. Data are also allocated to optimize data accesses before task execution according to both system's topology and tasks' dependencies. This model is centered in the runtime system and does not offer data locality control to users.

In [Al-Omairy et al., 2015], the authors extend the OmpSs programming environment to enhance work stealing scheduler with a NUMA aware strategy. The scheduler provides a queue per NUMA node. According to a first-touch NUMA policy, data locality is presumed from initialization tasks. By this way, data locality is related to the thread which performs this kind of tasks. At run time, an idle thread attempts to steal a task from another NUMA node task pool. To chose this pool, the scheduler refers to the NUMA distance from the current thread to the presumed victim NUMA node.

The authors of the paper [Olivier et al., 2012b] evaluates a hierarchical task scheduling policy for multi-core architectures. One centralized task pool is created per NUMA node, and is shared between all the threads of this node. If the list becomes empty, a work stealing scheduler is activated to steal tasks from another list of another NUMA node. Contrary to the previous reference, there is notion of distance between two node lists. They ensure local computations by limiting the number of remote steals with a centralized queue per NUMA node. They assume that remote computations only occur when there is a steal between two node queues.

The authors of [Virouleau et al., 2016a] enables to take into account data locality at task schedule time. The description of a task integrates data that is read and/or written. Until now, data are just used to compute dataflow dependencies. However, data can also be used to place task in order to favor local computatons. They integrate this scheduling policy in the X-Kaapi runtime system. This is similar to version [Gautier et al., 2013] of X-Kaapi work stealing policy for multi-GPUs to CPUs. According to the OpenMP formalism, X-Kaapi then extends the task's construction clause with the `affinity` keyword [Virouleau et al., 2016b]. It enables to provide scheduling hints for data locality computations. This clause can guide the execution by scheduling a task according to a specified NUMA node, a thread or the location of data.

3.2.2 Locality-aware computations in HARTS

All the previous works emphasize the importance of data locality computations to enhance memory accesses and thus application's performances. Initially, HARTS did not support this feature as the scheduling policy places tasks regardless of the data locality. Threads operate on a shared centralized queue without any information on data locality.

In collaboration from the API and HARTS, we develop a strategy at different levels to enhance data locality computations. At the API level, we provide users a collection of allocating tasks to distribute data among NUMA nodes while hiding data distribution. It thus corresponds to the `allocate` functions from the API's interface in Listing 3.1. The runtime systems then manages work at run time considering the previous data allocation information.

In the first subsection, we detail the data distribution among memory banks across the NUMA nodes. At the runtime system level, we then propose in the second subsection a distributed queue to enforce threads to perform tasks which operates on data as close as possible. In the last subsection, we develop the scheduling policy based on work stealing scheduler which encourages data locality first.

3.2.2.1 Transparency at the initialization

The `numactl` tool enables to distribute data among memory banks in a transparent way. However, it is not the best solution as it does not provide a fine control of data locality for users.

The API provides a collection of functions for both vector and matrices via the `allocate` function in Listing 3.1. New memory pages are reserved in the virtual memory address space. Then, data are initialized in parallel to benefit from a *first-touch NUMA policy*. We ensure by this way that data are close to the threads which perform the initialization tasks, i.e. within the same NUMA node. We discharge the load balancing issue to graph partitioning techniques. As it has been seen in section 3.1.2.1, graph partitioning attempt to balance the work among the various partitions. We expect that for a parallel execution on p threads, the partitioning tools will provide p balanced partitions. Therefore, we intend to fairly distribute memory among NUMA nodes.

From this step, we intend to evaluate the gain obtained by a data distribution among memory banks available on a compute node. We thus benchmark two different applications with various data initialization modes: in parallel or sequential. We select a Sparse Matrix Vector Product and a BiCGStab method performed via HARTS. We run our experiments on a large scale NUMA machine with 24 NUMA nodes of 8 cores Sandy Bridge processor clocked at 2.40GHz and 32Gb of memory each [1]. Time results on SpMV kernel are reported in Figure 3.9, while the performance of the BICGStab method is illustrated in Figure 3.10. For both experiments, we select a matrix coming from the M_{Lp} collection on a square mesh of size 5000×5000.

For both experiments, we can notice that performances of both initialization methods differ from 8 cores in the X-axis. It corresponds to a single processor, so there is no remote computations under 8 cores. Thereafter, we can see that performances of parallel initialization mode outperform methods ran after a sequential data initialization. Indeed, in this case data are distributed among NUMA nodes and threads grant a reduction of

1. Thanks to J.F. Mehaut who gives me access to this experimental machine

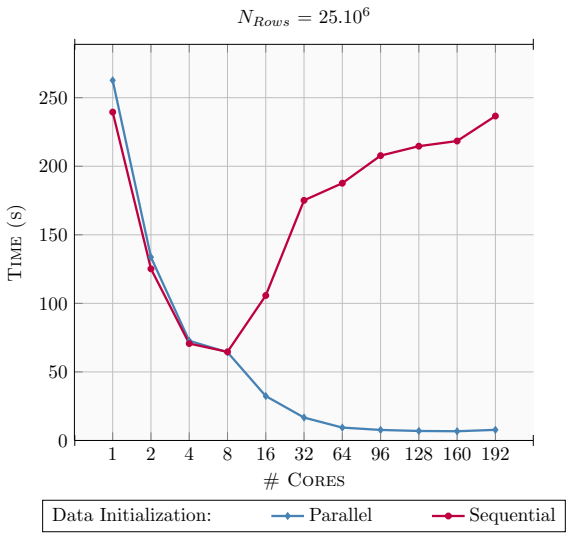


Figure 3.9 – *SpMV on 24 NUMA nodes*

memory traffic across interconnects between processors. With a sequential initialization, data fit in a single memory bank. Therefore, most of the threads operate on remote data and it results bus contention and memory access latency. In this case, parallel time grows up while it decreases with a parallel data initialization.

3.2.2.2 Work pushing strategy

At the runtime system level, we then implement a distributed task pool in HARTS. Each thread operates its own private queue and cannot access another one. When a task is marked as ready the task queue's owner, if idle, performs it. In this way, a thread can fill its queue with tasks that operates on close data. We can thus assume that a thread will always be responsible of tasks with the same partition identifiers.

At the instantiation of a task by the API, HARTS is then responsible to push it in a local thread's queue. This queue is thus chosen thanks to an affinity function. It takes in arguments a partition id, and returns the thread identifier that will own the task in its queue. Various strategies can be employed. We limit our study to place tasks in queues while ensuring that all threads will be responsible of an equivalent number of partitions. A smarter strategy can also be chosen according to the graph partition. In this case, we could fill threads' queue while trying to reduce data exchanges between sockets. We should attempt to place neighboring partitions tasks in threads on the same NUMA node. However, this is a minimization problem that is out of scope here.

As we mentioned above, threads are always responsible of the same set of partitions during the execution. By this way, we ensure that tasks with equal partitions id are always performed by the same thread. Therefore, data stored in memory are always initialized by the same threads. Two memory blocks belonging to the same partition are located in the

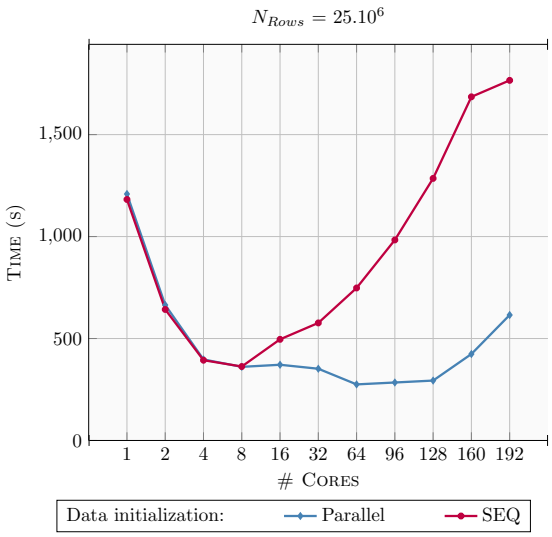


Figure 3.10 – *BiCGStab on 24 NUMA nodes*

same memory banks in a NUMA context. However, there is no strategy here to dynamically balance the work if a thread becomes in idle state.

3.2.2.3 Dynamic strategy at the execution

The execution model presented in section 3.2.2.2 does not allowed a thread to pick up tasks in another queue. It is a scheduling policy based on private queues without dynamic load balancing. The private distributed queue model shows its limits at run time. In the case of unbalanced work load, private queues hamper efficient computations. We thus implement a work stealing scheduling policy similar to Cilk[Blumofe et al., 1995]. By default, a thread launches a steal request to another thread according to ascending threads' numbering. However, this basic strategy generally is not the best because it does not take into account data locality. As we operate the data distribution before the beginning of the parallel region, we can thus know in advance where data are located. This information can be used at run time to decide the placement of a task according to data locality computations enhancement. By this way, we can make scheduling hints before tasks execution while avoiding system calls to know data locations. Our objective is to guide a thread to select a victim to steal following a steal strategy. For a given thread, we build a priority list of thread's ids that are good candidate for work stealing. At steal request a thread will browse its list to find the best victim according to a predefined strategy. If steal fails, the thread will check again its own queue. And if it is empty, the work stealing scheduler will be executed.

In order to build the threads' lists according to data locality, we used the notion of NUMA distance, previously explained in section 2.2.3.1. We make the assumption that data are initialized in parallel, so distributed among NUMA nodes. Thanks to the distributed queue and affinity function, tasks which are contained in the neighborhood of a thread thus operates on local data. Therefore, we can conclude that threads with a minimal NUMA

distance between them operates on close memory locations. The threads list is built by sorting threads by NUMA distance. A thread having a low NUMA distance from the list's owner will thus be selected with a higher priority than a more distant thread.

In order to validate our approach in HARTS, we benchmark a BiCGStab method with various task pools: a centralized and a distributed queue with work stealing strategy as described above. For both strategies, data are distributed among memory banks of the various NUMA nodes the machine is composed. We also select the NUMA aware work stealing strategy we set up with the distributed queue. The matrix we use comes from a realistic reservoir simulation: the SPE10 benchmark [Christie and Blunt, 2001]. At run time, we count the number of tasks a thread operates depending on the task partition id. As we know which thread is responsible of a partition id, we can thus deduce if the thread operates on a local or a remote memory bank. We thus compute the part of the computations that are local or remote, in percentage. We run our experiments on a dual socket machine, each composed of two Broadwell 14 cores processor configured with Cluster-on-die mode. So the machine is viewed as a 4 NUMA nodes, each one is composed of 7 cores and 32Gb of memory. We report the results we obtained in Figure 3.11a for the centralized task pool, and in Figure 3.11b for the distributed one.

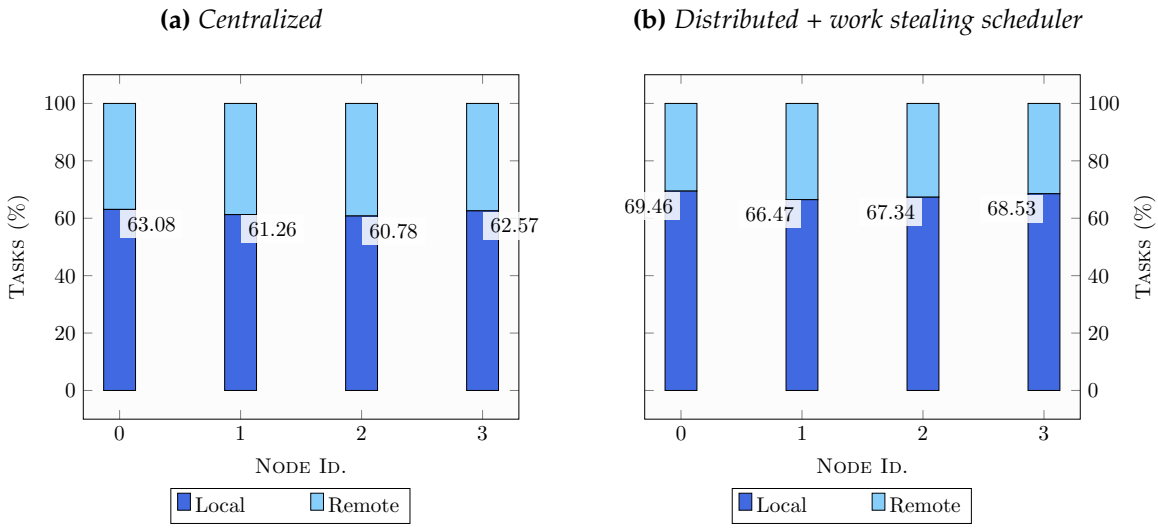


Figure 3.11 – *SPE10 – BiCGStab with Diagonal preconditioner*

In order to unbalance the work between processors, we used here a row partitioner which distributes rows among processors regardless of the number of nonzero elements per partition. We thus encourage work stealing for distributed task queue. If the sub-graphs are better balanced, steal requests are less numerous and threads operate more their own queue (filled with local tasks). Here, our objective is to emphasize our work steal strategy's benefit.

We can mention that we improve local computations from 5 to 7% with the dynamic work stealing scheduler. Threads perform up to 70% of local computations with this later policy. These results are quite moderate but still show data locality enhancement.

From this step, all the experiments with HARTS will be made with a distributed task pool with both work-stealing scheduler and the NUMA-aware scheduling hints.

3.2.2.4 Positioning with related works

As we previously seen in section 3.2.1, many works exist to take advantage of NUMA machines through scheduling policy.

The purpose of X-Kaapi is to get information from data at task execution time to schedule it according to memory location. Contrary to our strategy, there is no assumption on data placement before scheduling the tasks of a parallel region. By this way, we avoid system requests at run time to know where data are. OpenStream relies on the same strategy which consist to evaluate memory location at run time. However, they proposed a way to initialize data through tasks to distribute data among memory banks.

OmpSs enables users to detect initializing tasks which are supposed to distribute data among NUMA memory banks. By this way, the runtime system can make assumptions on data to efficiently schedule the tasks. It also provides runtime system calls to hint scheduler on data location before task instanciation. In addition of that, they enable work stealing scheduler to avoid load imbalance. The main difference to our strategy comes from the queue design. In our work, we distribute the queue among threads while OmpSs distribute them among nodes.

3.3 Monitoring and performance tools in HARTS

A parallel execution is a succession of tasks performed by various threads. Each task execution can be viewed as two events associated to start and end times. Performance counters and monitoring tools are employed to gain a more refined understanding of such an execution. We implement it inside HARTS, at several levels. During execution, we collect information about tasks before and after their execution. It happens when a task is marked as ready state and then picked up by a thread in its queue. At the end of the program, we gather data and then analyze them to analyze on the application behavior.

This modest work has not the pretension to be concurrent face to tools such as Tau [Shende and Malony, 2006] or Paraver [Servat et al., 2013]. While these tools enable to trace analysis from multi-level parallelism application on heterogeneous architectures, our study is limited to multi-threading analysis on shared memory machines. Tau enables to track parallelism in the application without instrumentation. We enable it too by instrumenting directly HARTS and it is limited to the use of this runtime system. Paraver with its Extrae extension enables to profile OpenMP or OmpSs applications. It enables to extract statistics from task-based parallel applications and may draw DAG of the application. We can also produce some statistics from task programming model in HARTS.

3.3.1 Monitoring in HARTS

All the collected information are first stored in memory, and then flushed in several files at the end of the execution. We create one chronological file per thread in CSV format. We set up points of interest (POI) to only focus on a part of the execution. A set of events can thus be gathered together and statistics can be made especially on this POI. An example is given by the Figure 3.12, which represents the chronology of a program performed by several threads. In this example we only focus on time measurements based on start/end timestamps collected from hardware clock. However any event can collect various information at run time. Task type and granularity, task placement or work stealing ratio are also precious data that can be helpful to understand application's dysfunctions. We set task granularity by user-defined values. We handle a list of thread's identifiers that refer to physical core ids. By this way, task placement is then deduced by storing thread id which performs the tasks in the event structure. When the distributed pool of HARTS is employed with work stealing scheduler, we also count how many steals are performed per thread. Tasks are also characterized by dependencies. A parallelism issue may be detected by analyzing the links between tasks grouped in the same DAG. The critical path is thus the longest path from the root of the DAG to the leaves. We denote the critical time by T_c which is the sum of the duration from the nodes of the critical path. DAGs and execution times are collected during the execution, whereas critical path is computed at the parallel region ending. We do not consider overheads generated by runtime systems in the critical time computation.

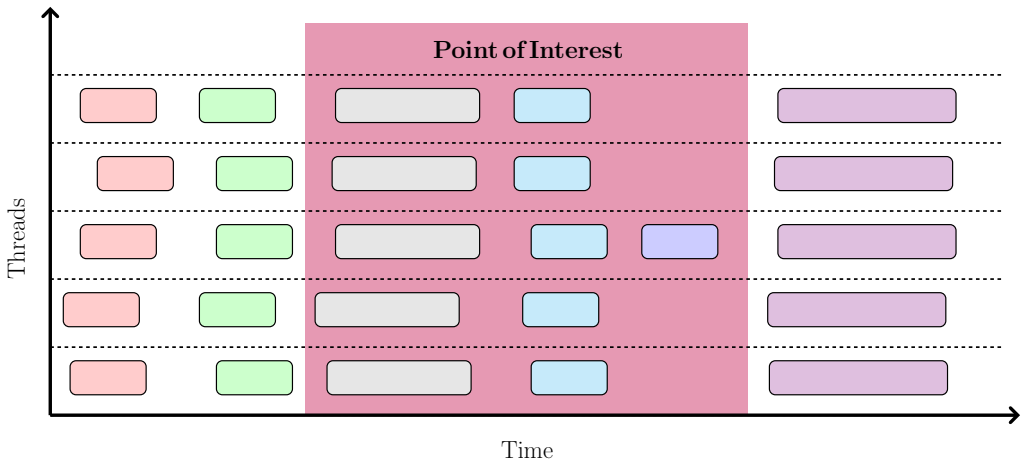


Figure 3.12 – *Gantt chart example*

Metrics

From collected data during the execution, critical path is not the only one data that can be computed at the end of the program. From timestamps, we can build a Gantt chart

such as the previously exposed one, both in CSV and Paje [2] formats. Moreover, we denote by W the addition of all the tasks duration which gives the total amount of work performed without runtime systems overheads. The duration of a task θ_k is given by:

$$w(\theta_i) = \text{end}(\theta_i) - \text{start}(\theta_i).$$

And so,

$$W = \sum w(\theta_i).$$

Classical studies assume that $W(p) = W(1)$ so E reflects the parallel efficiency of the algorithm with respect to the sequential code. Note that due to NUMA effect, work may vary with the number of cores in use [Olivier et al., 2012a]. So we consider W in function of the number of partitions N_{part} in which we decompose our problem, but also in function of the number p of threads in use. The computation efficiency $\text{E}_{\text{comp}}(N_{part}, p)$ of an execution is given by the formula:

$$\text{E}_{\text{comp}}(N_{part}, p) = \frac{W(N_{part}, p)}{pT_p}.$$

We can note that in the case where $W(N_{part}, p) = W(1, 1)$, E_{comp} corresponds to the parallel efficiency $\text{Eff}(p) = T_{seq}/(pT_p)$.

Because we log the tasks dependencies, we are able to compute the critical time T_c. We can thus detect a parallelism default by computing the ratio between W and T_c that represent the potential parallelism, i.e. W/pT_c. If T_c is high regarding W/p, then W/pT_c is low. So, we can deduce that T_c is the limiting factor so the application may suffer from a bad parallelism. In the same way, T_c is a lower bound for the parallel time T_p. If the two given times differ, we can thus deduce that the application suffers from inactivity. We notice that $W(N_{part}, p)$ does not vary a lot face to N_{part} parameter (See Figure 3.8 in Section 3.1.5.3). For more simplicity, we can let $W(N_{part}, p) = W_p$.

Scheduling activities

A thread can be inactive because of a lack of parallelism or scheduling inefficiency. For a given thread, we compute its idle time by the formula:

$$\text{Idle}(k) = \{T_p - \sum D(\theta_i) \mid \forall \theta_i\}.$$

From task types we can identify which one is the most time consuming, and how many times the task is invoked. We can do some statistics per threads on a task type, or in a global context. Task placement is one of these statistics we can make. In this model, we assume that both allocation and initialization are done in parallel following the assumptions made in section 3.2. Tasks with different types but with the same partition identifier are thus performed by the same thread. Hence our assumption let us to say that data are located in the same memory address space. As we collect both partition and thread identifiers per task, we can thus deduce if a thread requests a data to a local or a remote memory bank.

2. https://github.com/schnorr/poti

By this way, we avoid system calls to ask where the data are physically located. We can then plot statistics per thread about data locality computations.

3.3.2 First analysis

We now give an overview on the information we can extract from an execution with monitoring tools. We thus run a BiCGStab method on a multi-core machine while varying both the number p of used cores and the size of the input matrix. We partition data from Metis graph partitioner. We are able to do some statistics to have a good understanding of performances we obtain from the traces we collect at run time. We made our experiment on a bi-socket Broadwell computing node. Each processor is configured with cluster-on-die mode, so it is exposed as 2 NUMA nodes where each one has 7 cores and 32Gb of memory.

3.3.2.1 Overhead of Instrumentation

We first expose how intrusive the counters are during the execution. For a given number of threads, we compare two executions: with and without performance counters. For each one, we collect the parallel elapsed time to compute the BiCGStab algorithm. We then compute the time increasing from a classical execution to the one on which counters are turned on. We reduce it in percentage, and then illustrate the results in the Figure 3.13. As a simple BiCGStab does not converge without preconditioner, the method iterates up to 1000 times. Hence, it generates a lot of tasks and the benchmark is perfect to measure how sensitive are the performances according to code instrumentation.

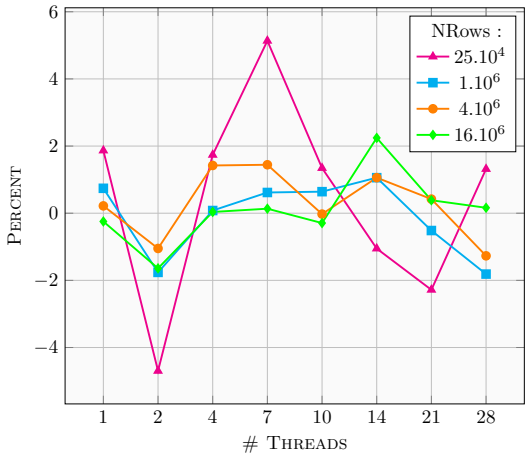


Figure 3.13 – *Counters' overhead evaluation*

We can see that it mainly affects the smallest input matrix, which can reach up up to 5% of increasing. In this case, the application does not provide a sufficient computing effort to hide overheads. For the other matrix sizes, out monitoring tools does not degrade application performances. The overall performances does not exceed more than 2%.

3.3.2.2 Iterative method analysis

Then, we illustrate performance evaluation in terms of executed flop performed per second in Figure 3.14a and of parallel efficiency in Figure 3.14b.

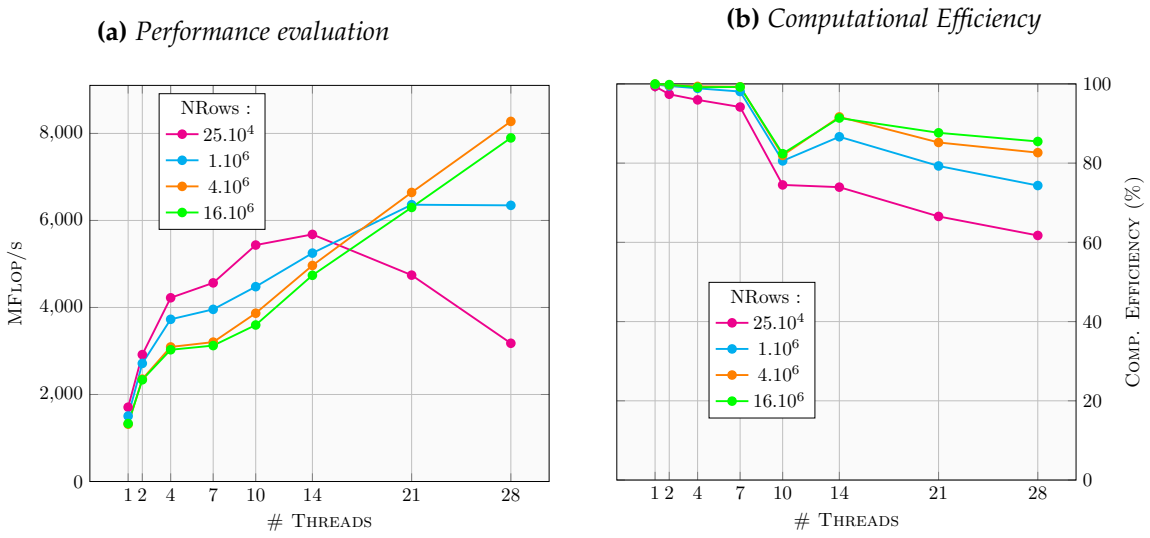


Figure 3.14 – *BiCGStab with Diagonal preconditioner*

We can first notice that for the smallest test case, performances increase up to 14 threads, and decrease after. It results a decrease in the parallel efficiency up to 50%. For the other input matrices, we see an increase in the parallel performances in function of the number of threads. For the biggest case, we decrease up to 85% in computational efficiency, while the other reach 82% and 75% respectively in descending order. Size is the only varying factor between the execution, while the matrix pattern remains the same between each other. We can thus suppose that for the smallest benchmarks, computing efforts are inferior to data communication which degrades performances when the number of threads grows up. On the opposite, the bigger benchmarks succeed to hide data communication and peak performances are reached. We now look at the data locality policy for one of the benchmark, $N_{Rows} = 4.10^6$, with $p = 28$ threads. Results are shown in Figure 3.15a, on which we illustrate both local and remote computations per NUMA node. We can see that threads perform up to 69% of local computations, which enhance computing performances.

We then investigate on the scheduling efficiency with the same input matrix. We thus activate monitoring on the previous method with the maximal available number of cores, i.e. $p = 28$. We now look at the threads' activity in Figure 3.15b on which we report both compute and idle times grouped by NUMA nodes. We can first report that we obtain around 17.7% of inactivity per NUMA node. This result slightly corresponds to the loss of efficiency reported in Figure 3.14b, which is equal to around 18%.

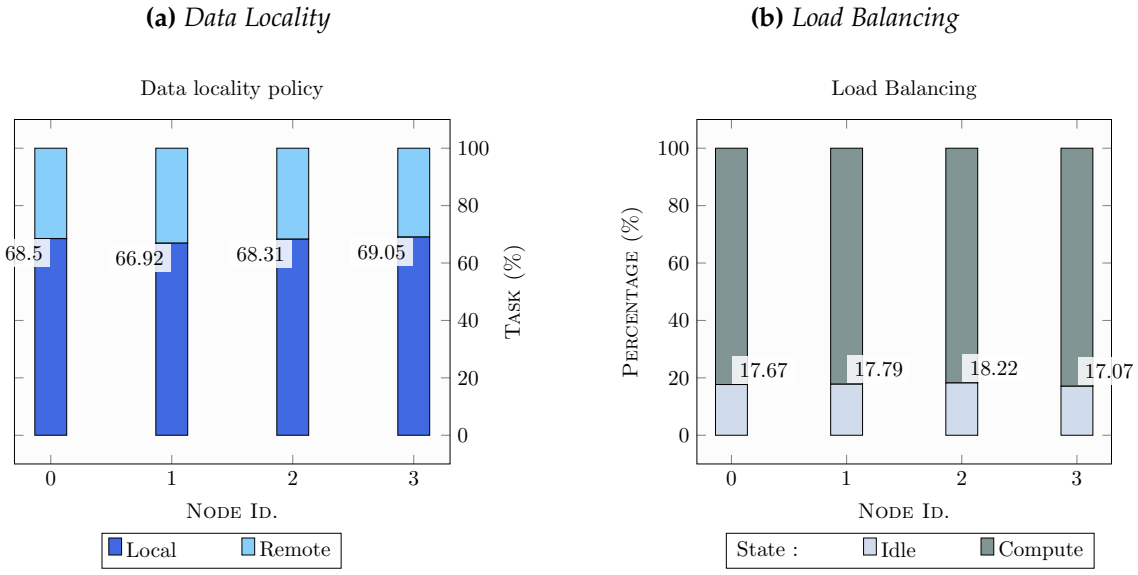


Figure 3.15 – *2000 x 2000 mesh – BiCGStab with Diagonal preconditioner*

3.3.2.3 Work inflation in a NUMA system

We split the computations into tasks thanks to graph partitioning methods. Until now, we build tasks according to the number of threads in use, i.e. $N_{part} = p$. In section 3.1.5.3, we already show that we do not observe a work inflation on 1 threads while varying the number of partitions. However, this experiment has not be done on multi-threaded executions yet. Thanks to monitoring tools implemented in the HARTS runtime system, we aim here to evaluate work inflation. For this experiment, we enable the work stealing scheduler we have developed. We benchmark a BiCGStab method with a matrix coming from the Laplace problem Finite Volume discretization. The size of the system is $N_{Rows} = 16.10^6$ rows.

From metrics we already introduced, work inflation is given by the formula:

$$\text{WInfl}_p(N_{part}) = \frac{W(N_{part}, 1) - W(1, 1)}{W(1, 1)}.$$

Results are gathered in Figure 3.16a. We present various executions which differ from the number of threads in use ($p = 7, 14, 28$), on which we vary the number of partitions.

We can notice that the working time increases from the initial working time $W(1, 1)$, regardless the number of threads we use. However, we observe that the working time inflation depends on threads. It only depends on the number of partitions too, but this is less important. Until now, we need more investigation to explain why the working time is dependent from the number of threads in use. However, we suppose that this phenomena is linked to memory accesses and bus contention.

(a) $N_{Rows} = 16.10^6$

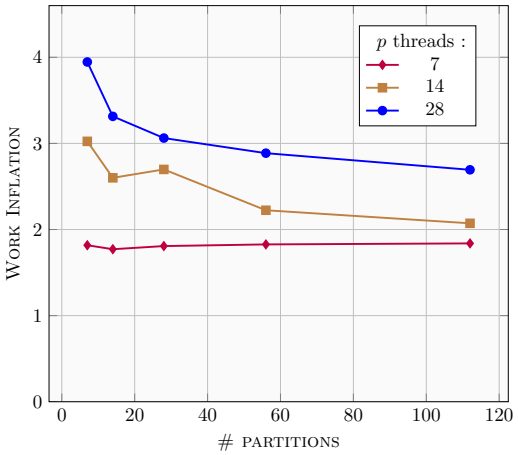


Figure 3.16 – *Work inflation of BiCGStab with Diagonal preconditioner*

Mesh size	N_{Rows}	N_{NZ}	Memory Footprint
1000 x 1000	1.10^6	4996000	488 Mb
2000 x 2000	4.10^6	19992000	1953 Mb
3500 x 3500	12.10^6	61236000	5980 Mb
4000 x 4000	16.10^6	79984000	7811 Mb

Table 3.2 – *Input matrices overview*

3.4 API Evaluation on various preconditioners

This section is devoted to the performance evaluation of the API we develop. We thus implement several preconditioners on top of the framework to enhance their parallel performances. The platform we use is based on the multi-core architecture. The machine is composed of one computate node equipped with two Intel Broadwell multi-core processors clocked at 2.4GHz. Each processor contains 14 cores, and is paired with 64Gb of memory. The processor is configured with Cluster-on-Die mode, which enables to split a processor in two distinct NUMA nodes. It results that the machine is viewed by the system as 4 NUMA nodes, each paired with 7 processors and 32Gb of memory.

For this study, we used standard linear systems which are not ill-conditioned. Matrices are coming from the M_{Lp} set, which are built from a Finite Volume discretization of a Laplacian operator on a unit cube mesh. We summarize information on the input matrices in Table 3.2.

3.4.1 Polynomial preconditioner

First of all, we analyze performances of one of the simplest preconditioner which is easy to build (See section2.1.2.2). However it is now used in simulations because of

its numerical inefficiency. This preconditioner is a succession of sparse matrix vector products, so it is easy to parallelize. We implement it through our linear algebra API, and benchmark it on various linear systems of different sizes. We then execute it on a number p of processors.

We first report performance evaluation. Time results are exposed in Figure 3.17a and operated operations per second in Figure 3.17b. We can observe that parallel time is decreasing in function of threads. It is more blatant on number of operations performed per second. On the Figure 3.17b, we can see that performances are growing up in function on the number of threads. We reach up to around 6GFlops for all matrices with the execution on 28 threads.

(a) *Time* **(b)** *MFlops*

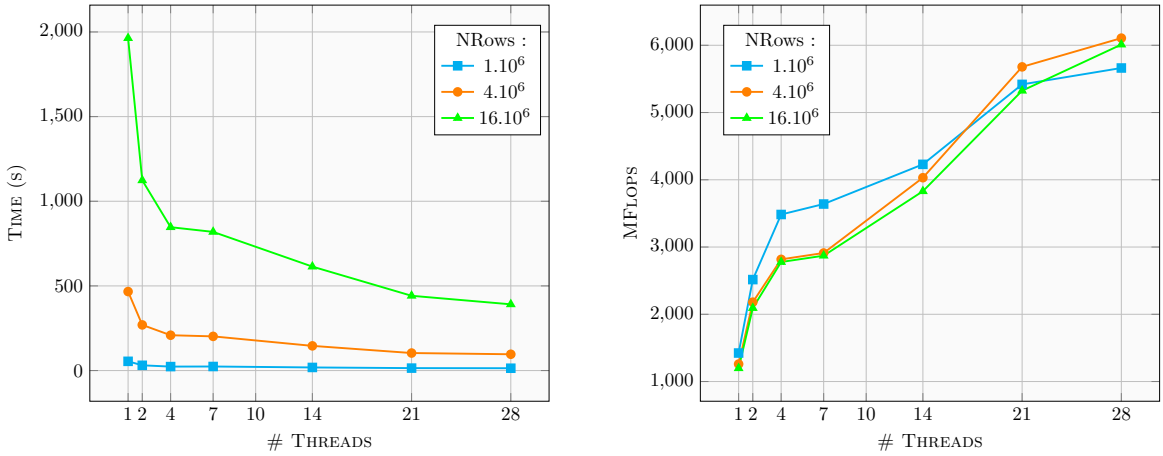


Figure 3.17 – *BiCGStab with Polynomial preconditioner parallel performances*

We can also see the parallel efficiency on both execution in Figure 3.14b. We observe that the curves go down up from 70 to 80% of efficiency, depending on the matrix size. We can see that the performances begin to decrease from an execution with 14 threads. This phenomena is induce by data communication because we run it on two NUMA nodes instead of only one before.

We are now looking at the cores' states during the execution on 28 cores of the largest matrix (i.e. $N_{Rows} = 16.10^6$) in Figure 3.18b. We can observe that cores waste around 20% of the time in idle state. This ratio corresponds to the percent of inefficiency we previously observed. It may be related to HARTS's overhead, or induced by waiting times on data for both SpMV and dot products.

3.4.2 Incomplete LU Factorization (ILU) preconditoner

We now attempt to evaluate the parallel performances of the ILU preconditioner we develop via our framework. The preconditioning operation, not naturally parallel, solves $LU.x = y$ with a backward substitution, then by a forward substitution. This preconditioner,

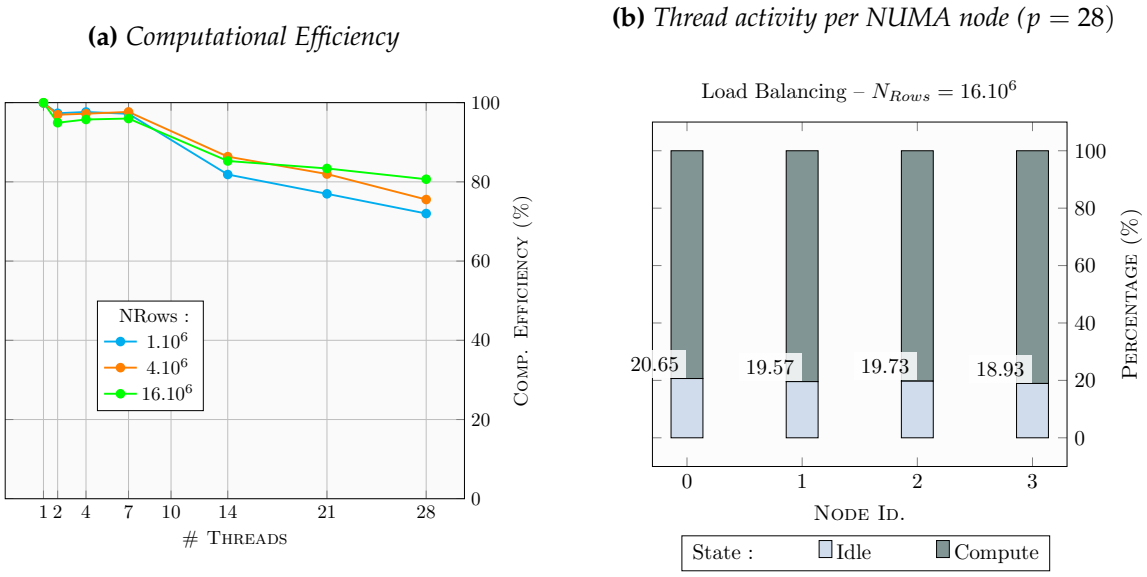


Figure 3.18 – *BiCGStab with Polynomial preconditioner parallel performances*

well known to be efficient for standard cases (i.e. not ill-conditioned and moderate problem size), is not naturally parallel, since its algorithm is recursive. Task programming associated to domain decomposition and renumbering techniques enable to extract different hidden levels of parallelism. Figure 3.19 illustrates the DAG of the backward and forward steps of ILU(0) algorithm parallelized with a 4 parts domain decomposition. A partitioner is used to split the global domain into 4 not connected interior domains and 4 interface domains connecting the 4 previous ones. The chosen partitioner algorithm aims to maximize the size of the independent interior domains and to minimize the size of interface domains creating dependencies between all the domains. Thus, tasks associated to interior domains can be executed in parallel, while the runtime system scheduler extracts automatically a two level degree of parallelism between tasks associated to interface domains. Data associated to local domains are stored contiguously in memory.

We run the ILU preconditioner in conjunction with a BiCGStab solver on several threads with input matrices of different sizes. As the previous experiment on polynomial preconditioner, matrices are coming from the M_{Lp} set. We report time results in Figure 3.20a.

We can first notice that the ILU preconditioner is less time consuming than the polynomial one. However, we can see that time results decrease at a very low speed for all the input matrices.

We now have a look to parallel efficiency which is illustrated in Figure 3.21a. We can notice that the efficiency tumbles from 7 threads and reaches the lowest efficiency with 28 threads from 80 to 65%, depending of the matrix. To better analyze this phenomena, we thus compute the ration W/pT_c as previously explained in section 3.3, between the working time W and the critical time T_c. We observe that all the curves decrease in function of the number of threads we use. With the maximal number of threads, all the test cases obtain the time W_p represents about 80% of the critical time T_c. It means that the parallelization is

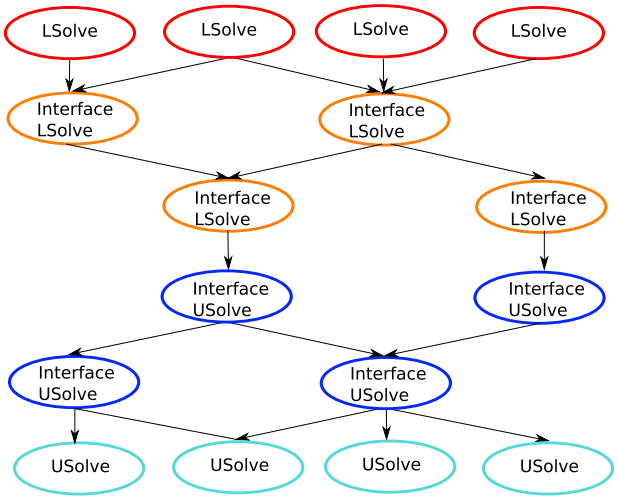


Figure 3.19 – *Incomplete LU Facto. DAG*

(a) *Time*

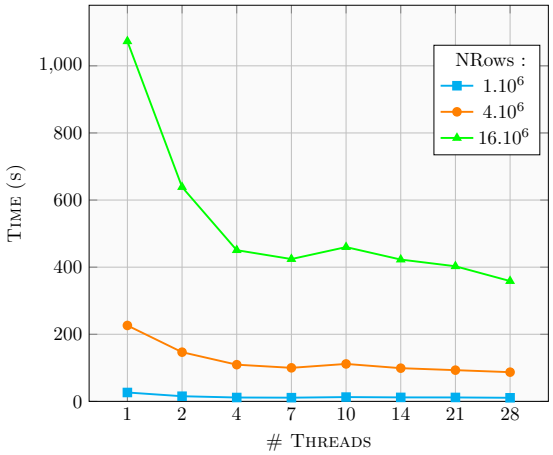


Figure 3.20 – *BiCGStab with ILU preconditioner parallel performances*

bounded by the critical time, but globally the application is well parallelized. We so then analyze threads activity in Figure 3.22, for the matrix of size $N_{Rows} = 4.10^6$ of an execution with 28 threads. In these results, we group both working and idle times by NUMA nodes. We can observe that idle time reach up to around 39% of inactivity. Even if the tasks are well balanced among NUMA nodes, because of the structure of the ILU algorithm dependencies generates many idle time in the execution.

3.5 Conclusion

In this chapter, we propose a way to write efficient parallel iterative methods to solve large and sparse linear systems. Through a unique semantic, we decompose the work into tasks that are then directed to a given runtime system. Thanks to the HARTS runtime

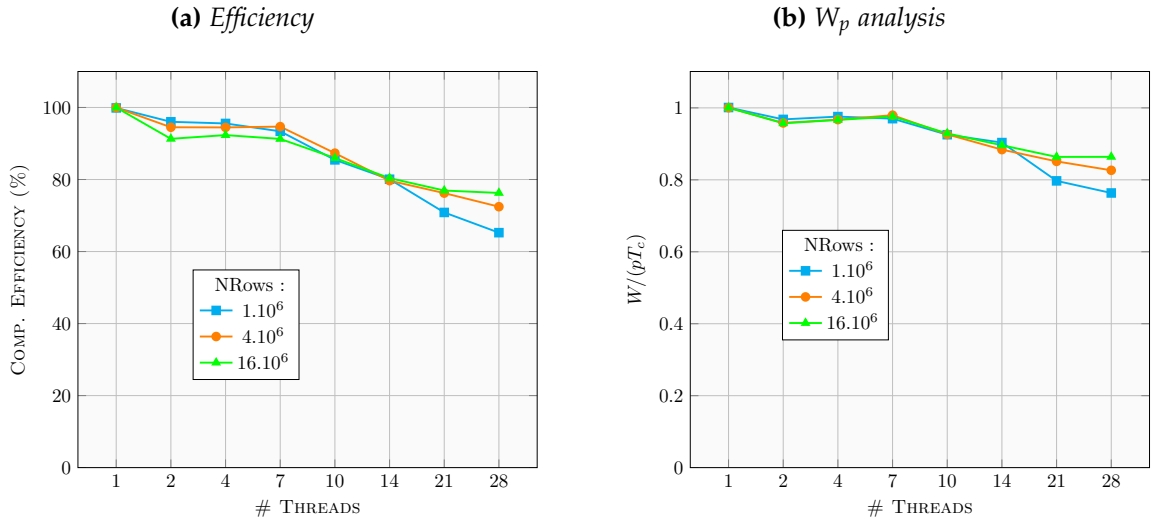


Figure 3.21 – *BiCGStab with ILU preconditioner parallel performances*

system, we enhance data locality computations to take advantage of multi-core systems based on a NUMA memory topology. We also develop monitoring tools in order to analyze and evaluate performances. By this way, we demonstrate the efficiency of the methods we implemented on top of the linear algebra framework we developed.

However, many-core systems imposes other challenges than multi-core platforms. Our concern is to develop a portable solution to write efficient algorithms regardless of the underlying architecture. Hence, we propose an extension of the API in the next chapter to take care of many-core architectures challenges such as code vectorization or memory management. As for the current chapter, we evaluate all these extensions on sparse iterative methods we use in oil reservoir simulations.

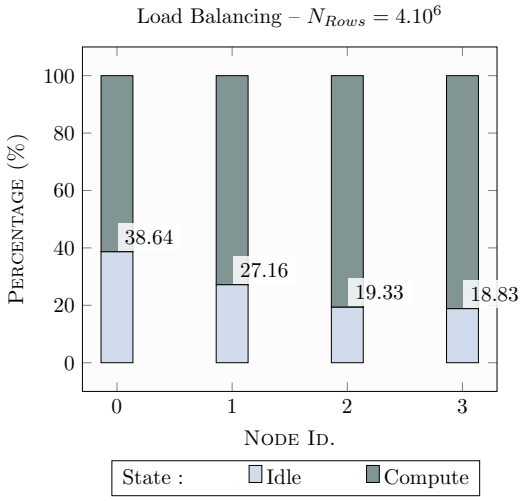


Figure 3.22 – *Thread activity per NUMA node ($p = 28$) for ILU precond.*

Chapter 4

Efficient programming on Many-Core architectures

Contents

Through the computer hardware evolution, multi-core chips tend to significantly increase the number of computational units per node. Many-core processors come from a core design simplification. The strategy is to handle many more but less powerful cores while aiming to outperform chips with less but more powerful cores. Another advantage is the low power consumption of such chips. However, programming challenges are different and require additional efforts on many-core systems to reach high performances.

GPUs are computing accelerator boards which provide a great efficiency face to intensive data parallelism thanks to their huge number of simple cores. Since several years, GPUs have been used efficiently in a large scope of domains such as numerical simulations. Programming such systems involves difficulties which mainly arise from the communication between the CPU and the GPU, but also from the SIMT model, which is close to vector processing. However, GPU programming is out from the concern of this work. Among the many-core processors, Kalray and the MPPA-256 (standing for Multi-Purpose Processor Array) architecture [de Dinechin et al., 2013] mainly focus on power efficiency while providing a huge number of cores per chip. Intel built its own many-core technology, first based on a accelerator board. Many integrated core (MIC) architecture was first handled by the Intel's Xeon Phi Knights Corner co-processor board [Chrysos, 2012]. Its main characteristic comes from the ring interconnect that links all the cores. A core handles several hardware threads (also known as hyper threads), and various execution units (both a scalar and a vector units). The challenge arising from this kind of chip is double. The architecture of this chip is complex, and reaching high performances comes from vectorization and overlapping communication by computations. Systems equipped with those accelerators are heterogeneous, so developers have to take care of data movements across PCI bus. Our first concern was to exploit this kind of architecture. However, following the evolution of its MIC architecture, Intel recently launches the Xeon Phi Knights Landing processor. Our interest thus switches on this many-core processor.

This chapter is devoted to the Knights Landing architecture programming to achieve high performance computing. From a performance portability perspective, we have upgraded the API we previously developed. By this way, it can address solutions to various programming challenges coming from hardware features without any formalism changes. For this purpose, we operate internal mechanisms changes inside the API. Hence, it does not impact the end users' way of programming.

In the first section, we make a short overview of the KNL many-core processor and the key elements of which it is composed. The way we take into account the various elements may have a significant impact on performances. However, we face several programming challenges to make a good utilization of hardware components.
We introduce in the second section the programming challenges we have faced to maintain performance portability while taking advantage of KNL many-core processors. After highlighting different challenges, we detail the approaches which are already adopted in other contexts.
In the third section, we detail the extension we introduce to benefit from vectorization on the

KNL processor in standard operations like the BLAS operation while keeping performances of the API. We explain the way we set up in our API while keeping transparency for users. We have noted that sparse structures we used until now are not well adapted for vectorization. After a short review of the most popular sparse matrix format in the fourth section, we explain how it is possible to integrate a new one in our API without any changes for developers.
The fifth section is devoted to memory management between various memory banks available in the KNL processor.
In the sixth section, we present the performance evaluation of the API on the KNL many-core processor. Thanks to various benchmarks, we thus demonstrate the relevance of our approach on the overall performances.
Eventually, we discuss on the improvements and the ongoing work aiming to enhance programming challenges management.

4.1 Knights Landing architecture Overview

Before overcoming programming challenges, it is necessary to well understand the architecture to know the challenges' origin and then to think about an appropriate solution. In this section, we overview the main features of the *Knights Landing* (KNL) micro-architecture. At a microscopic scale, a core is responsible of the computations. When we zoom out, interconnect that links all the cores enables communications between them and the memory. In the KNL processor, different kind of memory banks are available to gain in performance. In particular, Intel integrates a stacked memory onto the chip to attempt to reduce memory latency.

4.1.1 Core design and Cache levels

A core is the entity that handles instructions fetch and execution units. It is thus responsible of doing computations. KNL processor can handle from 64 to 72 cores in a chip. Cores are derived from the Intel Atom microarchitecture used in mobile devices, but adapted to HPC applications. They benefit from a lower frequency than the usual cores. Depending on the KNL processor model, the clock rate is from 1.3GHz to 1.5GHz while reducing power consumption. The KNL's cores use the x86 instruction set. Moreover, Intel extends the Advanced Vector Extensions (AVX) to support 512 bits wide instructions. At a clock time, an instruction can thus operates on 16 floating numbers in single precision, or 8 floating numbers in double precision. A core handles 2 vector processing units (VPU) that are the vector and floating point execution units of KNL.
Each core is composed of four hardware threads, also called hyper-threads [Marr et al., 2002]. This feature enables to possibly benefit from several execution contexts on the same core.

Cores are paired by two to constitute a tile. A tile shares a large 1Mb L2 cache between the two cores. Additionally, each core of the tile owns its own private L1 cache.

4.1.2 High Bandwidth Memory (HBWM)

The KNL processor supports two different kinds of memory banks: MCDRAM and classical DRAM. First, the processor is equipped with a stacked memory on the chip, named MCDRAM. This memory bank intends to reduce memory latency while being closer to the cores. MCDRAM has a capacity of 16Gb, which is scattered in 8 devices of 2Gb each. A device can be accessed from one of the 8 memory controllers on which the mesh architecture is composed. The MCDRAM delivers a performance of around 400Gb/s, that is 4.5 times faster than the DDR4 memory (˜ 90Gb/s). As a standalone processor, KNL also supports DRAM memory, up to 384Gb. There is 6 memory channels to this memory bank.

Two different memory modes are available to make a good cohabitation between the two memory banks. The memory mode is configurable at boot time. First, MCDRAM can be considered as a standalone memory within the *Flat Mode*. By this way, user has to explicitly allocate memory in the MCDRAM through the *Memkind library* [1] and allocator functions. It enables to devote memory management to end users.

However, the MCDRAM can also be viewed as a Last-Level Cache via the *Cache Mode*. If a data is not in the L2 Cache, a request is thus sent to the MCDRAM as a cache to ask if the data is present or not. If the data is in it, the data is thus returned. If not, a data request is sent to the main memory (DRAM) which will send back the data to the requesting core. Users does not have to take care about MCDRAM, and allocate memory as usual. The mechanism is only hardware managed.

Another hybrid mode is possible by configuring MCDRAM between Flat and Cache memory modes. We can split the MCDRAM in two parts, one as a Cache and another as a Flat memory. Users have just to define the proportion of each one when they turn on the *Hybrid Mode*.

4.1.3 2D Mesh interconnect

All the tiles are linked together via a cache coherent 2D mesh interconnect. The architecture is illustrated in Figure 4.1. The architecture handles a distributed tag directory which is dispatched among all the tiles. More precisely, L2 caches are connected all together with a mesh interconnect. The directory in each tile owns a portion of the address space based on a address hash.

Users can select various cluster modes to improve memory bandwidth by lowering the distance to retrieve a data when a L2 miss occurs. Indeed, in the case of a tile is looking for a data, it will first ask its own cache levels. If the data is not present, it will ask to tag directory which handles the memory address. If the data is no longer valid, the directory will then asks the main memory. The memory bank will be read from memory controller and then send to the tile requesting the data. So cluster modes attempt to reduce memory latency while minimizing the path from the tile to both the tag directory and the memory.

1. See https://github.com/memkind/memkind

In the *All-to-all* mode, the memory addresses are uniformly distributed among all the tag directories. Hence, a core may take a long path to first request a data to a tag directory and then maybe the memory.

For the *Quadrant/Hemisphere* clustering mode, tiles are divided in four quadrants or two hemispheres. In each part, a tag directory is affiliated to a set of memory controllers. When a cache miss occurs, a request is thus send to a tag directory containing the memory address. If it does not have a coherent data, it thus asks to the memory controller it is related to. It possibly may increase the path length if the tile which owns the memory address is not in the same part as the tile which requests data.

The *Sub-Numa Cluster* (SNC) mode extends the quadrant/hemisphere mode while affiliating a tile with the directory and the memory. Each part is exposed as a NUMA node. For a NUMA optimized application, a tile will access data from tag directories and memory controllers which are located in the same NUMA node. By this way, latency is thus reduced.

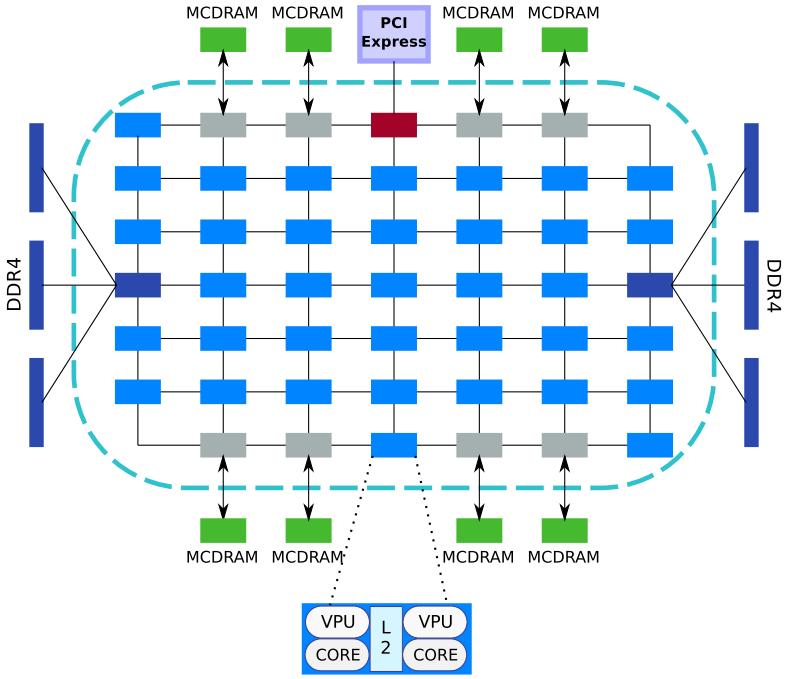


Figure 4.1 – *Intel ® Knights Landing processor architecture*

4.2 Programming Challenges

From hardware features of the KNL processor, some programming issues arise. Taking care of them is challenging, but it has a significant impact on performances. It sometimes needs to rewrite programs from scratch, for example, due to the change of algorithmic structure or data structures.

However, our interest here is to write portable application regarding performances

on various architectures while benefiting from the best optimization. Thanks to the internal structure of our API, it is easy for developers to integrate new structures or mechanisms without rewriting applications. By this way, we can fix issues introduced by a specific architecture transparently for users.

4.2.1 Multi-versioning of kernels for vectorization

Vector processing is one of the most powerful features enabled by the KNL processor. Vector registers increase up to 512 bits with this architecture. This size corresponds to 16 single precision floating numbers, or 8 double precision floating numbers. Each core handles two vector processing units 4.1.1. Clock rate of the KNL processor is slower than most of the standard actual multi-core processors. However, the 512 bits wide range vector processing enables to boost performances. It is a good compromise to overcome the slow down of the core frequency. It is thus important to take advantage of this feature to enhance performance applications from multi-core to KNL programming.

Several solutions exist to write efficient vectorized applications. For the simpler cases, it only requires to adapt the existing code to force the use of vectorized instructions rather than sequential ones. In some cases, an alternative algorithmic approach is favored, while keeping the same data representations.
However, in the most complex cases we are obliged to take other data structures to enable the vectorization of the associated operations.

4.2.1.1 Implementation

As our API is designed to support several architectures, we must be able to manage several versions of a kernel, possibly one per targeted architectures. In the case of the KNL architecture, a version of a kernel will designate a AVX-512 vectorized implementation's kernel. The factory pattern is employed to choose the appropriate version to the current architecture. The choice of the version can be made at two distinct times.

Runtime systems such as StarPU [Augonnet and Namyst, 2008] allows to attach meta data to a task, which enables a task to have several representation. A *codelet* is an abstraction of a task which may have several representations (e.g. one for classical CPU and another one for GPU). The choice of the version that will be performed is done at run time by the scheduler. This representation of multi-version kernels is mainly adopted in the case of the heterogeneous architectures because of the cohabitation of two distinct architectures at a time. Hence, the decision has to be dynamic and made at execution time.
X-Kaapi [Gautier et al., 2013] has a similar feature to provide multi-versions of task implementations through C++ template specialization.

Thanks to dynamic dispatch, the Intel MKL [Intel, 2017] library detects the appropriate version of BLAS kernels depending on the underlying architecture at execution time. By this way, it is fully transparent for its users and does not require any knowledge from them.

The choice can also be done at compile time, with conditional building application. By this way, there is only one kernel with an implementation that is frozen during the execution. The decision could not react to runtime variations. It is favored for regular applications. Dynamic adaptation of code is subject to future work.

4.2.1.2 Data Structure

Some specific kernel's implementations require to change data representation structures (e.g. vector or matrix). The API will thus have to manage several structures for the same data. The challenge is thus to manage distinct data representations at execution time.

The choice may be done at execution time, while using a factory pattern for all the data structures available. The use of a structure instead of another is dictated by the type of computational resources chosen by scheduling decision to perform the task. This choice is well adapted to heterogeneous architectures, in which we may need to handle several data structures at run time.

However, we sometimes are in the case where we need to handle only one structure per data. In this case, we do not require to make decision at run time but may be at compilation time. It can be implemented through several ways, as with API's template functions in C++. It only requires to redefine all the methods for all the structures.

4.2.2 Managing memory to control High Bandwidth Memory usage

High bandwidth memory enhances data accesses and thus overall performances. However, the size of such a memory bank is highly limited compared to standard memory. The coexistence of the two kinds of memory banks in compute node based on KNL processor imposes to find a compromise between each other. Indeed, memory management is needed to be sure to maximize performances while taking advantage of the best memory latency.

To benefit from the best latency, users need to allocate data in the appropriate memory bank. The purpose of the API is to provide a collection of features to abstract users from managing computations at hardware level. By this way, we aim to manage data allocations in a transparent way for users.

4.3 Various vectorized implementation for standard kernels: example with the BLAS

BLAS operations are the most used operations in the linear algebra area. These operations are particularly well adapted for vector processors, but it exists many ways to vectorize code. The API we develop enables to propose various implementations for a single BLAS kernel. In our case, the differences between each ones come from the way of vectorizing the code.

4.3.1 Different ways to vectorize a method

Vectorization of code may be produced by several ways. First, compilers may generate vectorized code through code analysis. Since OpenMP 4.0, the compiler may automatically generate vectorized code via the `#pragma omp simd` pre-processing directive. This directive enforces compiler to generate vectorization. But there are other directives which give hints to compiler. It then vectorizes the code or not, depending if the conditions are fulfilled. By this way, user does not have to care about loops unrolling. This method has also the advantage to be portable and non invasive. However, performances may be degraded if the application was not designed for vector processing. Moreover, the compiler will possibly not unroll the loop as expected by the developer. Indeed, the compiler may not detect the best option to unroll the loops to vectorize codes. In that case, it possibly needs the intervention of the developer to reorganize algorithmic structure of the method before code vectorization.

Some libraries are available to take advantage from vector processing units in a convenient way. Hence, developers do not have to care about vectorization because it is hidden behind a library call. One of the most popular library used in linear algebra field are the BLAS library such as OpenBlas [Xianyi et al., 2017], Atlas [Whaley and Dongarra, 1998] or the Intel Math Kernel Library (*MKL*) [Intel, 2017]. This latter provides BLAS operations while vectorizing many of them in a transparent way to increase application performances. The library enables to target several architectures by addressing specific computations with the available vector instruction set on the machine. By this way, a code targeting the KNL architecture does not need any changes from multi-core implementations.

However, when specific operations are required, manual vectorization by developers is needed. Most of the time, we prefer to use intrinsic functions to manage vector processing instructions at a lower level (close to the assembly) instead of automatic vectorization by compiler. However, the application lose its portability because of the intrinsic functions are hardware dependent. They are generally designed for only one instruction set. For example with the Advance Vector Extension (AVX) which is supported by most of the current processors. However, AVX2 and more recently AVX512 are supported by only a subset of recent processors. Functions differ from the input vector register sizes between two AVX instruction sets version. But the processor builders also add new features with specific vector register uses. Hence, developers have to maintain a version for each version of AVX they use. Each one will benefit from the hardware features of the underlying architecture.

4.3.2 Various implementations proposal in the API

In the case of level 1 BLAS operations, we operate on vectors or subvectors only. Hence, we do not need to change data structures to enable vectorization. However, the vectorized method differs from the previously implemented kernel. Hence, we need to maintain various implementations of a method in the API depending on the architecture or the instruction set we use. In our case, we need to handle a vectorized implementation of

kernel to take advantage of the KNL architecture, while keeping the other implementations for other architectures (e.g. multi-core architecture).

Thanks to our API, developers can select among several implementation of the kernel. In our case, the differences between each implementation come from the way to vectorize the operation or not. All the BLAS operations proposed by our API enable to target one of the three options of vectorization proposed by the previous sub-section. As it has been explained in section 4.2.1.1, several options exist to enable multi-versioning of a kernel in the API. Depending on the compiler option we provide to build the API, a flag is raised. By this way, kernel implementation is frozen all along the execution. Indeed, contrary to heterogeneous architectures we do not require the coexistence of two various implementation at run time. We suppose that the selected implementation is valid all along the execution.

The choice of the implementation we used is specified at API's tasks creation. For each task formalism, developers have to specify a function pointer to the work to do at task execution time. The compiler flags activate one implementation among the available ones. The choice of one implementation is here dictated by the libraries or compiler we used to build the API.

Hence, when the implementation is available we can activate it. In the absence of parameters, the vectorization is disabled for all the tasks. It thus corresponds to the basic multi-core API version. Otherwise, if BLAS library is detected at compile time we built the API around it for BLAS operation. If a specific intrinsic version is implemented, we can switch to this version if intrinsics functions are available. We can also select the auto-vectorization provided by the scheduler.

Actually, the instruction set AVX512 is only supported by the KNL processor. It enables to operate on 512 bits wide registers. It doubles the AVX2 capacity (i.e. 256 bits), which encourage developer to vectorize codes. However, the way of generating vectorized code is not the only one issue we have. Indeed, vector instructions need to perform the same instruction on a range of elements at a given time. All the structures are not adapted to it and need reorganization. Sometimes, memory accesses can not be performed on a regular range of elements. It is thus hard to find a way to correct it, and then benefit from vector processing. The solution can be found from algorithmic structure of the application. In this case, we attempt to change the order of execution of the operations to process instructions by vectors. However, it is not always possible. Sometimes, we must change data representation to impact the elements access pattern in a more convenient way for vector instructions.

Let's take the example of the *axpy* kernel, which can be performed thanks to the several ways we cited above. The Listing 4.1 illustrates how we can chose between each implementations. In this way, we decide to select the implementation of the *axpy* kernel when a task descriptor execution is invoked. However, we also could select it by changing the task descriptor we push in the `Sequence` object. In Listing 4.2 we illustrate the kernel's implementation via the compiler hints. Listing 4.3 shows the use of Intel MKL to compute

Listing 4.1 – *Axpy's descriptor trigger execution*

```
1  void axpy_Descriptor::exec()
2  {
3    #ifdef USE_MKL
4      task_axpy_mkl(A, B, C, val, begin, end);
5    #elif defined(USE_AVX512)
6      task_axpy_intrin(A, B, C, val, begin, end);
7    #else
8      task_axpy_auto(A, B, C, val, begin, end);
9    #endif
10 }
```

Listing 4.2 – *Axpy's taks auto-vectorization*

```
1  void task_axpy_auto(double* __restrict__ A,
2                      double* __restrict__ B,
3                      double* __restrict__ C,
4                      double val, int begin, int end)
5  {
6    int k;
7    #pragma vector always
8    for(int i = begin; i < end; ++i )
9      C[i+k] = val * A[i] + B[i];
10 }
```

the result. Eventually, the Listing 4.4 sketches an implementation made with AVX-512 intrinsics.

We can observe that the kernel's implementation using intrinsic requires much more effort than the two other ones. Moreover, in this case we do not consider the case where `size` is not multiple of 8, which requires `mask` operations (but we implemented it in the API). Hence, it is not so easy to implement a method with intrinsics.

4.3.3 Experiment on *Axpy* kernel

Most of the operations coming from linear solvers are BLAS level 1 or 2. Hence, these functions operate on vectors, and they do not need any changes. Let's take an example of an *Axpy* kernel such as proposed by our API in Listing 3.1. The function browses two vectors at a time, and store the result in another vector. Instead of browsing vector one element per step, the vectorized version will operate several entries per vector and per instruction. To illustrate the gain we can obtain from vectorization on simple kernels, we vectorize an *Axpy* kernel with different methods: no vectorization, auto-vectorization given from compiler, intrinsic and the Intel MKL. Thanks to the API we develop, we benchmark it

Listing 4.3 – *Axpy's taks MKL*

```
void task_axpy_mkl(double* __restrict__ A,
                   double* __restrict__ B,
                   double* __restrict__ C,
                   double val, int begin, int end)
{
  cblas_daxpy((end-begin), val, A, 1, B, 1);
}
```

Listing 4.4 – *Axpy's taks Intrinsics*

```
void task_axpy_intrin(double* __restrict__ A,
                      double* __restrict__ B,
                      double* __restrict__ C,
                      double val, int begin, int end)
{
  int i;
  __m512d a,b,c,alpha;
  alpha = _mm512_set1_pd(val);
  #pragma novector
  for(int i = begin; i < end; i += 8)
  {
    a = _mm512_load_pd(&A[i]);
    b = _mm512_load_pd(&B[i]);
    a = _mm512_add_pd(_mm512_mul_pd(a, alpha), b);
    _mm512_store_pd(&C[i], a);
  }
}
```

with the OpenMP 4.0 runtime system. All the data are allocated in the MCDRAM memory bank. The application is compiled with Intel v.2017 compiler. Vector are partitioned through following a row partitioning. We select the number of rows to be a multiple of the vector register width. Results are gathered in Table 4.1, in GFlops units, for an input vector with $N_{Rows} = 10.10^6$.

# threads	Auto-Vectorization	MKL	Intrinsics
1	0,96	0,96	0,96
16	8,73	8,75	8,09
32	16,75	16,80	15,60
64	25,71	25,79	25,03

Table 4.1 – *Various vectorized ways for Axpy kernel (GFlops)*

As we can see, both implementation succeed to take advantage of AVX-512 instruction set. Between the various implementations, results are equivalent. However, we show in the previous subsection that the effort to write each of them is not the same. In particular for intrinsic kernel's version which requires a huge programming effort.

4.4 Adaptation of sparse structures to vector processing

Dense structures do generally not need any data reorganization for efficient programming on vector processors. At the opposite, sparse data formats suffer from a lack of performances for vector processing. Most of the widespread sparse structures does not optimize entries accesses pattern. Hence, operations related to sparse matrices does not fully benefit from vector processing and performances are generally quite moderate.

4.4.1 Comparison and discussion on sparse matrix format

Until now, sparse matrices are stored via the Compressed Sparse Rows (CSR) format [Williams et al., 2009] in our interface. This format is generally favored because it minimizes memory footprint while storing only non-zero entries (i.e. N_{nz} values). It is thus used in most of the well-knows sparse linear algebra software as Eigen [Guennebaud et al., 2010], PetSc [Balay et al., 1997] or Sparse BLAS [Carney et al., 1994] in the Intel MKL [Intel, 2017]. However, this data structure is not well adapted to vector processors. It may be particularly inefficient when a matrix has few non-zeros values per row. If many computations are done on smaller number of entries than the vector register sizes, the instruction throughput will be under used. If this phenomena is repeated many times per row, and the matrix is potentially big too, the vector processing will be inefficient.

Let's take an example of the product of a sparse matrix by a vector, which is one of the basic kernel in linear algebra. Given a sparse matrix as it is shown in Figure 4.2, its corresponding access pattern of CSR-formatted matrix's entries is then illustrated in Figure 4.3a.

Ellpack (ELL) [Grimes et al., 1980] is another format which is specially designed for vector processing. Matrix entries are stored following a column major fashion, and computations are then packed on vector register size. In order to build such a format, we first look at the largest row in the matrix. Then, all the rows are padded to this width. It is thus assumed that the padding is negligible regarding to the gain obtained by vectorization. Nevertheless, this format may generate many more entries in the matrix and degrade both the memory footprint and the performances. It is true especially for matrices in when there is too much differences between the widths of the rows. Let Avg_{rows} be the average width of all the rows of a sparse matrix, and N_{max} be its widest row. It is thus easy to see that if $N_{max} \gg Avg_{rows}$, then we will add a lot of zero entries. Hence, the matrix will consume more memory but also the application will perform much more operations. An example is given in Figure 4.3b, where $N_{nz} = 54$, $N_{max} = 8$ and $Agv_{rows} = 3.375$. In this case, we add 74 elements which correspond to an increase in 137% of additional entries.

Sparse matrices arising from oil reservoir simulation are unstructured. It means that we cannot know in advance the shape of the matrix, and if its values are well balanced among rows or not. So we are looking for a format that enables efficient code vectorization while minimizing the number of padded values, whatever the input matrix. The Sell-C-σ [Kreutzer et al., 2013] storage format limits the memory footprint while keeping a vector friendly access pattern. The format is based on the Ellpack we previously described, while limiting the number of padded zero values. In a first step, we sort rows by width in descending order per packets of size σ in order to reduce the fill-in in the chunks of size C. If we take a large value σ, sorting rows will be time consuming if N_{Rows} is high but we are sure to minimize the fill-in. A strategy is established to find a good value for σ in [Kreutzer et al., 2013]. Then, matrix coefficients are stored in a column major fashion. However, rows are stored by chunks of size C. This argument generally corresponds to the number of elements that can be processed per instructions. Indeed, it specifies the number of rows (one element per row) that will be processed per instruction. All the rows in a chunk have the same width. We insert padding to the rows which have a smaller width than the largest one of the chunk. As the rows were first sorted by the rows' width, we thus intend to limit the padding. Two different configurations of the format are illustrated in Figures 4.3c and 4.3d. It respectively corresponds to an increase in 70% and 41% of additional entries from the number of non-zero values.

All the main information about sparse matrix representations are summarized in the Table 4.2.

Feature	CSR	Ellpack	Sell-C-σ
Access pattern	Row Major	Col. Major	Col. Major
Vector friendly	No	Yes	Yes
Padding	No	Yes	Yes
Padding Limitation	No	No	Yes

Table 4.2 – *Sparse Format summary*

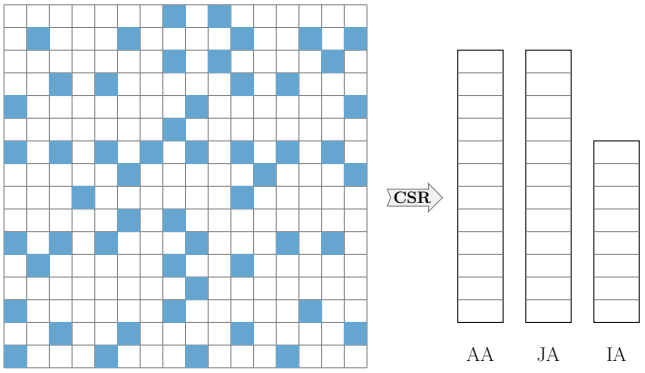


Figure 4.2 – *Example of any sparse matrix*

4.4.2 Switching to specific matrix format for a target architecture

Thanks to the structure of our API and its functions based on C++ templates, developers only need to write new matrices types and the corresponding operations to be integrated. Changing one element can thus have a performance impact, while keeping the same semantic. The Sell-C-σ format, which is supposed to have greater performances on KNL processor, can thus be selected by user when implementing an input matrix. The matrix is then passed as argument to all the API functions. End-users does not have to care about the internal structure, as all operations have been developed.

Until now, we only have to manage one structure per data at execution time. However, because of the API's capability to have several representation for one data, at run time two different tasks may operate on the same data but with various structures. The way to fix this issue mainly depends on the runtime systems we use in the API. With HARTS, we previously seen in Section 3.1.3.2 that we built the DAG according to the linear system's graph partitioning. In that case, we do not care about data structure we use. We can thus have various data representation at run time with the HARTS API's implementation. However, our API can take advantage of other runtime systems as the ones based on the OpenMP task programming interface. With these tools, tasks dependencies are built regarding pointers specified in the `depend` clause of the task construct. If we use several structures, pointers will thus be different and the dependencies will not be correctly expressed. To fix this issue, we propose to create a new structure that gather all the representations of a data. At task dependencies expression (i.e. the `depend` clause), we will thus give pointer to the factorized structure. By this way, we will homogenize all the pointers to build the DAG and we will not generate data race at run time. Until now, we do not face to this situation and does not require such a mechanism. However, we expect to implement it soon to prevent developers from this issue.

4.4.3 Experiments

This subsection is devoted to the performance evaluation of the Sparse Matrix Vector product kernel we implement through the API. This operation impacts the overall execution

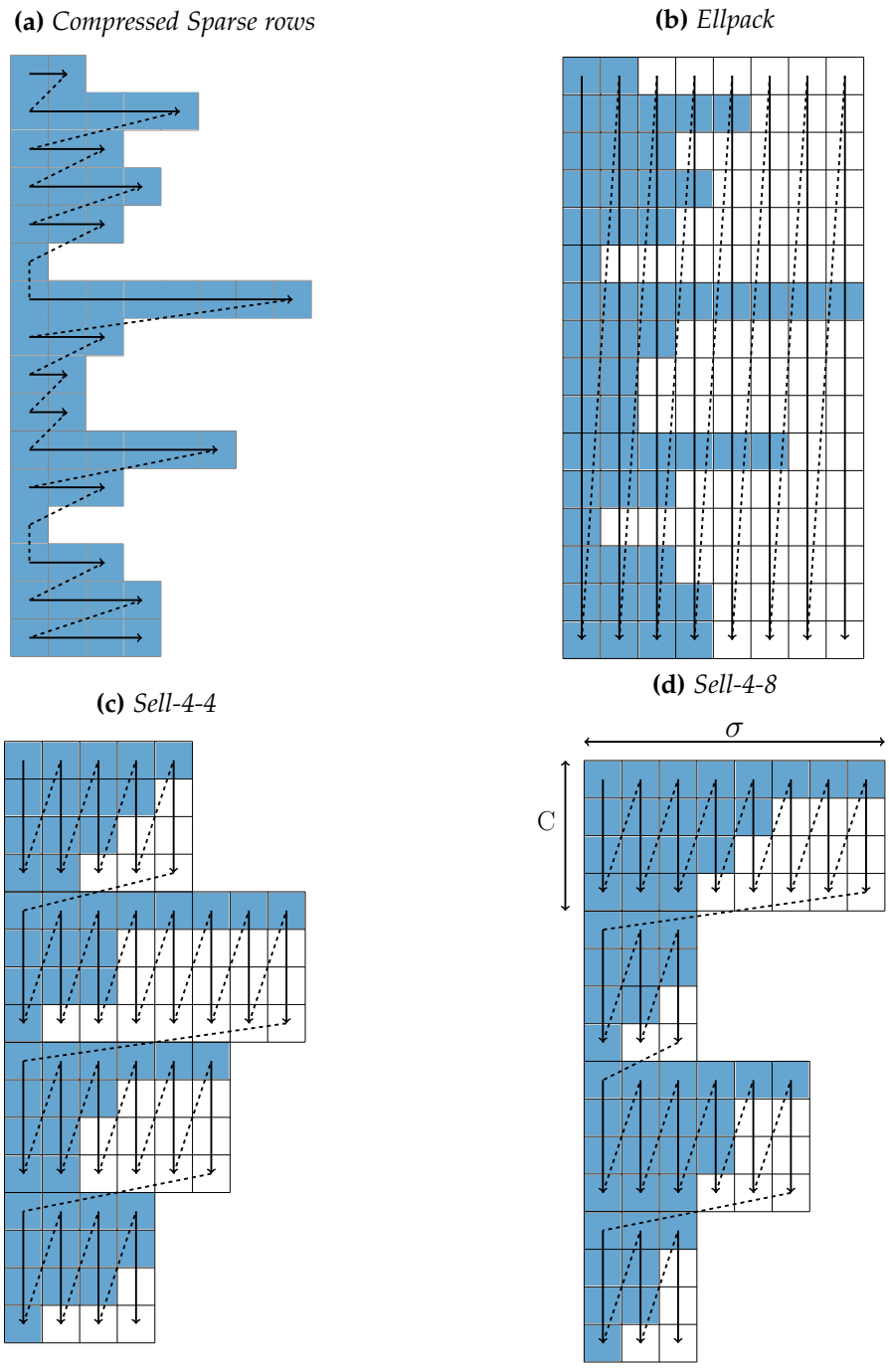


Figure 4.3 – *Various sparse matrix format for vector processors*

time of numerical iterative methods.

For both experiments, we do not reorder the matrix via the Sell-C-σ (i.e. $\sigma = 1$). By this way, we still take advantage of fill-in reducing face to Ellpack format and we avoid to reorder all the other data structures too. We initialize the matrix structure with chunks of width $C = 16$. By this way, we can load 16 indexes of the input vector per SpMV iterate. We thus decompose the iterate in two blocks of 8 floating points numbers in double precision.

We first present the improvement we can attempt from a vectorization of the SpMV using a single vector processing unit with various sparse matrix representations. Then, we evaluate the potential parallel gain using the Sell-C-σ format while varying sizes. Eventually, we propose a comparison between two parallel executions with two distinct sparse formats.

4.4.3.1 Vectorization benefit from various sparse format

This test is devoted to the performances comparison of SpMV using CSR and Sell-C-σ format. For the CSR format, we benchmark two various implementations. The first one is the one we implemented with auto-vectorization compiler hints, while the second one is provided from the Intel Sparse MKL library. The Sell-C-σ multiplication comes from a vectorization made with AVX-512 intrinsics. The results we obtained from both implementations with different sizes of Laplacian matrices are reported in Figure 4.4.

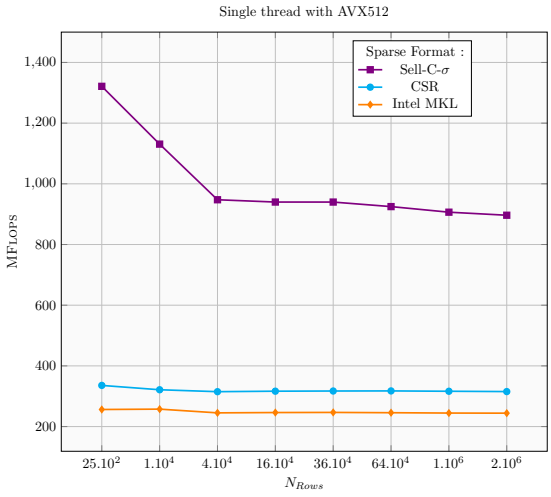


Figure 4.4 – *Performances on SpMV implementations on 1 thread*

We can observe that the SpMV kernel with the Sell-C-σ format outperforms the two other implementations using CSR format. With these later implementations, we report performances of around 300 MFlops regardless the matrix size. However, we reach a peak performance of around 970 MFlops with the Sell-C-σ implementation with the smaller benchmark. The performances seems to be stabilized to around 900 MFlops when the matrix size grows up.

4.4.3.2 Multi-threaded execution

In order to illustrate the benefit we can attempt from vectorization in sparse matrices computations, we now compare two SpMV distinct parallel executions on a KNL processor. The first one is computed thanks to the widespread CSR format. The second SpMV's execution is performed thanks to the previously presented Sell-C-σ sparse format. Each implementation benefit from vectorization using AVX-512 intrinsic. We run both of them on several threads, from 1 to 64. We then collect the results in terms of time and floating point operations per second.

(a) *CSR* **(b)** *Sell-C-σ*

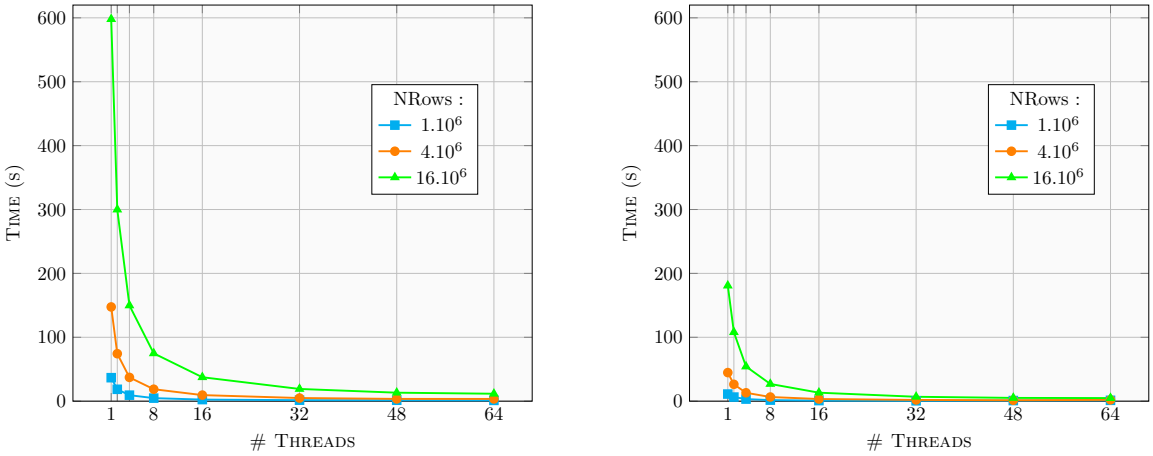


Figure 4.5 – *SpMV's time results*

Time results for CSR format are presented in Figure 4.5a, while those for Sell-C-σ are illustrated in Figure 4.5b. If we only focus on these times, we can first stare on the results with only one thread. Instantly, we can observe that the CSR format achieve around 3 times worst performances than its concurrent on the biggest input matrix. However, this case is not unique and also works for all the other tested matrices. The conclusion remains the same while increasing the number of threads we use. For the biggest matrix, the CSR format execution based performed with 64 threads ends the execution after 12 seconds. The same execution with the Sell-C-σ ran during around 5 seconds.

We have now a look on operations performed per second. The CSR format results are sketched in Figure 4.6a while the ones for the Sell-C-σ are presented in Figure 4.6b. We can observe that results for the CSR format have difficulties to grow up while increasing the number of threads we use. Indeed, the performances does not exceed 15 GFlops. At the opposite, the Sell-C-σ executions does not have these difficulties, and reach up to 33 GFlops of performances. On the other hand, we can observe decreases from 32 and 48 threads for two smaller test cases. One hypothesis is that the data communications costs can not be hidden by computations. Indeed, computations are not too intensive to achieve it. In this case, performances are thus leaded by data retrieval more than computations. Until now, we do not have verified the veracity of this hypothesis.

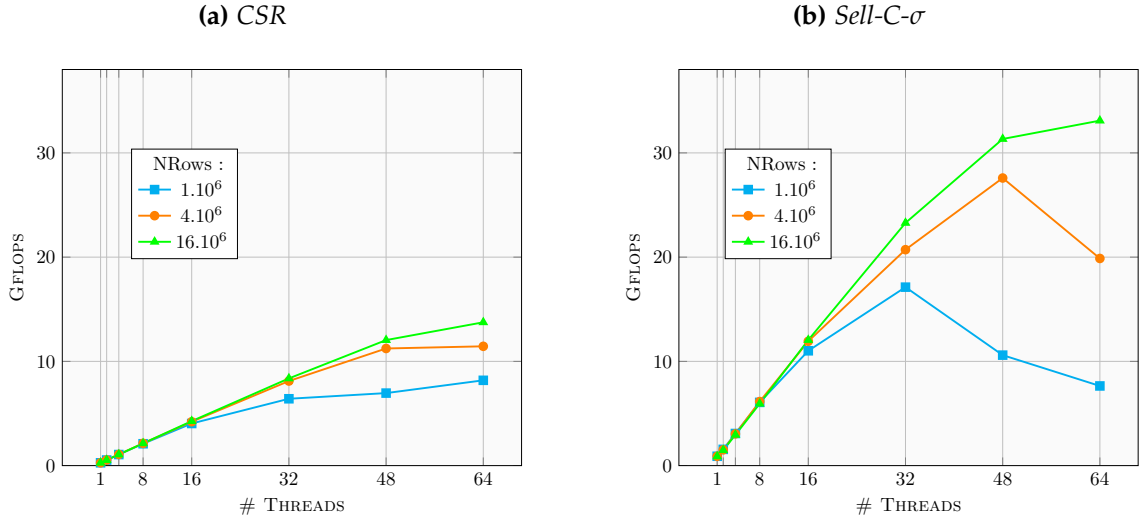


Figure 4.6 – *SpMV's Flops*

4.5 Flexible memory allocation for various memory banks

Memory bandwidth is considered to be one of the most critical bottleneck in HPC systems. It has been accentuated since the rise of accelerators in computing nodes. Stacked memory with a limited size is the option adopted by the KNL. However, the way to manage it efficiently is not so easy because of its limited size. Several configuration modes are available to attempt to take advantage of this memory, previously described in section 4.1.2. Moreover, clusters mode (see section 4.1.3) also have also an influence the way of reaching memory and eventually application performances.

4.5.1 Memory management depending on the memory mode

We previously explained in section 4.1.2 that it exists two major memory modes selectable at boot time. First, the *Cache mode* enables to use the high bandwidth memory as a last level cache. This mode does not require any particular attention from developer because data are allocated via standard functions like `malloc` or `posix_memalign`. The system is then responsible to the cache eviction policy. An application whose all the data fit in the 16Gb of the MCDRAM bank will not need any changes to increase its performances. If the amount of data is bigger than the capacity of the MCDRAM, data are not guaranteed to be inside. Data accesses could possibly slow down the application and thus degrade the performances.

At the opposite, the *Flat mode* offers distinct views between MCDRAM and standard DRAM memories. By this way, developers may have a finer control on data allocation. They can specify where the data must be placed, depending on their knowledge of the application. Hence, allocation in DRAM memory will be processed via a call to `malloc`. Memory allocation in MCDRAM will be addressed by a call to `hbw_malloc` function, given

by the Memkind library.

From Flat memory mode, two problems arise. First, how can we manage data allocation inside our API without users' understanding. Then, how can we decide in which memory bank a data should be placed.

4.5.2 Taking care of memory modes in the API

We previously seen in section 3.2.2.1 that the memory allocations are managed through the API via the `allocate` function. We generalize the `Allocator` concept to easily switch between various allocating functions. It is responsible to reserve memory space in the virtual address space, and free memory space too. By default, the `Allocator` object uses standard memory allocation functions (i.e. `mmap` and `munmap`). We now extend it to take in account the allocation in high bandwidth memory. Hence, we build an `Allocator` object that inherits from the default `Allocator` implemented for multi-core architecture. This new `Allocator` uses the functions provided by the *Memkind* library to allocate and free memory space in the MCDRAM memory bank (i.e. `hbw_malloc` and `hbw_free`).

At compile time, developers can select the `Allocator` object passed as an argument of the API. If nothing is specified, the default `Allocator` is chosen. We can also select the one based on the `hbw_malloc` function for KNL processor. However, the developer is responsible on performance optimization. Hence, he can decide to privilege an allocator if it deems it necessary. In that case, the developer has to instantiate a new `Allocator` object, which will be different from the one handled by the API. Unfortunately, it requires a good understanding of the application. Let's take an example of a heavy data which is needed only once. It can take place for other data that will be operated many times during problem solving phase. A smart strategy is thus to place data for initialization in DRAM memory. By this way, we can save enough place for other data in the MCDRAM.

Data distribution between the two kinds of memory is not easy. In our strategy, we allow developers to allocate memory as they wish while using `Allocator` objects. However, we can describe the strategy we employed in our API model. We distinct data that are used for structure initialization from data used for linear solver execution. Indeed, some data are only operated to initialize other data and then they are less or never used again. Let's take a simple example with matrices which represent linear systems. We first read data from file, and then partition it in several sub-domains. We thus handle two structures, but the non-partitioned one will not be used for parallel execution and takes a lot of memory. Storing this matrix does not guarantee that the partitioned system will have enough place in MCDRAM. We thus decide to store in DRAM memory the original matrix, and the partitioned system in MCDRAM. We then initialize the partitioned system from the original matrix. By this way we maximize chances to fit all the data in MCDRAM.
In this model, we do not consider data that do not fit entirely in MCDRAM memory bank. Until now, all the data we used fit inside. However, we are conscious that it will not be always the case. We thus consider the question in the discussion's section, see 4.7.

4.5.3 Experiment

In this context, we can wonder if the high bandwidth memory handled by KNL processor has a strong impact on SpMV performances. Hence, we run a parallel SpMV operation 10^3 times while using the Sell-C-σ matrix format. The cache is not flushed between the various iterations. We attempt to allocate memory in the two various memory banks of the KNL processor. First, data are physically set on classical DRAM memory. The benchmark is performed another time but data are now physically in the MCDRAM memory. For both execution, the KNL processor was configured in the SNC4 cluster mode. We compute the speed-up ratio between the execution where data are in MCDRAM and the new one by the formula: $R = T_{DRAM}/T_{MCDRAM}$. We thus plot the results in Figure 4.7 while varying the linear system sizes.

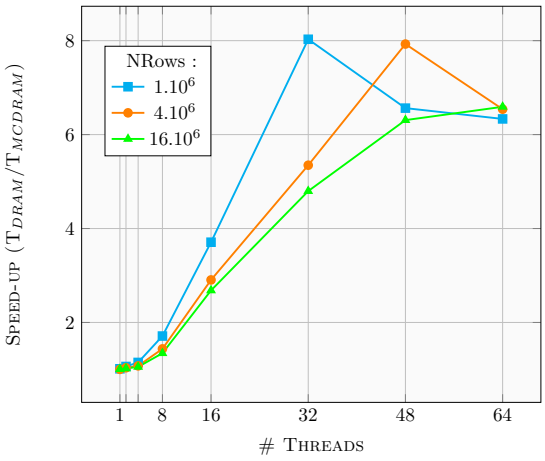


Figure 4.7 – *SpMV performances using the two memory banks of the KNL processor*

We can first observe that independently from the system size, the use of the KNL's stacked memory improve performances of the SpMV methods. The two smaller systems are able to reach up to 8 times more performances with the MCDRAM use compared to the classical DRAM memory use. We also observe decreases from respectively 48 and 64 threads on the two smaller benchmarks. In this case, we suppose that the data communication cost are high face to computations. The largest input case reaches up to 6.5 of speed-up ratio between MCDRAM and DRAM memory banks.

We now compare the two previous executions with flop per second measurements, on $p = 64$ cores. Results are sketched in Figure 4.8 while varying system sizes.

We can observe that performances of the SpMV execution where data are in the MCDRAM are increasing up to the 25.10^6 rows matrix. After this, we do not increase performances anymore. Hence, we have to deal with huge data to reach the best performances of the MCDRAM memory bank. In that case, SpMV performances reach up to 40 GFlops. The DRAM memory reaches the best performances earlier, since matrices of size $N_{Rows} = 16.10^6$. Moreover, slower memory bandwidth shows its limitations while achieving up to 5.7 GFlops of performances on the SpMV benchmark.

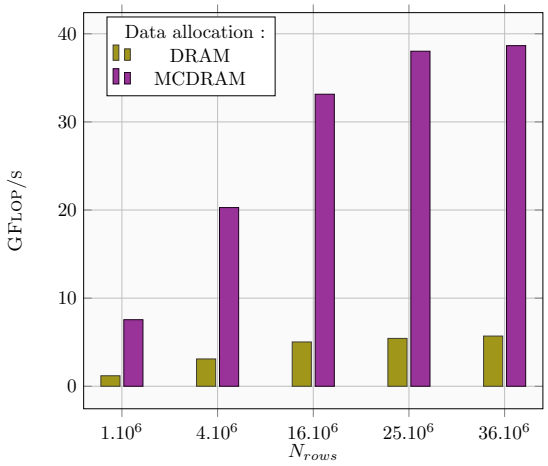


Figure 4.8 – *Performances on SpMV benchmarks using KNL memory banks*

4.6 Experiments on sparse iterative methods

The overall objective of our initiative is to take advantage of linear solver on many-core systems. All the extensions we proposed in the previous sections impact the performances of solvers we wrote through the API we develop. To benefit from these extensions, it is fully transparent for end users. BLAS operations are performed via the Intel MKL library. We use the Sell-C-σ format for matrix operations. All the data required to perform the iterative methods are allocated in the MCDRAM memory.

This section is devoted to the performance evaluation of the linear solver we developed with various preconditioners. As for the multi-core systems, we benefit from performance counters (see section 3.3) and statistics we extract at execution time to upgrade the understanding of our results.

For all the experiments, we use the matrices M_{Lp} coming from the Finite volume discretization of a 2D Laplace problem with various mesh sizes. Although simpler than in realistic oil reservoir simulations, the size of these systems can be easily tuned.

For these experiments, we used a 64 cores KNL processor which has 192 Gb DRAM memory [2]. The many-core processor is configured with the 4 Sub-Numa Cluster mode. The 16 Gb of MCDRAM memory are exposed according to the Flat memory mode. As explained in [Jeffers et al., 2016], we turn off the Turbo mode to benefit from the maximal performances when using vector intensive computing. Indeed, it can generate many frequency variations and thus degrade performances. The Hyper-threading mode is enabled, but we do not experiment hyper-threading here (See section 4.7). By default, we pin the first processing unit of the core in the appearance order through the Hwloc library.

2. Thank you to the LIP laboratory and the INRIA's AVALON Team to grant me the access to this machine, which is part of the *crunch* machines

4.6.1 BiCGStab

4.6.1.1 Performance evaluation

We first intend to evaluate the parallel implementation of the BiCGStab algorithm on a KNL processor. Given various linear systems with different sizes, we run the iterative method on a sub-set p of processors. First, we extract parallel times of system resolution and illustrate it in Figure 4.9a. From time results, we then compute the speed-up ratio and illustrate it in Figure 4.9b.

(a) *Time* **(b)** *Speed-Up*

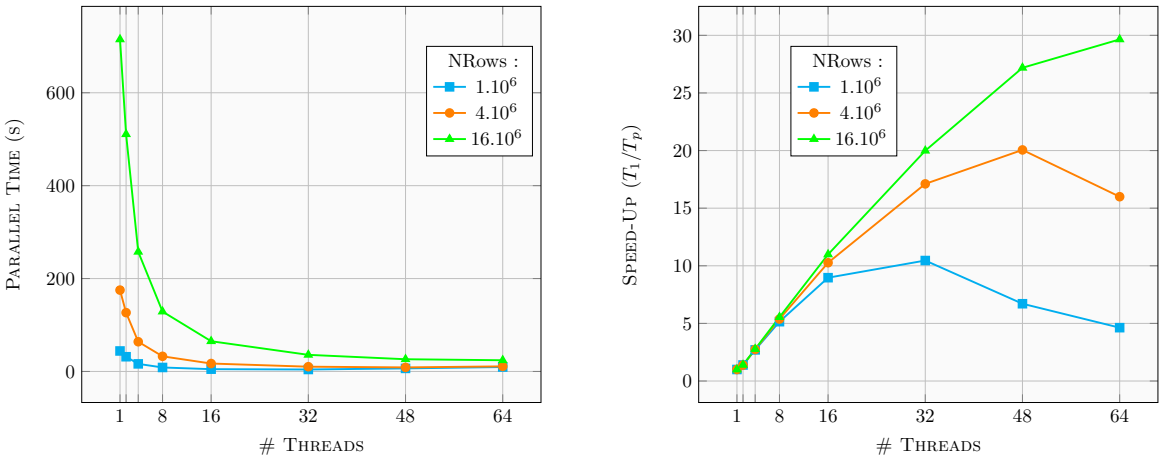


Figure 4.9 – *BiCGStab parallel performances*

We can observe that for all the systems, the time is decreasing while the number of threads increases. For the two smallest matrices, speed-up ratio is decreasing respectively from 48 and 64 threads. We can thus deduce that increasing the performances is relative to the size of the system. For the smaller test cases, data communication from memory bus degrade the performances when data are too small.

We can now have a look to the performance evaluation through the use of counters we implement. The number of operations performed per second is thus illustrated in Figure 4.10a, while the computational efficiency is shown in Figure 4.10b.

As we previously saw, performances are decreasing for the two smallest benchmark, from respectively 48 and 64 threads. A peak of performance of 33 GFlops is reach for the bigger system we use. Its computational efficiency decreases down to 81%. For the smaller benchmark, which most suffers from data communication, efficiency reach around 50% of efficiency with the maximal number of threads.

(a) *Operations per second* **(b)** *Computational Efficiency*

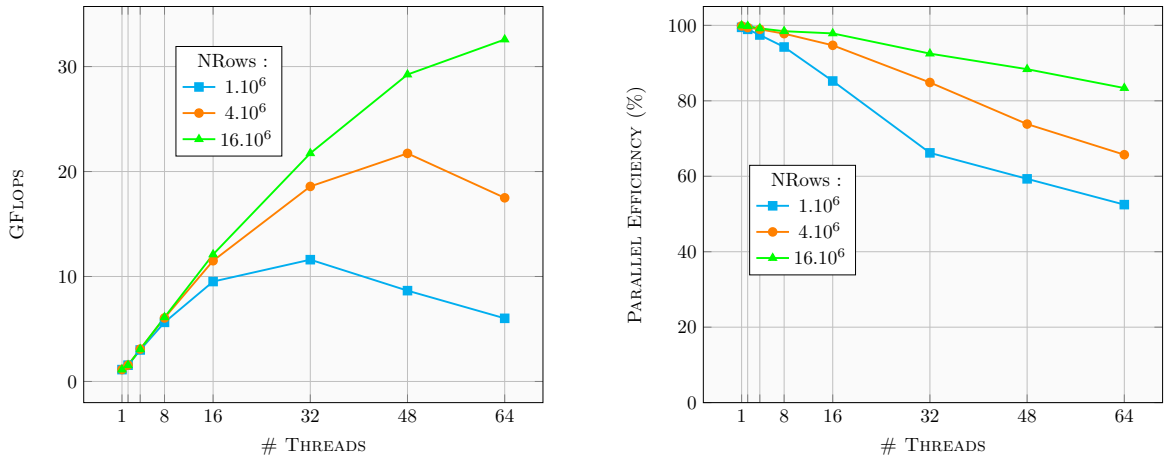


Figure 4.10 – *BiCGStab parallel performances*

4.6.1.2 Cluster mode impact

We now aim to compare the performances of the previously presented BiCGStab method while configuring the KNL processor with various cluster modes. We thus configure the same system with modes among All-to-all, Quadrant and 4 Sub-NUMA clusters (SNC4). We present results while varying the size of the input system in the Figure 4.11.

(a) $N_{Rows} = 4.10^6$ **(b)** $N_{Rows} = 16.10^6$

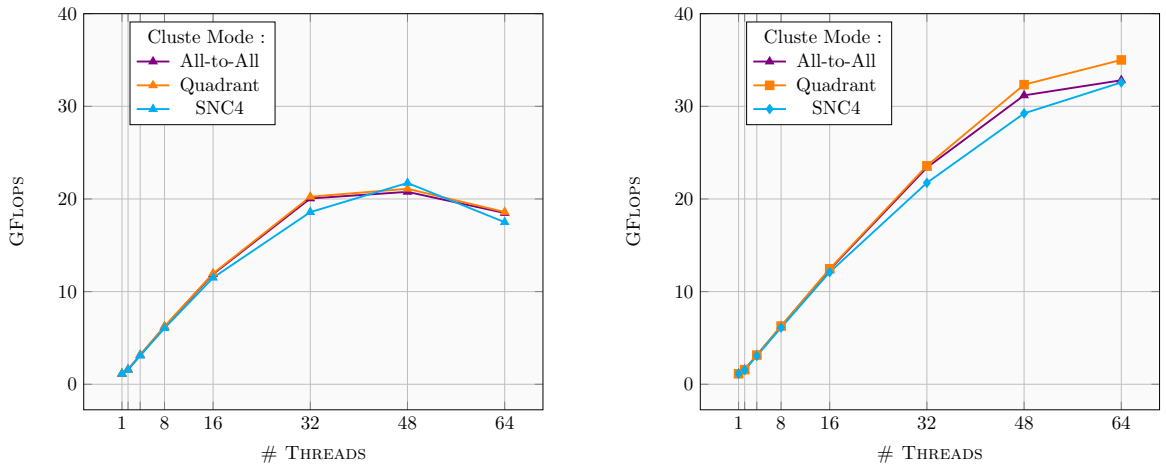


Figure 4.11 – *Comparison between cluster modes of the KNL on a BiCGStab algorithm*

We can notice that there is no significant differences between each cluster modes on the BiCGStab method's performances, regardless the size of the input system. For more convenient way, we configure the KNL processor with the 4 Sub-Numa clusters mode as we have previously made optimizations on NUMA architectures.

4.6.2 Preconditioned BiCGStab

4.6.2.1 Polynomial preconditioner

We now focus on the performance evaluation of the BiCGStab method with a polynomial preconditioner. Our purpose is to know if our parallel implementation via the API we developed presents some good properties on the KNL many-core processor. As it been mentioned before in the previous chapter (see section 3.4.1), the polynomial preconditioner performances mainly relies on the SpMV performances. Thanks to a metis graph partitioning, we perform parallel execution of this benchmark on several input matrices of different sizes while varying the number of threads. Time results are illustrated on the Figure 4.12a, while speed-up ratio evolution is shown in Figure 4.12b.

(a) *Time* **(b)** *Speed-Up*

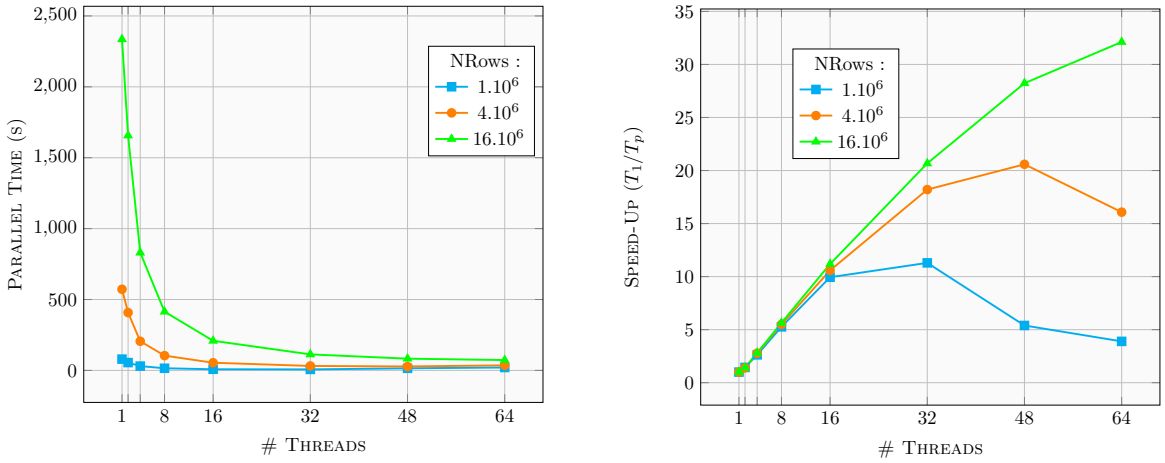


Figure 4.12 – *BiCGStab w. polynomial precon. parallel performances on a KNL processor*

Whatever the input matrix, time results are decreasing while the number of threads increases. On the other hand, we can see that speed-up ratio is increasing for both input matrices up to 32 threads. Beyond this step, the speed-up ratio of the smaller system is decreasing. The intermediate sized system presents the same behavior from 48 threads. The bigger matrix does not suffer from this phenomena, and reaches up to 33 of speed-up with 64 threads. The proportions are not the same, however the results have the same behavior as the SpMV parallel performances previously presented in section 4.4.3.2.

Now, we can have a look at other measurements such as operations operated per second or computational efficiency. These metrics are collected at run time by HARTS runtime system, with the methodology previously described in section 3.3. Results are gathered in Figures 4.13a and 4.13b.

We can first observe the same phenomena as speed-up ratio evolution curves as for the performances measured in GFlops. The largest input system reaches a peak performances of around 32 GFlops. On the computational efficiency results, we observe that the bigger

(a) *Operations per second* **(b)** *Computational Efficiency*

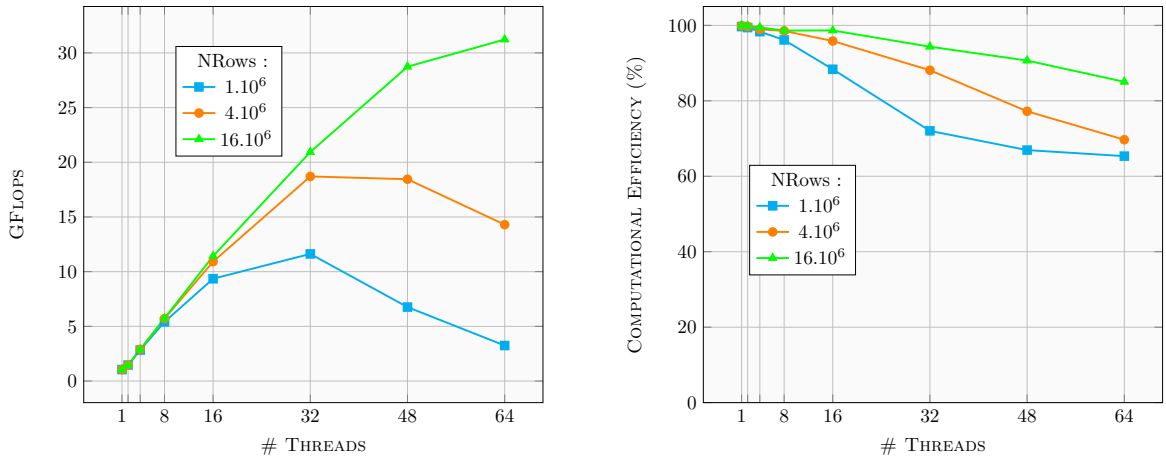


Figure 4.13 – *BiCGStab w. polynomial precon. performance counters on a KNL processor*

benchmark reaches the best efficiency among all the input systems. The computational efficiency's curve related to this system is slower decreasing than the two others.

As we only vary sizes between the executions, we can thus deduce that performances depend on the system size. Data are partitioned according to the number of threads (1 partition per thread). We can thus suppose that partitioned data become too small for the smaller benchmark when we use too much threads. In this case, computing effort may be insufficient face to data communications. It can explain why smaller data are much more impacted than the bigger ones.

4.6.2.2 ILU preconditioner

We now experiment the parallel implementation of the ILU(0) preconditioner on the KNL processor. We previously introduced the parallel implementation of this method in section 3.4.2. We benchmark linear systems coming from the finite volume discretization of the Laplacian problem with various mesh sizes. We first illustrate time results in function of the number of threads we use in Figure 4.14a. We compute the speed-up ratio by the formula $S = T_1/T_p$, and present our results in Figure 4.14b.

In the first hand, we can notice that time is decreasing while the number of threads increase, regardless of the linear system size. We can observe on speed-up curves that the performances decrease from 48 to 64 threads for the smallest benchmark. For the other benchmarks, we reach a speed-up ratio more than 20.

We now looking at the computational efficiency, which measures the efficiency regarding the computational effort (see section 3.3 for more details). Results are sketched in Figure 4.15a.

We can observe that computational efficiency sharply decrease while the number of

(a) *Time* **(b)** *Speed-Up*

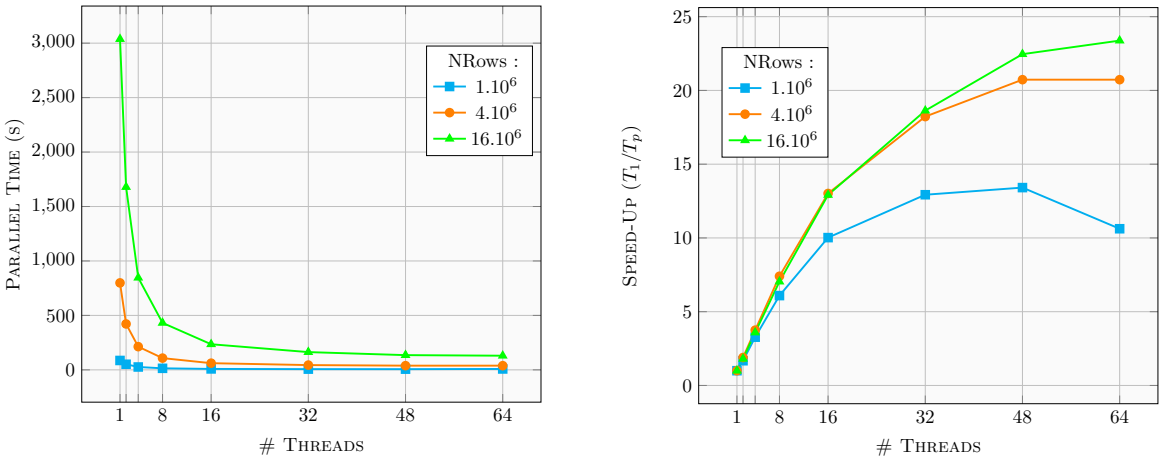


Figure 4.14 – *BiCGStab w. ILU precon. parallel performances*

(a) *Computational Efficiency*

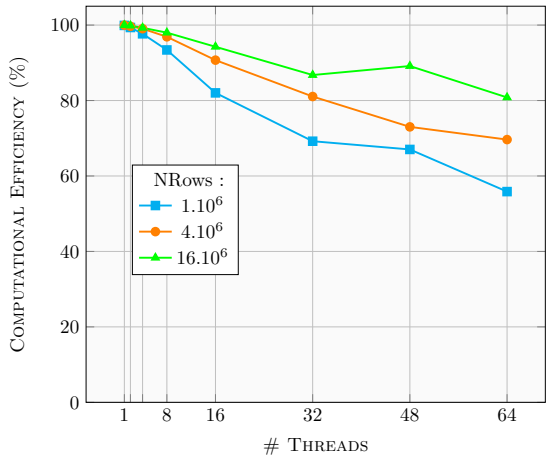


Figure 4.15 – *BiCGStab w. ILU precon. parallel performances*

cores in use increase, regardless the size of the system. In the best case, performances are decreasing up to 80% for the bigger system. At the opposite, the smallest benchmark reach around 60% of efficiency. This inefficiency corresponds to the work imbalance caused by the algorithmic structure of the method. It has already been sketched for the multi-core implementation via the API we develop in Section 3.4.2.

4.7 Discussion

Until now, we have only focused on the problems we encountered on the KNL many-core processor and their integration in the API we develop. However, much more work is

ongoing in order to expect better performances. This section is devoted to expose a list of the bottlenecks we need to take in consideration for the future. We first have a look on the MCDRAM memory management when we face big data. We then attempt to benefit from hyper-threading, that could provide 4 times more hardware contexts.

4.7.1 Distributed Shared Memory for MCDRAM

As previously seen in section 4.5, we only focus our work on data that entirely fit in the MCDRAM memory. However, it is not always the case. Indeed, applications that work on large data sometimes require more memory than offered by the MCDRAM memory. A lazy solution is to setup the KNL processor with Cache memory mode which offers a cache hardware management. However, we do not know the cache eviction policy offered by this mode.

One of the solution previously adopted in heterogeneous computing was to implement a software cache. X-Kaapi [Gautier et al., 2013] proposed a software cache based on a *Least Recently Used* (LRU) policy. They managed the software cache to avoid intensive data communication from CPU to GPU. If a data is not present in the cache, memory will be allocated and then transferred to GPU. This model is also based on the asynchronous memory transfers to prepare data before processing.

Another way was introduced by [Augonnet and Namyst, 2008] which is based on sub-data manager. As accelerators only operate on a piece of data at a time, they do not need to fit all the data in limited memory. By this way, they avoid to waste memory with unused data. They enforce a cache software to manage data movements between CPU and GPU while ensuring coherency through a MSI protocol.

We expect that a similar approach can be used for the MCDRAM management. Contrary to GPU, the KNL processor can operate on data that are not present in MCDRAM bank. Choosing between the two memory banks of the KNL is only a performance issue.

We expect to manage this issue at several layers. Until now, we have already operated on sub-partitions given by the Partitioner objects of the API. However, we do not guarantee that data related to one partition fit in a specific memory bank or not. We can thus extend the Partitioner objects to take care of memory bank with limited size. By this way, we will split data in which a sub-partition data set can fit in high bandwidth memory. At runtime system level, we can make several suggestions. First, a software cache based on the LRU policy. This tool will be responsible to allocate, deallocate and move data. In addition, it ensure the coherency between the memory banks. At scheduler level, we can imagine a scheduling policy that favors work with data that are allocated in the higher bandwidth memory. By this way, the scheduler ensures that the execution benefits from the best memory latency with the knowledge acquired from the system. If a data is not present in the cache, the scheduler may be advised from a performances prediction model. By this way, it can thus decide if it is necessary to retrieve data from a memory bank to another. Depending on the data sizes, these hints according to system information can be used to

evaluate the benefit to move a data. If a data is too small, the scheduler will not decide to move data. If not, the work will wait for the software cache manager to move data.

4.7.2 Hyper-Threading

We previously introduced hyper-threads in section 4.1.1. We mentioned that a KNL's core is composed of four hardware contexts, and two VPUs. Until now, we have taken advantage of one thread per core. On the later generation of Intel Xeon Phi product, authors of [Jeffers and Reinders, 2013] highly recommend to benefit from two hyper-threads per core to take advantage of all the core resources.

As a KNL's core handles two VPUs, we can thus expect to benefit from two times more performances while pinning two threads per core. We can thus expect to take advantage of hyper-threading through our linear algebra API. At first glance, two conditions are supposed to be fulfilled. First, vector intensive applications are required to feed enough the two vector processing units per core. Then, tasks granularity has to be enough small to not perturb L2 cache share between the two cores of a tile.

Unfortunately, until now we made some experiments that does not have a great success. The ongoing work will study the feasibility of taking advantage of hyper-threading. However, the authors of [Jeffers et al., 2016] already mentioned that most of the applications reach the best performances with one thread per core.

4.7.3 Preconditioners

Until now, we have demonstrated the scalability performances of the API both on multi-core and many-core systems. To reach this purpose, we implemented classic numerical methods as polynomial or ILU preconditioners. By this way, we succeed to evaluate our work in a convenient way, on concrete implementations.

However, these methods show their numerical limits on ill-conditioned linear systems coming from the oil reservoir simulations. Other emerging methods seems to be more robust while their performances scales up to hundreds of processing units. Thanks to these methods, and our API, we are confident to provide robust and efficient parallel methods on various architectures to API's users.

Chapter 5

Domain Decomposition Methods

Contents

Linear systems coming from oil reservoir simulations are ill-conditioned because of their unstructured pattern and coefficients heterogeneity. Until this chapter, we have only focused our work on the way to write efficient and portable algorithms on various architectures. We have evaluated our approach on well-known preconditioners such as ILU. However, these methods suffer from numerical and/or scalability issues. Two important numerical properties are required for preconditioners while facing ill-conditioned systems: robustness and extensibility. A preconditioner is said to be robust when it enables to converge in a reasonable number of iterations. Moreover, if the convergence rate does not depend on the size of the system, so the preconditioner is said to be extensible.

Preconditioners such as Algebraic Multigrid (AMG) [Brandt et al., 1984, Stüben, 2001] method are well-known to be robust and extensible. However, AMG suffers from strong scaling issues. It requires a huge programming effort to be parallelized efficiently on multi-core systems [Feng et al., 2013, Park et al., 2015, Baker et al., 2011].

Domain Decomposition Methods (DDM) are methods that are naturally parallel. More recently, these methods regain in interest because of the popularization of massively

parallel architectures in simulation field. However, these methods have suffered from a lack of numerical robustness until recent works on the addition of the 2-level methods [Spillane et al., 2014, Dolean et al., 2015]. Thanks to the divide-and-conquer strategy, DD methods achieve highly efficient parallelism on massively parallel machines [Jolivet et al., 2013].

Our concern is here to propose a task-based parallel implementation of a multi-level domain decomposition preconditioner written through the API we have developed. The API targets both multi-core and many-core systems. Hence, we aim to prove the parallel efficiency of our implementation regardless the architecture in use. We also compare our performances to the AMG preconditioner.

5.1 Overview and motivations

Different classes of DD methods exist. Among them, substructuring methods aim to split the whole system in distinct subdomains without overlaps contrary to the DD methods with overlap. Alternating Schwarz methods [Schwarz, 1870] are part of the overlapping domain decomposition methods. However, some alternative methods offer to avoid overlap as [Gander et al., 2002].

In this section, we focus on the Additive Schwarz method which is well-known to be easy to parallelize. We aim to highlight the numerical performances of such methods on problem coming from oil reservoir simulations. In the first subsection, we focus on the classic Additive Schwarz Method and on the 2-level methods. In the second subsection, we present DDML numerical performances we obtain on systems coming from oil reservoir simulations.

5.1.1 Multi-Level Domain Decompositon method

5.1.1.1 1-level method Additive Schwarz Method

Additive Schwarz method (ASM) [Schwarz, 1870] is a fully parallel domain decomposition method. The main principle of such methods is to split the whole domain in smaller ones. The system solution is therefore computed from the solutions on smaller and easier problems on each sub-domain. However, this method suffers from a lack of numerical robustness and extensibility. The number of iterations required to converge depends on the number of sub-domains. It is mainly due to the fact that at each iteration data are exchanged only between a neighbors sub-domain.

We here focus on the method with no overlapping sub-domains. We consider the ASM algorithm as linear solver of the equation:

$$Ax = b, \tag{5.1}$$

where A is the coefficient matrix, b the right-hand side vector and x the unknowns vector.

The Additive Schwarz method is thus given by:

$$u^{m+1} = u^m + M_j^{-1}(b - Au^m), \tag{5.2}$$

where M_j is the Jacobian matrix, composed of block diagonal elements of A. Given the partitioning of A following the graph partition. We consider the matrix A_{int}^k be the internal matrix belonging to the partition k, and A_{ext}^k be the matrix of external contribution from the partition k. Internal matrices refers to the set of elements Vi_k for a given partition k coming from data partitioning (previously presented in Section 3.1.2.1, on page 40). At the opposite, external matrices are built from the boundary vertices Vb_k of the partition k.
We can thus rewrite equation 5.2 as:

$$u^{m+1} = u^m + A_{int}^{-1}(b - (A_{int} + A_{ext})u^m) \tag{5.3}$$

$$= A_{int}^{-1}(b - A_{ext}u^m). \tag{5.4}$$

From equation 5.4, if we take $u_0 = A_{int}^{-1}b$, we can thus obtain the algorithm illustrated in Algorithm 5.1.

Algorithm 5.1: ASM Algorithm

```
x = 0;
do
    x̃ = A_int^{-1} b;
    x = x + x̃ ;
    b = −A_ext x̃;
while (||b|| < ε);
```

At each step of the algorithm, we have to solve a linear system given by $A_{int}\tilde{x} = b$, and a matrix vector product. All of these steps can be performed independently in parallel.

5.1.1.2 2-level Additive Schwarz Method

This problem has been fixed by the introduction of a two-level method via a coarse space correction. A coarse small problem is added to couple all the sub-domains at each iteration.

The coarse grid correction is given by the equation:

$$u^{m+1} = u^m + ZE^{-1}Z^T(b - Au^m), \tag{5.5}$$

where E is the matrix corresponding to the discretization of the initial equation on the coarse grid, and Z is restrict operator from the fine to the coarse grid. The matrix Z is called the deflation matrix, while the $m \times m$ matrix E is the coarse operator. The linear system to solve involved by E in Equation 5.5 as $m \gg N_{Rows}$. The 2-level Additive Schwarz algorithm corresponding to the Equation 5.2 is given in Algorithm 5.2.

Algorithm 5.2: 2-level ASM Algorithm

```
x = 0;
do
    b̃ = A_ext Z^T E^{-1} Z b;
    b = b − b̃;
    x̃ = A_int^{-1} b;
    x = x + x̃ ;
    b = b̃ − A_ext x̃;
while (||b|| < ε);
```

At each iteration of the method, two linear systems are solved. From the 1-level method, a second system is induced by the coarse operator E. It has to be solved to couple all the sub-domains. It is a quite moderate sized system as m is supposed to be small regarding the original system size. However, the resolution is repeated at each iteration. Each variant of *Multi-level Domain Decomposition* (**DDML**) methods [Dolean et al., 2015] differs from the chosen coarse operator.

Nicolaides coarse operator [Nicolaides, 1987] is one of the most simpler coarse operator. It is based on the indicator functions per subdomain. Its construction is not complex, but is well-known to be inefficient face to ill conditioned systems.

From a linear algebra point of view, the stagnation of the convergence rate corresponds to a few very low eigenvalues in the spectrum of the preconditioned system. Our work is focused on the GenEO [Spillane et al., 2014] coarse operator, which identifies and incorporates the low frequency modes in the coarse grid construction. GenEO is based on a overlaped domain decomposition method. The coarse operator select, per subdomain, the first N_{ev} eigenvalues coming from an eigenvalue solved built in the overlaps of the domain decomposition. The size of E is then given by the formula: $Size(E) = N_{ev} * N_{part}$. By this way, it is theoretically proved that GenEO coarse operator's convergence does not depend on the matrix coefficients, but only on the matrix size. The coarse operator was originally designed for finite element discretizations. We have adapted the method to handle at the algebraic level matrices from application using Finite Volume discretizations. The setup phase of DDML preconditioner mainly relies on the construction of the coarse problem. Since the resolution of an eigenvalue problem is not trivial, this phase is time consuming.

5.1.2 Numerical performances

One of the main interest of the 2-level Additive Schwarz method is its ability to converge in a feasible time on complex linear systems. Given an appropriate coarse operator, the convergence rate of the DD method depends on it. This subsection aim to validate the numerical properties of the GenEO coarse operator which motivate this work. Hence, we study the convergence rate of various preconditioners such as ILU(0), AMG[1]

1. available from the Hypre library [Falgout et al., 2006]

and 2-level ASM with both GenEO and Nicolaides coarse operator. We benchmark these preconditioners on two classes of problems.

The first class presented is the discretization of the Laplace problem on a unit cube mesh. The second category is coming from the oil reservoir industry.

5.1.2.1 Laplacian systems

To prove the numerical robustness of DDML preconditioners for large heterogeneous problems, we have benchmarked various preconditioning methods with a matrix coming from the finite volume discretization of a Laplacian problem. The matrix size is set to $N_{Rows} = 10^6$. We have drawn the evolution of the residual error at each iteration to reach a 10^{-15} precision. The results are illustrated in Figure 5.1.

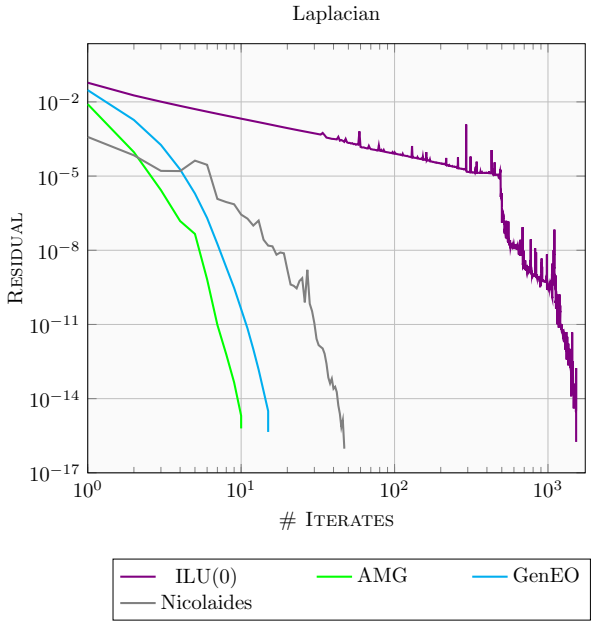


Figure 5.1 – *LP Homogeneous*

We notice that ILU(0) fails to converge in a reasonable time. For the three other ones, convergence is reach approximately at the same time. We can conclude that for easy problems as Laplacian systems, the DDML method with GenEO coarse operator is numerically robust and is as robust as the AMG reference method. We can also see that the DDML method with the Nicolaides coarse operator requires a bit more iterates to converge than AMG and the GenEO based DDML method.

5.1.2.2 Oil reservoir simulations

However, oil reservoir simulators does not deal with the Laplacian ones. To validate our approach on realistic oil reservoir simulations, we benchmark the various precondi-

tioners previously tested on two linear systems well-known cases from the oil reservoir industry: SPE9 [Killough, 1995] and SPE10 [Christie and Blunt, 2001].

The convergence rate results obtained from the SPE9 matrix are illustrated in Figure 5.2a, while those of the SPE10 problem are shown in Figure 5.2b.

(a) *SPE9* **(b)** *SPE10*

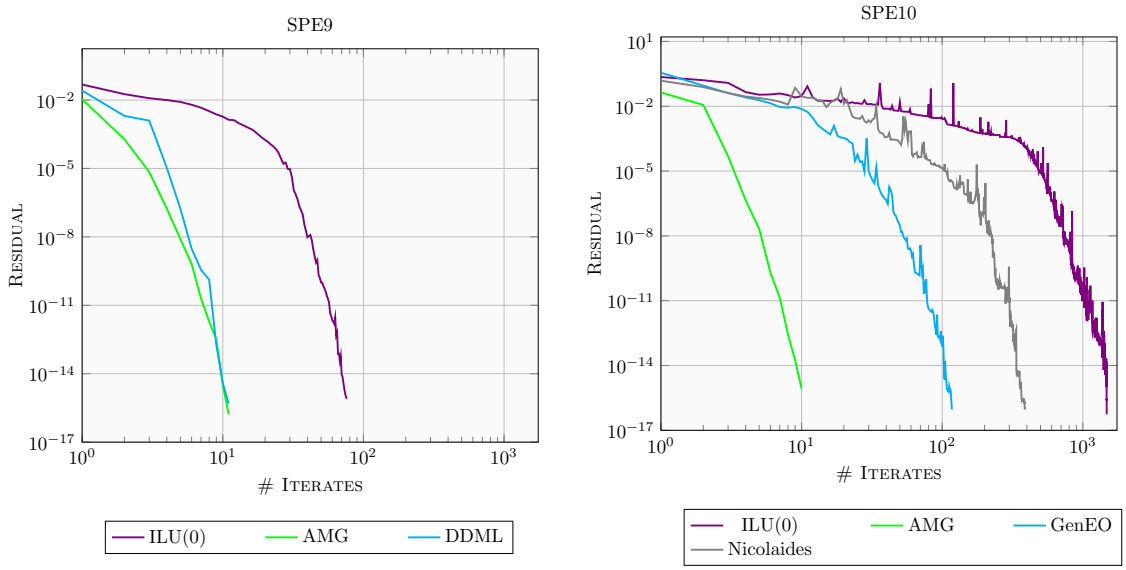


Figure 5.2 – *SpMV with CSR format*

For the first benchmark, AMG and DDML presents both an effective rate of convergence (around 10 iterations). Contrary to ILU(0) which failed to converge as fast (76 iterates). The curve draws a plateau that illustrates its slow convergence rate.

The larger benchmark, SPE10, shows some disparities between all the convergence rate curves. AMG achieves convergence after around 10 iterates while DDML requires 117 iterations. As previously seen, ILU(0) curve draws a plateau before reaching convergence after more than 1000 iterates. The DDML with the GenEO coarse operator's performances does not equalize AMG on this benchmark. However, the method still converges in a convenient number of iterations, at the opposite of the ILU(0) preconditioner. We can also notice that the Nicolaides coarse operator requires 387 iterations to converge, which is more than the GenEO coarse operator.

5.2 Parallel task-based implementation of DD methods

DDML preconditioner is robust regarding ill conditioned linear systems coming from oil reservoir simulation. Although the domain decomposition preconditioners rely on a parallel structure, programming such methods remains challenging.

In the first subsection, we describe the translation from the algorithm to the task

decomposition and therefore the DAG organization. The second subsection is devoted to the DAG's structure analysis to detect any issue encountered at method's parallelization. From this analysis, we experiment a 2-level ASM with the GenEO coarse operator in order to achieve a good performance of the method.

5.2.1 Task Decomposition

The main difference between the 1-level and the 2-level ASM algorithms relies on the addition of the coarse operator on line 3 of the Algorithm 5.2. However, lines 5-7 of this algorithms also corresponds to the classical AS method. In a first step, we translate the classic ASM algorithm sketched in Algorithm 5.1 (see page 99) into the API formalism. It can be viewed as two major parts: local solver part (line 3), and result update (lines 4-5). The later part groups several operations: 2 *axpy* kernels and a matrix vector product. All these steps can be performed in parallel independently and can be decomposed in tasks. Local solver resolution is operated only on internal nodes of the input matrix. As the matrix is reordered and partitioned, this local solving phase can thus be split into N_{part} partitions. The same treatment can thus be done for both *axpy* and *SpMV* kernels. One operation is thus divided in N_{part} tasks, as it has been discussed in the section 3.1.2. The different steps of the ASM algorithm are at the 5th and 6th levels of the 2-level ASM DAG illustrated in Figure 5.3.

We now focus on the additional operations of the Algorithm 5.2 (lines 3-4) induced by the insertion of the coarse operator of the 2-level Additive Schwarz Method. The line 4 is translated by an *axpy* kernel, which corresponds to the 4th level of the DAG. A particular attention is required for the line 3 as it involves a linear combination of deflation operations coming from the multiplications coming from Z and Z^T. We respectively name these steps *Interpolation operator* and *Restriction operator* and are at the 1st and 3rd levels of the DAG. Another matrix vector multiplication is also induced A_{ext}. These operations can be done in parallel. In addition of that, we also have a linear system to solve induced by the coarse operator E, that couple all the sub-domains. This operation could be performed in parallel, however the linear system is supposed to be small. Hence, we operate this solving phase in a unique task, named *Coarse Op*, which synchronizes all the partitions as it is sketched at the 2nd level of the DAG in Figure 5.3.

5.2.2 Bottleneck identification

The most costly operations of the 2-level Additive Schwarz algorithm are linear solvers introduced in the resolution phase (i.e. coarse operator and local solvers). Both systems are supposed to be of small size, and can thus be solved by direct solvers. In our approach, we solve linear solvers which appeared from 2-level ASM with the *Eigen* library [Guennebaud et al., 2010].

The size of the coarse operator E increases according to the number subdomains. In the case of GenEO coarse operator, it also relies on the number N_{Ev} of eigenvalues we

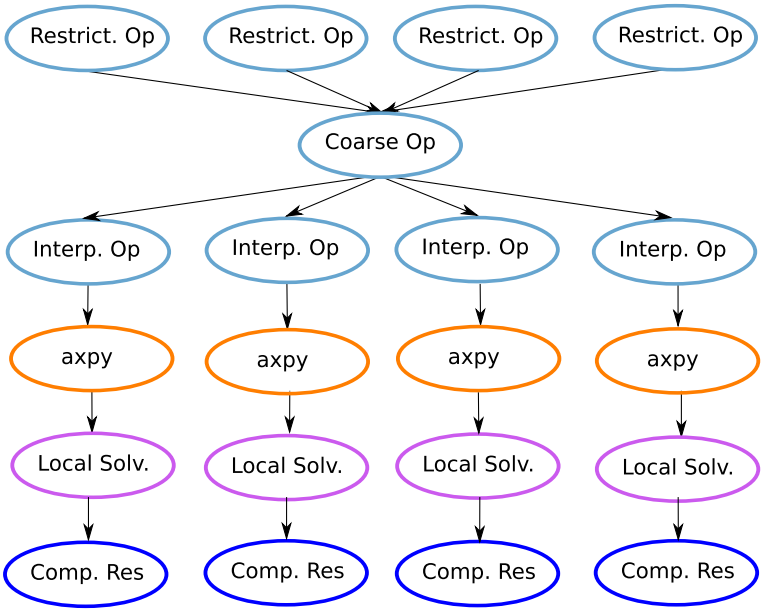


Figure 5.3 – *2-level ASM DAG*

incorporate per subdomain. The size of the coarse operator is thus given by the formula: $Size(E) = N_{part} * N_{Ev}$. If the coarse operator sizes increase too much, the coarse operator system will be more difficult to solve. The time complexity of a dense direct solver is $O(n^3)$. In this case, it means that the coarse resolution may synchronize all the threads because it is a blocking task as it couples all the subdomains at a time.

On the other hand, the size of a system A_{int}^k coming from local solver operation on the k-th subdomain depends on the size of the partition k, i.e. its number of internal nodes. The partition's size then rely on the number of partitions we use. Indeed, we expect that the size of local solvers decrease when the number of subdomains increase. According to graph partitioning techniques, we assume that the average number of rows of a system A_{int}^k is N_{Rows}/N_{part}. If the number of partitions is high, thus the number of internal nodes per partition will decrease. Hence, the local linear system to solve per partition will be simpler and the resolution will be faster.

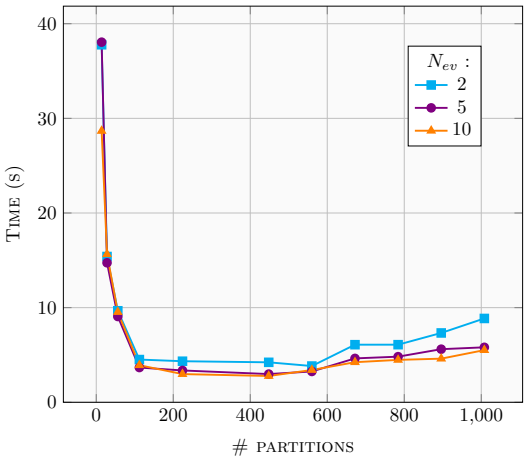
Hence, the granularity of both problems are connected together. We have to find a compromise between the sizes of both problem. From one side, we have to incorporate a sufficient numbers of subdomains to enhance communication between them, but not too much to not synchronize too much time the threads per iteration. At the other side, we have to partition in a sufficient number of partitions to make the local solver simpler.

5.2.3 Experiments

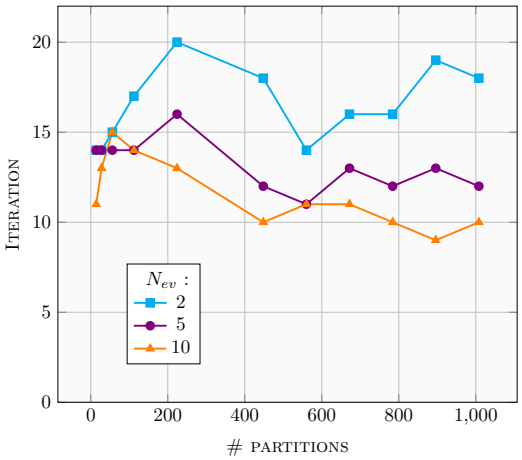
We explained in the previous section that it exists a compromise to do between the sizes of the coarse operator and the local internal systems A_{int}^k. We intend to illustrate this

compromise in terms of time results while varying both solvers' sizes. We performed a 2-level ASM preconditioner with the GenEO coarse operator on the SPE10 oil reservoir benchmark. We fixed the number of threads to 28, and run it on a 2 sockets nodes each composed of a 14 cores processor clocked at 2.40GHz and 64Gb of memory. The GenEO coarse operator's size depends on the number of partitions and the number of eigenvalues we incorporate. Hence, we vary the number of eigenvalues. For a given number of eigenvalues incorporated in the coarse operator, we vary the number of partitions we used. We illustrate in Figure 5.4a the time results we obtained from this execution. We illustrate in right part of the Figure 5.4c the evolution the number of elements of the coarse operator regarding both the number of partitions and eigenvalues. We then compute the ratio $R = (\sum N_{Rows}(A_{int}^k))/N_{part}$ which gives the average number of rows of the systems A_{int}^k and plot the results regarding N_{part} in the left y axis of the Figure 5.4c.

(a) *Performances' impact from partitioning*



(b) *Partitioning impact on convergence rates*



(c) N_{Nrows} *of local solvers*

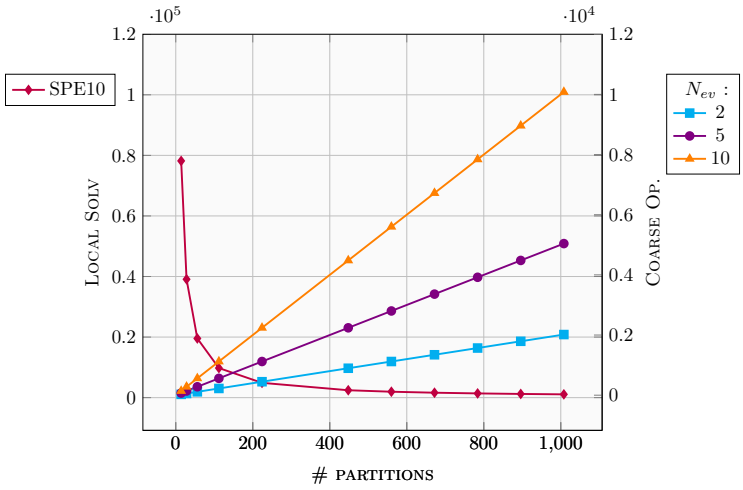


Figure 5.4 – *Partitioning information*

From a theoretical point of view, we can first notice that we can attempt to find an

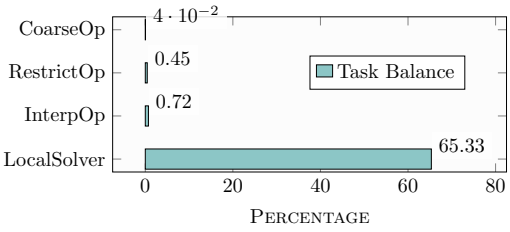
optimum between the two linear system's sizes (i.e. local and coarse systems). It can be defined on the Figure 5.4c at the intersection between the sizes of local and coarse problems. By this way, we can attempt to find the optimum at 112 partitions while $N_{Ev} = 10$, 150 subdomains when $N_{Ev} = 5$, and eventually $N_{part} = 224$ when $N_{Ev} = 2$.

We can see on Figure 5.4a that all the curves decrease from 14 to 112 partitions. It is cause from the high size of local solvers that takes too much time and then slow down the overall performances. When the average size of the A^k_{int} systems decrease, time results decrease too. From 112 to 560 partitions, time results are stabilized before growing up beyond 560 partitions. The best performances are obtained from 112 to 560 subdomains, independently from the size of the the coarse operator E. Between these partitions' sizes, its size vary from 224 to 5600 elements. Hence, we can suppose that the most important factor here is to find a good size for the local solvers to not degrade performances. Here, the average of rows of the systems A^k_{int} is from 9771 with 112 partitions to 1954 with 560 partitions.

In order to have a better understanding on the origin of the minimal time values arising from 112 to 560 subdomains, we are now looking to the tasks' distribution of the two executions. In Figures 5.5a and 5.5b we illustrate the time spend to perform each task types with the incorporation of 5 eigenvalues per subdomain in the coarse operator. We only focus on tasks related to the Coarse operator addition. Let ω_j be a task type, and we gather all the tasks of type ω_j in the set Ω. The function $D(t_i)$ gives the duration of the task t_i. W denotes the sum of the duration of all the task performed, $W = \sum D(t_i)$. For each task type ω_j, we compute the ratio,

$$R_{\omega_j} = \frac{100 * \sum D(\Omega_i)}{W}.$$

(a) *48 subdomains*



(b) *448 subdomains*

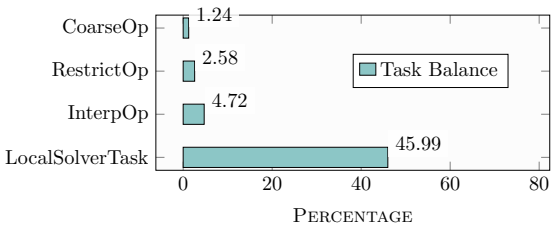


Figure 5.5 – *DDML's tasks percentage*

We can first notice that the most time consuming tasks are the local solver tasks. Coarse solver operation only represents around 1%, which does not involve a too much aggressive synchronization for all the threads. When the system is partitioned in 48 subdomains, the local solver tasks represent 65% of the global time. They represent 20% less when using 448 subdomains. This analysis confirms our assumption made in the previous paragraph. Indeed, the time decrease observed from 14 to 560 partitions is caused by the local solver's time reduction.

We assume that the time increase from 560 subdomains and beyond is induced by the increase in the size of the coarse operator E. When its size grows up, the coarse solver task is supposed to be more time consuming and synchronizes much more threads because of its blocking property. We thus compare both compute and idle times for the execution of 448 and 1008 subdomains with the incorporation of 5 eigenvalues per subdomain in E. Results are sketched in Table 5.1.

Subdomains	Idle	Compute
448	8.27 %	91.73 %
1008	11.90 %	88.10 %

Table 5.1 – *Load Balancing analysis*

We observe that idle time increases according to the increase of the number of subdomains. It is caused by the cost of the coarse solver which increases too. Indeed, we observe that the *CoarseOp* task represent 2.11% of the overall execution for $N_{part} = 1008$, instead of only 1.24% for $N_{part} = 448$.

To conclude this experiment, we can notice that the number of subdomains N_{part} enable to find an optimum between both local and coarse solver tasks. Theoretically, we can find an optimum between 112 and 224 subdomains, depending on the number of eigenvalues we incorporate per subdomain in the coarse operator. Experimentally, we show that the optimum is reach when N_{part} is between 112 and 560 regardless N_{Ev}. Hence, the theoretical analysis is not biased from the experimental one and can give a first hint to find the best compromise between the two task granularity.

5.3 Experiments on multi-core based systems

Our experiment are made on a 2 sockets compute node, each composed of a 14 cores Broadwell processor clocked at 2.40Ghz and 64Gb of memory. On this compute node, Cluster-on-Die mode is enabled in order to expose a processor as two various NUMA nodes. We compile the application with the GCC 6.1.0 compiler and we use Intel MKL to operate BLAS operations through the API we developed. We partition data through Metis partitioner. For these experiments, we use the 2-level ASM preconditioner with the BiCGStab algorithm. At each iterate, we incorporate two iterations of the DDML preconditioner.

5.3.1 2-level ASM method performances

We first intend to evaluate the 2-level DDM preconditioner performances through the use of the GenEO coarse operator. We analyze preconditioner's behavior while varying sizes of the input matrices via systems coming from the finite volume discretization of a 2D Laplace problem. For both problems, we partition the system in 252 subdomains.
We report our results on the Figure 5.6. Figure 5.6a illustrates the parallel time T_p according

to the number of threads p. From time T_p we plot the speed-up ratio, $S = T_1/T_p$, according to p on the Figure 5.6b.

(a) *Elapsed Time* **(b)** *Speed-Up*

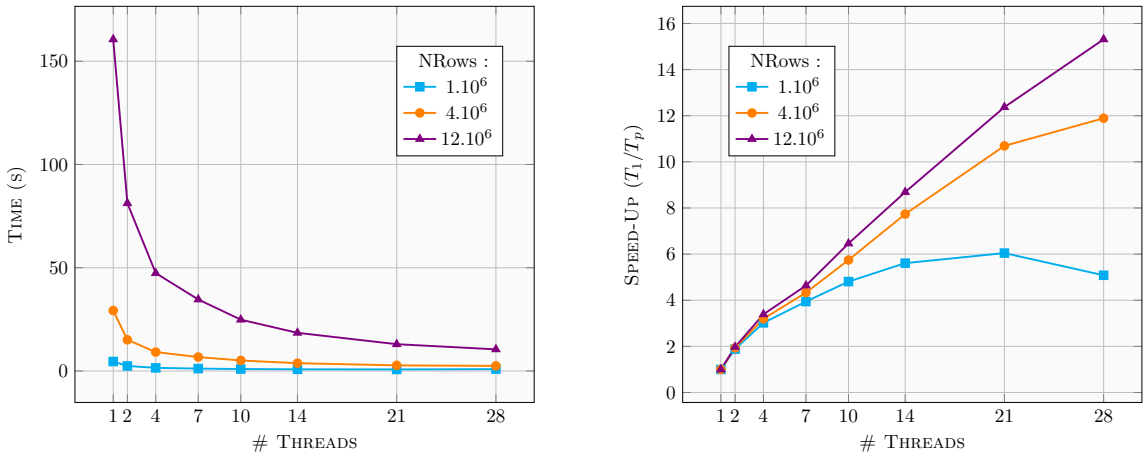
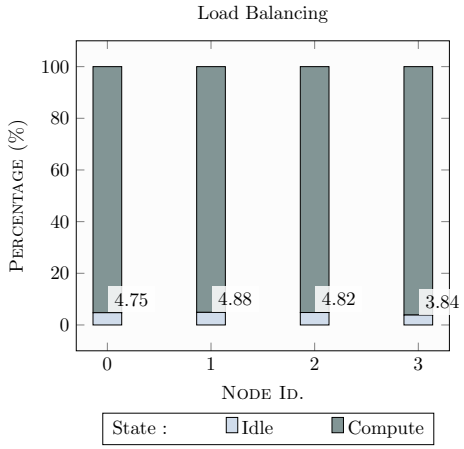


Figure 5.6 – *2-level AS preconditioner on Laplacian matrices*

Whatever the input system, the parallel time is decreasing while the number of cores p is increasing. On the speed-up curves, we can observe that the smallest benchmark's performances increase up to 14 threads, then slow down and eventually decrease. As its work is not high, we can thus suppose this benchmark reach its parallel limits at 14 threads. However, beyond 14 threads the application run on the two sockets of the machine. Even if data are distributed among NUMA nodes, we can suppose that data communication induced by SpMV products or coarse operator synchronization are too expensive face to computing efforts. For the two other benchmarks, parallel performances are still increasing in function of the number p of cores. For the bigger input matrix, we reach a speed-up peak of 15.3 with 28 threads while the intermediate size matrix reach 12 with the same number p of cores.



5.3.2 Oil reservoir simulation's case

We now aim to validate our approach by evaluating 2-level DD method with GenEO coarse space regarding widespread preconditioners used in oil reservoir simulation. Hence, we compare parallel time results obtained from the parallelization of the given preconditioners: ILU(0), AMG from the Hypre [Falgout et al., 2006] library and our custom implementation of the DDML preconditioner. We benchmark both preconditioners while using the SPE10 [Christie and Blunt, 2001] linear system, which is representative to linear systems arising from oil reservoir simulations. We plot results according to various numbers p of processors in Figure 5.7. Elapsed parallel time, T_p, is sketched in Figure 5.7a, while the speed-up ratio $S = T_1/T_p$ is illustrated in Figure 5.7b.

(a) *Elapsed Time* **(b)** *Speed-Up*

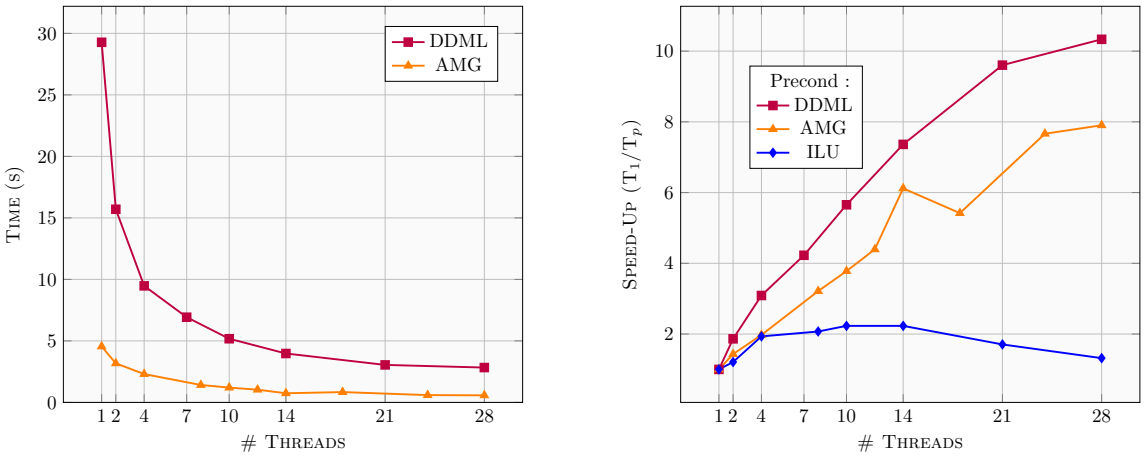


Figure 5.7 – *2-level AS preconditioner on the SPE10 system*

We can observe on both curves that the AMG preconditoner offers the best performances for the reservoir case. It can be explained by the few iterations required to converge. We can notice that DDML preconditioner achieve good performances, but not as good as the AMG preconditioner. Even if we require a bit more number of iterations than the AMG method, DDML's iterations remain time consuming because of the local linear solver's phase. Indeed, we previously seen in section 5.2.3 that this step represent up to 50% of the parallel execution in the case where $p = 28$ cores. In the last position, ILU(0) preconditioner fails to converge in a reasonable number of iterations and it impacts a lot its performances. We can also see that its performances tend to decrease from 14 threads, because of its difficulties to be efficiently parallelized.

5.4 Experiments on the KNL many-core processor

For these experiments, we used a 64 cores KNL processor embedded in a system which handles 192Gb of DRAM memory. The many-core processor is configured with the 4

Sub-Numa Cluster mode. The 16Gb of MCDRAM memory are exposed according to the Flat memory mode. We turn off the Turbo mode to not perturb the vector processing unit frequencies. By default, we pin the first processing unit of the core in the appearance order through the Hwloc library.

We aim here to evaluate the performances of the DDML method with the GenEO coarse operator on the KNL many-core processor. We intend to use systems coming from the finite volume discretization of the Laplace problem in order to evaluate the method on various system sizes. We benefit from the improvement we made on the API to take care of the KNL architecture. Indeed, we use here the Sell-C-σ format while vectorizing SpMV operations. The BLAS kernel are vectorized too thanks to the use of the Intel MKL library. All the effective data used for iterative method computations are allocated in the high bandwidth memory bank of the KNL, i.e. the MCDRAM. For each system size, we run a BiCGStab algorithm with a DDML preconditioner based on the 2-level ASM method and the GenEO coarse operator.

Results are presented in the Figure 5.8. Time results are illustrated in Figure 5.8a. From these results, we compute the speed-up ratio via the formula $S(p) = T_1/T_p$ and show its evolution in Figure 5.8b.

(a) *Elapsed Time* **(b)** *Speed-Up*

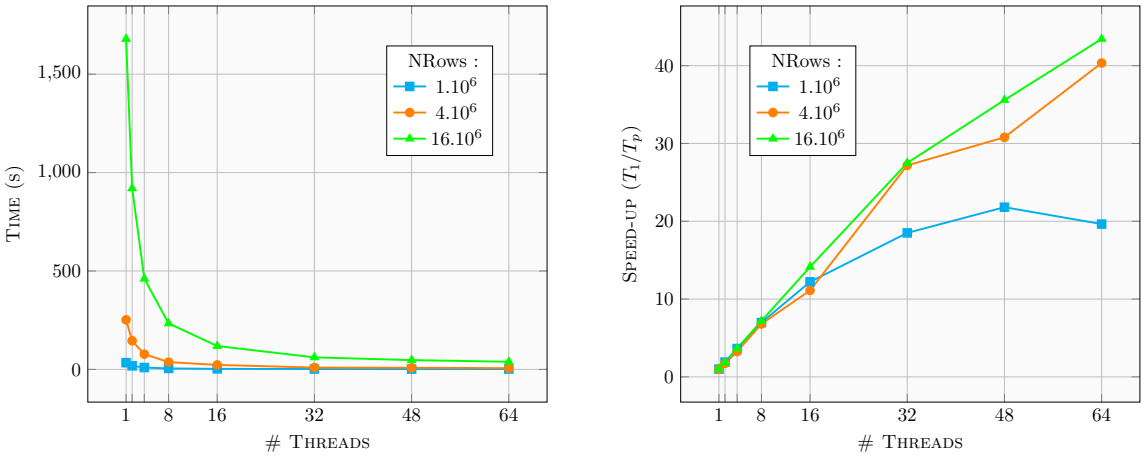


Figure 5.8 – *2-level AS preconditioner with various Laplacian systems on KNL processor*

We can first observe that time decrease while the number of threads in use increase. On the speed-up curves, we can see that the two biggest input systems, the application benefit from a speedup factor of around 40. We do not achieve such performances for the smaller system which reach no more than 20 of speedup ratio. For this case, performances are decreasing from 48 to 64 threads. We suppose that it is due to the difficulty to provide enough data to feed the KNL processor.

Thanks to monitoring tools we implemented, we now have a look the computational efficiency of the DDML method on the same input systems. Computational efficiency has already been defined in the section 3.3. Results are gathered in the Figure 5.9.

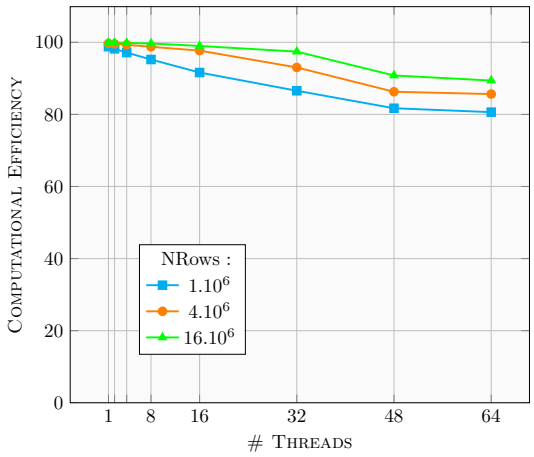


Figure 5.9 – *Computational Efficiency of the DDML method on a KNL processor*

We can notice that the computational efficiency decreases while threads in use increase, regardless the system size. For the smaller system, the efficiency decreases up to 80% while the biggest one reaches 90% at its minimum value. We can thus conclude that our DDML implementation does not suffer from treads' inactivity caused by the coarse system's solver.

Thanks again to monitoring tools, we then analyze task balancing at execution time. In Figure 5.10, we gather the task distribution of two executions on 64 threads while varying the system's size.

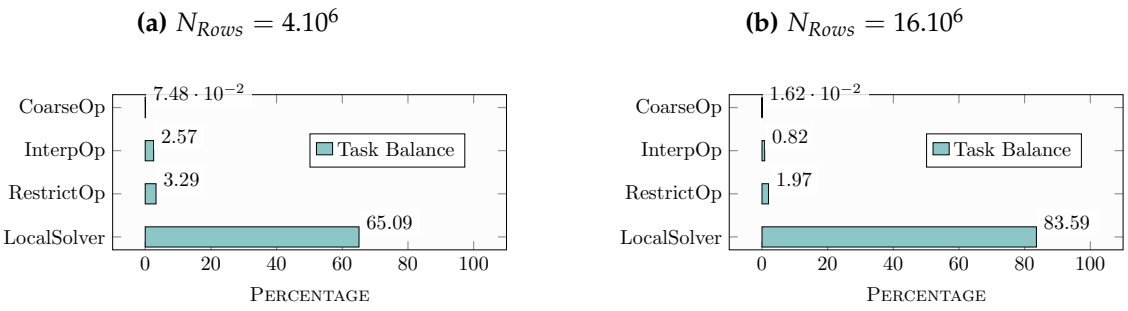


Figure 5.10 – *DDML's tasks percentage on KNL processor*

We can observe that for both system sizes, *LocalSolver* task leads the performances of the DDML's parallel execution. The cost of this task may represent up to 84% of the total time execution for the bigger test case. We can thus conclude that this task still represent a bottleneck for parallel performances. We already have seen in section 4.4.3 that tasks as the SpMV operation may be enhanced by vectorization on the KNL architecture. Until now, *LocalSolver* task is still not vectorized and its performances may be improved while taking advantage of AVX512 instructions. A reflection on this problem is still in progress to enable sparse direct solver enhancement by vectorization.

5.5 Discussion

We have seen in this chapter that DDML preconditioners are robust facing ill conditioned systems coming from oil reservoir simulators. Thanks to the API we developed, we propose here a task-based parallel implementation of a 2-level ASM preconditioner. By this way, we are able to propose both multi-core and many-core implementations of the preconditioner to simulator's users.

Thanks to our implementation, we show by experiment that the granularity of two task types are linked together. The cost of each one is not negligible as the time complexity is of $O(n^3)$. We thus find a compromise to not compromise performances.

On multi-core architectures, DDML preconditioner's performances does not suffer from the increase in number of cores in use. On realistic simulation case, it successes to be robust while getting high performances. Compared to the AMG preconditioner, DDML is a bit more expensive.

The KNL many-core processor enable to enhance performances while multiplying the number of concurrent processing units in use. Thanks to our API, we demonstrate that the DDML methods are efficient on this kind of systems thanks to their natural parallel structure. However, some improvements on vectorization are still in progress to fully exploit the capabilities of such many-core processor. Thanks to this ongoing work, we expect to increase performances of the DDML preconditioner face to AMG.

Chapter 6

Conclusion

The performance of numerical methods used in numerical simulations in reservoir engineering relies on the performances of iterative methods used to solve large and sparse linear systems. In this case, performances have several meanings: few iteration to converge, low memory footprint, robustness face to ill-conditioned systems, low execution time.

Our work is focused on the objective to reduce execution time of iterative methods while using parallelism induced by computer architectures. We experiment the benefit of task parallel programming model to implement efficient methods, especially domain decomposition preconditioner. This programming model is useful in dynamic scheduling which adapts the execution face to the variation of the environment.

In a first time, we develop a sparse linear algebra framework which aims to provide task based parallel implementation of iterative methods used to solve sparse linear systems. Its semantic is sequential while the parallelism is implicit. The API is built around the `Sequence` concept, which is an ordered list of tasks that represents a succession of operations. The `Sequence` allows capture the sequential semantics of the execution of tasks. At runtime, the sequence allows to compute data flow dependencies and thus it detects at run time independent tasks that could be performed concurrently. Most of the BLAS like kernels used by our algorithms in sparse linear algebra are decomposed by a sequence of tasks. The API and implementation could be easily extended to support new basic algorithm. The design of the API follows a factory pattern to trigger task creation and management to a specific runtime systems in charge of scheduling and balance the work load.
We based most of our development effort on the HARTS runtime system developed at IFPEN. Due to strong data locality impact in our algorithms, we extend HARTS in order to be able to distribute data among memory nodes thanks to the first touch policy. Then we add a work stealing scheduler which uses precomputed good victims to steal when a thread becomes idle. By experimentation, we demonstrate that our scheduler improve data locality computations compared to a classical scheduler based on a centralized task pool. In order to have a better understanding of the application's behavior, we implement monitoring tools in the HARTS runtime system. We add instrumentation to the HARTS runtime. It allows to collect some metrics such as task duration and task placement. From

these extracted information, we can analyze post-mortem the performance of the execution of the application. By this way, we are able to analyze the parallel execution through various performance tools such as threads' activity, Gantt chart and critical path analysis. Eventually, we evaluate the performances of the iterative methods we built on top of the API we develop.

The goal of the API was to make easy writing application in our context. Moreover, the API aims at making portable performances among architecture. Our work is focused on the recent Knights Landing many-core processors. These processors simplify the core design to handle many more cores per processor. The vector register size increased to face to the decreasing in core frequency. They also embed a High Bandwidth Memory (HBWM) in order to reduce memory latency. We identify two main needs to be efficient on such an architecture: code vectorization with SIMD instructions and high bandwidth memory management. We propose to froze the implementation of kernels at compile time among several versions for a same function. By this way, we fully benefit from vectorized kernels while taking advantage of vector capabilities of the KNL processor. However, some data structures are not adapted to vector processing. Hence, we handle several structures for a same data at the API level. By this way, we are able to reach up to 3 times more performances on a SpMV kernel by changing the sparse matrix data structure from classical CSR format to the Sell-C-σ. Moreover, we also extend the API to let the choice to developers to select in which memory bank data are supposed to be. Thanks to `Allocator` concept, we thus abstract users from data allocating function. We highlights the importance of the MCDRAM in the solver performance. We have benchmarked various linear solvers to highlight the performances of the API on the KNL many-core processor.

Although we are able to write portable applications on both multi-core and many-core systems, we only evaluated parallel performances of sparse iterative methods regardless the input linear system. However, classical methods have difficulties to converge in a reasonable time face to ill conditioned systems arising from reservoir simulation. Thanks to the API we developed, we are able to propose a parallel task-based implementation of a 2-level Domain Decomposition preconditoner. By experimentation, we show that the DDML preconditoner performances are leaded by two kinds of task that represent direct solvers. The granularity of the these two categories of tasks are linked as they both depend on the number of subdomains. Hence, we investigate on the optimum between the two sizes of problems in order to reach the best performances. We also demonstrate that the DDML preconditioner has no difficulty on parallel scalability facing the increase of number of cores on multi-core systems. Compared to the AMG preconditioner, DDML is a bit more costly but remains more efficient than ILU(0). Thanks to the increase in cores of the KNL many-core chip, we show that the DDML preconditioner naturally fit on massively parallel processor thanks to its algorithmic structure. The performances of our parallel implementation reach up to around 80% of computational efficiency. It highlights that this task based parallel implementation enable to take advantage of such a method on complex architectures.

To sum up, we propose an abstract linear algebra API to developers at IFPEN, which

hide parallelism without perturbing performances while being adapted for two kind of architectures. This evaluating work has been done in the linear solver context developed through the MCGSolver library [Anciaux-Sedrakian et al., 2014], which is integrated in the Arcane software [Grospellier and Lelandais, 2009]. These evaluations have been proposed thanks to the instrumentation of the HARTS runtime system. By this way, we have been able to propose two kinds of optimization. The first one is the integration of the NUMA concern in HARTS, including a specific scheduler. The second one is the optimization of the Sparse Matrix Vector kernel on the KNL architecture.

During this work, we face to the problem of code portability. Regardless the complexity of emerging architectures, some optimizations are specific to only one architecture. However, rewriting from scratch application is time consuming. Our choice is to adopt a static strategy chosen at compile time. However, this solution is not optimal because it requires knowledge about architecture from end-users to switch on some parameters. In a long-term prospect, we hope to develop a more flexible solution that can adopt decision at execution regarding the underlying architecture. First, this solution can be provided at the runtime system with the help of auto-tuning parameters. By this way, it can make decision on optimizations that are the most favorable on the current architectures such as multi-versioning kernels. However, this answer to the problem can also come from the programming model. Within the API, we only experiment task-based programming model and it is adapted to our application, but we have no guarantee that it will be always right. This "code portability" is general and most people who maintain codes are faced without support. The study of programming model and tools for providing solution or methodology to manage code evolution among different architecture variations has been submitted into an ANR project named "KOALA" leaded by my PhD supervisor.

Meanwhile, the short-time prospect is the study of our solution with the OpenMP standard. By this way, we attempt to understand and then limit the work inflation phenomena we previously observed. Our first hypothesis is that the work inflation comes from bus contention. The solution may come from various sources as scheduling to favor cache reuse. We will also try to increase computation load to limit memory transfers, and hide them by computations.

Bibliography

[Al-Omairy et al., 2015] Al-Omairy, R., Miranda, G., Ltaief, H., Badia, R., Martorell, X., Labarta, J., and Keyes, D. (2015). Dense matrix computations on numa architectures with distance-aware work stealing. *Supercomputing Frontiers and Innovations*, 2(1).

[Anciaux-Sedrakian et al., 2014] Anciaux-Sedrakian, A., Gottschling, P., Gratien, J.-M., and Guignon, T. (2014). Survey on Efficient Linear Solvers for Porous Media Flow Models on Recent Hardware Architectures. *Oil and Gas Science and Technology*, 69(4):pp. 753–766.

[Asanović et al., 2006] Asanović, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. (2006). The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

[Augonnet and Namyst, 2008] Augonnet, C. and Namyst, R. (2008). A unified runtime system for heterogeneous multicore architectures. In *2nd Workshop on Highly Parallel Processing on a Chip (HPPC 2008)*, Las Palmas de Gran Canaria, Spain.

[Augonnet et al., 2011] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2011). StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198.

[Ayguadé et al., 2010] Ayguadé, E., Badia, R. M., Bellens, P., Cabrera, D., Duran, A., Ferrer, R., Gonzàlez, M., Igual, F., Jiménez-González, D., Labarta, J., Martinell, L., Martorell, X., Mayo, R., Pérez, J. M., Planas, J., and Quintana-Ortí, E. S. (2010). Extending openmp to survive the heterogeneous multi-core era. volume 38, pages 440–459.

[Ayguade et al., 2009] Ayguade, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., and Zhang, G. (2009). The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418.

[Baker et al., 2011] Baker, A. H., Gamblin, T., Schulz, M., and Yang, U. M. (2011). Challenges of scaling algebraic multigrid across modern multicore architectures. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 275–286.

[Balay et al., 1997] Balay, S., Gropp, W. D., McInnes, L. C., and Smith, B. F. (1997). Efficient management of parallelism in object oriented numerical software libraries. In Arge,

E., Bruaset, A. M., and Langtangen, H. P., editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press.

[Bleuse et al., 2014] Bleuse, R., Gautier, T., Lima, J. V. F., Mounié, G., and Trystram, D. (2014). *Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*, chapter Scheduling Data Flow Program in XKaapi: A New Affinity Based Algorithm for Heterogeneous Architectures, pages 560–571. Springer International Publishing, Cham.

[Blumofe et al., 1995] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1995). Cilk: An Efficient Multithreaded Runtime System. *SIGPLAN Not.*, 30(8):207–216.

[Blumofe and Leiserson, 1994] Blumofe, R. D. and Leiserson, C. E. (1994). Scheduling multithreaded computations by work stealing. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 356–368.

[Bosilca et al., 2012] Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., and Dongarra, J. (2012). Dague: A generic distributed dag engine for high performance computing. *Parallel Comput.*, 38(1-2):37–51.

[Brandt et al., 1984] Brandt, A., Mccormick, S. F., and Ruge, J. W. (1984). Algebraic multigrid (AMG) for sparse matrix equations. Cambridge University Press, New York.

[Broquedis et al., 2010] Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., and Namyst, R. (2010). hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In IEEE, editor, *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa, Italy.

[Carney et al., 1994] Carney, S., Heroux, M. A., Li, G., and Wu, K. (1994). A revised proposal for a sparse blas toolkit. Technical report.

[Christie and Blunt, 2001] Christie, M. and Blunt, M. (2001). Tenth spe comparative solution project: A comparison of upscaling techniques. *SPE Reservoir Evaluation & Engineering*, 4(2):308–317.

[Chrysos, 2012] Chrysos, G. (2012). Intel xeon phi coprocessor (codename knights corner). In *2012 IEEE Hot Chips 24 Symposium (HCS)*, pages 1–31.

[Ciarlet, 1998] Ciarlet, P. (1998). *Introduction à l'analyse numérique matricielle et à l'optimisation*. Collection Mathématiques appliquées pour la maîtrise. Dunod.

[Davis, 1978] Davis, A. (1978). *Data Driven Nets: A Maximally Concurrent, Procedural, Parallel Process Representation for Distributed Control Systems*. University of Utah, Department of Computer Science.

[de Dinechin et al., 2013] de Dinechin, B. D., de Massas, P. G., Lager, G., Léger, C., Orgogozo, B., Reybert, J., and Strudel, T. (2013). A distributed run-time environment for the kalray mppa®-256 integrated manycore processor. *Procedia Computer Science*, 18:1654 – 1663. 2013 International Conference on Computational Science.

[DeVito et al., 2011] DeVito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., Elsen, E., Ham, F., Aiken, A., Duraisamy, K., Darve, E., Alonso, J., and Hanrahan, P.

(2011). Liszt: A domain specific language for building portable mesh-based pde solvers. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12.

[Dolean et al., 2015] Dolean, V., Jolivet, P., and Nataf, F. (2015). An Introduction to Domain Decomposition Methods: algorithms, theory and parallel implementation. Lecture.

[Drebes et al., 2014] Drebes, A., Heydemann, K., Drach, N., Pop, A., and Cohen, A. (2014). Topology-aware and dependence-aware scheduling and memory allocation for task-parallel languages. *ACM Trans. Archit. Code Optim.*, 11(3):30:1–30:25.

[Falgout et al., 2006] Falgout, R. D., Jones, J. E., and Yang, U. M. (2006). *The Design and Implementation of hypre, a Library of Parallel High Performance Preconditioners*, pages 267–294. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Feng et al., 2013] Feng, C., Shu, S., and Yue, X. (2013). *An Improvement to the OpenMP Version of BoomerAMG*, pages 1–11. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Ferreira Lima et al., 2015] Ferreira Lima, J. V., Gautier, T., Danjean, V., Raffin, B., and Maillard, N. (2015). Design and analysis of scheduling strategies for multi-CPU and multi-GPU architectures. *Parallel Computing*, 44:37–52.

[Filippone and Colajanni, 2000] Filippone, S. and Colajanni, M. (2000). Psblas: A library for parallel linear algebra computation on sparse matrices. *ACM Transactions on Mathematical Software (TOMS)*, 26(4):527–550.

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

[Forum, 1994] Forum, M. P. I. (1994). Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA.

[Frigo et al., 1998] Frigo, M., Leiserson, C. E., and Randall, K. H. (1998). The Implementation of the Cilk-5 Multithreaded Language. *SIGPLAN Not.*, 33(5):212–223.

[Gander et al., 2002] Gander, M. J., Magoulès, F., and Nataf, F. (2002). Optimized schwarz methods without overlap for the helmholtz equation. *SIAM Journal on Scientific Computing*, 24(1):38–60.

[Garey and Johnson, 1990] Garey, M. R. and Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

[Gautier et al., 2007] Gautier, T., Besseron, X., and Pigeon, L. (2007). KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *2007 international workshop on Parallel symbolic computation*, pages 15–23, Waterloo, Canada. ACM.

[Gautier et al., 2013] Gautier, T., Lima, J. V. F., Maillard, N., and Raffin, B. (2013). Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 1299–1308, Washington, DC, USA. IEEE Computer Society.

[Goglin, 2014] Goglin, B. (2014). Managing the Topology of Heterogeneous Cluster Nodes with Hardware Locality (hwloc). In *International Conference on High Performance Computing & Simulation (HPCS 2014)*, Bologna, Italy. IEEE.

[Gratien, 2013] Gratien, J.-M. (2013). An Abstract Object Oriented Runtime System for Heterogeneous Parallel Architecture. In *IPDPS Workshops*, pages 1203–1212. IEEE.

[Grimes et al., 1980] Grimes, R., Young, D., Kincaid, D., of Technology (Australia). Computer Centre, R. M. I., and of Texas at Austin. Center for Numerical Analysis, U. (1980). *ITPACK 2.0: User's Guide*. Center for Numerical Analysis: Center for Numerical Analysis. R.M.I.T. Computer Centre.

[Grospellier and Lelandais, 2009] Grospellier, G. and Lelandais, B. (2009). The arcane development framework. In *Proceedings of the 8th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, POOSC '09, pages 4:1–4:11, New York, NY, USA. ACM.

[Guennebaud et al., 2010] Guennebaud, G., Jacob, B., et al. (2010). Eigen v3. http://eigen.tuxfamily.org.

[Henon et al., 2002] Henon, P., Ramet, P., and Roman, J. (2002). Pastix: a high-performance parallel direct solver for sparse symmetric positive definite systems. *Parallel Computing*, 28(2):301 – 321.

[Intel, 2017] Intel (2017). Intel Math Kernel Library. http://software.intel.com/en-us/articles/intel-mkl/.

[Jeffers and Reinders, 2013] Jeffers, J. and Reinders, J. (2013). *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

[Jeffers et al., 2016] Jeffers, J., Reinders, J., and Sodani, A. (2016). *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Elsevier Science.

[Johnston et al., 2004] Johnston, W. M., Hanna, J. R. P., and Millar, R. J. (2004). Advances in dataflow programming languages. *ACM COMPUT. SURV*, 36(1):1–34.

[Jolivet et al., 2013] Jolivet, P., Hecht, F., Nataf, F., and Prud'Homme, C. (2013). Scalable Domain Decomposition Preconditioners for Heterogeneous Elliptic Problems. In *SC13 - International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 80:1–80:11, New York, NY, USA, France. ACM.

[Karypis and Kumar, 1995] Karypis, G. and Kumar, V. (1995). Metis – unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report.

[Kelley, 2003] Kelley, C. T. (2003). *Solving nonlinear equations with Newton's method*. Fundamentals of Algorithms. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA.

[Killough, 1995] Killough, J. (1995). Ninth spe comparative solution project: A reexamination of black-oil simulation. In *SPE Reservoir Simulation Symposium*. Society of Petroleum Engineers.

[Korch and Rauber, 2004] Korch, M. and Rauber, T. (2004). A comparison of task pools for dynamic load balancing of irregular algorithms. *Concurrency and Computation: Practice and Experience*, 16(1):1–47.

[Kreutzer et al., 2013] Kreutzer, M., Hager, G., Wellein, G., Fehske, H., and Bishop, A. R. (2013). A unified sparse matrix data format for modern processors with wide SIMD units. *CoRR*, abs/1307.6209.

[Kurzak et al., 2010] Kurzak, J., Ltaief, H., Dongarra, J., and Badia, R. M. (2010). Scheduling dense linear algebra operations on multicore processors. *Concurr. Comput. : Pract. Exper.*, 22(1):15–44.

[Lacoste et al., 2014] Lacoste, X., Faverge, M., Ramet, P., Thibault, S., and Bosilca, G. (2014). Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. Research Report RR-8446.

[Marr et al., 2002] Marr, D. T., Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A., Miller, A. J., and Upton, M. (2002). Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1).

[Moore, 1965] Moore, G. E. (1965). Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117.

[Nichols et al., 1996] Nichols, B., Buttlar, D., and Farrell, J. P. (1996). *Pthreads Programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

[Nicolaides, 1987] Nicolaides, R. A. (1987). Deflation of conjugate gradients with applications to boundary value problems. *SIAM Journal on Numerical Analysis*, 24(2):355–365.

[Nvidia Corporation, 2012] Nvidia Corporation (2012). CUDA Compute Architecture: Kepler GK110. Technical report.

[Olivier et al., 2012a] Olivier, S. L., de Supinski, B. R., Schulz, M., and Prins, J. F. (2012a). Characterizing and mitigating work time inflation in task parallel programs. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–12.

[Olivier et al., 2012b] Olivier, S. L., Porterfield, A. K., Wheeler, K. B., Spiegel, M., and Prins, J. F. (2012b). Openmp task scheduling strategies for multicore numa systems. *The International Journal of High Performance Computing Applications*, 26(2):110–124.

[Olivier and Prins, 2010] Olivier, S. L. and Prins, J. F. (2010). Comparison of openmp 3.0 and other task parallel frameworks on unbalanced task graphs. *International Journal of Parallel Programming*, 38(5):341–360.

[OpenMP Architecture Review Board, 2008] OpenMP Architecture Review Board (2008). OpenMP application program interface version 3.0.

[OpenMP Architecture Review Board, 2013] OpenMP Architecture Review Board (2013). OpenMP application program interface version 4.0.

[Park et al., 2015] Park, J., Smelyanskiy, M., Yang, U. M., Mudigere, D., and Dubey, P. (2015). High-performance algebraic multigrid solver optimized for multi-core based distributed

parallel systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 54:1–54:12, New York, NY, USA. ACM.

[Pellegrini and Roman, 1996] Pellegrini, F. and Roman, J. (1996). *Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs*, pages 493–498. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Roussel, 2016] Roussel, A. (2016). Comparaison de moteurs exécutifs pour la parallélisation de solveurs linéaires itératifs. In *Conférence d'informatique en Parallélisme, Architecture et Système (Compas'2016)*, Lorient, France.

[Roussel et al., 2016] Roussel, A., Gratien, J.-M., and Gautier, T. (2016). Using Runtime Systems Tools to Implement Efficient Preconditioners for Heterogeneous Architectures. *Oil & Gas Science and Technology - Revue d'IFP Energies nouvelles*, 71(6):65.

[Saad and Schultz, 1986] Saad, Y. and Schultz, M. H. (1986). Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869.

[Schwarz, 1870] Schwarz, H. A. (1870). Uber einen grenzubergang durch alternierendes verfahren. pages 272–286.

[Servat et al., 2013] Servat, H., Teruel, X., Llort, G., Duran, A., Giménez, J., Martorell, X., Ayguadé, E., and Labarta, J. (2013). On the instrumentation of openmp and ompss tasking constructs. In *Proceedings of the 18th International Conference on Parallel Processing Workshops*, Euro-Par'12, pages 414–428, Berlin, Heidelberg. Springer-Verlag.

[Shende and Malony, 2006] Shende, S. S. and Malony, A. D. (2006). The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311.

[Spillane et al., 2014] Spillane, N., Dolean, V., Hauret, P., Nataf, F., Pechstein, C., and Scheichl, R. (2014). Abstract robust coarse spaces for systems of pdes via generalized eigenproblems in the overlaps. *Numerische Mathematik*, 126(4):741–770.

[Stüben, 2001] Stüben, K. (2001). A review of algebraic multigrid. *Journal of Computational and Applied Mathematics*, 128(1):281 – 309. Numerical Analysis 2000. Vol. VII: Partial Differential Equations.

[Topcuouglu et al., 2002] Topcuouglu, H., Hariri, S., and Wu, M.-y. (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274.

[van der Vorst, 1992] van der Vorst, H. A. (1992). Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13(2):631–644.

[Varga, 1960] Varga, R. S. (1960). Factorization and normalized iterative methods. *Boundary problems in differential equations (R. E. Langer, ed.)*, pages 121–142.

[Virouleau et al., 2016a] Virouleau, P., Broquedis, F., Gautier, T., and Rastello, F. (2016a). Using data dependencies to improve task-based scheduling strategies on NUMA architectures. In *Euro-Par 2016*, Euro-Par 2016, Grenoble, France.

[Virouleau et al., 2016b] Virouleau, P., Roussel, A., Broquedis, F., Gautier, T., Rastello, F., and Gratien, J.-M. (2016b). Description, Implementation and Evaluation of an Affinity Clause for Task Directives. In *IWOMP 2016*, IWOMP 2016 - LLCS 9903, Nara, Japan.

[Virouleau et al., 2016c] Virouleau, P., Roussel, A., Broquedis, F., Gautier, T., Rastello, F., and Gratien, J.-M. (2016c). Description, Implementation and Evaluation of an Affinity Clause for Task Directives. In *IWOMP 2016*, IWOMP 2016 - LLCS 9903, Nara, Japan.

[Von Neumann, 1945] Von Neumann, J. (1945). First draft of a report on the EDVAC. Technical report, Los Alamos National Laboratory.

[Whaley and Dongarra, 1998] Whaley, R. C. and Dongarra, J. (1998). Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*. CD-ROM Proceedings. **Winner, best paper in the systems category.** URL: `http://www.cs.utsa.edu/~whaley/papers/atlas_sc98.ps`.

[Williams et al., 2009] Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., and Demmel, J. (2009). Optimization of sparse matrix–vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178 – 194. Revolutionary Technologies for Acceleration of Emerging Petascale Applications.

[Xianyi et al., 2017] Xianyi, Z., Qian, W., and Saar, W. (2017). Openblas, an optimized blas library. http://www.openblas.net/.

Résumé

Les méthodes en simulation numérique dans le domaine de l'ingénierie pétrolière nécessitent la résolution de systèmes linéaires creux de grande taille et non structurés. La performance des méthodes itératives utilisées pour résoudre ces systèmes représente un enjeu majeur afin de permettre de tester de nombreux scénario.

Dans ces travaux, nous présentons une manière d'implémenter des méthodes itératives parallèles au dessus d'un support exécutif à base de tâches. Afin de simplifier le développement des méthodes tout en gardant un contrôle fin sur la gestion du parallélisme, nous avons proposé une API permettant d'exprimer implicitement les dépendances entre tâches : la sémantique de l'API reste séquentielle et le parallélisme est implicite.

Nous avons étendu le support exécutif HARTS pour enregistrer une trace d'exécution afin de mieux exploiter les architectures NUMA, tout comme de prendre en compte un placement des tâches et des données calculé au niveau de l'API. Nous avons porté et évalué l'API sur les processeurs many-cœurs KNL en considérant les différents types de mémoires de l'architecture. Cela nous a amené à optimiser le calcul du SpMV qui limite la performance de nos applications.

L'ensemble de ce travail a été évalué sur des méthodes itératives et en particulier l'une de type décomposition de domaine. Nous montrons alors la pertinence de notre API, qui nous permet d'atteindre de très bon niveaux de performances sur des architectures multi-cœurs et many-cœurs.

Mots clés: Calcul parallèle, Support exécutif, Décomposition de domaine, multi-cœurs, many-cœurs, programmation par tâches

Abstract

Numerical methods in reservoir engineering simulations lead to the resolution of unstructured, large and sparse linear systems. The performances of iterative methods employed in simulator to solve these systems are crucial in order to consider many more scenarios.

In this work, we present a way to implement efficient parallel iterative methods on top of a task-based runtime system. It enables to simplify the development of methods while keeping control on parallelism management. We propose a linear algebra API which aims to implicitly express task dependencies: the semantic is sequential while the parallelism is implicit.

We have extended the HARTS runtime system to monitor executions to better exploit NUMA architectures. Moreover, we implement a scheduling policy which exploits data locality for task placement. We have extended the API for KNL many-core systems while considering the various memory banks available. This work has led to the optimization of the SpMV kernel, one of the most time consuming operation in iterative methods.

This work has been evaluated on iterative methods, and particularly on one method coming from domain decomposition. Hence, we demonstrate that the API enables to reach good performances on both multi-core and many-core architectures.

Keywords: Parallel computing, runtime system, domain decomposition, multi-core, many-core, task programming