# Scalable and Efficient Algorithms for Unstructured Mesh Computations

Loïc Thebault

▶ **To cite this version:**

Loïc Thebault. Scalable and Efficient Algorithms for Unstructured Mesh Computations. Distributed, Parallel, and Cluster Computing [cs.DC]. Université Paris Saclay (COmUE), 2016. English. NNT : 2016SACLV088 . tel-01701925

**HAL Id: tel-01701925**

**https://theses.hal.science/tel-01701925v1**

Submitted on 6 Feb 2018

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

université PARIS-SACLAY

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

université PARIS-SACLAY

NNT : 2016SACLV088

# Thèse de doctorat de l'Université Paris-Saclay préparée à l'Université de Versailles

École doctorale n°580
Sciences et Technologies de l'Information et de la Communication
Spécialité de doctorat : Informatique

par

M. Loïc Thébault

## Algorithmes Parallèles Efficaces Appliqués aux Calculs sur Maillages Non Structurés

Thèse présentée et soutenue à Versailles, le 14 octobre 2016.

*Composition du Jury :*

| | | |
|---|---|---|
| M. Florian De Vuyst | Professeur, ENS Cachan | (Président du Jury) |
| M. Emmanuel Jeannot | Directeur de Recherche, INRIA | (Rapporteur) |
| M. David Defour | Maître de Conférences<br>Université de Perpignan | (Rapporteur) |
| M. Vincent Moureau | Chargé de Recherche CNRS, CORIA | (Examinateur) |
| M. Quang Dinh | Ingénieur de Recherche<br>Dassault Aviation | (Examinateur) |
| M. William Jalby | Professeur, Université de Versailles | (Directeur de thèse) |

*Membres Invités :*

| | | |
|---|---|---|
| M. Éric Petit | Ingénieur de Recherche, Intel | (Encadrant) |
| M. Victor Lee | Ingénieur de Recherche, Intel | |

# Remerciements

Ces nombreuses années passées à la fac m'ont permis de rencontrer un grand nombre de personnes d'horizons divers mais partageant tous une ouverture d'esprit et une sympathie commune. Je tiens donc à remercier tous ceux qui ont contribué à rendre ces années agréables et enrichissantes.

Je tiens tout d'abord à remercier mon directeur de thèse, William Jalby, de m'avoir accueilli dans de très bonnes conditions au sein du LI-PARAD et de m'avoir fait confiance durant toutes ces années. J'espère que le pot lui plaira même s'il n'est pas estampillé *Chez Juliette* !

Je souhaite ensuite remercier l'ensemble des personnes ayant accepté de faire partie de mon jury de thèse et de faire le déplacement jusqu'à l'université de Versailles. Toute ma gratitude à mes rapporteurs, messieurs Emmanuel Jeannot et David Defour, pour avoir pris le temps de lire attentivement mon manuscrit ainsi que pour leurs retours. Un grand merci également à messieurs Florian De Vuyst, Vincent Moureau et Victor Lee d'avoir accepté d'être examinateurs de ma thèse.

Je remercie sincèrement Quang Dinh, pour sa disponibilité et sa sympathie durant toutes ces années, mais aussi pour avoir accepté d'être examinateur de ma thèse. Je remercie par la même occasion Dassault Aviation (et leur cantine !) pour leur collaboration lors de mes travaux de thèse.

Je tiens tout particulièrement à remercier Éric Petit, mon encadrant, sans qui je n'écrirais très certainement pas ces quelques lignes aujourd'hui. Il a su m'orienter et m'aider dès ma 1ère année de master. Jamais en manque d'idées novatrices et avec un esprit critique acéré, il m'a énormément appris. Un très grand merci à toi *Sensei*. J'en profite aussi pour remercier Solène, reine du cookie, et fournisseuse officielle du bureau ! Je voudrais également remercier tous les membres du laboratoire avec qui j'ai pu collaborer ou simplement discuter, et en particulier Pablo De Oliveira pour sa bonne humeur et son rire communicateur qui passe au travers les murs et les étages ! Je remercie également Marc Tchiboukdjian pour ses idées et son aide au début de ma thèse.

Mais toutes ces années auraient été bien moins amusantes sans tous les potes de bureau qui ont su détendre l'atmosphère en toutes circonstances. Il sera difficile de retrouver une pareille ambiance de bureau… Un grand merci donc aux camarades de CS, Age of Empires, Tamoul, Seven Wonders… Mais aussi de batailles d'Ikoulas ! Futurs occupants du bureau 220B, si vous trouvez des poissons dans le faux plafond, ne vous inquiétez pas, c'est normal. Je remercie ainsi mes professeurs : JB, professeur émérite de CS, et Chadi, chercheur actif à LoL. Je sais, je n'ai pas été très bon élève… Merci également à Mihail, avec qui on a pu partager et relativiser sur pas mal de galères propres aux thésards. Un grand merci à Nathalie (et Sylvain) pour tous ces coups de main au boulot mais aussi pour ses barbecues à Orléans et à Arcueil. Merci également à Seijilo, notre ex-Dieu du bureau, pour ces quelques mois de grandes théories sur le monde et les forces obscures qui l'entourent. Merci enfin à mon voisin de bureau et de pallier, Clément, mais aussi à la *Girl next door*, Laura, pour tous ces repas et ces soirées jeux improvisées d'un côté ou de l'autre du couloir ;-). Ces quelques mois de voisinage auront été bien cool !

Pour finir, je tiens à exprimer toute ma gratitude envers mes parents, qui ont toujours été là pour moi et qui ont su faire en sorte que je puisse en arriver jusque là aujourd'hui. Des gros bisous à vous deux.

Enfin, je ne peux finir ces remerciements sans Poupou, a.k.a Béa, qui a su à maintes reprises me soutenir et m'encourager, mais aussi me supporter pendant toutes ces longues années ! Merci Poupou de t'être occupée de relire méticuleusement ma thèse, d'avoir mené à la baguette la préparation du pot, et plus encore d'avoir été tout simplement là au quotidien depuis tout ce temps.

P.S. : Merci à toi aussi ! Oui, toi qui te lances dans la grande aventure de la lecture de ma thèse, ou peut-être juste des remerciements, mais c'est déjà un bon début !

*La dure vie de péon-thésard...*

As tu lu le dernier article que je t'ai envoyé ?

Et tu as corrigé le bug dans ton code ?

Il me faudrait aussi les dernières courbes à jour !

Et commences à refléchir à ton manuscrit !

Ca y est, Je suis mort !

Oui monseigneur !

Du travail, encore du travail...
Je n'ai pas le temps de jouer !

*Et un beau jour...*

Travail terminé !!

# Résumé

L'informatique est devenue un outil central dans de nombreux domaines scientifiques. La majorité des recherches menées dans le domaine scientifique et technologique reposent sur la simulation numérique. Ce besoin croissant en simulation a conduit à l'élaboration de supercalculateurs complexes et d'un nombre croissant de logiciels hautement parallèles.

Ces supercalculateurs requièrent un rendement énergétique et une puissance de calcul de plus en plus importants. Les récentes évolutions matérielles consistent à augmenter le nombre de noeuds de calcul et de coeurs par noeud. Cette augmentation continuera certainement jusqu'à atteindre plusieurs milliers de noeuds de calcul, eux-mêmes composés d'un millier de coeurs. Certaines ressources n'évoluent cependant pas à la même vitesse. La multiplication des coeurs de calcul implique une diminution de la mémoire par coeur, plus de trafic de données, un protocole de cohérence plus coûteux et requiert d'avantage de parallélisme. De plus, les architectures actuelles sont de plus en plus hétérogènes.

De nombreuses applications et modèles actuels peinent ainsi à s'adapter à ces nouvelles tendances. En particulier, générer du parallélisme massif dans des méthodes d'éléments finis utilisant des maillages non structurés, et ce avec un nombre minimal de synchronisations et des charges de travail équilibrées, s'avèrent particulièrement difficile. Les approches actuelles basées sur la décomposition de domaine et le coloriage se retrouvent confrontées à ce problème, en particuliers avec les architectures hautement multicoeurs. Il devient donc nécessaire d'explorer de nouvelles approches parallèles.

Afin d'exploiter efficacement les multiples niveaux de parallélisme des architectures actuelles, différentes approches parallèles doivent être combinées. Le parallélisme massif de données se limite habituellement aux problèmes réguliers pouvant être décomposés en grilles de calcul. Ces problèmes sont adaptés aux exécutions de type *programme unique, données multiples* (SPMD) sur CPUs ou *flux d'instruction unique, flux d'exécution multiples* (SIMT) sur GPUs.

Cette thèse propose plusieurs contributions destinées à aller au-delà de cette limitation en adressant les codes et les structures irrégulières de manière efficace. Nous avons développé une approche parallèle hybride par tâches à grain fin combinant les formes de parallélisme distribuée, partagée et vectorielle sur des structures irrégulières.

De plus, une application industriel pouvant difficilement être intégralement réécrite, nous avons exploré le concept de proto-application en offrant une représentation simplifiée. Nous avons développé Mini-FEM, une proto-application représentative de l'application DEFMESH développée par Dassault Aviation. Nous avons ensuite développé la librairie D&C à partir de cette proto-application, puis l'avons validée sur DEFMESH. Nous avons également porté la librairie D&C sur AETHER, un autre code de mécanique des fluides développé par Dassault Aviation. Les résultats obtenus sur la proto-application ont ainsi pu être reproduits sur des applications grandeur réelle utilisant des schémas de calcul similaires.

Nous avons testé notre approche sur des multicoeurs Xeon classiques et sur le Xeon Phi type KNC. Sur 512 coeurs Sandy Bridge avec seulement 2000 sommets par coeur, D&C dépasse l'approche purement MPI de 3.47× et atteint 77% d'efficacité parallèle. Sur 4 KNC, D&C obtient 96% d'efficacité parallèle et une accélération de 2.9× comparé à l'approche MPI commune basée uniquement sur la décomposition de domaine. De plus, la performance obtenue avec D&C est équivalente à 96 coeurs de type Xeon Sandy Bridge. En réduisant l'intensité arithmétique du code, l'efficacité parallèle de D&C sur les 4 KNC descend à 92% mais l'écart avec la version purement MPI augmente à 6.56×.

**Mots-clés:** HPC, multitâche, FEM, maillage non structuré, D&C, Cilk, vectorisation, PGAS, GASPI

# Abstract

Computing science is at the center of a wide range of scientific domains. Almost all current scientific and technological research activity relies on numerical simulations to solve new problems or to design new products. This growing need for numerical simulations results in larger and more complex computing centers and more HPC softwares.

Actual HPC system architectures have an increasing requirement for energy efficiency and performance. Recent advances in hardware design result in an increasing number of nodes and an increasing number of cores per node. In future post-exascale systems, one can reasonably foresee thousands of nodes composed of thousand cores. However, some resources do not scale at the same rate. The increasing number of cores and parallel units implies a lower memory per core, higher requirement for concurrency, higher coherency traffic, and higher cost for coherency protocol. Moreover, current trends result in an increasing usage of heterogeneous architectures.

Most of the applications and runtimes currently in use struggle to scale with the present trend. In the context of finite element methods, exposing massive parallelism on unstructured mesh computations with efficient load balancing and minimal synchronizations is challenging. Current approaches relying on domain decomposition and coloring exacerbate these issues, especially with parallel manycore architectures. HPC users have to explore new paradigms for applications, runtimes, and programming models.

To make efficient use of these architectures, several parallelization strategies have to be combined together to exploit the multiple levels of parallelism. Parallelization approaches exposing massive data parallelism are usually bounded to regular problems. These problems can be decomposed in compute grids and are well suited to Single Program Multiple Data (SPMD) executions on CPUs or Single Instruction Multiple Threads (SIMT) executions on GPUs.

This P.h.D. thesis proposes several contributions aimed at overpassing this limitation by addressing irregular codes and data structures in an efficient way. We developed a hybrid parallelization approach combining the distributed, shared, and vectorial forms of parallelism in a fine grain task-based approach applied to irregular structures.

Moreover, since very large industrial codes cannot be rewritten from scratch, we experimented the concept of proto-application as a proxy between computer scientists and application developers on a real industrial use case. We developed the Mini-FEM proto-application representative of the DEFMESH application from Dassault Aviation. Then, we built the D&C library on top of the proto-application and validated it on the original DEFMESH application. We also ported the D&C library to another fluid dynamic application, AETHER, also developed by Dassault Aviation. The results show that the speedup validated on the proto-application can be reproduced on other full scale applications using similar computational patterns.

We experiment our approach using standard Xeon multicores and Xeon Phi KNC manycores. On 512 Sandy Bridge cores, we overpass the pure MPI approach by up to 3.47× and reach 77% of parallel efficiency with only 2000 vertices per core. By running an intensive computation kernel on 4 Xeon Phi, we achieve an excellent parallel efficiency of 96% and a 2.9× speedup compared to the common approach only based on MPI domain decomposition. By reducing the arithmetic intensity by a factor of 100×, the parallel efficiency of the D&C library decreases to 92% but becomes 6.56 times faster than the pure MPI version. Finally, running on 4 Xeon Phi, D&C has similar performance to 96 Intel Xeon Sandy Bridge cores.

**Keywords:** HPC, multithreading, FEM, unstructured mesh, D&C, Cilk, vectorization, PGAS, GASPI

# CONTENTS

**Introduction** **1**

**I State of the Art** **9**

**1 Heterogeneous Hardware Architectures** **11**

1.1 Introduction . . . . . 11

1.2 80 Years of Innovations Leading to the Multicore . . . . . 12

1.3 GPU Architecture . . . . . 15

1.4 Manycore Architecture . . . . . 16

1.5 Supercomputers . . . . . 18

1.6 Experimental Environment . . . . . 19

1.6.1 Intel Sandy Bridge . . . . . 19

1.6.2 Curie . . . . . 20

1.6.3 Anselm and Salomon . . . . . 21

1.6.4 MareNostrum . . . . . 22

**2 HPC Parallel Programming Models** **23**

2.1 Introduction . . . . . 23

2.2 MPI Model - Message Passing Interface . . . . . 25

2.2.1 Two-sided Communications . . . . . 26

2.2.2 Collective Communications . . . . . 27

2.2.3 One-sided Communications . . . . . 28

2.3 PGAS Model - Partitioned Global Address Space . . . . . 31

2.3.1 Pros and Cons of One-sided Communications . . . . . 32

2.3.2 GASPI and its GPI-2 Implementation . . . . . 33

2.4 OpenMP Worksharing Model . . . . . 36

2.5 Task Model . . . . . 37

2.5.1 OpenMP Tasks . . . . . 38

2.5.2 Cilk Plus . . . 40
2.5.3 An Example of CilkView Usage . . . 42

**3 Parallelization Strategies for Finite Element Methods 43**
3.1 Introduction . . . 43
3.2 FEM Computation over Elements . . . 44
3.3 Compressed Storage Formats for Sparse Matrices . . . 45
3.4 Domain Decomposition Method . . . 47
3.4.1 Halo Exchange . . . 48
3.4.2 Multilevel Graph Partitioning . . . 49
3.5 Hybrid Parallelism . . . 50
3.5.1 Coloring of Unstructured Meshes . . . 51
3.5.2 Divide And Conquer Parallel Algorithms . . . 53
3.6 Vectorization on Unstructured Meshes . . . 54
3.7 Optimizing Locality in Mesh Computation . . . 56

**II Contributions 59**

**4 Introduction to Code Modernization 61**
4.1 Introduction . . . 61
4.2 Dassault Aviation’s Industrial Applications . . . 62
4.2.1 DEFMESH Mesh Deformer Application . . . 62
4.2.2 AETHER Aerodynamic and Thermodynamic Application . . . 63
4.2.3 List of Use Cases . . . 63
4.3 Proto-Application Concept . . . 64
4.4 D&C Library Integration in AETHER . . . 65
4.5 Conclusion . . . 65

**5 Node Level Parallelism 67**
5.1 Introduction . . . 67
5.2 Shared Memory Divide and Conquer on Unstructured Meshes . . . 68
5.2.1 Notations Used . . . 69
5.2.2 Recursive Bisection . . . 69
5.2.3 Locality Improvement . . . 70
5.2.4 Numerical Stability of the Results . . . 72
5.2.5 D&C Matrix Storage Format . . . 73
5.2.6 Precomputation of the D&C Tree . . . 73

5.2.7 Cilk Plus Implementation . . . . . 74
5.2.8 OpenMP Implementation . . . . . 75
5.3 Experimental Results . . . . . 77
5.3.1 Experimental Setup . . . . . 77
5.3.2 FEM Assembly . . . . . 78
5.3.3 Solver . . . . . 83
5.3.4 Comparison Between Cilk Plus and OpenMP 3.0 Tasks . . . . . 83
5.4 Conclusion . . . . . 87

**6 Core Level Vectorization** **89**
6.1 Introduction . . . . . 89
6.2 Coloring for Efficient Vectorization . . . . . 90
6.2.1 Implementing the Vectorization . . . . . 90
6.2.2 Color-Based Vectorization Model . . . . . 91
6.2.3 Longest Colors Strategy . . . . . 92
6.2.4 Bounded Colors Strategy . . . . . 93
6.2.5 Reduction of Memory Consumption and Synchronization Needs . . . . . 95
6.2.6 Structuredness, Vectorization and Locality . . . . . 96
6.2.7 Vector Length Sensitivity Study . . . . . 97
6.3 Experimental Results . . . . . 98
6.3.1 Single Node Experiment . . . . . 99
6.3.2 Strong Scaling Experiment with Increased Arithmetic Intensity . . . . . 100
6.3.3 KNC Experiments . . . . . 101
6.4 Conclusion . . . . . 104

**7 Distributed Level Parallelism** **107**
7.1 Introduction . . . . . 107
7.2 Potential Gain of Communication Optimization . . . . . 108
7.2.1 Benefits and Limitations of Communication Overlap . . . . . 109
7.3 Bulk-Synchronous Communications Using GASPI . . . . . 110
7.4 Asynchronous and Multithreaded Communications Using GASPI . . . . . 111
7.4.1 Communication Level Version . . . . . 112
7.4.2 Bounded Communication Size Version . . . . . 114
7.5 Experimental Results . . . . . 118
7.5.1 Impact of the Number of Processes per Node . . . . . 118
7.5.2 Bulk-Synchronous Comparison on a Single Node Experiment . . . . . 119
7.5.3 Strong Scaling Experiment Using Standard Arithmetic Intensity . . . . . 121

7.5.4 Multiple KNC Experiment Using Standard Arithmetic Intensity . . . . . . . . 123
7.5.5 All-in-One D&C Version . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124
7.6 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 127

**Conclusion and Perspectives** **131**

**List of Figures** **139**

**Bibliography** **143**

# INTRODUCTION

# General Context

Computing science is at the center of a wide range of scientific domains. Most of research programs rely on numerical simulations to solve new problems and to design new products. Many examples can be found in various domains. In the context of aviation, planes, which were originally designed through drawing-boards and improved after flight tests, have since early 1970s started to be computer-aided designed. Today, they are fully conceived through computers and involve different domains of research such as structural mechanic, aerodynamic, and electromagnetism. At the end of 2013, a large human brain project aimed to simulate and better understand the human brain has started and should continue until 2024. Numerical simulation has also advantageously replaced nuclear weapons tests which were made up to 1974 in France. Recently, deep learning approach on HPC system has defeated the best world player of the Go game. Nowadays, simulation is present everywhere. It includes the elaboration of car engines and car design, the conception of new pharmaceuticals, the simulation of chemical reactions, the analysis of huge amount of data, and so on.

## Computational Fluid Dynamics

The domain of Computational Fluid Dynamics (CFD) in particular is closely linked to computing science. CFD is a branch of fluid mechanics which consists in simulating the interaction of liquids and gases through the resolution of numerical equations involving the fluid flows. Fluid mechanics is a broad topic. It has a wide range of applications including for instance mechanics, chemistry, geophysics, aerospace, astrophysics, biology, life science, or yet numerical weather prediction.

The first step of a CFD problem is the definition of the geometry representing the initial problem. Then, the computational domain is transformed into a mesh. A mesh is a discretization of a space into cells, typically millions to tens of billions, which can have different shapes, e.g. triangle in 2D and tetrahedron in 3D. Meshes can be regular, i.e. all the cells have the same shape and geometry, or not. Irregular meshes as the one illustrated in Figure 1 are mostly used in CFD problems. Indeed, some parts of the computational domain, such as the surfaces generating turbulences and their neighboring areas, are more important than others. The cells are therefore smaller and in bigger quantities in those areas. Several discretization methods exist such as Finite Volume Method (FVM), Finite Element Method (FEM), or Finite Difference Method (FDM). In this thesis we focus on FEM applications. They are presented in more details in Section 3.2.

Once the mesh is constructed, the numerical methods to use have to be defined. Depending on

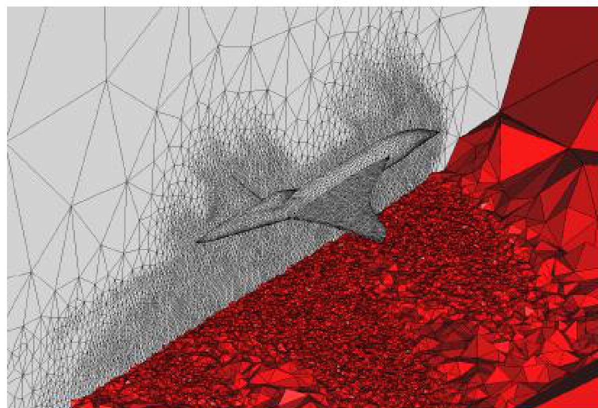


Figure 1: Example of 3D unstructured mesh.

the problem to solve and the desired level of approximation, the numerical equations may vary. The fundamental CFD numerical equations are the Navier-Stokes equations. They are used in most of CFD simulations. However, these equations can be simplified into Euler equations, simplified again into full potential equations, or linearized potential equations. There is a trade-off between the desired accuracy of the problem to solve and the resources and amount of time allocated to it.

Lastly, the boundary conditions and the initial values are defined. The former corresponds to the fluid properties at the boundaries of the problem. The numerical simulation can start once all these preprocessing steps are completed.

### High Performance Computing

The needs in accuracy and the complexity of problems, as those encountered in CFD applications, are increasing as well as the size of problems to solve. This leads to a growing need for performance in numerical simulations and as a result to larger and more complex computing centers. Before 2000s, the evolution of computers performance was mostly due to the increase of frequency of the Central Processing Unit (CPU) and of the memory bus. But this race for higher frequencies has reach a wall of energy consumption and heat dissipation. The performance evolution has therefore take a turn which consist in increasing the number of computing units composing a CPU, i.e. the cores, instead of increasing their frequency. However, designing such new architectures and developing applications able to efficiently make use of them is complex. This is the essence of High Performance Computing (HPC).

### Evolution of Hardware Architectures

The evolution of hardware architectures driven by the increasing requirement for performance and energy efficiency has led to more and more complex HPC systems. Recent advances in hardware design result in an increasing number of distributed compute nodes, an increasing number of cores per node, and larger vectorial instruction units. In future post-exascale systems, one can reasonably foresee thousands of nodes with thousand cores. However, some resources do not scale at the same rate. The increasing number of parallel units implies a lower memory per core, higher requirement for concurrency, higher coherency traffic and higher cost for coherency protocol. Additionally, the development of new parallel architectures has explored different ways. On the one hand, there is the classical CPU model which consists in a small number of powerful cores aiming at executing as quickly as possible a sequential workload. On the other hand, there is the Graphic Processing Units (GPUs) model originally designed for graphic purpose. It is composed of an important number of simple cores and relies on the available parallelism within an application to hide computation latency. However, while CPUs are evolving to become more and more parallel, GPUs have became more generic and more programmable. These two models may in the future lead to a unified architecture model. The recent Xeon Phi manycore from Intel is at the frontier between these two models and illustrates this new trend. The different evolutions of parallel models result in an increasing usage of heterogeneous architectures. Modern supercomputers are composed of many distributed compute nodes able to contain at the same time several multicore CPUs and several manycore or GPU accelerators.

### Parallel Programming Models

In order to exploit this variety of computing resources, a variety of parallel programming models has emerged. Recent architectures and their associated parallel execution models involves multiple cooperating processes simultaneously executing the same program on different data. Parallel programming approaches have to exploit distributed and shared memory systems, but also vector operators.

Processes are commonly used across distributed compute nodes. They are independent compute flows with their own private memory. Interactions between processes are handled by messages evolving through the network. The message passing model and its Message Passing Interface (MPI) Application Programming Interface (API) has become a standard to handle communications and synchronizations [1, 2, 3]. MPI provides an API to perform one-to-one and collectives operations. However, other approaches exist such as the Partitioned Global Address Space (PGAS) model. The Global Address Space Programming Interface (GASPI) API [4, 5, 6] is one of the most advanced implementations today. PGAS model consists in a global memory space shared among the distributed processes. Remote process can directly write to another process memory through Remote Direct Memory Accesses (RDMA).

On the other hand, threads are used to generate parallel streams of execution in a shared memory context. Threads are entities within a process which rely on its virtual address space and executable code. They are therefore lighter than the processes and exploit the shared memory to communicate or synchronize themselves. Several programming interface for multithread parallelism exist. The most common are the POSIX threads, a.k.a. pthreads, the widely used OpenMP pragma-based model [7, 8, 9, 10], or yet the Intel Cilk Plus [11, 12, 13] and Threading Building Blocks (TBB) [14] task-based models.

The lower memory consumption and lower overheads of the multithread approach have advocated for hybrid programming models. Only one process is map to each compute node, or eventually processor, while all shared resources within the node are handled by threads. Distributed memory libraries such as MPI or GASPI have adapted to handle multithreaded communications.

In addition to these hybrid process and thread models, the vector resources located at core level advocate for Single Instruction Multiple Data (SIMD) vectorization. Moreover, GPUs commonly uses the Single Instruction Multiple Thread (SIMT) model in which a large amount of threads execute in parallel exactly the same instruction flow on different data. As a result, combining all these programming models in an efficient way leads to a severe challenge for performance scalability.

## Problematic

To make efficient use of recent heterogeneous architectures, several parallelization strategies have to be combined together to take advantage of the multiple levels of parallelism. The resulting hybrid programming models combining distributed communications and shared memory threading models are difficult to implement. Each layer of parallelism have to efficiently exploit the corresponding architecture level, while the different layers have to properly cohabit with each others. The message passing model, e.g. MPI, constraints to split and duplicate work and memory which may strongly degrade the code efficiency at scale. The OpenMP loop parallelization is commonly used in addition to MPI to reduce the number of duplications. However, this loop parallelization approach lets many sequential parts in the code which also degrades the scalability. For instance, according to Amdahl's law, an application parallelized at 97% and executed on 256 cores, can only attain a 29.6$\times$ speedup with 11% of parallel efficiency. Even with 99.9% of the code parallelized, the maximal speedup is 204$\times$ with only 79% of parallel efficiency. This is illustrated in Figure 2.

As a result, most of the applications and runtimes currently in use struggle to scale with the present trend [9]. In the context of finite element methods, exposing massive parallelism on 3D unstructured meshes computation with efficient load balancing and minimal synchronizations is challenging. Current approaches relying on domain decomposition and mesh coloring exacerbate these issues, especially with new manycore accelerators. In particular, the FEM assembly state-of-the-art parallelization approaches [15, 16, 17, 18], discussed in details in Chapter 3, show inherent limitation preventing them to be efficient on future manycore systems.

To take benefit from the new systems, two major aspects prevail when designing an application:

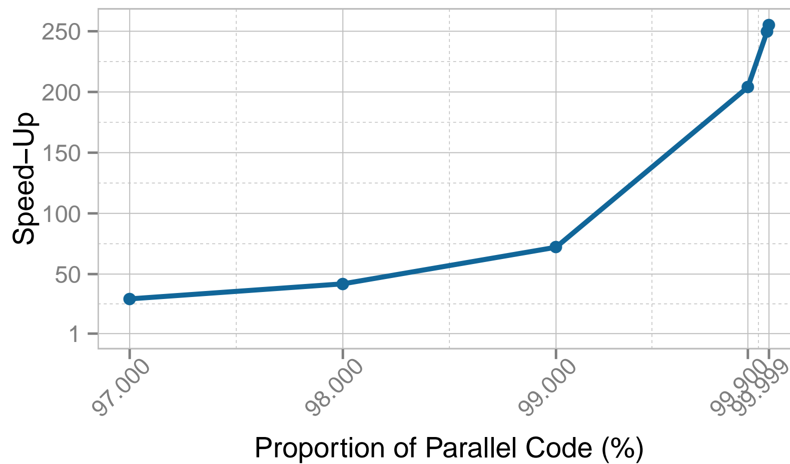
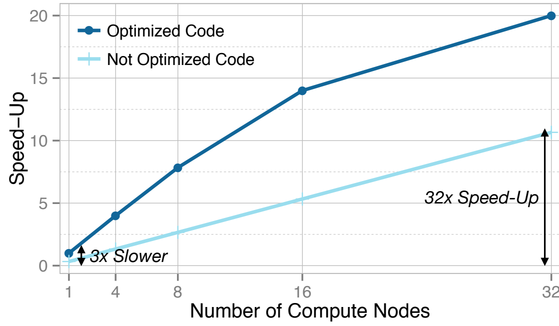


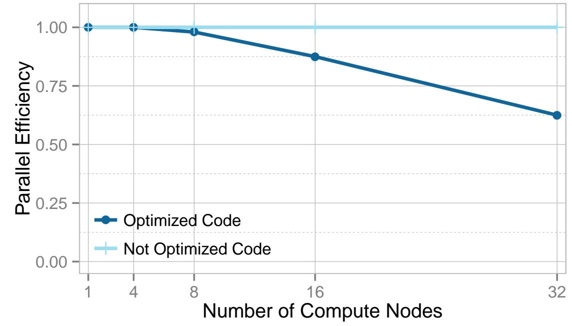

Figure 2: Speedup according to the proportion of parallelized code on a 256 cores execution.

concurrency and locality. Concurrency consists in creating enough independent compute tasks to feed all the concurrent resources, i.e. nodes, cores and vectors. Every single serialization and sharing on the critical path is a future bottleneck. Remember Amdahl: the more parallelism in the system, the higher the time proportion of the sequential code is. We also need to expose locality at core level, i.e. caches, socket level including hardware accelerators, and network level. But, we cannot only rely on static locality at core, socket and network level, we should also ensure data communication locality. The implicit assumption that memory coherency protocol between cores is a free resource for data sharing does not hold anymore [19, 20]. The communication cost to exchange data and to synchronize in shared memory has to be taken into account when conceiving parallel algorithms. Therefore, shared memory programming is getting conceptually closer to distributed programming.

Moreover, a well performing application on a small number of cores, e.g. a compute node, is not necessarily efficient on a larger number of compute nodes. Conversely, a perfectly scaling application does not necessarily exploit efficiently the underlying resources and can be far from the peak performance of the compute nodes. As illustrated in Figure 3, an application developed with a badly performant threading model and having poor node performance, will easily scale without loosing in parallel efficiency compared to its sequential execution. In the opposite, a well optimized application which exploits efficiently the shared resources of a compute node, will be difficult to scale up without sacrificing parallel efficiency. Indeed, a perfectly efficient application which runs on a single thread and uses 100% of the computation and memory resources, will necessarily reduce its resource usage per thread when increasing their number. Therefore, when developing new applications, if developers have a high scalability but a low usage of the FLOP peak performance, it is time to optimize the core performance and the threading model at node level. If the optimized threading model does not scale anymore, it is then time to rework on the distributed communication pattern. Both node performance and scalability are necessary to efficiently take advantage of supercomputer resources. Another important aspect when developing new applications is the programmability. An algorithm optimized with the greatest care but impossible to develop and maintain will highly loose in interest.




(a) Speedup compared to best sequential performance




(b) Parallel Efficiency

Figure 3: Typical speedup and parallel efficiency depending on the level of code optimization.

# Contributions

In this thesis, we propose a hybrid parallelization approach for finite element methods. Our hybrid approach targets the three main levels of parallelism within a supercomputer. The distributed memory parallelism between compute nodes, the shared memory parallelism inside a compute node, and lastly the vectorial parallel units within a compute core. It is based on domain decomposition at distributed memory level using asynchronous and multithreaded one-sided communications, recursive Divide & Conquer (D&C) at shared memory level, and finely tuned coloring heuristic for vectorization at core level. The rational is to adopt the best suited strategy at each level of the architecture while using architecture oblivious design for the algorithms. Therefore, our code can adapt with a low intrusion and few architecture aware parameters to the underlying hardware characteristics such as cache sizes or vector lengths.

The first contribution concerns the shared memory parallelism. We replace the rigid loop-based approach by a versatile and efficient recursive approach with architecture oblivious design. This approach relies on the divide & conquer principle coupled with the Intel Cilk Plus runtime to generate concurrency and locality with unstructured meshes used in FEM applications. The D&C recursive approach naturally exposes parallelism and allows to improve locality both in the mesh computation and in the associated sparse matrix system. In this method, we recursively bisect the mesh and permute the elements, which results in a parallel and recursive tree of tasks. The Cilk Plus runtime allows to generate a large amount of parallel tasks and to minimize the synchronizations between them. It also efficiently handles dynamic load balancing through a work-stealing scheduler. This first contribution resulted in a publication published at the PARCO workshop [21] and a second one at the HPCC workshop [22].

The second contribution targets the vectorization in very small data partitions of unstructured meshes. To exploit the vectorial resources of compute cores, we propose an approach which efficiently exposes vectors of independent elements within each previously generated task. This approach is based on mesh coloring and uses the Cilk Plus Array Notation to generate the vectorial instructions. This second contribution together with the D&C approach led to another publication published at the PPoPP international conference [23].

The third contribution concerns distributed memory parallelism. In our approach, we use the standard domain decomposition approach coupled with GASPI asynchronous one sided communications. We propose a communication pattern compatible with the D&C fine grain task parallelism used at shared memory level to multithread the communications and overlap them with computation. This contribution is about to be submitted in a journal.

Finally, since very large industrial codes cannot be easily rewritten from scratch without a proven reason and measured risks, we experiment the concept of proto-application as a proxy between computer scientists and application developers on a real industrial use case. We developed a proto-application, called Mini-FEM, representative of our target CFD applications from Dassault Aviation, named DEFMESH. Then, we developed on the proto-application an open-source library, called DC-lib, implementing the three contributions described above. The D&C library have been successfully ported back into the original DEFMESH application and validated in another industrial application, AETHER, developed by Dassault Aviation. The results show that the speedup obtained on the proto-application can be reproduced on other full scale applications using similar computational patterns. Dassault Aviation is currently integrating the D&C library in its production codes and starts to reproduce the same proto-application based code modernization on other applications. This last contribution has led to a publication at the Alchemy workshop [24].

We have evaluated our D&C library on different systems based on standard clusters of NUMA nodes using Xeon multicores and on a cluster of Intel Xeon Phi manycores. D&C achieves a high parallel efficiency, a good data locality, an improved bandwidth usage, and competes on current nodes with the state-of-the-art pure MPI version with a minimum 10% speedup. By using 512 Sandy Bridge cores and only 2000 vertices per core, D&C achieves 77% parallel efficiency and overpasses the pure MPI approach by 3.47×. On 4 Xeon Phi, D&C has a performance similar to 96 Intel E5-2665 Xeon Sandy Bridge cores and 96% parallel efficiency on the 240 physical cores. It shows an impressive 373× strong scaling speedup and is 2.9× faster than the common approach using only MPI domain decomposition. Using a lower intensive computation kernel, the speedup over the pure MPI version grows up to 6.56×.

## Overview

The first part of this thesis presents the context and the state-of-the art in the domain of parallelization of irregular applications based on unstructured meshes. The Chapter 1 details the evolution of hardware architectures which has led to heterogeneous and complex architectures more difficult to exploit efficiently. Chapter 2 presents the existing programming models in use to parallelize applications at distributed, shared, and core level of such new architectures. Finally, Chapter 3 describes the different approaches used in FEM applications to generate concurrency and to improve data storage locality.

Then, the second part contains the contributions made during this thesis. Chapter 4 explains the concept of proto-applications used to ease the development of new algorithms and largely exploited during the thesis. It also presents the different industrial applications which have been modified to incorporate our contributions and their associated use cases. The Chapter 5 targets the contributions made at node level on shared memory parallelism and presents in details our D&C approach. Then, we have focused on the development of a new vectorization heuristic to improve the vectorization ratio of small data sets fitting in core caches as detailed in Chapter 6. Lastly, we propose in Chapter 7 a new approach to communicate between distributed processes in an asynchronous manner using the PGAS model and one-sided communications. This approach exploits the D&C multithreaded tasks to parallelize the communications and to recover communication and computation.

The last concluding part summarizes all the contributions to give a global picture of the work achieved during this Ph.D. thesis and proposes future works.

# Part I

# State of the Art

CHAPTER 1

# HETEROGENEOUS HARDWARE ARCHITECTURES

**Contents**


**1.1 Introduction . . . . . 11**
**1.2 80 Years of Innovations Leading to the Multicore . . . . . 12**
**1.3 GPU Architecture . . . . . 15**
**1.4 Manycore Architecture . . . . . 16**
**1.5 Supercomputers . . . . . 18**
**1.6 Experimental Environment . . . . . 19**
1.6.1 Intel Sandy Bridge . . . . . 19
1.6.2 Curie . . . . . 20
1.6.3 Anselm and Salomon . . . . . 21
1.6.4 MareNostrum . . . . . 22


## 1.1 Introduction

The need for computation aid has arisen thousands years ago. It first appeared as counting devices such as the Sumerian abacus around 2500 BC. While computing devices have hardly evolved during millenniums, they have considerably changed our lifestyle starting from the second half of the 20th century. Nowadays, the needs for computation are entirely handled by computers. The progresses made in the different domains of research have drastically increased the computation power but also the complexity of computing devices. The general purpose Central Processing Unit (CPU) model which marked the beginning of the computers has pursued its evolution. It results in smaller manufacturing processes, higher frequencies, and later, in more parallel resources. This is detailed in Section 1.2. But other models have appeared. The specific needs of parallelism in the domain of graphic processing have led to the Graphic Processing Unit (GPU) model described in Section 1.3. More recently the Intel Xeon Phi manycore model, presented in Section 1.4, has appeared as a compromise between these CPU and GPU model. The larger computing centers, a.k.a supercomputers, are a clustering of distributed computed nodes composed of these different technologies. Most of modern supercomputers compute nodes contains at the same time several multicore CPUs, and several accelerators which can be either GPGPUs or Xeon Phi. It results in a severe challenge for performance scalability. The most powerful supercomputers are presented in Section 1.5 as well as those used during this thesis in Section 1.6.

## 1.2 80 Years of Innovations Leading to the Multicore

Charles Babbage is often cited as the father of the computer. In 1833, he conceived the idea of an Analytical Engine composed of mechanical arithmetic units and punched card memory. This Analytical engine is considered as the first general-purpose computer. A century later, in 1936, Alan Mathison Turing conceived a mathematic model aimed to manipulate a list of symbols recorded on a strip according to a table of rules. This model known as the Turing machine was at the origin of what will become the modern computer. Then, the Second World War was an important driving force in the elaboration of computers. In 1943, many scientists including John von Neumann searched for a way to increase the computation rate of ballistic trajectory originally handled by humans and analog computers. Research resulted to the von Neumann architecture which is still used in modern computers. This model, illustrated in Figure 1.1, is composed of an Arithmetic-Logic Unit (ALU), a control unit which schedules the operations, a memory containing the program and the data, and input and output (I/O) operations. Their work led in 1945 to a huge machine of 160 square meters named Electronic Numerator, Integrator, Analyzer and Computer (ENIAC). The ENIAC was able to sum 5000 numbers in one second and to compute in 20 seconds ballistic trajectory which would have required 3 days for humans. In 1945, inspired by the von Neumann works, Alan Mathison Turing wrote the first detailed project of a computer, the Automatic Computing Engine (ACE).

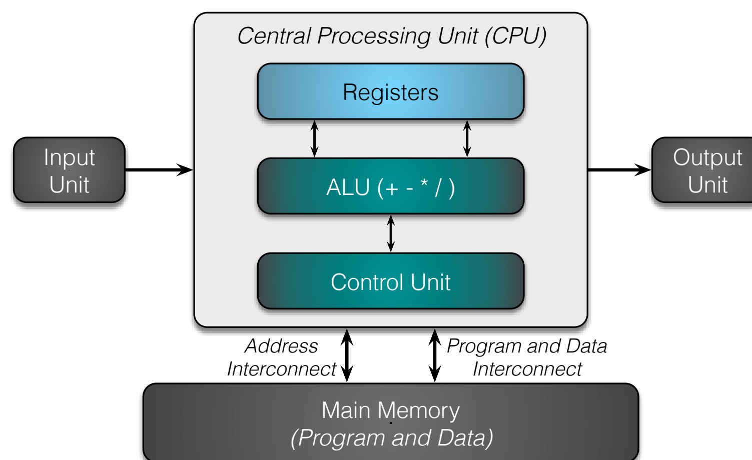



Figure 1.1: Representation of the Von Neumann architecture. It is composed of an Arithmetic-Logic Unit (ALU), a control unit, a main memory, and I/O operations.

Later, in 1971, Marcian Hoff, an Intel's engineer, designed the first processor integrated in a single ship which rapidly resulted in the first public microprocessor, the Intel 4004. In 1977, Apple released its first computer and four years later, IBM presented its first Personal Computer (PC). The microprocessor has then quickly evolved, following a law conceived by Gordon Moore, a co-founder of Intel Corporation. The Moore's law indicates that the number of transistors per chip doubles every two years. Following this law had been a challenge for the industry during 50 years but it had relentlessly continued to apply until recently. As illustrated in Table 1.1, in 1971, the Intel 4004 microprocessor executed 60,000 instructions per second, i.e. around 2.5 million times less than the actual Intel Core i7 6700K. The manufacturing process has been divided by more than 700 resulting in 760,000 times more transistors per chip which are smaller than most viruses.

In parallel to the increasing number of transistors, several new technologies have emerged. Originally, an instruction flow was sequentially executed by a processor. Each instruction requires one to several clock

Table 1.1: Evolution of microprocessors over 45 years.

| Date | Name | Transistor count | Process | Frequency | Address space | MIPS |
|---|---|---|---|---|---|---|
| 1971 | Intel 4004 | 2300 | 10 $\mu m$ | 108 $kHz$ | 4 $bits$ | 0.06 |
| 2015 | Intel i7 6700K | 1,750,000,000 | 14 $nm$ | 4 $GHz$ | 64 $bits$ | $\sim 160,000$ |

cycles. In 1960, IBM introduced the instruction parallelism, also known as pipelining [25]. Pipelining permits the execution of several independent instructions in a same clock cycle. The processors using this technology were named superscalar processors. Tomasulo *et al.* from IBM also tried to reduce the execution time of a set of instructions by sorting them while preserving the dependencies in order to fill all the pipeline stages [25]. This optimization is called out-of-order execution and is still in use in modern CPUs. The Control Data Corporation (CDC) 6600 created in 1965 was the first supercomputer designed by Seymour Cray making use of multicore superscalar out-of-order processors. Intel had waited until 1993 to produce its first superscalar processor, the P5 Pentium.

Later, to improve memory access latencies, the idea appeared to reduce the distance between the main memory and the processor. A small amount of very fast memory, called cache memory, was integrated to the CPU [26]. Least recently accessed data from main memory are stored in caches by continuous chunks of memory, called cache lines. If any data in the cache line is reused in the near future before being evicted by new data, there is no need to access main memory again. This way, the short distance with the caches enables fast memory accesses. However, the capacity of these caches is reduced. Indeed, as illustrated in Table 1.2, there is a trade-off between memory capacity versus latency and bandwidth. The closer to the memory is the processing unit, the faster and the smaller it is. Therefore, to benefit from this acceleration, it is required to enable spatial and temporal locality. Several optimizations appeared to improve the cache benefits such as the prefetching, which aims to bring data blocks to cache before it is actually needed. An ensuing branch predictor optimization, such as the L-TAGE branch predictor proposed by Seznec *et al.* [27], has followed to anticipate which branch in a code is most likely to be taken and enable appropriate prefetching.

Table 1.2: Trade-off between memory capacities and latencies.

| Memory | Register | Cache | Main Memory | Storage Disk |
|---|---|---|---|---|
| Capacity $(byte)$ | $\mathcal{O}(Kilo)$ | $\mathcal{O}(Mega)$ | $\mathcal{O}(Giga)$ | $\mathcal{O}(Tera)$ |
| Latency $(cycle)$ | $\mathcal{O}(1)$ | $\mathcal{O}(10)$ | $\mathcal{O}(100)$ | $\mathcal{O}(10,000)$ |

Moreover, to save cycles during context switch or when waiting for new instructions, a Simultaneous Multi-Threading (SMT) technology has emerged [28]. It permits the simultaneous execution of multiple independent threads. In early 2000s, Intel introduced with its Pentium 4 a two-thread SMT technology, known as Hyper-threading. Two threads can be simultaneously executed and share the same pipelines, caches and registers.

In the early 2000s, when the manufacturing process continued below 90 nanometers, the automatic benefit brought by the increasing number of transistors and the higher frequencies began to fail because of overheat issues. To overcome this new limitation in sequential performance gains, the performance gain has continued, not by increasing frequency but by multiplying the number of cores [29]. The microprocessors evolution has led to the actual multicore CPUs. The IBM POWER4 created in 2001 was the first commercially available multicore chip. However, while in the past applications could freely benefit from most of the previous hardware evolution, this new trend necessitates to raise parallelism in

Table 1.3: Increasing vector length of modern microarchitectures and their associated Instruction Set Architecture (ISA).

| ISA | SSE | AVX | AVX-2 | AVX-512 | IMCI |
|---|---|---|---|---|---|
| Size (*bit*) | 128 | 256 | 256 | 512 | 512 |
| Year | 1999 | 2011 | 2013 | 2016 | 2012 |
| Microarchitecture | Pentium III | Sandy Bridge | Haswell | KNL | KNC |

the code to get performance improvements. Additionally, microprocessors start to integrate larger and larger vectorial units which allow to apply simultaneously a same instruction on a vector of data up to 512 bits. This is illustrated in Table 1.3.

But the increasing number of cores accessing a shared memory at a same time has induced critical memory contentions. Non Uniform Memory Access (NUMA) architectures have been introduced to address this problem by providing separate memory for each core. As described in Figure 1.2, each distributed compute node, i.e. NUMA node, has its own separate memory. Inside the NUMA nodes, each core has its own private Level 1 (L1) cache, and one or two additional cache levels shared with the other cores of the same processor. The main Random Access Memory (RAM) is shared among all the processors of the node. While data exchanges between distributed processes are handled trough communications as explained in the next chapter, every core within a shared memory space has a global vision of the main memory. However, the access times may vary a lot according to the proximity of the accessed data. Moreover, since multiple cores can work on a same cache line, NUMA systems provide a costly coherency protocol to move data between memory banks in order to maintain cache coherence across shared memory. This protocol uses inter-processor communications between cache controllers to keep a consistent memory image over the caches storing a same memory location. This may induce important overheads if several cores attempt to access the same memory region in rapid succession. Data locality is therefore more and more critical as the size of the NUMA nodes increases to reduce memory duplication and bandwidth contention. AMD implemented NUMA in 2003 with the Opteron using HyperTransport interconnection, while Intel provided NUMA compatibility in 2007 with its Nehalem using Intel Quick Path Interconnect (QPI) interconnection.

Similarly, the increasing number of cores and larger caches within a same processor led to studies on Non Uniform Cache Access (NUCA) [30, 31]. The traditional cache design consists of a centralized

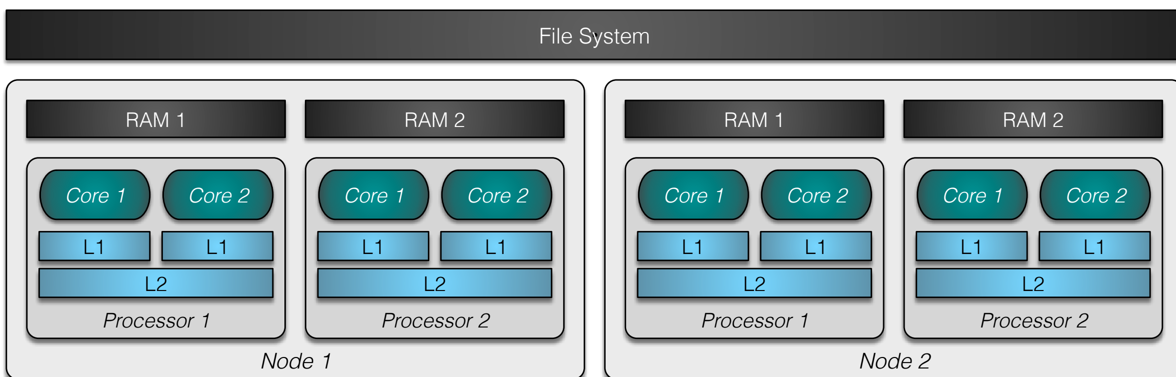



Figure 1.2: Cluster representation with two distributed NUMA nodes, two processors per node, and two cores per processor.

decoder which drives physically partitioned memory banks accessed with a homogeneous time. Data access is therefore limited by the slowest memory bank. The increasing wire delays induced by larger and more distant caches increase the access times. In NUCA architectures, the cache is broken into smaller memory banks which can be accessed at different latencies. The D-NUCA design proposed in [30] has better scalability properties since accesses are serviced with different close banks.

Despite all the recent innovations, after 50 years of efforts, the end of Moore law has been officially announced. The *More than Moore* industry roadmap released in 2010 [32] will for the first time lay out a research and development plan that is not centered on Moore's law. The projection of the law to 2020 would lead to a two or three nanometers process. This represents about ten atoms length. At this scale, with the actual silicon CMOS technology, transistors are going to be unreliable due to quantum effects. The recent extreme UV process with its 13.5 nm wavelength, compared to the actual 193 nm, will not overcome this.

However, it is unlikely that the evolution stops. It will lead to innovations in other domains. Several new approaches are currently developed in laboratories and may bring back the scaling of the last decades. Among them, we can cite the research made on carbon nanotubes and graphene transistors, but also on superconductors. The quantum computing might also bring significant speedups in certain domains of application. There are also the tri-gate transistors or yet the 3D chips with multiple layers of components on a single die. Lastly, more and more devices are likely to be integrated inside CPUs as it is already the case with the integrated GPU-CPU approach named Accelerated Processing Unit (APU). The lower distance between devices will result in faster data transfer and lower power requirements.

## 1.3 GPU Architecture

Image processing naturally offers large amount of parallelism. To speed up such parallel applications, specific architectures called Graphics Processing Units (GPUs) have been designed. GPUs are composed of a large number of simple dedicated cores efficient to handle this kind of applications. These cores have low frequencies and small caches and allow to execute a large number of threads in parallel. They exploit SIMD paradigm where a same operation is applied to a vector of contiguous data. They are used as complementary devices, known as accelerators, linked to the CPU through the parallel PCI-Express interconnect.

Graphic chips were initially used to accelerate the memory-intensive work of texture mapping and rendering polygons of arcade games since 1970. Then, exploiting GPU architecture stayed very specific to graphical applications such as video games. However, due to their performance potential on embarrassingly parallel applications, GPUs started to become popular especially in domains involving matrix and vector operations. In order to be easier to program, actors have developed simpler dedicated programing languages such as CUDA for Nvidia GPUs, or OpenCL, for a wide range of architectures including both Nvidia and AMD GPUs. GPUs have then evolved to incorporate new features enabling general purpose programming and getting conceptually closer to CPUs. Among them, there is the integration of double precision units for scientific application, and the apparition of hardware-managed multi-level caches. Most modern supercomputers at the top of the Green500 [33] which rewards the most power efficient machines are equipped with GPUs.

The Nvidia Maxwell GM200 architecture created in 2015 is illustrated in Figure 1.3. It is composed of 24 Streaming Multiprocessors (SMM) grouped in 6 Graphics Processing Cluster (GPC), a shared L2 cache of 3 MB, and 6 64-bit memory controllers resulting in a 384-bit memory interface. Each SMM is composed of a 96 KB shared memory, 32 load / store units, 32 Special Function Units (SFU) (e.g. sin, cosine, reciprocal, square root), and 128 cores resulting in a total of 3072 cores and 8 billions transistors. The Nvidia GeForce GTX Titan X is the only card embedding the Maxwell GM200 GPU in addition to

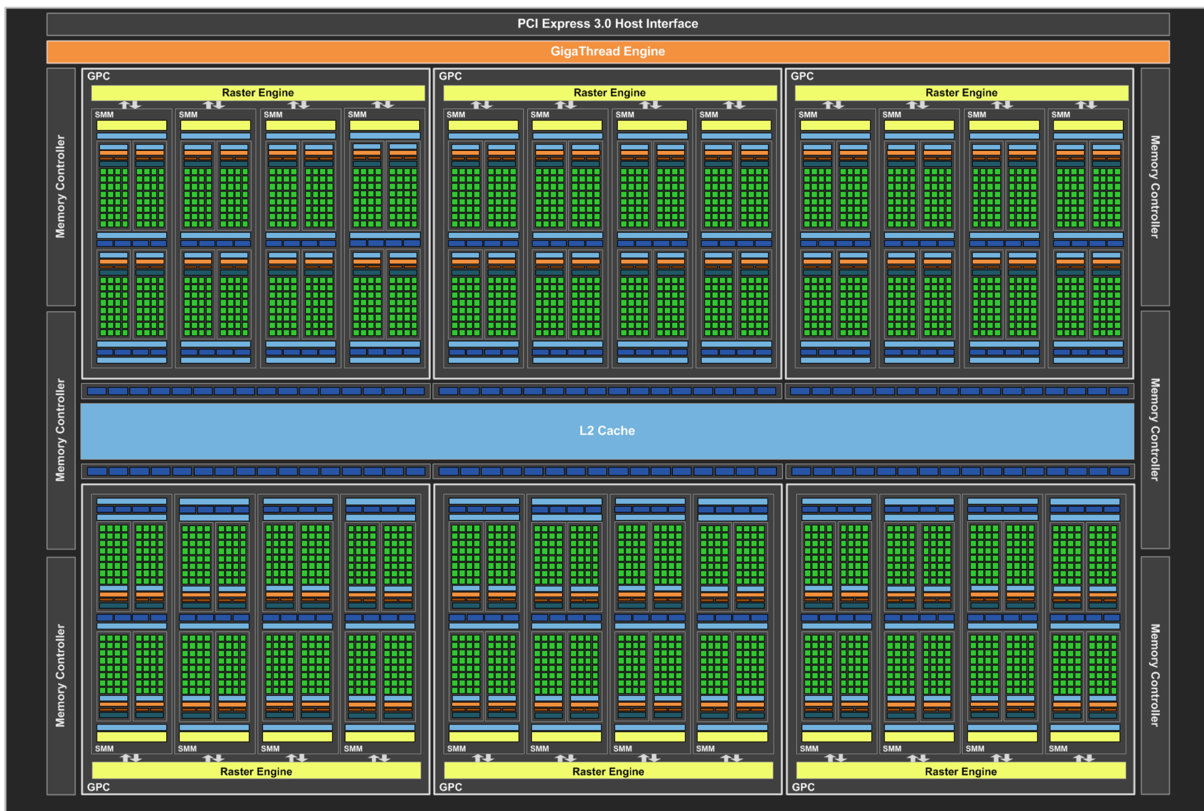



Figure 1.3: Nvidia Maxwell GM200 Architecture.

12 GB of GDDR5 VRAM clocked at 1753 MHz Quad Data Rate (QDR), i.e. with 4 data transfers per clock cycle, which leads to the equivalent of 7 GHz. It can reach a processing power of 6156 Gflops in single precision and of 206 Gflops in double precision.

## 1.4 Manycore Architecture

Recently, Intel has proposed with its Many Integrated Core (MIC) architecture a trade-off between GPUs and classical CPUs. The Intel Xeon Phi is the latest commercial release of the MIC architecture. It is a x86 compatible coprocessor connected to the PCI-Express interface and composed of several Atom like processors with extended vectorial operations. This x86 compatibility is aimed at simplifying code porting but in practice, it requires important optimization effort. Although it can be seen as an accelerator such as GPUs, it can be standalone and an Operating System (OS) can directly be installed on it. It is designed to exploit existing x86 parallel applications originally conceived for standard multicores. However, most applications designed for multicores will not have good performance when running on a Xeon Phi. In the opposite, when an application has been finely tuned to exploit efficiently the Phi, we can expect that this application will have good performances on a multicore architecture.

The MIC architecture inherits many design elements from the Larrabee research project from 2008 [34]. A first prototype, called Knights Ferry (KNF), has been released to developers in 2010. It was composed of 32 cores clocked at 1.2 GHz and allowing to run up to four threads per core. It was built at a 45nm process size and possessed 2 GB of GDDR5 memory.

In 2011, Intel officially released the Knights Corner (KNC) architecture under the Xeon Phi commercial name. It is built at a 22nm process size equivalently to the Intel Ivy Bridge multicore. Different

variants of the KNC architecture with minor distinctions have been released. The memory architecture of the KNCs experimented during our evaluations is illustrated in Figure 1.4. It is composed of 61 cores clocked at around 1 GHz and possesses 8 to 16 GB of GDDR5 memory depending on the version. Since the KNC can run an OS inside, a core is often used to service requests like interrupts and it may end up with 60 cores available for the user application. Each core has its own L1 cache of 32KB and a coherent L2 cache of 512KB connected by a bidirectional ring interconnect. To exploit ILP despite the in-order architecture, the cores support four hyper-threads totalizing 240 threads on a single KNC card. Unlike standard Xeon, the Intel documentation claims that reaching optimal performance on the KNC requires the hyper-threads. KNC proposes also large SIMD units of 512 bits and an adapted x86 Instruction Set Architecture (ISA) called Initial Many Core Instructions (IMCI). Contrary to GPUs, the IMCI ISA allows irregular memory accesses through built-in gather and scatter operations making easier the execution of irregular applications. However, large access ranges involve more cache lines and therefore negatively impact the performance [35]. There is also support for masked instructions, where specific vector lanes can be excluded from an instruction. This KNC architecture has been integrated in many modern supercomputers present in the Top500 [36]. Twice a year, the 500 most powerful supercomputers in the world are listed by the Top500 as detailed in the next section.

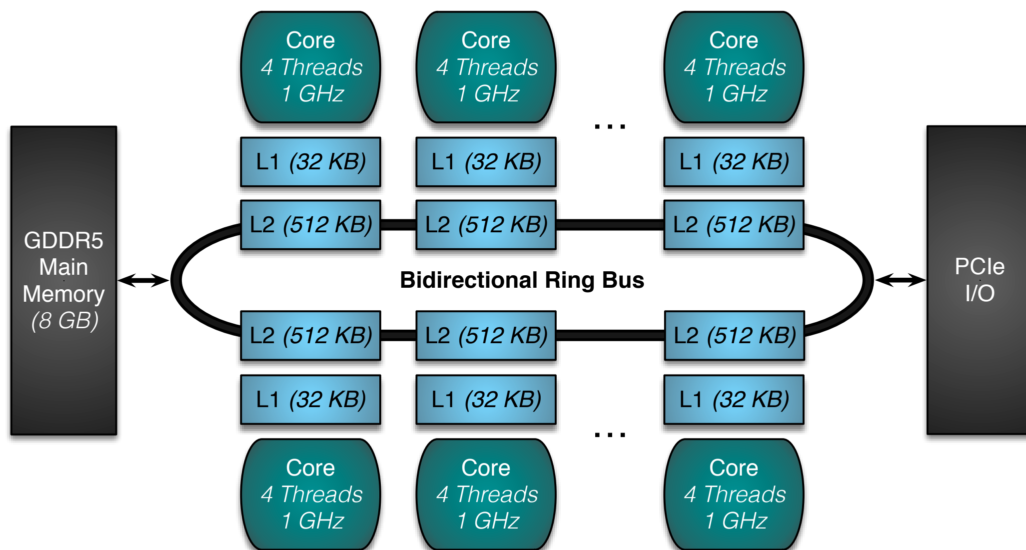



Figure 1.4: Memory architecture of the Intel KNC.

A performance evaluation of the SpMV kernel on the KNC manycore reveals the importance of data locality among cores but also inside cores [37]. According to the authors, the SpMV kernel on this architecture is more memory latency bound than memory bandwidth bound. Well separate the data used among the different cores in order to avoid cache conflicts and resulting data moved by the coherency protocol through the interconnection ring is primordial to obtain good performance. We show in Section 6.3.3 the importance of blocking the data in the L1 caches of the KNC. However, the recent studies in NUCA architectures introduced in Section 1.2 could mitigate this problem in future architectures. Additionally, exploiting the wide 512 bits SIMD instructions enables an acceleration factor up to 16 on single precision values.

A new architecture, named Knights Landing (KNL), is still waiting for commercial release. It will be composed of up to 72 Atom cores built at a 14 nm process size and will have up to 16 GB of 3D memory. It will still use 512 SIMD units but with the more generic AVX-512 ISA.

## 1.5 Supercomputers

Supercomputers are large and powerful computational centers, a.k.a clusters. The first supercomputer was designed by Saymour Cray. At this time it was composed of a small number of processors and of specific vectorial computing units. In the 1990s, supercomputers reached thousands of processors gathered in several distributed computing nodes interconnected by a high performance network such as Myrinet, RapidIO, or Quadrics. Nowadays, Infiniband and Gigabit Ethernet interconnection networks are present in most supercomputers. Each compute node is composed of several processing units following a NUMA topology as explained in Section 1.2. Almost all modern supercomputers are heterogeneous, which means that they are composed of various type of compute units. Most of the compute nodes are composed of one or several multicores, but they can also contain accelerators such as GPGPUs or manycores as the Xeon Phi.

Supercomputers performance are measured in FLoating-point OPerations per Second (FLOPS). As stated in previous section, the 500 most powerful supercomputers around the world are ranked twice a year in lists made by the Top500 organization [36]. They are ranked according to their maximal performance, RMax, achieved using the High Performance LINPACK (HPL) benchmark [38] proposed by Dongarra *et al*. HPL consists in generating a dense linear system of equation and solve it using LU decomposition. However, LINPACK benchmarks in general have been criticized because of their non representativeness of common production codes. The resolution of dense linear system does not stress memory bandwidth and communication networks as sparse irregular applications would do. To address this issue, Dongarra *et al.* propose a new benchmark more representative of real case applications, named High Performance Conjugate Gradient (HPCG) [39]. HPCG consists in solving a preconditioned conjugate gradient parallelized with MPI and OpenMP, both described in the next chapter.

According to the last list of November 2015 summarized in Table 1.4, the actual most powerful supercomputer is for the six time Tianhe-2, meaning Milky Way-2, from China. It is composed of 3,120,000 cores coming from Intel Ivy Bridge Xeon multicores and KNC manycores. It reached a maximal performance of 33.86 PFLOPS while consuming 17.8 MW.

In addition to RMax, the theoretical peak performance, RPeak, is also indicated. The ratio between

Table 1.4: Top 5 supercomputers from Top500 list of November 2015.

| Rank | Country | Name | System | Cores | RMax *(PFLOPS)* | Power *(MW)* |
|---|---|---|---|---|---|---|
| 1 | China | Tianhe-2 | NUDT TH-IVB-FEP<br>*Intel Xeon E5*<br>*& Intel Xeon Phi* | 3,120,000 | 33.86 | 17.8 |
| 2 | USA | Titan | Cray XK7<br>*AMD Opteron*<br>*& Nvidia K20x* | 560,640 | 17.59 | 8.2 |
| 3 | USA | Sequoia | IBM BlueGene Q<br>*Power BQC* | 1,572,864 | 17.17 | 7.9 |
| 4 | Japan | K computer | Fujitsu<br>*SPARC64* | 705,024 | 10.51 | 12.7 |
| 5 | USA | Mira | IBM Blue-Gene Q<br>*Power BQC* | 786,432 | 8.59 | 3.9 |

RMax and RPeak gives an indication of the efficiency of the system. While the peak performance has almost followed the Moore's law, the sustained performance achieved by applications is far behind [40]. Reaching high efficiency with real-world applications on such machines is almost impossible. Achieving 10 to 15% efficiency of the peak performance for a production code at scale is considered as acceptable. Showcasing applications optimized to sustained petaflopic performance, achieve 30 to 50% efficiency. Even the highly parallel HPL benchmark achieved only 62.3% efficiency on the rank 1 Tianhe-2 supercomputer. The second Titan machine based on AMD multicores and Nvidia GPUs and the third ranked Sequoia BlueGene machine provide a slightly better power efficiency. Another important metric is the power efficiency of the machine given by the ratio between performance and power consumption in FLOP/watt. Similarly to the Top500, the most power efficient supercomputers are ranked in the Green500 lists [33].

The first supercomputer to reach the petascale, i.e. $10^{15}$ FLOPS, was the Roadrunner, built by IBM in 2008. The number of cores has then continued to grow until reaching the million with the IBM Sequoia supercomputer and more than three millions with Tianhe-2. Given the current speed of progress, supercomputers are projected to reach the exascale around 2023. However, mainly due to economical reasons, it is unlikely to reach the exascale by just increasing the number of parallel resources exploiting actual technologies. The projection of actual technologies used in Tianhe-2 or Titan into exascale machines leads to energy consumption of around 500 MW which is closed to the electric production of the first nuclear stations. Evolutions driven by energy efficiency will be needed at hardware level to enable a theoretical peak performance of 1 EFLOP. This will probably concern compute units architecture, memory technologies, interconnects, or yet cooling systems. But this will not be enough. Changes at software level will also be required to efficiently exploit such new architectures. More details on the evolutions of runtimes and programming models are given in the next chapter.

## 1.6 Experimental Environment

During this thesis, all the experiments were made with four different supercomputers: Curie, Anselm, Salomon, and MareNostrum, described in more details in the following sections. These supercomputers have similar architectures mainly based on Sandy Bridge multicores, briefly presented in the next section, and of some Intel Xeon Phi manycores (KNC architecture) previously detailed in Section 1.4.

### 1.6.1 Intel Sandy Bridge

The Intel Sandy Bridge is a 32 nm CPU microarchitecture developed by Intel since 2005 and released in 2011. Its core architecture is illustrated in Figure 1.5. This architecture succeeding to the Intel Nehalem has provided several ameliorations.

Among them, the apparition of the Advanced Vector eXtensions (AVX) with a 256 bits instruction set and vectorial units replacing the previous Streaming SIMD Extensions (SSE) instructions. Sandy Bridge additionally introduced a new cache able to store around 1500 decoded micro-operations ($\mu$ops). It is aimed to optimize the energy consumption by avoiding frequent decoding of $\mu$ops. The branch predictor has also been improved with a bigger and more efficient historic. Furthermore, the Sandy Bridge architecture embeds a graphic unit aimed to compete with low-end GPUs from Nvidia or AMD.

Concerning cache memories, in addition to the L0 like $\mu$op cache, each core has its own L1 instruction cache and L1 data cache, both of 32 KB, and a L2 data cache of 256 KB. The Last Level Cache (LLC) of 8 MB which may correspond to a L3 cache is shared between the different cores and the graphic unit. The cores, the graphic unit, the LLC and the system agent, a.k.a uncore, are interconnected through ring bus to handle coherency and minimize latency. The system agent is the set of functions, such as the memory

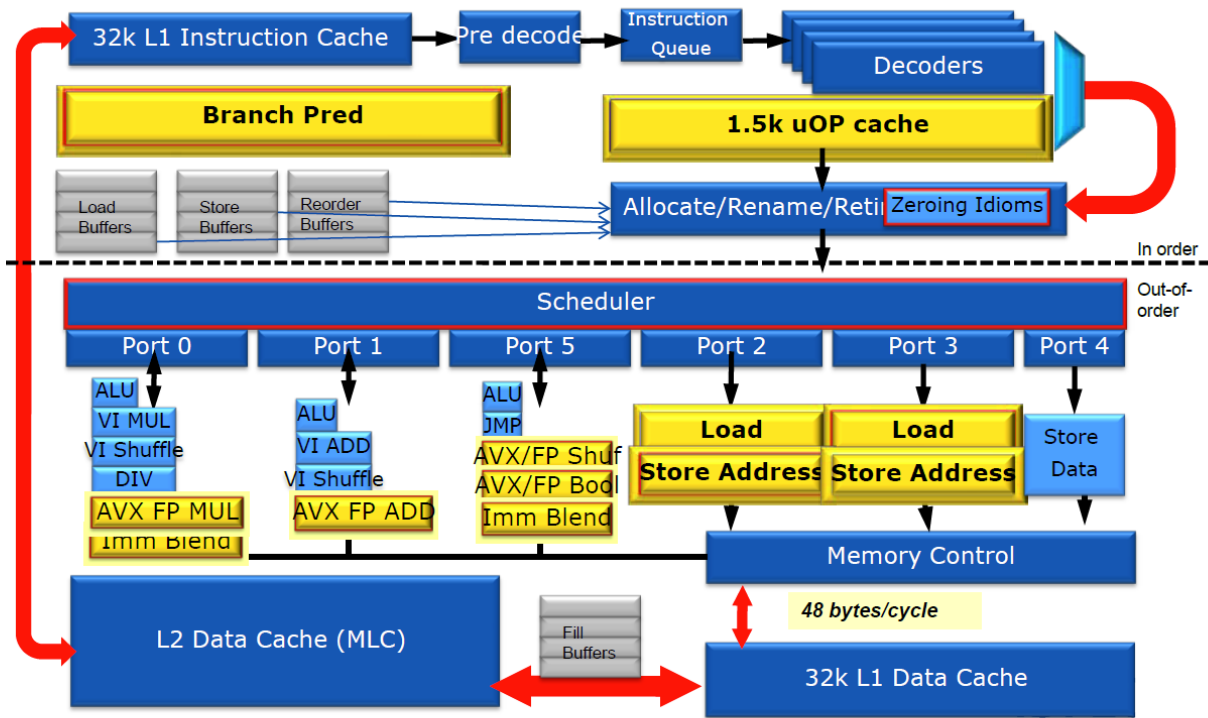



Figure 1.5: Presentation of the Intel Sandy Bridge core architecture from Intel Developer Forum.

controller or the PCI-Express controller, which are not in the core but which must be closely connected to it to achieve high performance. Sandy Bridge also enables the hyper-threading technology allowing to assign two different threads to a single core and the Turbo Boost technology. Turbo Boost consists in increasing the frequency depending on the activity of the CPU and on its Thermal Design Power (TDP), i.e. its maximum amount of heat generated.

Several variants of the Sandy Bridge microarchitecture exist. The Xeon versions present in the supercomputers used during our experiments are octa-cores used without hyper-threading and without Turbo Boost. Indeed, these optimizations are often disabled on supercomputers due to their lack of efficiency and the poor acceleration ratio on applications that are not CPU bound. Moreover, using the Single Program Multiple Data (SPMD) model, two threads having similar workloads will require the same resources. If applied to a same core, there will be nothing to gain and these threads may conflict with each other.

### 1.6.2 Curie

Curie is a French supercomputer designed by Bull and owned by the Grand Equipement National de Calcul Intensif (GENCI). It is located at Bruyères-le-Châtel into the Très Grand Centre de Calcul (TGCC). In June 2012, it was the ninth most powerful computer in the world with a 1.67 PFLOPS peak performance and a consumption of 2.25 MW.

As illustrated in Figure 1.6, Curie is composed of different kinds of compute nodes. There are 360 fat nodes composed of four sockets of Nehalem-EX X7560 octa-cores totalizing 11,520 cores. Curie also contains 5045 two-sockets compute nodes using 10,080 Sandy Bridge E5-2680 octa-cores, totalizing 80,640 cores. And lastly, it contains 144 BullX racks composed of 288 Westmere multicores and 288 Nvidia M2090 T20A GPUs. All these nodes are interconnected through InfiniBand Quad Data Rate (QDR) network with a fat tree topology. We also made some additional experiments on extra compute

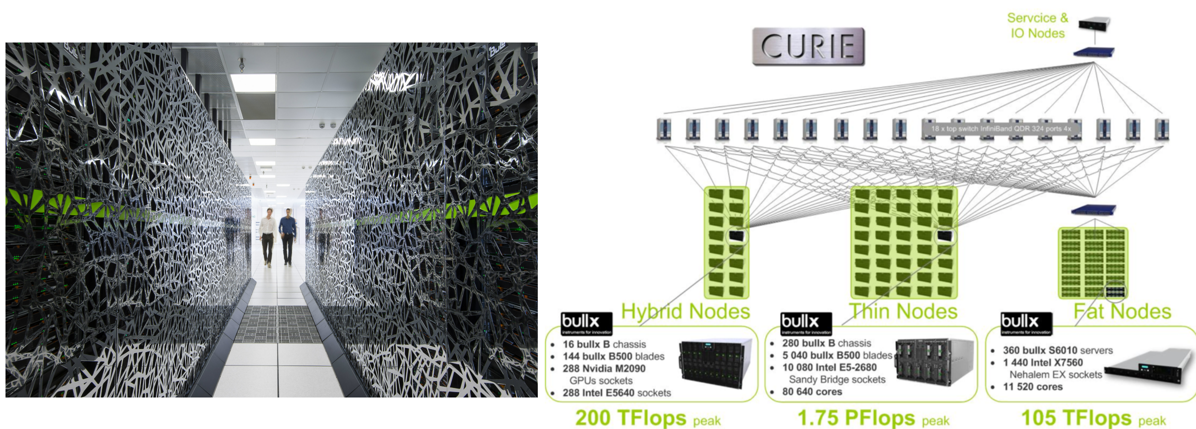



Figure 1.6: Presentation of the Curie supercomputer taken from CEA website [41].

nodes from the Cirrus cluster equipped with KNC manycores.

### 1.6.3 Anselm and Salomon

Anselm and Salomon are two distinct supercomputers from the IT4I computing center at Technical University of Ostrava in Czech Republic. The Salomon cluster, illustrated in Figure 1.7, consists of 1,008 compute nodes, totaling 24,192 cores with 129 TB RAM and giving over 2 PFLOPS theoretical peak performance. All nodes share a 0.5 PB NFS disk storage and a DDN Lustre shared storage of 1.69 PB. There are 576 regular nodes composed of two twelve-core Intel Xeon E5-2680v3 processors running at 2.5 GHz and 128 GB RAM and 432 nodes equipped with two additional Intel Xeon Phi 7120P. Nodes are interconnected by 7D enhanced hypercube topology with Infiniband FDR56 and Ethernet network.

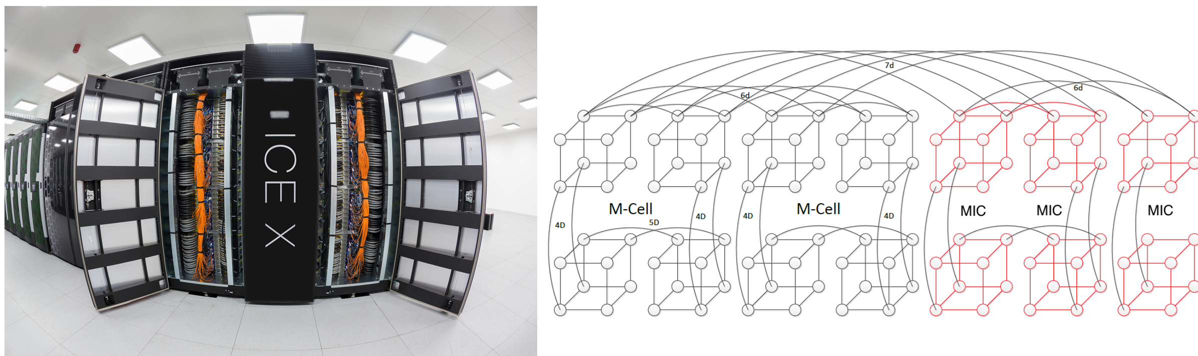



Figure 1.7: Presentation of the Salomon supercomputer and of its 7D enhanced hypercube topology taken from IT4I website [42].

Anselm is smaller than Salomon with only 209 compute nodes, totalizing 3,344 cores, 15 TB RAM, and 506 TB of disk storage separated in two shared file systems. It has a theoretical peak performance of 94 TFLOPS . There are 180 regular nodes composed of two octa-cores Intel Sandy Bridge E5-2665 running at 2.4 GHz and 64 GB of physical memory. There are also 23 additional nodes equipped with a GPU Kepler K20 accelerators, 4 nodes with an Intel Xeon Phi P5110, and lastly 2 fat nodes equipped with 512 GB RAM and two 100 GB SSD drives. Nodes are interconnected by a fat tree topology using Infiniband and Ethernet network.

### 1.6.4 MareNostrum

The last supercomputer used during this thesis is MareNostrum, which is the Roman name for the Mediterranean Sea meaning "our sea". It is part of the Barcelona Supercomputing Center and of the PRACE Research Infrastructure and results from a partnership between IBM and the Spanish government. It is the most powerful supercomputer in Spain. As illustrated in Figure 1.8, the supercomputer is housed in the deconsecrated Chapel Torre Girona at the Polytechnic University of Catalonia at Barcelona.

Figure 1.8: Presentation of the MareNostrum supercomputer.

It has a peak performance of 1.1 PFLOPS and was ranked 29 in the Top500 list of June 2013 [36]. It is composed of 6112 Sandy Bridge E5-2670 octa-cores clocked at 2.6 GHz which are divided into 3,056 two-sockets compute nodes and it totalizes 48,896 cores. It also includes 42 compute nodes with two additional Xeon Phi 5110P. MareNostrum has 115.5 TB of DDR3 memory and 2 PB of GPFS disk storage. It consumes 1015.60 kW. The nodes are interconnected through Infiniband FDR-10 and Ethernet network.

CHAPTER 2

# HPC PARALLEL PROGRAMMING MODELS

**Contents**


**2.1 Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **23**
**2.2 MPI Model - Message Passing Interface** . . . . . . . . . . . . . . . . . . . . . **25**
2.2.1 Two-sided Communications . . . . . . . . . . . . . . . . . . . . . . . . . 26
2.2.2 Collective Communications . . . . . . . . . . . . . . . . . . . . . . . . . 27
2.2.3 One-sided Communications . . . . . . . . . . . . . . . . . . . . . . . . . 28
**2.3 PGAS Model - Partitioned Global Address Space** . . . . . . . . . . . . . . . . **31**
2.3.1 Pros and Cons of One-sided Communications . . . . . . . . . . . . . . . 32
2.3.2 GASPI and its GPI-2 Implementation . . . . . . . . . . . . . . . . . . . 33
**2.4 OpenMP Worksharing Model** . . . . . . . . . . . . . . . . . . . . . . . . . . . . **36**
**2.5 Task Model** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **37**
2.5.1 OpenMP Tasks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
2.5.2 Cilk Plus . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
2.5.3 An Example of CilkView Usage . . . . . . . . . . . . . . . . . . . . . . . 42


## 2.1 Introduction

To make efficient use of the recent heterogeneous architectures presented in the previous chapter, several parallelization strategies have to be combined together to exploit the multiple levels of parallelism. Usually at top level, there is the distributed memory parallelization model which consists in distributing the problem across a number of compute nodes with explicit message passing between them for synchronizing the application or exchanging data between the remote processes. The Message Passing Interface (MPI) presented in Section 2.2 has become the default implementation of this model. A recent alternative to message passing is the Partitioned Global Address Space (PGAS) model presented in Section 2.3. It consists in splitting a memory region over distributed processes and providing them a direct remote access.

Until recently, many applications only use distributed memory parallelization model by assigning one process to each CPU core. However, on modern multicore CPUs, it becomes difficult to maintain such a coarse level of parallelism due to the increasing number of cores on a single compute node. By relying solely on a distributed memory model, the contention for these shared resources results in

increasing communications and memory bottlenecks. Multiplying the number of processes leads to memory overheads caused not only by the large number of processes created and their associated file descriptors but also by the required data and work duplication at program level, as detailed in Section 3.4. The cores of a same compute node share several resources such as memory and caches which should be used to replace message communications and data duplications.

Distributed model has to be combined with shared memory parallelism to make efficient use of the shared resources and to limit the number of process per node. A shared memory parallelization approach allows to reduce the number of communications between cores and to lower the overhead of data duplications. However, it makes the application development more complex. Threads are used as the execution unit inside each node, and the processes are only used between the distributed compute nodes. Multiple threads can share a block of memory while a process maintains its own private memory block. Multithreading allows to independently execute various threads in a single program but with their own execution path and their own data locations. The most popular threading interface is the POSIX Thread (PThread) programming interface used by several higher level programming models. The pragma-based OpenMP runtime detailed in Section 2.4 has became prominent thanks to its simplified way to parallelize applications. However, task-based parallelism is getting popular. The Cilk Plus runtime [11, 12] proposed by Intel and described in Section 2.5.2 and the Intel Threading Building Blocks (TBB) [14] implement this model. OpenMP has also evolved to handle task parallelism as detailed in Section 2.5.1.

Modern architectures have also brought the Single Instruction Multiple Data (SIMD) model up to date. Past vector machines with large data vectors have been put aside. Vector units have been reduced to small size vectors at core level using limited set of vector operations. But recently, we have seen the emergence of manycore processors such as the Intel Xeon Phi using large 512 bits vector instructions. More generally, as illustrated in the Figure 1.3 of previous chapter, the vector units embedded in modern microarchitectures are getting larger. SIMD vectorization consists in packing blocks of data into vector registers and applying a same operation to an entire data vector. To maintain correctness of the code and performance gain when dealing with SIMD parallelism, data packed into vectors must have distinct memory locations to avoid data corruption and have to be contiguously stored to efficiently use the cache memory.

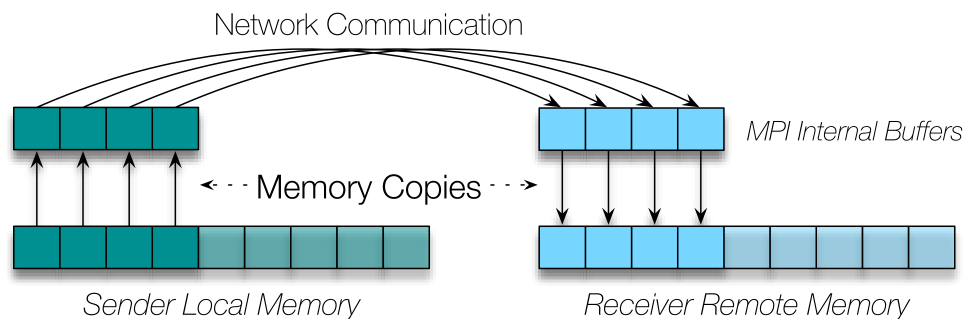
Additionally, many supercomputers make use of General Purpose GPUs (GPGPUs) mostly from Nvidia. These GPGPUs consist of a large number of low frequency cores with small caches which allow to execute a large number of threads in parallel. The standard programming model for GPGPUs is the Single Instruction Multiple Thread (SIMT) model implemented by Nvidia. SIMT implies a large number of threads which execute the same instructions at the same time. But when different threads take divergent flow paths, their execution is concatenated. All the threads execute the flow path corresponding of the first branch condition, then the second one, and so on. The threads not supposed to be active during the currently executed branch are ignored by the hardware using masks. In a same way, threads have to access coalesced memory addresses with a single memory transaction to avoid important impacts on performance. This GPGPUs are addressed using specialized programming languages such as CUDA [43], which only handle the Nvidia's GPUs, or the OpenCL [44] open-source language aimed to support a large variety of architectures.

When developing large scale applications, several of these programming models have to be combined to achieve high performance. However, managing efficiently the parallelization at each level and the interactions across the different levels is challenging.

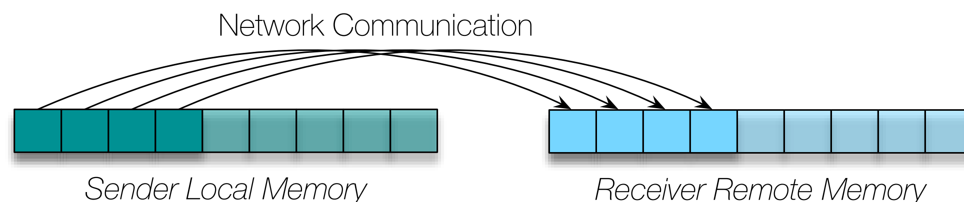
## 2.2 MPI Model - Message Passing Interface

To exploit the ensuing parallelism between the previously created subdomains, the MPI model has become a standard. Several MPI implementations exist such as MPICH [1], OpenMPI [45], or Intel MPI [46]. The idea of MPI is to provide an interface to parallelize an application by creating several processes, called *MPI ranks* or *tasks*, and to exchange data between them using messages. Each MPI rank is mapped to a distinct compute core. A subdomain part of the problem is assigned to each MPI rank. Historically, the first version of MPI has been designed to exploit distributed computing units linked together by network, e.g. InfiniBand or Gigabit Ethernet, using two-sided communications. Two-sided communication means that both sender and receiver processes have to participate to the communication. There are different kinds of point-to-point and collective two-sided communication functions respectively detailed in Sections 2.2.1 and 2.2.2.

However, MPI also handles shared memory resources to communicate between ranks on a same NUMA node. Furthermore, thread-based version of MPI such as TMPI [47], Adaptive MPI (AMPI) [48], or MPC-MPI [49] have appeared. They map MPI ranks to threads instead of processes inside each NUMA node using process virtualization. Nevertheless, this process virtualization method requires to restrict the use of global variables since all the MPI ranks associated to threads share a unique address space. The goal is to reduce the memory overheads induced by a large number of processes by light-weight threads, but also to reduce the context switch and synchronization costs. The communications are optimized to take benefit from the shared resources, but the MPI programming model core concept remains unchanged. As shown in Figure 2.1a, MPI enforces data duplications and intermediate message buffers which lead to




(a) MPI-1 copy-based mechanism




(b) MPI-2 RMA mechanism

Figure 2.1: Comparison between MPI standard copy-based mechanism and zero-copy mechanism using remote memory accesses.

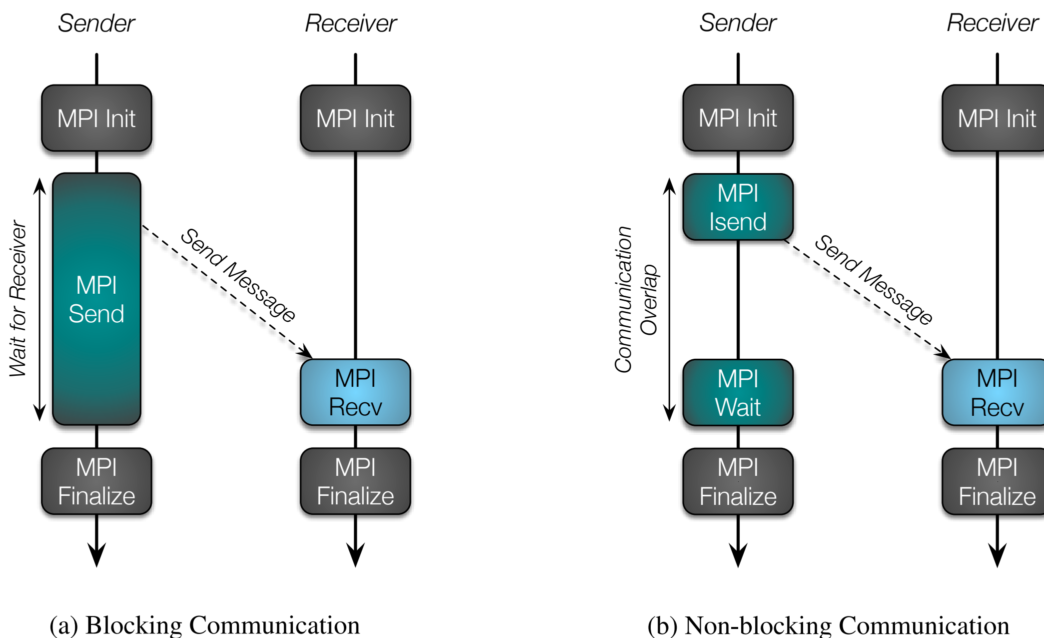
increased memory usage. Moreover, the MPI send / receive model induces synchronization overheads between the two communicating processes. This becomes a bottleneck on large clusters especially when transferring big messages.

To tackle this problem, a Remote Memory Access (RMA) mechanism, a.k.a zero copy message transfer, has been added to MPI [50, 51] in its second version. RMA relies on lower communication layers involving pinned physical memory regions which can be shared with other remote processes. RMA mechanism enables one-sided communications requiring only one process to transfer data. Sender local memory is directly transferred to the remote pinned memory regions of the receiver without any memory duplication. This also reduces the synchronization needs between sender and receiver processes. This is illustrated in Figure 2.1b. Moreover, recent network hardwares support Remote Direct Memory Accesses (RDMA). Using RDMA, the data to communicate can be directly handled by the network and neither the sender nor the receiver CPUs or Operating System (OS) are involved in the communications. One-sided communications are presented in further details in Section 2.2.3.

### 2.2.1 Two-sided Communications

The standard MPI communication model is based on two-sided point-to-point communications. In this model, data are exchanged between two distinct processes with their private memory region. Both sender and receiver have to participate to the communications, which requires synchronization between the processes. The sender process has to explicitly send to its receiver a given memory location and the message size. This memory location is copied to an internal communication buffer, sent to the receiver, and is copied again to its memory. For each message sent, there must be a corresponding reception call from the remote process. The sender is therefore forced to wait for the receiver to be ready before sending the data, as seen in Figure 2.2a.

To mitigate this problem, MPI proposes two different synchronization modes, a blocking one, and a non blocking one. Blocking communications, illustrated in Figure 2.2a, are done using the *MPI_Send* and

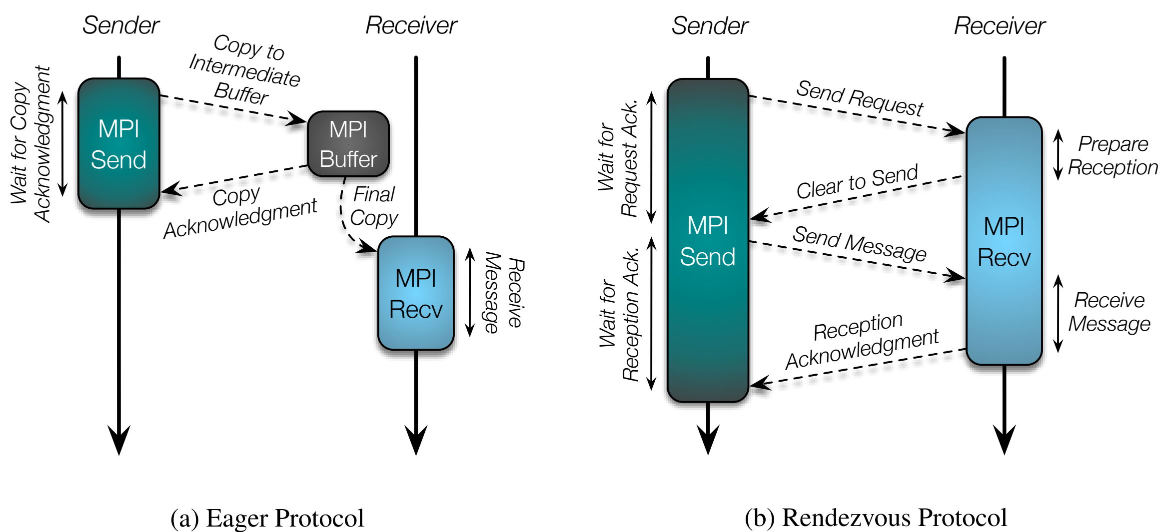



(a) Blocking Communication (b) Non-blocking Communication

Figure 2.2: MPI two-sided synchronization modes.

*MPI_Recv* functions. The call to *MPI_Send* does not return until MPI has duplicated the transferred data in a communication buffer, or until data have been received by the remote process. Similarly, *MPI_Recv* only returns when the communication is complete. Non-blocking communications, illustrated in Figure 2.2b, are done through the *MPI_Isend* and *MPI_Irecv*. These functions return immediately. This way, the sender is free to continue its computation while the receiver process gets ready. The completion of the communication is ensured by the *MPI_Wait* function which may be called further. The *MPI_Probe* function can also be used to see whether the communication has finished and go back to computation if it is not the case. This allows computation and communication to overlap and therefore, hide at least partially, the communication latencies.

In both blocking and non blocking modes, MPI can use two different communication protocols depending on the size of the message. For smaller messages, MPI generally uses the *Eager* protocol illustrated in Figure 2.3a. With the *Eager* protocol, the data to communicate are copied into an intermediate communication buffer. This reduces the synchronization delays, since the sender can continue its work before the completion of the communication. As soon as it has been acknowledged that the data to communicate have been copied into the intermediate buffer, the sender can reuse this memory region without waiting for the receiver acknowledgment. However this induces memory waste and copy delay for both sender and receiver, and it assumes that receiver has enough place to unpack the data. For bigger messages, MPI uses the *Rendezvous* protocol shown in Figure 2.3b. The sender process requests the receiver for an incoming communication of a given size. Then, the receiver allocates the required memory if necessary and confirms to the sender that it is ready to receive. Lastly, the sender can send the message and wait for reception acknowledgment. This way, the sender is sure that the remote process is able to receive the message. Moreover there is no extra data copy nor memory duplication in an intermediate communication buffer. But this synchronization between sender and receiver is costly since it implies several remote communication latencies.




(a) Eager Protocol

(b) Rendezvous Protocol

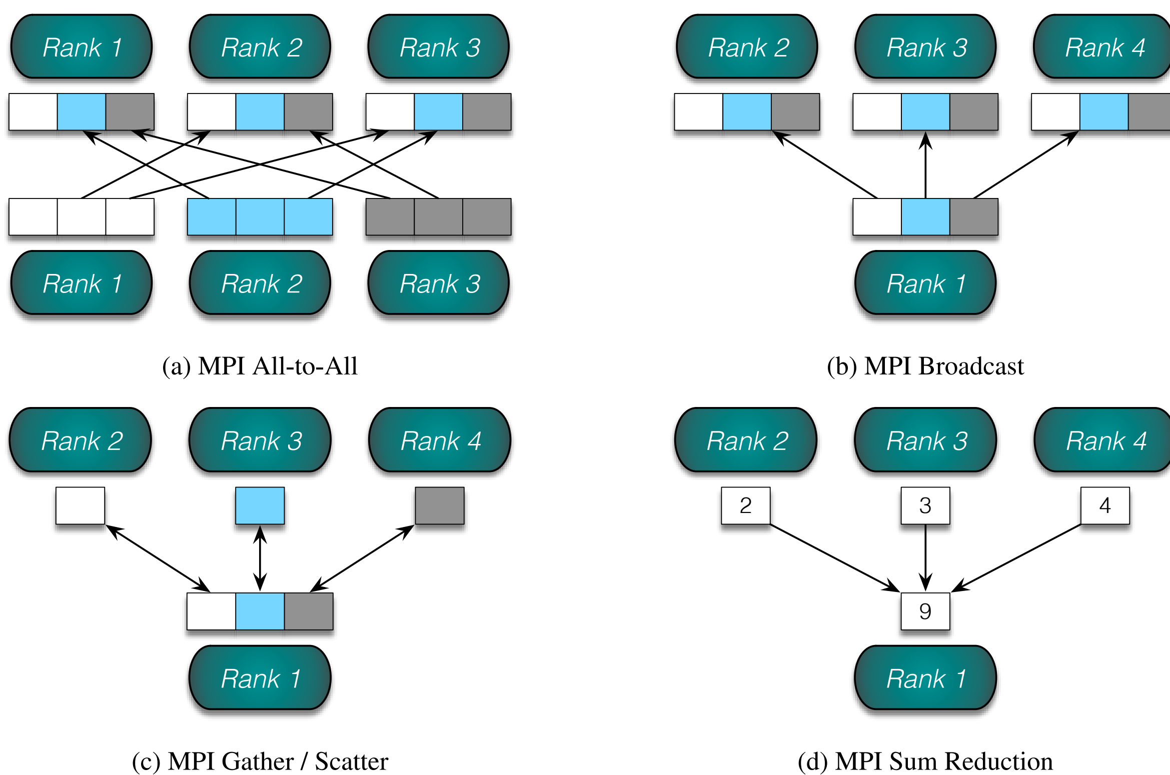
Figure 2.3: MPI two-sided communication protocols.

### 2.2.2 Collective Communications

In addition to point-to-point communications, MPI proposes a set of collective communications involving a group of MPI ranks. The groups, called communicators, contain at least two MPI ranks. Since MPI-3,

collective communications can be either blocking or non-blocking. Several categories of collective communications have been implemented in MPI. Some of them are used to exchange data between processes within an MPI communicator, while others are used to apply operations on distributed values spread among the communicator's processes.

Concerning data movement collectives, there is the *All-to-All* illustrated in Figure 2.4a in which data are switched between all processes. The *i-th* part of the local buffer is sent to the *i-th* process while data coming from the *j-th* process are stored to the *j-th* position of the local buffer. The *Broadcast* represented in Figure 2.4b is used to send a local data to all the other processes within the communicator. There are also the *Gather* and *Scatter* operations which are two opposite functions both illustrated in Figure 2.4c. The *Gather* consists in aggregating data coming from the other processes into their corresponding rank index. In the opposite, the *Scatter* is used to split local data to the other remote processes. A piece of data located at the *i-th* position is sent to *i-th* rank.




(a) MPI All-to-All

(b) MPI Broadcast

(c) MPI Gather / Scatter

(d) MPI Sum Reduction

Figure 2.4: Set of MPI global communications.

Concerning the reduction collectives, MPI implements several operations aiming at computing the minimum, maximum, sum, or even logical operations between the data of an MPI communicator. An example of the *Sum Reduction* is given in Figure 2.4d. Lastly, MPI provides a collective synchronization used to block all MPI ranks of the communicator until they are all at the same point. All these collective communications are very sensitive to load balancing issues since the processes involved in the communications have to wait for the slower one.

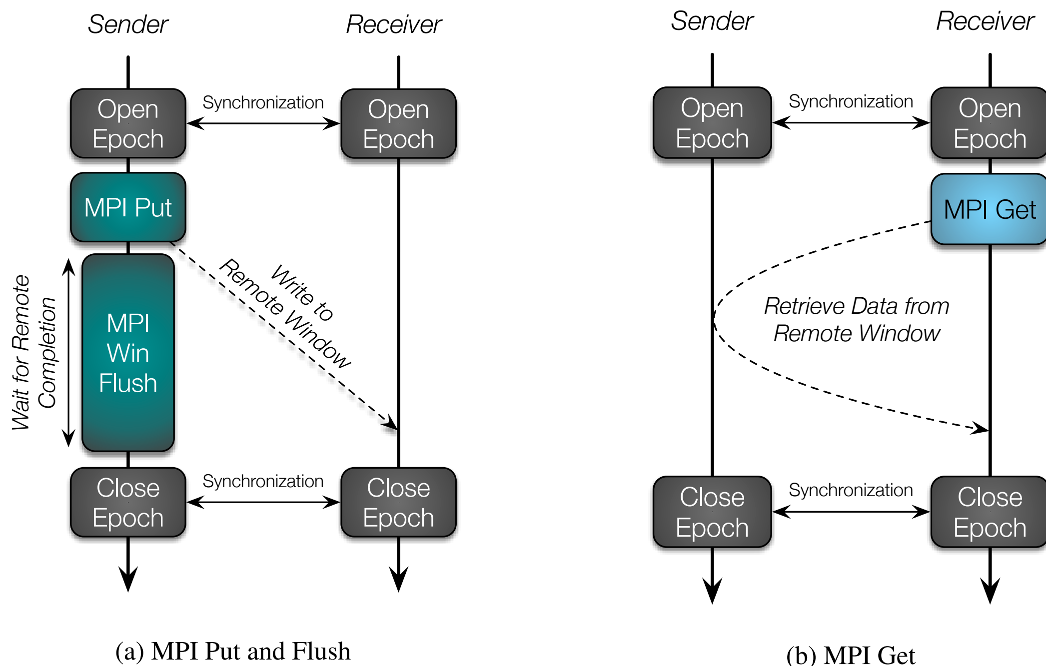
### 2.2.3 One-sided Communications

As stated in previous sections, two-sided communications induce memory copies and costly synchronizations which negatively impact the overall performance. To address this issue and take advantage

of RMA capabilities, MPI has extended its traditional two-sided communication model with one-sided communications. This version 2 of MPI is described in more details in [2, 52]. MPI-2 uses abstract objects called *Window* to specify memory regions of processes made available for remote operations by other processes. Creation and deletion of windows are done through collective operations, *MPI_Win_create* and *MPI_Win_free*, implying all the processes intending to use this window. One-sided RMA operations are applied to a window, an offset in the window, and a target rank. But the creation of the window does not make the corresponding memory regions available to the other processes. An RMA epoch must be opened through a synchronization call to make the window accessible. All RMA operations of a given window take place in a single epoch delimited by a start and an end calls at a given point of the code. There are three different one-sided operations:

- The remote write operation *MPI_Put* illustrated in Figure 2.5a, which transfers data from the sender local window to the receiver remote window.
- The remote read operation *MPI_Get* seen in Figure 2.5b and used to retrieve data from a remote window and to copy it to the local window.
- And a remote update operation called *MPI_Accumulate* which allows to accumulate a local value to an other remote value. Making a call to *MPI_Accumulate* is equivalent to retrieve a remote data using *MPI_Get*, update it locally, and send it back to the remote process using *MPI_Put*. Several accumulation operations are available such as the sum reduction.

These operations are non-blocking, however, a synchronization call is required to ensure that the communication is achieved. Indeed, the read or written memory region should not be updated before the end of the data transfer. The *MPI_Win_flush* function can be used to ensure the completion of all one-sided operations initiated by the caller process to a remote receiver process. In any case as shown in Figure 2.5, the end of the epoch guarantees the completion of all occurring one-sided operations.



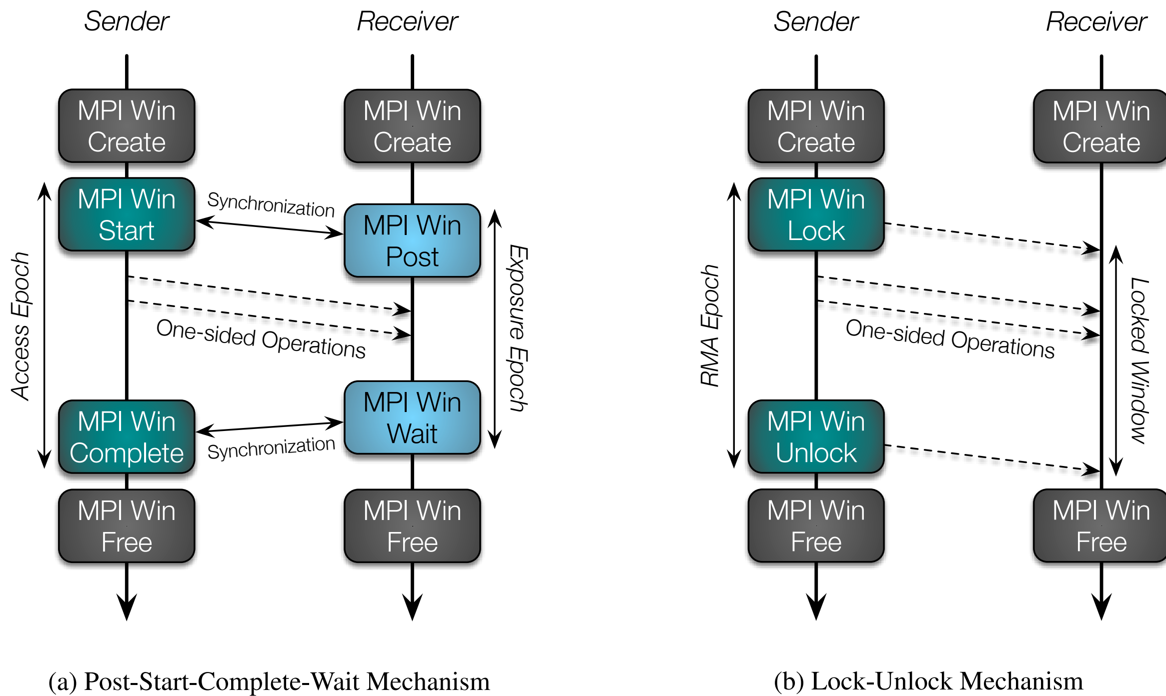

(a) MPI Put and Flush

(b) MPI Get

Figure 2.5: MPI one-sided operations.

The synchronization between emitter and receiver ranks depends on the chosen mode between active or passive target. Active target implies that the emitter and the receiver processes call the synchronization epoch functions. Two different synchronization calls are available in active target mode, the *Fence Synchronization* and the *Post-Start-Complete-Wait (PSCW)*. *Fence Synchronization* is very similar to a standard MPI barrier. All the processes involved in an epoch have to call the collective *MPI_Win_fence* at the beginning and at the end of that epoch. This model is well adapted to bulk-synchronous paradigm where computation and communication are separated but is incompatible with asynchronous programming.

To avoid global synchronizations for each epoch, MPI provides a *Post-Start-Complete-Wait* model illustrated in Figure 2.6a. In PSCW, emitters have to call the *MPI_Win_start* and *MPI_Win_complete* functions which are similar to *MPI_Win_fence* excepted that they only involve a subgroup of processes. Receivers must call the corresponding *MPI_Win_post* and *MPI_Win_wait* functions to acknowledge the start and the end of an epoch. The period between the start and complete calls defines an *Access Epoch* during which RMA communications are possible. The period between the post and wait calls defines an *Exposure Epoch* corresponding to the time in which the target window is exposed or accessible to RMA calls. This model is similar to the non-blocking mode used in two-sided communications and reduces the synchronization needs between the processes participating to a same epoch. But the cooperation between emitter and receiver processes destroys the possibilities of true one-sided asynchronous communications.




(a) Post-Start-Complete-Wait Mechanism

(b) Lock-Unlock Mechanism

Figure 2.6: MPI one-sided synchronization modes.

The last type of synchronization is the *Lock-Unlock* passive target mode shown in Figure 2.6b. Here, only the emitter process calls the synchronization functions to delimit an epoch. In this case the synchronization corresponds to a lock on the window using the *MPI_Win_lock* and *MPI_Win_unlock* functions. This lock restricts the access to a target window to only one process at a time preventing from concurrent interference during the communication. Locks can be either shared or exclusive. Exclusive locks enforce mutual exclusion on the window and the associated RMA operations within the epoch. No other process can access that epoch or create another epoch until the lock is taken. If the lock is shared,

other processes can open other concurrent epochs. However users have to ensure that there is no different RMA operations updating data on the same window memory location at the same time to avoid errors.

To improve the performance of MPI-2, several optimizations have been proposed [53]. Later, a third version of MPI has been released to revise the existing one-sided communication model in order to better exploit the recent RDMA network possibilities [54, 55]. MPI-3 has also added some extensions to this model. Among them, MPI-3 has introduced a request-based mechanism to the one-sided operations. This mechanism enables the use of different *test* and *wait* functions, as those used in traditional two-sided point-to-point communications. MPI-3 has also added a global lock-unlock mechanism which allows to simultaneously control access to all processes sharing a window. These new features help programmers to overlap parts of the communications with computation. Another novelty of MPI-3 are the shared memory windows which take benefit from the shared resources to handle communications and synchronizations. Programmers have to ensure that all the processes intending to use a same shared memory window are on a same NUMA node. This is *de facto* reducing the interest of thread-based MPI approaches such as TMPI [47], AMPI [48], or MPC-MPI [49].

All-in-all, MPI one-sided communication model is complex to use. Memory emplacements specified by windows cannot be updated by a remote RMA operation and locally stored during the same epoch, even on non-overlapping locations. Moreover, RMA operations within an epoch are not allowed to access the memory of different processes. This would require to create a distinct window for each target process and to serialize the operations. Furthermore, we observe that while the documentation indicates that all the RMA operations are ensured to be completed at the end of an epoch, in practice, all the RMA operations posted during an epoch are only handled when calling the end of this epoch. With the current implementation, even if an RMA operation is ready to be sent at the beginning of an epoch, it will be delayed with all the other RMA operations occurring during that epoch to the end call.

To overcome these drawbacks, a new version of MPI is being created. MPI-4, which is proposed by Hoefler *et al.* [56], will reuse a notification driven model enabling remote completion already demonstrated in GASPI [4, 5]. In this model, once a sender has complete its one-sided writes, it can send a notification identified by a unique ID. The receiver can then wait for incoming notifications and be ensured of the incoming communication completion. This model in use in the GASPI library will be presented in more details in Section 2.3.2.

## 2.3 PGAS Model - Partitioned Global Address Space

A more recent approach used to exploit distributed memory parallelism is the *Partitioned Global Address Space (PGAS)* model. PGAS model consists in allocating a global memory space which is partitioned among the distributed processes resources. Each process has a local part of the segment and a direct access to the other remote parts of the segment. It is therefore possible to access, both in read and write, the memory of remote processes without their active involvement. These processes can be assigned to heterogenous architectures.

In our case, we use the GASPI API [4, 5] implemented in the GPI-2 library [57] described in Section 2.3.2. But many other PGAS languages exist such as Unified Parallel C (UPC) [58], Titanium [59], Co-Array Fortran [60], Chapel [61], or yet X10 [62]. MPI-RMA could also have been used for PGAS languages, but the strong restrictions on memory access pattern, the lack of semantic guarantees, and the absence of the remote completion concept restrain it [63].

### 2.3.1 Pros and Cons of One-sided Communications

PGAS languages exploit one-sided communications and are well suited to clusters allowing *Remote Direct Memory Access (RDMA)*. RDMA enables direct memory accesses to all processes sharing data on the partitioned global address space without involving the operating system nor the CPU. Communications are directly done through network interface controllers. This approach brings several advantages and is well suited to unstructured applications with irregular communication patterns [64].

- It allows to reduce the data movement and memory duplication. Using one-sided operations, a sender process specifies the communication parameters both for the local and the remote side and then directly writes data to the receiver's memory. In a two-sided approach, the process sending data needs to pack its local data and copy them to an intermediate communication buffer. Then, the receiver has to wait for the incoming of this buffer and to copy it again to its local memory. This requires twice as much data movement than a process directly writing to its receiver's memory, increasing by the same way the memory bandwidth requirements.

- One-sided communications using RDMA compatible network consume less CPU resources. Neither the sender, nor the receiver CPU is involved in the communication. This way, there is no context switch, no cache pollution, and no CPU cycles wasted. This is directly handled by the network controller.

- One-sided operations also reduce the needs of synchronization. In two-sided approach, for each call to a send, there must be a corresponding receive call. A sender process stays involved in the communication while the remote process has not acknowledged the reception. Moreover, the sender cannot reuse its communication buffer until the receiver has unpacked its data. Using one-sided, as soon as a piece of data is ready to communicate, the sender can directly write it to the receiver's memory and continue its execution even if the latter is not ready to receive. Receiver has just to check when it is ready whether the incoming data are ready or not.

- Lastly, the communications are handled in parallel with the rest of the computations which allows to recover more easily the network delays by computation. Overlapping communication with computation becomes crucial to scale on modern architecture. This problem occurs in various domains. For instance, Ghysels *et al.* have developed a pipelined version of the conjugate gradient, Pipelined CG [65]. It aims at removing the global synchronization step at each iteration and increasing the overlap possibilities, trading numerical accuracy for a better scalability.

However, using one-sided communications also brings new complications. Once again, in two-sided approach, the process sending data only needs to gather its local data and pack them to communication buffers. Concerning the receiver, it receives the buffer and unpacks the data to its local position. In both cases, all the informations required to achieve these tasks are known by the process handling it. When using one-sided communications, the sender has to know its local information to pack its data, but also all its receivers information. If a process sends scatter data to several other processes having different offsets, it needs to know the destination offsets of each of its receivers.

As previously stated, most of the scientific applications have already been designed following a domain decomposition pattern using the MPI library. Although it is possible to mimic the behavior of two-sided bulk-synchronous pattern using one-sided communications, to fully exploit their underlying possibilities of asynchronism and communications recovery, the communication pattern needs to be fully redesigned as well as the data layout. This may imply deep changes in the code which can be blocking on large industrial applications.

For instance, in the DEFMESH application from Dassault Aviation presented in Section 4.2.1, all processes have an interface index giving them the local emplacement of all the elements on each of their interfaces. With the actual MPI two-sided implementation, the receivers loop over the interfaces and wait for incoming data. This way, when a communication is complete, the receiver knows exactly where to store its data. When passing to one-sided communications, each process needs to know the destination emplacement of all its neighbors, i.e. needs to have their interface index, which increases the memory requirements of the application.

### 2.3.2 GASPI and its GPI-2 Implementation

GASPI, for Global Address Space Programming Interface, is an API based on the PGAS model enabling asynchronous and fault tolerant data-flow programming using non-blocking one-sided communications [4, 5]. It has been developed by Fraunhofer ITWM since 2005 and was originally named Fraunhofer Virtual Machine (FVM). In 2009, FVM was renamed into Global Address Programming Interface (GPI) and in 2011, the GASPI standard was created. GPI-2 is the first open-source implementation of this standard [57].

As a PGAS model, each GASPI process has a local and a global memory region which is partitioned among the distributed processes resources. These distributed resources can be either NUMA partitions, HDDs, SSDs, or even GPUs memory and more recently Xeon Phi memory. The local memory of a GASPI process is private while the global memory space, called *segment*, is shared with the other processes and has to be allocated at the beginning of the application. Each GASPI process has read and write accesses to the whole segment and can directly access the memory of remote processes without their active involvement through RDMA interconnect. This is illustrated in Figure 2.7. The GASPI *segment* are comparable to the MPI windows described in Section 2.2.3. However, these segments can be mapped to memory of heterogeneous architectures such as GPGPUs, Xeon Phi, and multicores. The GPI-2 implementation supports both InfiniBand and Ethernet interconnects.

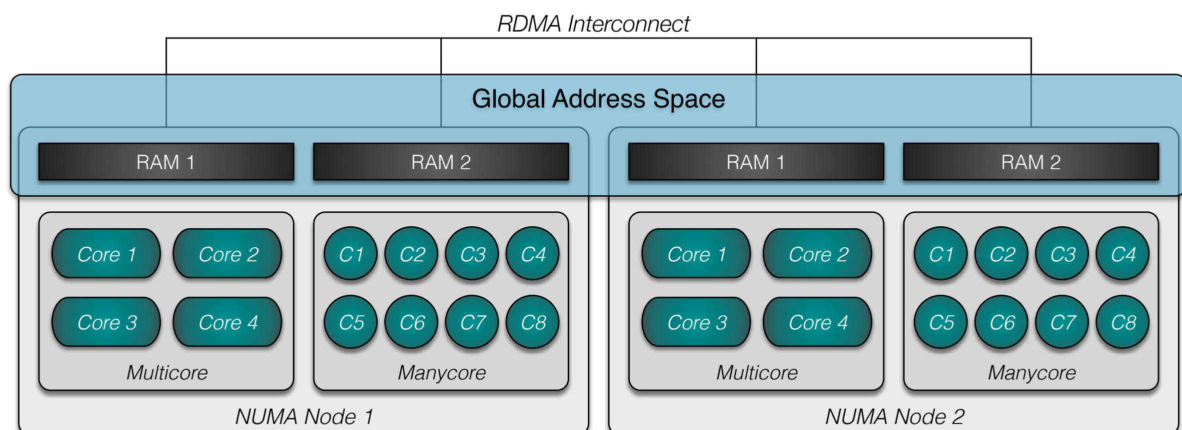


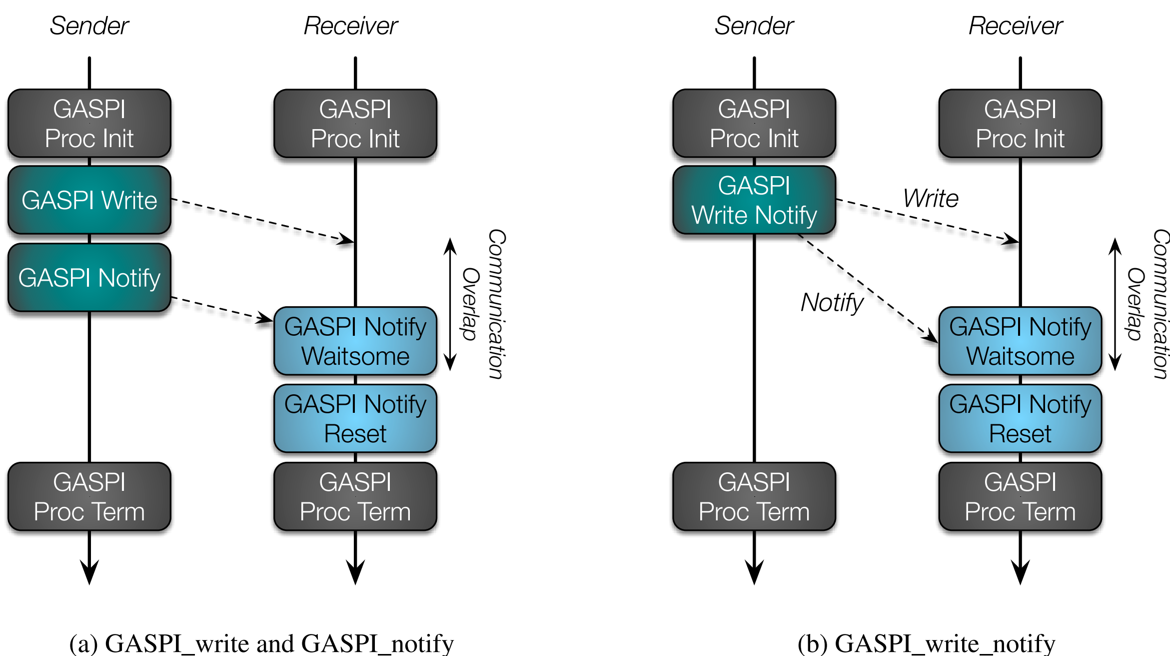

Figure 2.7: *PGAS model* - Each process owns a local part of the segment and has a direct access to the other remote parts of the segment through the RDMA interconnect.

GASPI API provides several ways to exchange data between processes. A process can directly read from, or write to, a remote process memory address located inside a GASPI segment, using the *gaspi_read* or *gaspi_write* functions. It is also possible to communicate data from a list of distinct memory addresses using the *gaspi_read_list* or *gaspi_write_list* functions. Similarly to MPI, GASPI provides both synchronous and asynchronous collective operations involving a group of GPI processes. As for MPI, the default group contains all the ranks. Two collectives are provided in GASPI: the *gaspi_barrier* and the

*gaspi_allreduce*. The *all reduce* function comes with a list of pre-made operations such as the *min, max* or *sum*, but it also offers the possibility to apply a user reduction operation using the *gaspi_allreduce_user* function.

A complete communication involves two calls, the first one initiates the communication and posts a communication request directly to the network. All the communication requests are enqueued by default into a unique queue. However, many different queues can be created simultaneously to separate different types of requests and synchronize them independently. The second call waits for the completion of the communication. Unlike MPI one-sided using specific epochs to communicate, GASPI uses remote completion through notifications. Data may be asynchronously written as soon as it is ready and followed by a corresponding notification identified by a unique ID, aimed at informing the remote process of the completion of all the foregoing communications in a given queue. Once the notification is sent, the sender process is done with the communication. As soon as it is ready, the remote process can wait for a range of notification IDs and can reset them using the *gaspi_notify_waitsome* and *gaspi_notify_reset* functions as viewed in Figure 2.8. It is then ensured that all the corresponding incoming data are locally available. A process can also wait for the completion of all the requests stacked in a given queue using the *gaspi_wait* function. An example of two equivalent communication patterns using *GASPI_write* and *GASPI_notify* functions or *GASPI_write_notify* is given in Figure 2.8. This GASPI mechanism is highly more scalable than the MPI model based on windows and epochs and brings several advantages. Remote completion using notifications enables more true asynchronous programming and allows to better overlap communication with computation. It also avoids the ping-pong between sender and receiver and therefore the number of messages on the network and the resulting latency overhead. Incidentally, this idea has been picked up by Hoefler *et al.* in their MPI 4.0 model [56].

GASPI supports the notion of non coalesced data accesses with the *GASPI_write_list*. *GASPI_write_list* consists in sending to a remote process a list of local offsets, each of them with a specific size. Additionally, GASPI provides atomic operations which can be used to manipulate variables as if they were




(a) GASPI_write and GASPI_notify

(b) GASPI_write_notify

Figure 2.8: Presentation of *GASPI_write*, *GASPI_notify*, and *GASPI_write_notify* one-sided operations.

global shared variables in order to synchronize processes or events. Only one process at a time can access and modify these variables. These atomic operations are the *gaspi_atomic_fetch_add* and the *gaspi_atomic_compare_swap*. The *fetch_and_add* can be used to get the value of a global variable and then to increment it by any value. The *compare_and_swap* is used to compare the value of a global variable to any other value and if equal, apply the given value to the global variable.

Lastly, GASPI is tolerant to hardware failure. Unlike MPI applications, if one or more nodes fail during a GPI run, the remaining processes are not stuck when trying to communicate with a failing node. GASPI offers the possibility to the user to detect that the remote nodes do not respond and adapt his algorithm to handle such issues. Indeed, GASPI communication calls are non-blocking and it provides a timeout mechanism for blocking procedures. There are two predefined timeout values: *GASPI_BLOCK* value blocks the procedure call until completion while *GASPI_TEST* stays in the procedure call for the shortest time possible.

Since GASPI is likely to be used in a hybrid parallelism context in addition to multithreading libraries such as OpenMP or Cilk presented later, it allows multithreaded communications. All the operations presented above can be called in parallel from different threads. In our case, as described in Chapter 7, we make parallel calls to the write and notify functions inside the leaves of our divide and conquer recursive tree. The wait and the reset of the incoming notifications, so as the unpacking of incoming data, are also handled in a parallel loop.

In order to convince developers to use the GASPI library, even if they have already developed MPI applications, GASPI can interoperate with MPI. GASPI sections can be created inside MPI sections. In this case, MPI starts the binary and creates the processes. Then, at the beginning of the GASPI section, the MPI processes are recovered by GASPI, and at its end, the processes are given back to MPI. The only limitation is that MPI communications cannot coexist with GASPI communications at a same time. All MPI operations must be terminated at the beginning of the GASPI sections and vice versa. This way, users can try GASPI on a specific part of the application without having to modify the whole application.

#### GASPI Performance Comparison

GPI-2 has recently been experimented to parallelize a SpMV kernel and a fluid flow solver in [66] and adaptive local search approaches in [67]. They show that the algorithms have to be redesigned to fully exploit GPI. But when properly used, this allows to recover an important part of the communications and gain a significant speedup compared to MPI implementation. Simmendinger *et al.* [68] have compared several halo exchange implementations using MPI and GASPI in strong scaling experiments. They have implemented many different communication patterns and experimented them on standard Xeon Ivy Bridge processors and on Xeon Phi manycores. The Ivy Bridge experiment is illustrated in Figure 2.9. It compares 10 distinct implementations:

- The communication free version which uses no communication at all, only the compute phase. This is the ideal performance to reach by entirely overlapping communication by computation. This version is called *comm_free* in the Figure 2.9.

- The standard MPI bulk-synchronous approach using two-sided communications, in which the communication phase only starts at the end of the computation phase. It is called *mpi_bulk_sync*.

- An early-receive MPI version using two-sided communications where the calls to the non-blocking receive procedures are posted before the beginning of the computation phase. Similarly to the bulk-synchronous version, the sends are only executed once the computation phase is completed. This version is named *mpi_early_recv*.

- A two-sided asynchronous MPI version similar to the previous early-receive version but where the sending operations are processed as soon as the data involved in the corresponding communication are available. This is the *mpi_async* version.

- A bulk-synchronous version and an asynchronous version, both using MPI one-sided and fence synchronization as explained in Section 2.2.3. They are respectively named *mpi_fence_bulk_sync* and *mpi_fence_async* in the Figure 2.9.

- And two other bulk and asynchronous MPI one-sided versions, but using the PSCW synchronization mechanism explained in Section 2.2.3. These versions are respectively called *mpi_pscw_bulk_sync* and *mpi_pscw_async*.

In addition to this communication free version and these 7 MPI versions, they have implemented 2 different GASPI communication patterns:

- A bulk-synchronous version similar to the MPI one, but using GASPI one-sided communications. It is named *gaspi_bulk_sync*.

- And an asynchronous GASPI version, called *gaspi_async*, in which the data involved in the communication are directly written to the appropriate neighbor, as soon as they are available.

The conclusions are similar on both of these architectures. The overlap possibilities of the asynchronous versions increase the gap compared to bulk-synchronous version as the number of cores increases. These experiments also highlight the poor performance of the MPI one-sided communications compared to GASPI ones, or even to standard MPI two-sided.

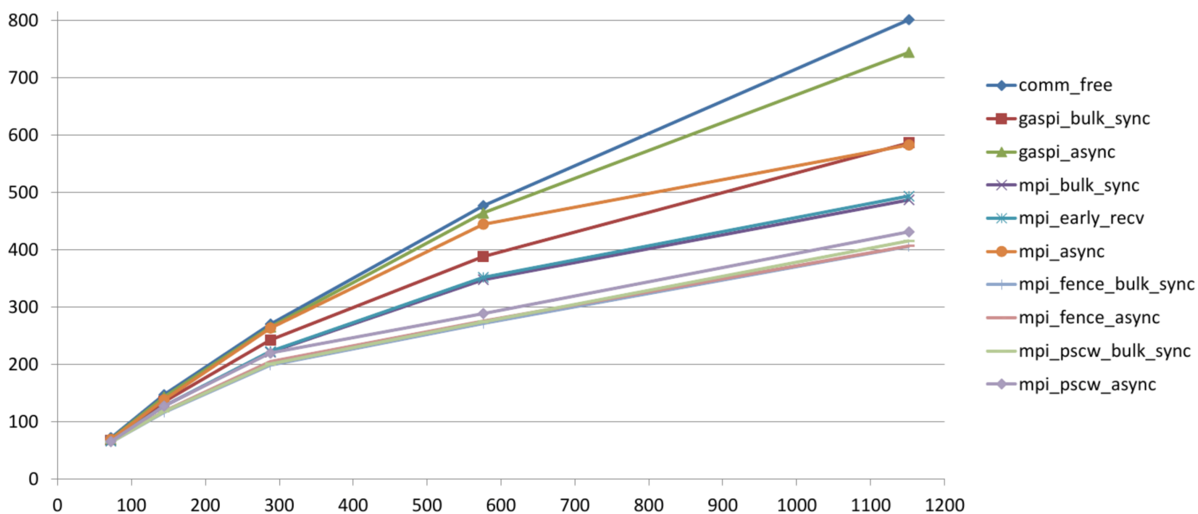



Figure 2.9: Comparison between different communication patterns on 48 two-socket nodes of 12-core Xeon Ivy Bridge EP. With courtesy of Simmendinger *et al.*.

## 2.4 OpenMP Worksharing Model

OpenMP [7] is a shared memory parallelization runtime using pragmas to create and manage threads. It first appeared in late 1990s and was initially published for the Fortran language and quickly after released

for C and C++ languages. It is a bulk-synchronous fork-join model originally based on loop parallelization. As illustrated in Figure 2.10, the master thread creates as many new threads as required by the user at each new parallel region and synchronizes them at their end. This is done by the *omp parallel* pragma. To parallelize loops, the *omp parallel for* pragma splits the loop iterations into independent chunks and schedules then among the different threads. Scheduling can follow different heuristics:

- *Static* - The loop is divided in equal-sized chunks and a tail chunk statically distributed to the threads.
- *Dynamic* - The chunks are dynamically distributed to the threads with a synchronization at each assignment. This allows to dynamically balance the load of irregular loop iterations but it generates an important overhead.
- *Guided* - Similar to *Dynamic* but with decreasing chunk size in order to increase load balancing opportunities.
- *Auto* - The compiler automatically choose between the three previous scheduling strategies.

The pragma approach of OpenMP makes it easy to use and has been widely used during the last decade. However, its loop-based model is in essence limited by the Amdahl law. The sequential remaining parts of the computation will be more and more critical as the parallelism increase. The bigger the compute nodes will be, the less efficient this model will be. Nevertheless, OpenMP is well suited to handle dense regular application composed of hot loops representing an important proportion of the total execution time on a small number of cores. In [69], authors experiment OpenMP loop parallelization in an FE academic application, called FEAP. Synchronizations are handled using the atomic directive. They obtain good speedup and efficiency at 12 cores but they only measure the time spent in the parallelized loop and do not scale beyond 12 cores. Exploring strong scaling experiments would be necessary to have an idea of the potential bottlenecks of their algorithm.

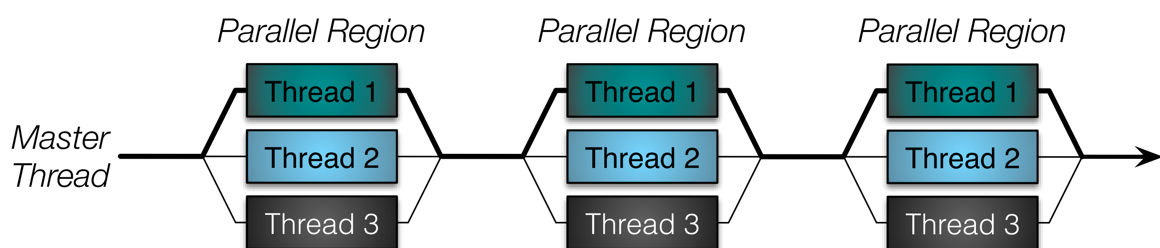



Figure 2.10: Bulk-synchronous fork-join parallelization model using 3 threads per parallel region.

## 2.5 Task Model

Task-based parallelism enables easier expression of unbalanced applications with complicated control structures and dynamic data decompositions. The task model has been explored both at shared and distributed memory levels.

At distributed memory level, several runtimes such as StarPU [70], PaRSEC [71], or OpenCL [44] have appeared to generate parallel tasks able to be executed over various heterogeneous architectures such as multicores, manycores, or GPUs. Using StarPU [70], the programmer has to specify the data accessed by each task and how they are accessed, e.g. read, write, or read an write. Then, the StarPU

runtime automatically generates the dependencies between tasks and schedules them over the different units, depending on the cost of the tasks and the charge of the units, in order to exploit as much as possible the different resources and to schedule their charge. StarPU has also been extended to handle MPI communications between tasks [72]. At higher level, the PaRSEC runtime [71] expresses an algorithm as a dataflow of tasks, i.e. a flow of tasks scheduled according to their data dependencies and represented as a Direct Acyclic Graph (DAG). Then, PaRSEC manages and schedules these tasks over distributed architectures. At the opposite, the low-level OpenCL programming model [44] requires the programmer to manually define parallel tasks, manage the memory allocation, and perform the transfers between hosts and devices. Lastly, the YML framework and its associated YvetteML workflow language [73] provide an interface to develop and execute parallel applications. Similarly to PaRSEC, the YvetteML language permits to describe an algorithm as a dependency graph of tasks. YML does not rely on specific middleware and aims to stay independent of the underlaying environment.

At shared memory level, task-based parallelism allows programmers to easily assign concurrent blocks of code to independent tasks rather than directly creating, joining, and managing threads. The runtime automatically handles the generated tasks and schedules them on the different threads. This way, the number of spawned tasks can be arbitrarily large and is not linked to the number of threads and the targeted hardware architecture. This model is well suited to recursive splitting which results in a large amount of parallel tasks. This fine-grain parallelism enables very efficient load balancing using for instance work-stealing schedulers as those used in the Intel TBB library [14] or the Intel Cilk Plus runtime [11, 12] presented in Section 2.5.2. OpenMP has also evolved to handle task-based parallelism in its third version presented in the next section. Moreover it has been extended by Duran *et al.* with the OmpSs programming model [74]. Contrary to OpenMP, OmpSs generates all the threads at the beginning of the execution. The master thread executes normally while the other worker threads wait for work generated by the master thread. But worker threads can generate nested work. Moreover, OmpSs extends the OpenMP task model with automatic dependencies generation. Similarly to StarPU, the user defines the tasks as input, output, or inout tasks and OmpSs schedules them according to the resulting dependency graph. Recently, a novel approach has been proposed by Gonnet *et al.* with the QuickSched scheduler [75]. It differs from the implicit task dependencies model used in Cilk, TBB, or OpenMP 3.0, which is inherently defined by the spawning and the synchronization of the tasks. Instead, it extends the dependency-only model using automatic generation of task dependencies used in StarPU, PaRSEC, or OmpSs, with task conflicts. The concept of conflicts consists in explicitly defining the shared resources needed by a task to execute. Tasks requiring the same shared resources cannot execute concurrently. However, as soon as there is no explicit dependency, they can execute in any order. Once a task's dependencies have been resolved, it is put in one of the thread queues and is executed when it is not involved in any shared resources conflict. This lets more possibilities to the scheduler removing irrelevant execution order between independent tasks.

### 2.5.1 OpenMP Tasks

Standard OpenMP model has been very successful last decade to exploit regular loop parallelism. In order to address the growing need for irregular parallelism in modern applications, preliminary versions of OpenMP exploiting task parallelism have been explored. In [76] and [77] authors highlight the limitations of OpenMP model to handle nested parallelism. While OpenMP is initially designed to parallelize the outer level of parallelism, they show the importance of taking into account the multiple levels of parallelism available in the algorithm. To tackle this problem, they propose an algorithm aimed at creating tasks and explicitly assigning them to OpenMP threads. The work-queuing model proposed in [77] is representative of how OpenMP 3.0 has evolved later. It has been integrated as an extension into the Intel OpenMP compiler [78]. This model is basically composed of two pragmas. The first one, *taskq*, initializes a single empty queue and starts a block which specifies the environment within which the tasks will be

executed. The second one, *task*, is enclosed to the *taskq* block and specifies a unit of work, i.e. a task, which is enqueued on the previously created queue.

This results in the very similar OpenMP 3.0 tasking model presented in [79]. Firstly, programmers have to initialize a parallel region using the *omp parallel* pragma. Then he needs to restrict its execution to a single thread using the *omp single nowait* pragma. The tasks are generated by this single thread using the *omp task* pragma and enqueued into a workqueue. The runtime automatically schedules the tasks among the different threads as illustrated in Figure 2.11. Lastly, the synchronization of the tasks is handled by the *omp taskwait* pragma. This synchronization waits for the completion of the generated children tasks.

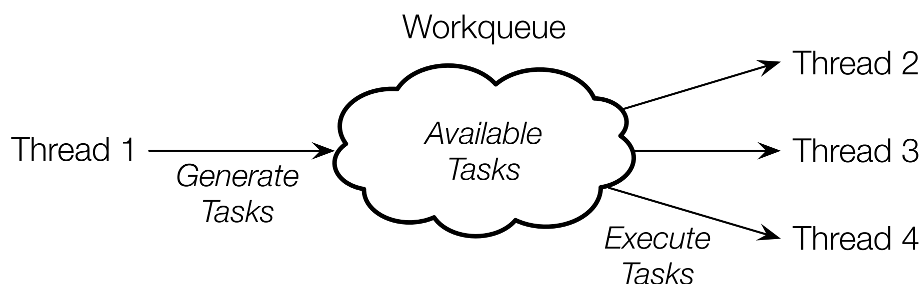



Figure 2.11: OpenMP 3.0 task workqueuing model.

An experimental evaluation of the OpenMP 3.0 tasking model is proposed in [80]. According to the authors, this new OpenMP model is promising for a wide range of irregular application parallelization. Although it is a recent model which still can be improved in programming facilities and performance, it can be easily used and provides good performance scalability. However in our sense, the generation of all the tasks by a single thread and their centralization in a single work-queue is in essence not scalable.

Duran *et al.* have compared different scheduling strategies for OpenMP tasks and concluded that work-first schedules provide the best performance [81]. However due to OpenMP restrictions, they opted for a breadth-first scheduler which gets along better with OpenMP. The breadth-first scheduling consists in placing each new created task into a pool of tasks associated with a team of threads. Only the threads of the team can execute tasks from that pool. The scheduling of these new tasks is delayed after the end of the parent tasks execution. This way, all tasks in the current recursion level are generated before a thread executes tasks from the next level. In OpenMP implementation, there are also tied tasks which are owned by only one thread and which are not shared in the team pool. These private tasks are scheduled in priority to the shared tasks located in the team pool.

As breadth-first scheduling generates an important amount of tasks before executing them, it is important to be able to limit their number. This is called a cutting off strategy. Two different policies are compared in [81]. The first one fixes a maximal number of tasks which can be created simultaneously inside each pool. Once the maximal number of tasks is reached, newly created tasks are directly executed. This maximal number is function of the number of threads. The second cutting off policy limits the maximal number of recursion levels. When a new task is created, if it exceeds the maximal number of ancestors, it is directly executed.

Different implementations of OpenMP 3.0 have been compared to the standard OpenMP implementation without tasks and to the Intel Cilk Plus runtime presented in next section [82]. They evaluated these different versions on the Unbalanced Tree Search (UTS) benchmark [83] aimed at testing the scheduling and load balancing strategies of parallel runtimes. They concluded on the poor behavior of the different OpenMP 3.0 implementations on this benchmark. The Intel implementation behaves significantly better

than the others in terms of load balancing but is still outperformed by the Cilk Plus runtime.

### 2.5.2 Cilk Plus

Cilk is a task-based runtime designed for multithreaded parallel programming and developed as an extension to the C programming languages [11, 12]. The C elision of a Cilk program produces a syntactically and semantically correct C code. Cilk was originally developed at the MIT in 1994 by Leiserson *et al*. The name of Cilk is an allusion to the silk, which can be described as "nice threads", and to the C programming language. In 2006, Leiserson launched Cilk Arts and decided to modernize it into Cilk++ [13]. Cilk++ integrates the support for the C++ language, loop parallelization, and hyperobjects such as reducers or thread local storages named holders [84]. Since 2009, Cilk Arts is part of Intel corporation and has been renamed into Cilk Plus.

Similarly to OpenMP, Cilk uses fork-join parallelism, a.k.a. fully strict parallelism. In this model, a task can create an arbitrary large number of children tasks which can freely execute in parallel. Then, it needs to wait for the completion of its children tasks before ending. But it cannot wait for tasks that are not its children. All the dependencies of a task subtree come from the subtree's root. This allows to simplify and alleviate the runtime. Moreover, Cilk enables nested parallelism. It means that any generated task applied over a set of values can in turn generates other parallel tasks over other sets of values. This model is well adapted to divide and conquer algorithms.

Cilk Plus allows users to generate and synchronize efficiently many parallel tasks using respectively the *spawn* and the *sync* keywords. Newly created tasks can be used to *spawn* again other tasks, forming a very large recursive tree in which the depth corresponds to the critical path and the width corresponds to the available parallelism. The synchronization of the tasks is local to the subtree of a given task. The task making the *sync* call is blocked until the completion of all its descendant tasks. Cilk Plus also provides a loop parallelization using the *cilk_for* keyword which automatically generates a recursive tree assigning the different loop iterations to distinct tasks. All these tasks are organized in queues and executed by threads, named *workers*.

Concerning load balancing, the Cilk Plus runtime uses a work-stealing scheduler. As explained in the previous section, work-first scheduling strategies obtain in general better performances [81]. However, such strategies were not chosen in OpenMP due to internal restrictions. In Cilk, there is no such restriction and a work-first scheduler has been implemented [11, 12]. The idea of the work-first scheduling is to follow the serial execution path to benefit from the data locality of the sequential algorithm. Moreover, it is better to remove overheads from work than from critical path. Indeed, in Cilk scheduler, the parallel running time can be bounded by the following formula [12]:

$$T_P \leq C_1 \frac{T_1}{P} + C_\infty T_\infty \tag{2.1}$$

Where $P$ is the number of processors, $T_1$ the sequential execution time, i.e. the work, $T_p$ the execution time on the $P$ processors, $T_\infty$ the execution time on an infinite number of processors, i.e. the critical path, and lastly $C_1$ and $C_\infty$ are constant due to overhead in the system. Even with a large number of processors, we have $\frac{T_1}{P} \ll T_\infty$. Therefore, in order to optimize performances, it is better to move the overhead from $C_1$ to $C_\infty$.

Contrary to the breadth-first scheduling strategy, in work-first schedulers, when a new task is created, the current thread directly switches from the parent task to the new task. The tasks which have been suspended, either because of the creation of a new task or because of a synchronization, are pushed into a pool local to the current thread. Threads execute in priority their local pool, following a LIFO strategy. When it is empty, they will steal parent tasks in priority and if not possible, they will steal tasks from other thread pools following a FIFO strategy. Work stealing algorithms are presented in more details in [85].

During the compilation, two clones of every Cilk procedure are produced, a *fast clone* and a *slow clone* [12]. The fast clone is almost identical to the C elision of the Cilk procedure and is executed in the common case. The slow clone, which is around 20 to 30% slower than the fast clone, is executed in the infrequent cases where a task is stolen from another worker thread. Since Cilk scheduler steals in priority parent tasks, a fast clone is ensured to be executed by the same worker than its parent task. However, a slow clone needs to load additional variables stored in a shadow stack for parallel bookkeeping. This is illustrated in Figure 2.12.

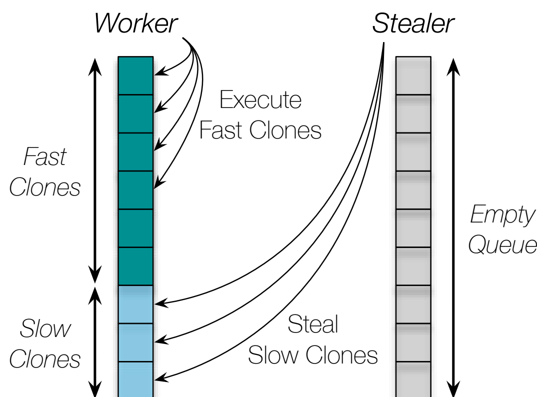



Figure 2.12: Cilk Plus work-stealing scheduler.

Cilk Plus is accompanied with two powerful and helpful tools which are part of the Intel Cilk Plus Software Development Kit. These tools are Cilkscreen [86] and Cilkview [87] which has recently been refactored as Cilkprof [88]. The Cilkview tool, and by extension Cilkprof, are designed to profile and estimate the scalability of Cilk Plus applications. Contrary to Cilkview which only collects data at application level, Cilkprof is able to collect data for each call site. Both of these tools monitor Cilk Plus applications and report parallel statistics and performance prediction on multi-processor systems. They analyze the logical dependencies within the computation and extrapolate speedups based on the algorithmic measures of the *work* and *span* metrics. The *work* is the total time spent to sequentially solve the initial problem. The *span* corresponds to the critical path length which is the execution time on an infinite number of processors. These metrics are used to predict how the application will scale. Cilkview and Cilkprof tools also analyze scheduling overheads which may be due to insufficient grain size of parallel tasks. Experiments made on CilkView during this thesis are presented below.

The Cilkscreen tool is designed to detect the race conditions which may occur between tasks executed simultaneously on different Cilk workers. In practice, it checks that every possible scheduling of the program execution produces the same behavior. It must determine if two tasks which access the same memory location can be scheduled at the same time, or if there is a serial relationship between the tasks. To do so, Cilkscreen dynamically instruments memory accesses of a Cilk binary. If no race is detected, Cilkscreen guarantees that there will not be any race condition, at least on the actual data set, no matter how it is scheduled. If a race exists, Cilkscreen localizes the bug and provides many debugging informations such as variable name, file name, line number, or dynamic context.

Finally, Cilk Plus has provided a useful array notation [89] which have been integrated in the Intel 2011 C++ compiler. It is described in more details in Section 6.2.1. As explained in the previous section, the evaluation made on the UTS benchmark shows the potential of the Cilk Plus runtime [82].

### 2.5.3 An Example of CilkView Usage

CilkView [87] monitors a Cilk Plus binary application and reports parallel statistics and performance prediction on multi-processor systems. During this thesis, we used this tool to analyze the unexplained poor scalability observed at some steps of the integration of the D&C library. To present it, we compare two reports obtained on two different applications, both of them using our D&C library and running a similar use case. Let us call these two generic applications, the *Application 1* and the *Application 2*. The two reports provided by CilkView are presented in Figure 2.13.

The *Work* corresponds to the number of executed instructions. This indicates that the *Application 1* has a much higher computation ratio than the second one. The *Span* and *Burdened span* indicate the number of instructions on the critical path. The second one includes the runtime's overhead for scheduling, stealing and synchronization. Due to the more complex computation, the runtime overhead is more than 25 times smaller in the first application. The number of spawns, synchronizations and strands, i.e. sequences of sequential instructions, are similar. It makes sense since the meshes have similar sizes and the D&C tree topologies are analogous.

```
Work: 72,508,999,092
Span: 273,655,307
Burdened Span: 275,295,307
Parallelism: 264.96
Number of spawns/syncs: 7,901
Average instructions/strand: 3,098,145
Strands along span: 66
Average instructions on span: 4,146,292
Total number of atomic inst.: 1,957,609
Frame count: 30,174,937
```

(a) Application 1

```
Work: 2,886,380,280 instructions
Span: 10,734,061 instructions
Burdened Span: 12,261,716 instructions
Parallelism: 268.90
Number of spawns/syncs: 7,515
Average instructions/strand: 128,021
Strands along span: 61
Average instructions on span: 175,968
Total number of atomic inst.: 7,518
Frame count: 22,659,845
```

(b) Application 2

Figure 2.13: Comparison between CilkView scalability reports.

The larger difference involves the number of atomic operations. The ratio between these atomic operations and the amount of work is 10 times higher for the first application. A rapid experiment putting some local variables on the stack by using local static declarations in conjunction with the *recursive* Fortran keyword, drastically reduced the number of atomic operations. We deduced that the high number of atomic operations was the result of the high number of dynamic allocations involved. Based on this observation, we privatized all the shared variables using the threads' private stack. This is explained in more details in a previous publication [24]. As a result, CilkView report shows a significant decrease of atomic operations, the remaining ones corresponding to the spawn and sync operations of the Cilk Plus runtime. Therefore, CilkView permitted us to detect a scalability bottleneck during the D&C integration and to restore the expected scalability.

CHAPTER 3

# PARALLELIZATION STRATEGIES FOR FINITE ELEMENT METHODS

**Contents**


**3.1 Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **43**
**3.2 FEM Computation over Elements** . . . . . . . . . . . . . . . . . . . . . . . . . . **44**
**3.3 Compressed Storage Formats for Sparse Matrices** . . . . . . . . . . . . . . . **45**
**3.4 Domain Decomposition Method** . . . . . . . . . . . . . . . . . . . . . . . . . . . **47**
3.4.1 Halo Exchange . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
3.4.2 Multilevel Graph Partitioning . . . . . . . . . . . . . . . . . . . . . . . . . 49
**3.5 Hybrid Parallelism** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **50**
3.5.1 Coloring of Unstructured Meshes . . . . . . . . . . . . . . . . . . . . . . . 51
3.5.2 Divide And Conquer Parallel Algorithms . . . . . . . . . . . . . . . . . . 53
**3.6 Vectorization on Unstructured Meshes** . . . . . . . . . . . . . . . . . . . . . . . **54**
**3.7 Optimizing Locality in Mesh Computation** . . . . . . . . . . . . . . . . . . . . **56**


## 3.1 Introduction

As explained in previous chapters, the evolution of the hardware architectures results in an increasing number of compute nodes and cores per node, and a decreasing memory per core. To make efficient use of these new concurrent resources, applications have to combine multiple levels of parallelism. In this thesis, we focus on irregular applications based on Finite Element Method (FEM), presented in Section 3.2, and working on 3D unstructured meshes. FEM applications commonly use the domain decomposition approach presented in Section 3.4 to generate data parallelism. Each process is assigned to the execution of a mesh subdomain and the communications between processes are usually handled by MPI. Unfortunately, most of the FEM applications currently in use exploit MPI domain decomposition both at distributed and shared memory levels. This results in increasing communication and memory bottlenecks. As explained in Section 3.4, in the FEM context, the large number of processes and subdomains induces data duplication of the frontier values, a.k.a halos, and a larger amount of communications.

The coloring approach, introduced in Section 3.5.1, is commonly used in shared memory systems to complement the domain decomposition approach. To exploit the shared memory parallelism, coloring is often associated to the OpenMP runtime. Divide & Conquer (D&C) algorithms have also been explored to

replace coloring at shared level. D&C consists in recursively dividing a problem into smaller independent subproblems and solve them in parallel. Several parallelization approaches based on D&C, similarly to the one developed during this thesis, are presented in Section 3.5.2. Moreover, in addition to distributed and shared parallelization strategies, modern supercomputers enable significant speedup by using their vector units. Researches on efficient vectorization are detailed in Section 3.6.

Lastly, to obtain good performance on modern architectures, it is crucial to optimize data locality. However, this is a challenging task when dealing with highly unstructured meshes parallelized among various levels of heterogeneous architectures. Data have to be uniformly distributed among the processing units and to fit into the lowest cache levels. Moreover, data brought to caches must be fully exploited before being evicted. Discussion on locality improvements strategies in mesh computation and how they differ from our approach is given in Section 3.7.

## 3.2 FEM Computation over Elements

In the context of finite element method, the main computational workloads consist of loops iterating over elements, nodes, or edges. These loops are non trivial to parallelize in shared memory since they access large amount of data, generate many indirections, and bring several dependencies between iterations. In addition, most of the industrial HPC applications work on unstructured meshes with irregular geometries and variable number of neighbors, making their parallelization even more challenging.

An illustration of the iterating process over elements in a simplified FEM application working a 2D regular mesh is presented in Figure 3.1 with its associated sparse matrix built from the mesh values. A coefficient is computed for each element from the values of their neighboring nodes or edges, depending on the application. Additionally, the calculation of the coefficient depends on the chosen operator. Once the coefficients are computed, the contribution of all the elements is reduced to the neighboring nodes or edges. The nonzero values of the resulting matrix correspond to these node and edge values. In Figure 3.1, the coefficient of the element $E1$ is computed from the values of the nodes $N1$, $N2$, $N3$, and $N4$. Then, the coefficient of the elements $E1$ and $E2$ are reduced to the nodes $N3$ and $N4$ and on the edge $(N3,N4)$. Let $\oplus$ be our reduction operation. Then we have, $N3 = E1 \oplus E2$, $N4 = E1 \oplus E2 \oplus E3 \oplus E4$, and $(N3,N4) = (N4,N3) = E1 \oplus E2$. The resulting symmetric and sparse matrix represents the linear system to solve. In 3D, even with a fixed number of edges per element, the number of neighboring elements can be arbitrarily large.

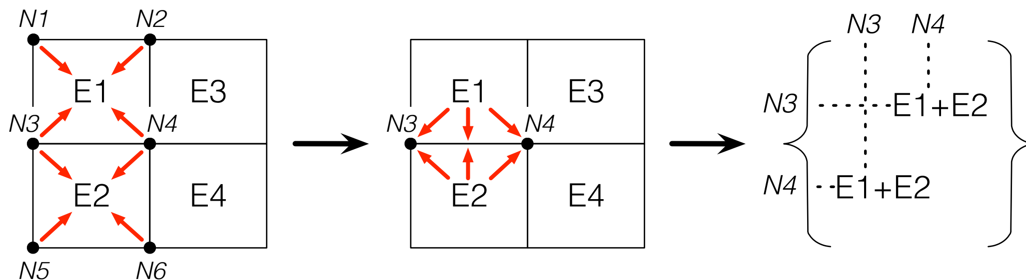



Figure 3.1: 2D illustration of a basic iteration over elements in FEM application working on a regular 2D mesh.

## 3.3 Compressed Storage Formats for Sparse Matrices

Sparse and symmetric matrices are mainly composed of zero values. In order to avoid the storage of these useless zero values and to save memory consumption and memory bandwidth when accessing the matrix elements, most algorithms use a compressed format. Let us consider a $n \times n$ sparse matrix $A$ composed of $nnz$ nonzero values. The naive approach, called coordinate format (COO), consists in storing for each nonzero values a tuple composed of the matrix element value, its row index, and its column index. However, this approach requires to store two additional values per non zero entry, resulting in $\mathscr{O}(3nnz)$ memory consumption.

The state-of-the-art storage format for sparse matrix is the Compressed Sparse Rows (CSR) [90] format. This format permits to store only $n+2nnz$ values. A first array dimensioned at $nnz$ contains the nonzero values of each matrix row stored consecutively. A second array dimensioned at $n$ contains the position in the value array of the first nonzero value of each row, as well as the total number of nonzero values at the end. In Figure 3.2, the first value of the first row, i.e. 1, is located at index 0 in the value array. The first value of the second row, i.e. 4, is located at index 3 in the value array, and so on until reaching the last ninth value. Lastly, a third array dimensioned at $nnz$, stores the column index of each nonzero value. The transposed version of the CSR format, called Compressed Sparse Columns (CSC) [90], also exists. It contains the nonzero values of each columns stored in consecutive memory locations, the index of the first nonzero value of each column, and the index of each row. The CSR storage of $A$ corresponds exactly to the CSC storage of $A^\top$. These three versions are compared in Figure 3.2.

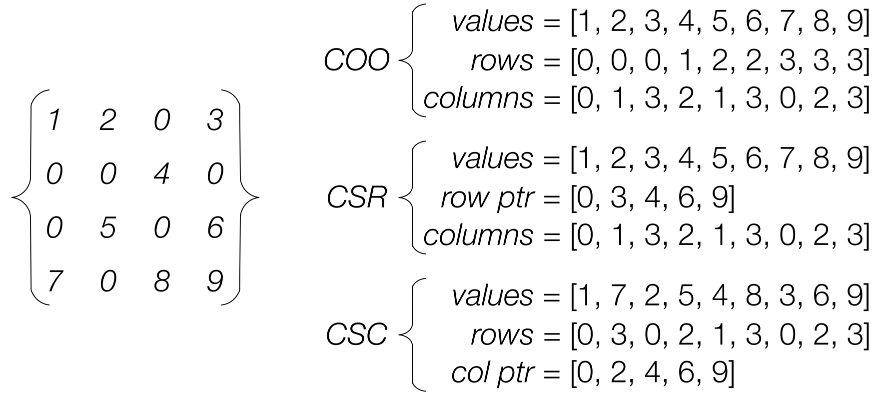



Figure 3.2: Presentation the COO, CSR, and CSC formats on a $4 \times 4$ sparse matrix.

However, applications dealing both with a matrix $A$ and its transposed $A^\top$, will at some point be penalized using either the CSR format favoring rows over columns or *vice versa* using CSC. Buluç *et al.* have proposed a new storage format, named Compressed Sparse Blocks (CSB) [91] which does not favor nor rows nor columns. CSB partitions the $n \times n$ matrix $A$ into square blocks, or submatrices, of size $\beta \times \beta$, with $\beta \ll n$. While CSR stores rows contiguously and CSC stores columns contiguously, CSB stores blocks contiguously. It is illustrated in Figure 3.3. Inside each block, nonzero values are stored following a Z-Morton ordering [92]. Top-left values are stored first, then top-right values, then bottom-left values and finally the bottom-right values. In Figure 3.3, the matrix values 1 and 2 are contiguously stored in first block, the 3 and 4 in the second block, and so on. Contrary to CSR or CSC format, the row and column indices indicate the position of a nonzero value within a block and not in the entire matrix. Therefore, the indices values are smaller and the row and column indices can be stored inside a unique integer. In Figure 3.3, the value 9 is located at row index 1 and column index 1 of the bottom right block. Lastly, a

block array dimensioned at $n^2/\beta^2$ indicates the first index of each block. For instance, the bottom right block starts at value 6 which is located at the sixth index of the value array. All-in-all, the CSB format uses the same amount of memory than the CSR and CSC formats. According to the authors, the CSB version benefits from a better scalability than standard CSR in a parallel Sparse Matrix Vector (SpMV) multiplication. However, it suffers of a higher consumption in memory bandwidth. Buluç *et al.* found a way to reduce the memory bandwidth consumption by up to two on SpMV using the CSB format [93]. They achieve that by using bitmasked register blocks and propose a new algorithm which better exploits symmetric matrices. In our D&C approach proposed in this thesis, the matrix values are gathered to the diagonal due to the applied permutations, as explained in more details in Section 5.2.3. This CSB format is then not the best suited to our needs. Indeed, since CSB partitions a sparse matrix into equal-sized blocks, using it with our D&C library would results in dense blocks along the diagonal and almost empty blocks beside.

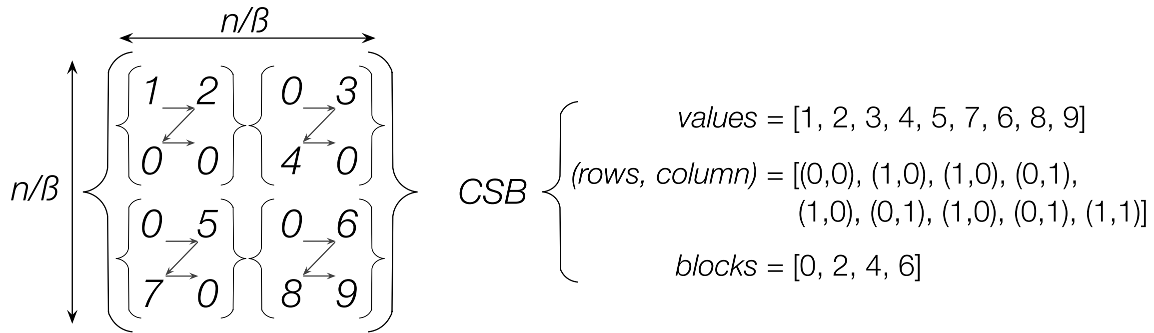
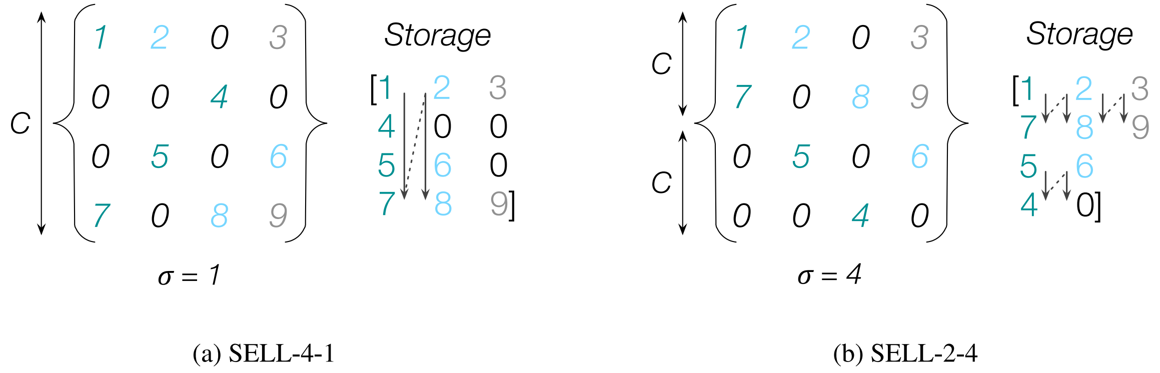



Figure 3.3: Presentation of the CSB format on a $4 \times 4$ sparse matrix with block size $\beta = 2$.

Another similar approach for cache blocking of sparse matrices was proposed by Nishtala *et al.* [94] as well as a block version of CSR, called BCSR [95]. Martone *et al.* also propose a recursive storage format for sparse matrices using Morton Z-ordering named Recursive CSR (RCSR) [96, 97] and a hybrid recursive format based on both CSR and COO formats, called RSB for Recursive Sparse Blocks [98, 99]. They evaluate their formats on sparse matrix-vector multiplication, BLAS operations, and matrix assembly. Other methods used to compress both the nonzero values and their indices in order to reduce the memory bandwidth requirements of the SpMV kernel have also been proposed [100]. Kreutzer *et al.* propose another sparse matrix format, called SELL-$C$-$\sigma$ [101], derived from a version of the ELLPACK [102] matrix format designed for GPGPUs. The SELL-$C$-$\sigma$ format is aimed to enable vectorization on wide SIMD units which can be found on modern multicores (e.g. IvyBridge), manycores (e.g. Xeon Phi), but also GPGPUs. The initial sparse matrix is cut into chunks of C rows. C contains the whole matrix rows in Figure 3.4a and half of them in Figure 3.4b. These chunks are consecutively stored in memory. The matrix nonzero values within a chunk are stored in column order. The first values of the rows form the first column, the second values form the second column, and so on. The columns of a chunk with less than C nonzero values are padded with zero values. The rows are also padded with zero values to match the length of the longest row of the chunk. In Figure 3.4a, the second values of each row, i.e. 2, 6, and 8, form the second column. Since the second row of the matrix has only one nonzero value, it is padded with zeros until reaching the longest row of size 3. The second column of value is then 2, 0, 6, and 8. This way, each column within a chunk forms an equal-sized vector of C nonzero values. To reduce the difference of size between rows of a same chunk, and therefore the padding overhead, the rows are sorted according to their length by chunks of size $\sigma$, where $\sigma$ is a multiple of C. In Figure 3.4b, there are 2 chunks composed of 2

rows and $\sigma = 4$, therefore, the 4 matrix rows are sorted together. First and last rows of length 3 are firstly stored, third row of length 2 next, and lastly the second row of size 1. This allows to save 2 extra zero values in the second chunk. Nevertheless, this approach can induce important padding memory overhead, especially on highly irregular meshes.




(a) SELL-4-1

(b) SELL-2-4

Figure 3.4: Presentation the SELL-$C$-$\sigma$ sparse matrix format.

Many sparse matrix storage formats exist and choosing the optimal one is really domain and hardware specific. With the actual heterogeneous architectures exploiting several parallelization patterns, this becomes a complex topic, especially on large industrial applications. Tools like Kokkos Array [103] appeared to help users to abstract the use of multidimensional arrays for computational kernels able to run on various architectures. Kokkos is a package of the Trilinos project [104] which is an API aimed to help the development of mathematical libraries. It provides several interoperable packages which cover a large range of applications, such as algebraic preconditioners or non linear solvers. The Kokkos Array library provides a collection of sparse and dense kernels, such as SpMV that are portable to multicores, manycores, and GPGPUs and a multidimensional array API. The selection of the appropriate data access pattern is internally handled by Kokkos at compile-time through C++ template meta-programming.

During this thesis, we investigated several matrix storage formats for our divide and conquer approach. In our opinion, a natural storage enabling efficient and scalable parallelization consists in storing the matrix values according to the leaves of the D&C tree. The nonzero values are reordered in order to follow the execution order of the D&C leaves. To avoid memory fragmentation, the matrix values accessed by the different leaves are contiguously stored in a single array. We have chosen the CSR format to store their indices. Our storage format is described in more details in Section 5.2.5.

## 3.4 Domain Decomposition Method

With the evolution of the computing resources and the increasing size of problems to solve, almost all scientific applications have taken the plunge of parallelism. But as stated in Section 3.2, in the context of *Finite Element Methods, FEM*, parallelism cannot be generated using a simple parallel loop due to the irregularity of the structure and the serialization of the reduction. The easiest solution to handle the reduction on mesh edges is to use locks or atomic operations. However, as the computation per edge is usually low, this approach is inefficient.

FEM applications are typically executed on 3D unstructured meshes as illustrated in the Figure 1 of the introduction. To generate parallelism, most of these applications use the *domain decomposition* method, illustrated in Figure 3.5. The initial mesh domain is split in as many subdomains as the number of cores required for the run. Each subdomain can compute its inner part in parallel. For each domain,

the frontier values are duplicated in a buffer, commonly named halo, which is exchanged after each computation phase.

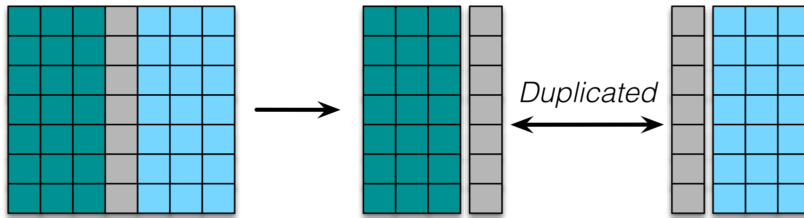



Figure 3.5: 2D illustration of domain decomposition.

### 3.4.1 Halo Exchange

As explained in Section 3.2, during the computation, values are computed on each cell of the mesh from the values of the neighboring edges or vertices, depending on the application. To maintain the correctness of the numerical results, the cells located at the frontiers between subdomains need to know the values of their neighbors located in another subdomain. Therefore, the slice of cells at the frontiers, called ghost cells or halos, are duplicated among the neighbor subdomains so that they can add their contributions. Then, these halos are exchanged at each time step to reduce the different contributions and obtain the final correct result. This is illustrated in Figure 3.6.

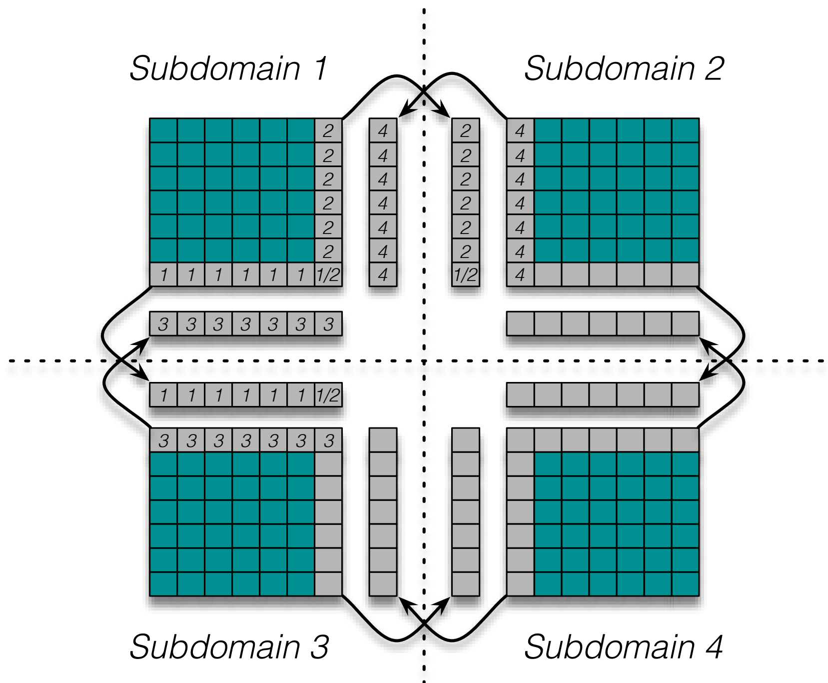



Figure 3.6: *Halo exchange* - At each time step, the halos are exchanged between neighbor subdomains.

### 3.4.2 Multilevel Graph Partitioning

To produce the subdomains, the first step consists in transforming the initial mesh into a graph. This graph can be either a primal or a nodal graph, depending if we want a partition based on vertices or on elements of the mesh. Once the graph is produced, it is then partitioned.

Graph partitioning is a broad topic and a NP-complete problem. There are many different approaches to partition irregular meshes. The most intuitive is the geometric partitioning where a graph is cut at fixed geometrical coordinates in order to produce equals parts [105, 106]. Some examples of geometric partitioner are the Recurse Coordinate Bisection (RCB) [107], the Recursive Inertial Bisection (RIB) [108], and the greedy algorithm [109]. However, if the geometry of the graph evolves during the computation, which is the case in the DEFMESH application for instance, the balancing between parts will change from an iteration to another. Most of the graph partitioners produce topological cuts, i.e. cuts done on vertices of the graph. Each of these partitioners focuses on different criteria such as the number of edges on the cut, the balancing between partitions, or yet the speed of the partitioning process.

In some problems, the topology of the mesh can also evolve during the computation. As an example, Adaptive Mesh Refinement (AMR) which is a method frequently used in numerical analysis, consists in dynamically adapting the accuracy of a problem to the most sensitive parts of the simulation. The mesh is refined by adding extra cells during the execution of the solver and it is therefore required to produce a new partitioning after each refinement step. To limit the impact on performance, the partitioning has to be quick and to induce few memory overhead. Recent researches on this topic have been explored such as the PaMPA project [110]. PaMPA aims at partitioning and remeshing in parallel. It uses the PT-Scotch partitioner [111] to transform a mesh into a dual graph and partition it. Partitions are then remeshed in parallel by third party sequential remeshers, e.g. MMG3D [112]. Lastly, the partitions are reintegrated together to reform the initial mesh.

The standard approach for topological partitioning is the Recursive Graph Bisection (RGB) [113]. Another well known technic is the multilevel spectral bisection [114]. However, this approach is slow

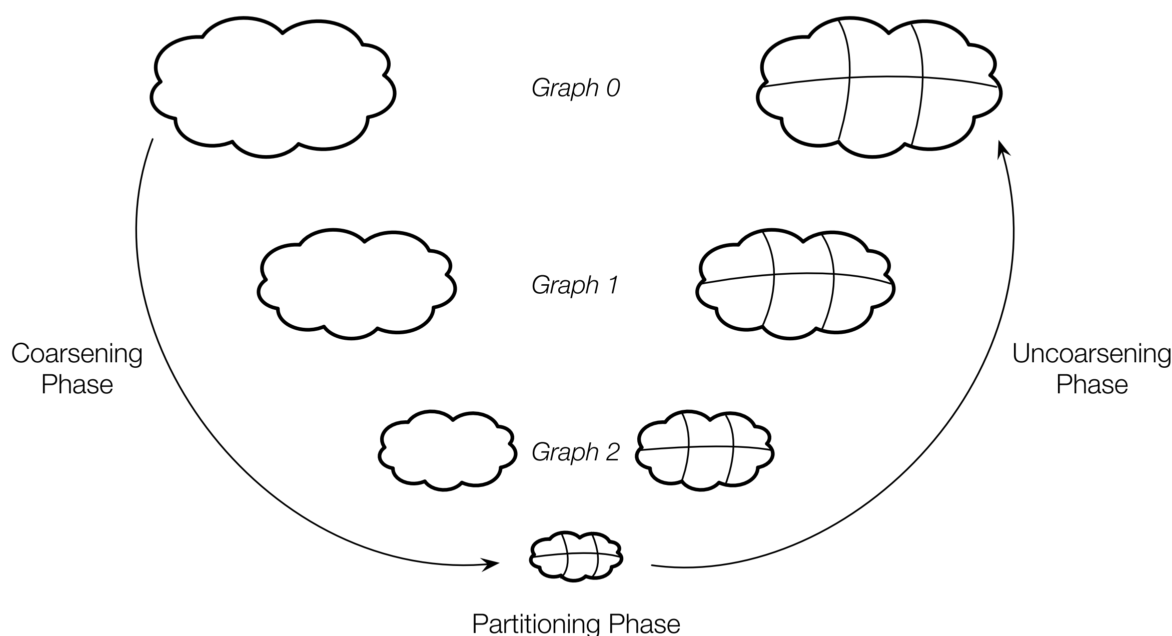



Figure 3.7: 2D illustration of the $k$-way partitioning process.

to produce partitions [115]. Most modern graph partitioners such as the well known METIS [116], use the multilevel $k$-way partitioning proposed by Karypis *et al.* [117, 118]. As many other partitioning methods [119, 120], $k$-way partitioning is based on the multilevel graph partitioning approach. In this approach, illustrated in Figure 3.7, the original graph is coarsened until it is composed of a small number of vertices. During this coarsening phase, a sequence of smaller graphs is constructed by collapsing together the vertices which are incident on each edge. Then the coarsened graph is cut in $k$ equal parts using one of the various existing partitioning approaches presented above. Lastly, the initial partition is successively refined and projected back to the original graph and is based on the Kernighan-Lin algorithm [121].

A distributed version of multilevel $k$-way partitioning has also been proposed by Karypis *et al.* [115]. They use graph coloring to produce independent parts of the graph and parallelize the coarsening and the refinement during the uncoarsening phase. Few years later, they propose another parallel version, called MT-METIS [122], based on OpenMP multithreaded parallelism instead of the previous ParMETIS version using MPI [123]. They observe a significant gain in memory consumption and a speedup of 2 compared to ParMETIS and PT-Scotch [111]. The benefits of exploiting shared memory parallelism in addition to distributed memory parallelism are explained in more details in the following sections.

## 3.5 Hybrid Parallelism

Using thread parallelism in addition to distributed domain decomposition brings several opportunities [124, 125]. This reduces the memory requirements, enables cache sharing, reduces the number of communications and I/O overhead. Indeed, at scale, a larger number of smaller subdomains leads to an increased communication volume and to load balancing issues. Moreover, the decreasing memory per core is not compatible with the increasing ratio of duplicated halos. For instance, as illustrated in Figure 3.8, on the 512 subdomains decomposition of our million vertices EIB use case, the ratio of nodes in halos to duplicate and to communicate at each iteration is over 60%. Solely relying on domain decomposition and distributed memory parallelism can limit the performance on current supercomputers, and this will

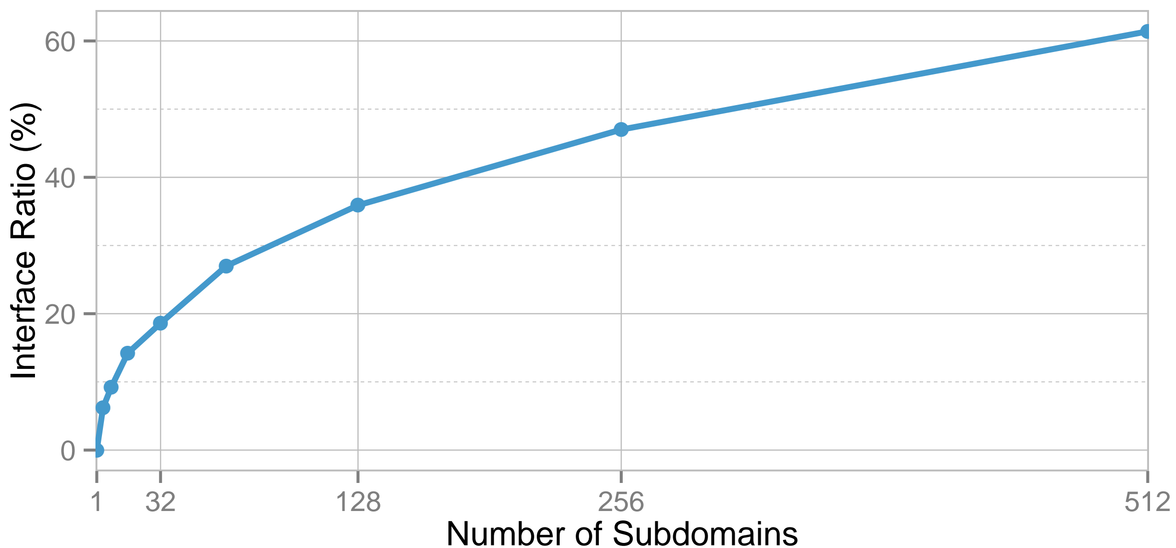



Figure 3.8: Ratio of the number of interface nodes to duplicate and communicate compared to the number of subdomains on the EIB use case.

even worsen on upcoming manycore systems. Williams *et al.* highlight the memory bandwidth bottleneck that occurs on many different parallel architectures when dealing with the SpMV kernel [124]. They list different ways to reduce the memory bandwidth requirements in SpMV, such as blocking, to reuse as much as possible the data bring to cache, or padding to avoid cache conflicts. They also highlight the importance of reducing the amount of communications on modern architectures using hybrid parallelism. Gourdain *et al.* also demonstrate the scalability issues of two different applications based on pure MPI domain decomposition [125].

We can also note that on certain iterative approaches, the domain decomposition negatively impacts the convergence [126, 127], regardless of whether it is implemented with processes or thread parallelism [127], and induces rounding errors [125, 128]. Therefore, using shared memory parallelism instead of domain decomposition can be beneficial for the convergence rate. This is not the case for the matrix assembly part and therefore, the observed improvements are only related to our new parallelization strategy. But this advocates for hybrid parallelization of the solver step inside each subdomain.

Several advanced parallelizations have been proposed for the FEM-assembly step. We identify four main groups of methods in the literature for mesh assembly parallelization in shared memory. Recent performance evaluations of all these methods on CPUs and GPUs can be found in [15, 16].

- *Assembly by mesh element with coloring* [17].
  This approach is discussed in details in next section.

- *Local assembly by mesh element followed by global assembly in a reduction step* [15].
  In this approach, all elements are computed in parallel without synchronization. To avoid data races, coefficients are duplicated and reduced in parallel during a second phase. Since each coefficient needs to be stored multiple times, this approach is limited by the available memory. Moreover, it increases the bandwidth requirements since the data have to be accessed twice.

- *Assembly by nonzero coefficient* [15, 18].
  This approach consists in creating parallelism between all the nonzero entries of the sparse matrix. This method has a very fine grain thread parallelism and therefore is better suited to GPUs than multicore architectures. Moreover, it introduces a large amount of redundant computation.

- *The local matrix approach* [16].
  In this approach, the reduction is deported to the solver phase. It requires no synchronization during the assembly but it increases the bandwidth requirements and the amount of computation in the solver. Moreover, it breaks the abstraction between the solver and the rest of the application.

### 3.5.1 Coloring of Unstructured Meshes

#### Thread Level Parallelism

A popular complementary approach to domain decomposition on current NUMA systems is to use shared memory parallelism. However, efficient parallelization in shared memory is challenging [127, 8] and recent manycore architectures expose the limit of current loop-level strategy [127, 9]. The common approach in use is mesh coloring [127, 17, 129, 10, 130].

Coloring avoids race conditions by assigning a different color to the elements sharing a reduction variable. Therefore, colors must be sequentially treated but elements of a same color are independent and can be processed in parallel by SIMD units, with no risk of race conditions. This is illustrated by a regular 2D coloring example in Figure 3.9 and a pseudo-code given in Algorithm 1.

Determining a minimal coloring is NP-complete. However, well-known heuristics can create efficient colorings with only a few more colors [17]. The state-of-the-art coloring algorithm, known as First-Fit,

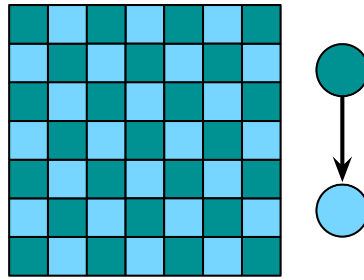


Figure 3.9: 2D illustration of mesh coloring.

**Algorithm 1:** Pseudo code for coloring and assembly.

```
1 NodeToElem ← Build_node_to_element (ElemToNode)
2 ElemToElem ← Build_element_to_element (NodeToElem)
3 ElemToColor ← Build_element_to_color (ElemToElem)
4 ColorToElem ← Build_color_to_element (ElemToColor)
5 foreach color ∈ ColorToElem do
6 |   foreach element ∈ color do in parallel
7 |   |   Compute_element_contribution (element)
8 |   end
9 end
```

or greedy coloring, is a polynomial algorithm which consists in visiting every element of the mesh and assigning it the smallest color available, i.e the smallest color which has not already been assigned to a neighbor element [127, 129]. To improve data locality and vectorization efficiency, elements of a same color can be contiguously stored. This algorithm produces the longest possible colors since as soon as a color is available, this color is reused.

In adaptive algorithms, the topology of the mesh regularly evolves. It is therefore necessary to recompute at each time a new coloring. Optimizations have been explored to reduce the compute time of the coloring algorithm by parallelizing it and removing costly synchronizations [131, 132, 133].

One drawback of this approach is the need of a costly global synchronization between each color. The more unstructured is the graph, the more colors are required [129], and therefore the more global synchronizations are on the critical path. The increasing core count will make global events induce prohibitive costs that must be avoided. Moreover, the global barriers make the approach sensitive to load balancing within each color.

Furthermore, since colors cover a large part of the mesh domain, they access almost all the nodes and edges of the domain, i.e. the CSR values. Therefore, treating one color will trigger the transfer of a large part of the mesh domain. Additionally, these CSR values are updated by several different colors and loaded multiple times into caches [127]. As large meshes do not fit in cache, data are accessed from the main memory. This multiplies the bandwidth requirement by a factor proportional to the number of colors. That can be mitigated by doing cache blocking before coloring. However, this multiplies the global barriers by the number of blocks, increasing the runtime overhead.

**Vectorization**

Originally, coloring was designed for vector machines. Indeed, the elements in a single color can be computed by vectors. In the current architectures, multicores and manycores are not pure vector machines, but, the CPU core itself contains vector operators and can be considered as a vector machine. While the coloring strategy is not efficient at system and node level for both data locality and synchronization, there are no such issues at core level. However, since unstructured meshes are reluctant to expose data parallelism, vectorization is a challenging task to solve. Furthermore, considering the block size required by the cache, either we color the domain before blocking and therefore, sacrifice locality, or, we color at block level. But as shown in Section 6.2.3, the data parallelism per block is too small for state-of-the art heuristics. Different approaches using coloring to enable vectorization are presented in Section 3.6.

### 3.5.2 Divide And Conquer Parallel Algorithms

Divide & conquer is a standard computation approach where the solution to a problem is obtained by recursively solving subproblems. D&C computations generate a tree structure where each node is associated to a problem instance and its children are associated to its subproblems. D&C computation are composed of two phases. The first one, called the divide phase, corresponds to the recursive division of an initial problem into subproblems. After the creation of the D&C tree, the leaves execute their local problem and return the results to their parents. The results of the local problem and of the subproblems are combined together until reaching the root of the tree. That's the combine phase.

Divide and conquer paradigm and recursion in general have been extensively explored for parallel algorithms [134]. One of the most famous algorithm using the D&C approach is the quick sort method [135]. Alternatively, the memory locality improvement brings by divide and conquer algorithms is known for a long time [136].

Martone *et al.* [99] describe a matrix assembly approach for Recursive Sparse Blocks, RSB, matrix on CPUs. Despite this format can provide good results in solver algorithms [98], the FEM assembly on RSB matrix is very demanding on memory bandwidth [99].

D&C algorithms have also been used to increase locality and to enable task parallelism for mesh visualization in [137, 138, 139] and for wave propagation in [140]. Despite the computation is not presenting patterns similar to mesh assembly, it might be possible to find other parts of FEM applications, such as mesh update or adaptation, which can use parts of their solutions. In [89], the authors propose a methodology based on Cilk with many similarities to ours. However, the study deals with a set of synthetic examples based on regular or naturally recursive structures, such as grids and trees. Therefore, it does not deal with revealing data parallelism for vectorization, partitioning unstructured meshes, and composing parallelism strategies with the distributed level. This limitation makes it difficult to apply to use cases coming from real applications.

Other examples using D&C can be found in dense linear algebra problems such as Qu *et al.* preconditioner [126] and Dongarra *et al.* LU [141]. They use nested dissection for numerical purpose and benefit from the concurrency. However, since it impacts the numerical results, they are limited in the level of dissection and therefore have limited concurrency and locality. Some gains can still be found in dense linear algebra [142]. In our approach, we use shared memory locality to reduce the synchronization cost of the original algorithm semantic and data dependencies. Therefore, we can produce concurrency without impacting the numerical results.

Frigo *et al.* [143] propose several cache oblivious algorithms, such as the matrix multiplication, matrix transposition, and Fast Fourier Transform (FFT) based on the recursive divide and conquer approach. They exhibit the performance advantage of using recursive algorithms over iterative algorithms for computers with caches, due to their native blocking properties. This observation is shared among several

researchers [144, 145, 146].

Divide & conquer algorithms are also used in the LAPACK library [147] to solve eigenvalues problems. In [148], authors adapt them to handle heterogeneous architectures composed of multicore CPUs and manycore GPUs. Their ideas are implemented in the Matrix Algebra on GPU and Multicore Architectures (MAGMA) project [149].

Similarly to our work, Goudin *et al.* propose a parallel algorithm for matrix assembly on irregular meshes [150]. They use nested dissection to produce a topological decomposition of a mesh into several subdomains and to reorder the mesh elements. They order the elements according to the number of subdomains updating one of their nodes. The non-local elements updated by a high number of processors are scheduled first and the local elements updated by only one processor are scheduled at last. The computation of the local elements only starts when all the processing of the non-local elements is complete. This way, the communications between the different processors contributing to a same non-local element can be recovered by the computation of the local elements mapped to only one process. This approach is also similar to Simmendinger *et al.* works [68]. However, to generate a large amount of parallelism, the mesh has to be divided in an important number of subdomains, leading to a higher proportion of non-local elements. This implies a larger number of communications and a smaller number of local elements, which can be used to recover the communications.

A recent work by Park *et al.* on HPCG [127] also finds the block coloring to have a bad locality. They propose the P2P approach, which aims at increasing data and communication locality while having one task per operation and coarsening them. This, in essence, is a D&C approach. However, they have to increase the number of parallel domains. In their case, this has a negative impact on the convergence rate and the final parallelism is much smaller than with block coloring. According to the authors, P2P is in general not suitable on Xeon Phi. Therefore they have to choose the coloring approach despite its bad locality.

Wu *et al.* provide a theoretical study on the communication complexity for divide & conquer parallel algorithms [151]. They quantify the trade-off between balancing the computation loads and minimizing the communication costs and establish lower bounds on the communication cost depending on the computation load balancing.

Most of the current D&C approaches search the concurrency in the algorithm instead of in the data [3]. In this thesis we advocate that, when applicable, producing efficient data decomposition and reordering to expose parallelism provides both locality and parallelism, without changing the numerical algorithm.

## 3.6 Vectorization on Unstructured Meshes

In addition to distributed and shared memory parallelism, vectorization units are present inside the compute cores. SIMD parallel units are available in processors for a long time, but this kind of parallelism has mainly been exploited by regular applications. Additionally, as explained in Chapter 1, the SIMD units are getting larger in both multicores and manycores. To achieve optimal performance on these new architectures, it is required to make an efficient use of the vector units. However, exploiting the underlying SIMD parallelism combined to the increasing MIMD parallelism provided by the larger number of cores of modern architectures is a challenging task. This is especially true when dealing with unstructured meshes where each vertex, edge, or element can be connected to an arbitrary large number of neighbors. The easiest way to vectorize an application is the compilers auto-vectorization which target loops iterating over arrays and try to vectorize them [152]. But auto-vectorizing compilers have a poor vectorization ratio, especially on irregular applications with non predictable memory accesses and computational patterns [153]. As soon as the compiler cannot ensure that the different iterations of a loop access independent memory locations, it does not vectorize this loop for semantic reasons. Fortunately,

more flexible programming APIs, such as the or Cilk array notation presented in Section 6.2.1, have emerged and provides new opportunities to tackle this challenge.

In a vectorized application, operations are handled by packed vector of values varying from 64 to 512 bits, depending on the target architecture. An example of vectorized sum of two arrays A and B into a third array C with a vector length of 4 integer values is given in Figure 3.10. When running an application dealing with double precision variables, such as those used during this thesis, on a Xeon Phi composed of 512 bits SIMD units, there is a theoretical speedup of 8× to reach by vectorizing perfectly the code. However, in order to load contiguous and large enough data to fill the SIMD units and that, on each one of the 60 physical cores of the KNC, it is crucial to save as much memory bandwidth as possible. Data blocking is then required to efficiently exploit data brought to caches, and cache line conflicts between cores have to be avoided. Moreover, while write conflicts can be protected through locking mechanisms at the price of costly overheads and critical contentions among threads, they are prohibited at SIMD level since there is no such locking mechanism.

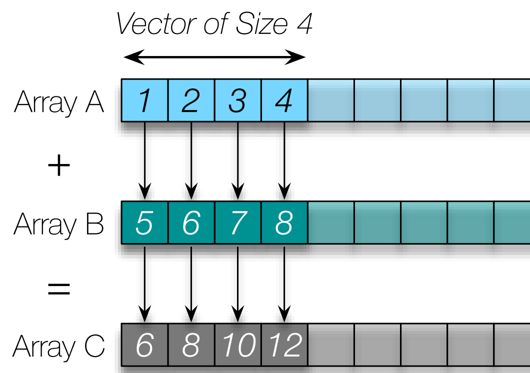



Figure 3.10: Example of a vectorized sum over 4 integer values.

Chen *et al.* explore good data placement in irregular applications to exploit recent SIMD architectures [35]. Their work is very close and posterior to our. They actually cite our work published at the PPoPP conference [23]. They show the impact on performance of the access stride size, i.e. the distant between accessed data, on the Intel Xeon Phi. Indeed, larger strides involve more cache lines and therefore more memory and bandwidth consumption. They propose a hierarchical matrix storage, named hierarchical tiling. The sparse matrix is stored into tiles, i.e. small square portions of the matrix, using the COO format presented in Section 3.3. Similarly to our observations made in Chapter 6, these tiles have to be large enough to fill the SIMD units, but small enough to fit in cache and have good locality. Since their tile sizes correspond to a number of values in the "virtually dense" sparse matrix and since the concentration of nonzero values is irregular, the tiles can have different sizes. A tile is processed by only one thread and exploit SIMD vectorization among values inside the tile. To avoid SIMD write conflicts, they partition the nonzero values inside each tile into conflict-free groups. The creation of these conflict-free groups is similar to the coloring approach presented in Section 3.5.1. But contrary to coloring where a nonzero value can belong to a color, i.e. a conflict-free group, if it has neither edge, nor vertex in common with the other nonzeros of this group, they build their conflict-free groups with nonzero values having neither common row ID, nor column ID. The conflict-free group is limited to the vector length of the target architecture. A similar approach is also applied between the different tiles in order to avoid thread conflicts among tiles.

Reguly *et al.* [154] highlight the difficulties of vectorizing efficiently unstructured meshes applications

on multicores and manycores. Their work makes use of the OP2 framework [155]. OP2 is a framework based on OPlus [156, 157], providing an API to develop applications based on unstructured meshes. It handles data partitioning, through ParMETIS and PT-Scotch, data dependencies, data layouts (SoA, AoS), and can support a large variety of architectures. OP2 supports MPI and OpenMP parallelism on multicore CPUs, and also CUDA and OpenCL on GPUs. Reguly *et al.* [154] use it to generate a wide range of vectorizing implementations for multicores and Intel Xeon Phi, and benchmark these different implementations. They explore the Intel implementation of OpenCL, which rely on TBB, to generate parallel tasks at the outermost level of parallelism. Then, they make use of the vector intrinsics of the Intel's Initial Many Core Instructions (IMCI) to vectorize the innermost level. IMCI is more flexible than the AVX instruction set and offers more vectorization possibilities. However, it is restrained to the Intel Knights Corner (KNC) architecture. Lastly, Löhner *et al.* propose a renumbering strategy aimed to minimize the cache misses and avoid memory contention for edge-based solvers running on unstructured grids [158]. Groups of independent edges are created similarly to a coloring technic applied to the edges. The edges are ordered according to their first node. When iterating over the edges, the first edge-node increases monotonically. Then, the second edge-node of current groups must be reused by the edges of the next group.

## 3.7 Optimizing Locality in Mesh Computation

In Section 5.2.3, we show the positive impact of D&C on data and communication locality, and by extension on bandwidth. Other approaches for locality in FEM computations have been explored. Most of these approaches consider the sparse matrix product locality since this is the most consuming part of solvers. Many softwares in use, as DEFMESH, are based on the Reverse Cuthill-Mckee (RCM) [159] ordering to improve the locality in SpMV. As shown in Section 5.2.3, we obtain a much higher locality in the studied problems by using our recursive D&C reordering.

In a previous study, Oliker *et al.* [160, 161] already evaluate METIS as a strategy for locality in sparse matrix and compare it to the Reverse Cuthil-McKee (RCM) approach [159] and to the self-avoiding walk (SAW) approach [162]. But they do not apply a recursive strategy to permute the nodes and they do not decompose the separator part. As a result, they have not fully exploited the sparsification of the elements outside the diagonal block, and they have not exploited the associated parallelism. According to Oliker, METIS does not appear to be the best strategy; still, when applying our recursive D&C strategy over it, the graph dissection provides better results on our use cases, as shown in Section 5.2. One of the best approaches tested by Oliker *et al.* is the Self-Avoiding Walk [162], SAW, a space-filling curve approach. The principle is to run over the elements with a pattern minimizing the spatial distance between the current element and the next one. We do not plan to use it directly for the ordering. However, as a future work, we consider using a similar approach for region coarsening as a cost effective alternative to METIS. The objective is to produce smaller partitions with a better load-balancing.

Yzelman *et al.* introduce a matrix reordering method based on a recursive partitioning of the initial matrix [163]. The matrix is initially stored in a variant of the CSR data format using a Z-ordering, a.k.a Morton ordering, space filling curve [164]. This format is aimed to prevent the jump between the last to the first column index when iterating over the matrix rows. In standard CSR order, values are stored from the first column to the last one. Once reordered, when iterating to a new row, the first value accessed is located at the same column index than the previous value from previous row. Nonzero values are then accessed in reverse order. This process is repeated at each new row, changing each time the access order. The matrix is then transformed into an hypergraph, recursively partitioned in one dimension, and reordered. Later, they extend their approach to exploit blocking properties. They process nonzero values block by block using two-dimensional partitioning and block data structure [165].

Nested dissection is used to improve locality in mesh computation for a long time [166]. In [167], authors use nested-dissection for reordering the columns of sparse matrices in sparse LU factorization. Authors show that nested-dissection reordering allows to reduce the amount of memory required and the amount of work.

# PART II

# CONTRIBUTIONS

CHAPTER 4

# INTRODUCTION TO CODE MODERNIZATION

**Contents**

**4.1 Introduction** . . . . . . . . . . **61**
**4.2 Dassault Aviation's Industrial Applications** . . . . . . . . . . **62**
4.2.1 DEFMESH Mesh Deformer Application . . . . . . . . . . 62
4.2.2 AETHER Aerodynamic and Thermodynamic Application . . . . . . . . . . 63
4.2.3 List of Use Cases . . . . . . . . . . 63
**4.3 Proto-Application Concept** . . . . . . . . . . **64**
**4.4 D&C Library Integration in AETHER** . . . . . . . . . . **65**
**4.5 Conclusion** . . . . . . . . . . **65**

## 4.1 Introduction

As we have seen in the previous part, to take full advantage of future HPC systems, hybrid parallelization strategies are required. However, very large industrial codes cannot be rewritten from scratch. To cope with the rapid evolution of the underlying hardware, these legacy codes need a profound refactoring, known as the code modernization issue. We experiment the concept of proto-application as a proxy for complex scientific simulation codes optimization. We explore the possibility of using proto-applications to allow early phase exploration of technologies and design options in order to make better and safer choices when it comes to modernize the full scale application. The Section 4.3 presents the proto-application concept and our implementation.

In this thesis, we propose an original approach for efficient parallelization of Finite Element Method (FEM) applications presented in the next chapters. This approach has been developed in a C++ library, called DC-lib, which can be interfaced with minimum intrusion to the original code. The D&C library has been used to modernize two industrial applications from Dassault Aviation. The DEFMESH mesh deformer presented in Section 4.2.1 and AETHER, a very large Computational Fluid Dynamics (CFD) application presented in Section 4.2.2 and using an approach comparable to DEFMESH. The modernization scenario of these Dassault Aviation applications is summarized in Figure 4.1.

As a first step we built Mini-FEM, a proto-application extracted from the assembly step of DEFMESH to simplify the development and the validation of our D&C library. Then, the performance gain brought

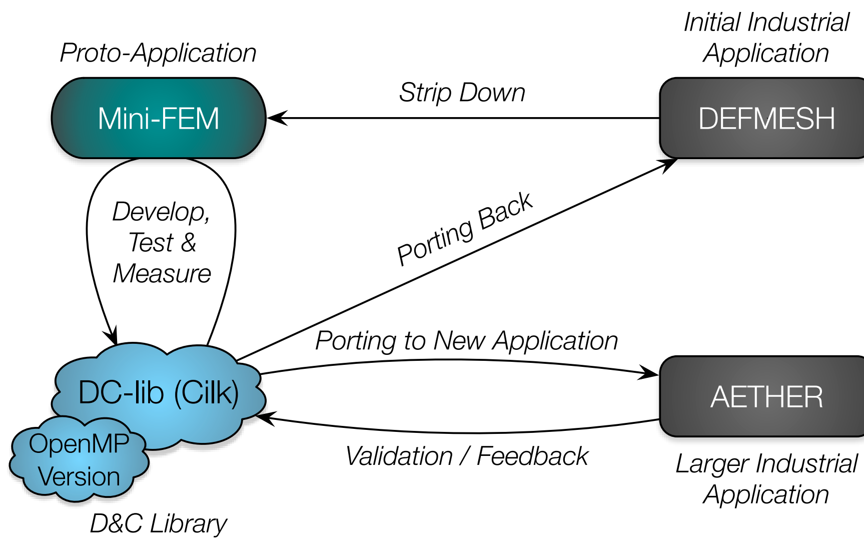



Figure 4.1: Code Modernization Scenario.

by our D&C library originally applied to Mini-FEM have been ported back to the original DEFMESH application. It produced impressive results in terms of speedup and efficiency on both superscalar multicore clusters and Xeon Phi manycores. However, it does not demonstrate the ability of a proto-application to generate generic approaches for a class of applications using unstructured meshes computation. Moreover, it was not clear if different runtimes and optimizations can be experimented with a single proto-application and have similar conclusion in the real applications. Therefore in a third time, we have ported the D&C library to the AETHER industrial code and validated the performance gain brought by the DC-lib. This is presented in Section 4.4.

Lastly, we made from the Mini-FEM proto-application an additional experiment which consists in replacing the Cilk Plus runtime by OpenMP 3.0 tasks. The proto-application approach allows a quick prototyping, implementation, and debugging of the OpenMP version. The comparison between these two runtimes is detailed in Section 5.2.8.

## 4.2 Dassault Aviation's Industrial Applications

### 4.2.1 DEFMESH Mesh Deformer Application

The DEFMESH application is an industrial fluid dynamics Fortran code based on FEM. It is originally parallelized using MPI domain decomposition as presented in Section 3.4. This unstructured mesh deformation code for CFD application is an important numerical module in Dassault Aviation aerodynamic optimization environment. It is also used in other simulations which may include surface variations of larger magnitude, such as in aero-elastic interactions or dynamics of moving bodies.

DEFMESH implements a three-dimensional elasticity-like system of equations from given surface data. These equations are solved by two different algorithms. The first one is a linear algorithm used when the magnitude of the surface data is small: the equations can be linearized into a system of linear equations. The second one is a non-linear algorithm where surface data of large magnitude are cut into a succession of small increments. The original equations are solved as a non-linear succession of linearized sub-problems, consisting of systems of linear equations. Each linear system is described as a symmetric

definite positive matrix. The systems are solved by a standard Conjugate Gradient (CG) algorithm.

DEFMESH implements two options for the definition of the operator for its system of volume equations. The first one is the laplacian operator. It calls three times the CG algorithm at each linearized step. Each CG call computes the solution of a scalar system of linear equations. The second operator is the elasticity one. This option implements the full 3D linear elasticity operator. It couples the three coordinates of the nodes. Therefore, at each linearized step, the CG algorithm has to solve a 3 by 3 system of linear equations. The elasticity operator permits smoother mesh deformations of greater magnitude than the laplacian operator.

The DEFMESH main kernel is performed in three steps. The first one is the FEM assembly, where mesh data are gathered into a sparse matrix. The second step is the solver which works on this matrix and computes optimal displacements. The final step is the update of mesh coordinates, using previously computed deformations.

### 4.2.2 AETHER Aerodynamic and Thermodynamic Application

AETHER is a large CFD simulation framework, developed for more than 30 years. It roughly contains 130,000 lines of code, combining different versions of Fortran and different programming styles. It solves Navier-Stokes equations with a finite element method on unstructured meshes. Like DEFMESH, it is composed of three main steps which consist in building the linear system of equations, solving it, and updating the mesh values for the next iteration.

AETHER is parallelized using MPI domain decomposition. For each domain, the frontier values are duplicated in a buffer exchanged after each computation phase. At scale, a larger number of smaller subdomains leads to an increased communication volume and to load balancing issues. For instance, as illustrated in Figure 3.8, on the 512 subdomains decomposition of our 1 million vertices EIB use case, the ratio of values to duplicate and to communicate at each iteration reaches 60%.

To take benefit from the SIMD architectures, AETHER is vectorized using a traditional coloring technique as the one presented in Section 3.5.1. In order to mitigate the data locality issue of the coloring approach, AETHER uses a function which retrieves data from a main global array and transfers them into a local and smaller block. This data block is temporarily stored in a contiguous layout in order to be treated vectorially. Results are then written back to the global structure. This method requires a lot of redundant random data accesses and leads to a significant overhead.

A preliminary version of hybrid parallelization using MPI and OpenMP has been proposed by Dassault. It reuses the blocks obtained with the coloring algorithm initially developed for vectorization. The main assembly loop is parallelized using the *omp parallel do* pragma. Since only loops are parallelized, the sequential part remains large. Therefore, the parallel efficiency is very low and cannot compete with the pure MPI version.

### 4.2.3 List of Use Cases

During this thesis, we performed our experiments on several unstructured meshes from Dassault Aviation.

- The first one, LM6, is a small mesh composed of 27,499 vertices and 152,086 elements used for developing and debugging.
- The second one, called EIB, is illustrated in the Figure 4.2 and represents the displacements of a fuel tank along an airplane fuselage. It is composed of 1,079,758 vertices and 6,346,108 elements.
- Another use case, named F7X and with a size similar to EIB, has also been used on the AETHER application. This F7X mesh represents the Falcon 7X jet.

- Lastly, the FGN use case is the larger mesh used during this thesis. It is composed of 7,173,650 vertices and 42,574,409 elements and it represents a generic Falcon jet.

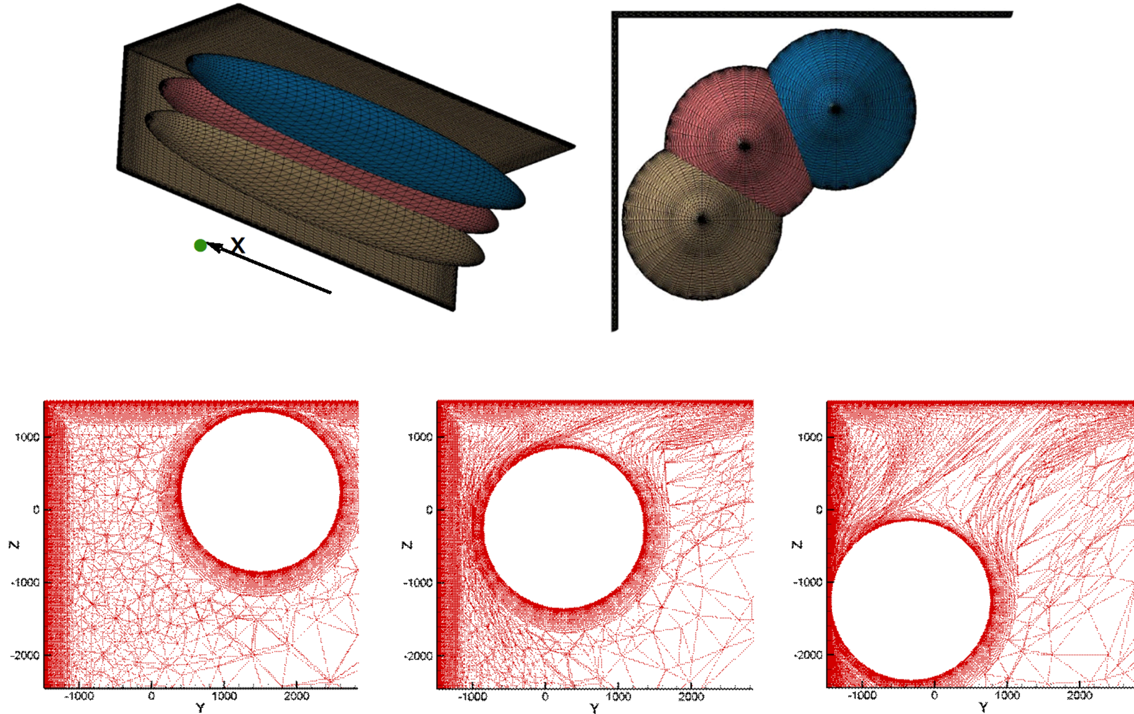



Figure 4.2: Illustrations of the 3D EIB fuel tank position optimization use case.

## 4.3 Proto-Application Concept

This section presents the proto-application concept [168, 169], also known as proxy-app (e.g. NERSC trinity, Argonne CESAR). The objective of a proto-application is to reproduce, at scale, a specific behavior of a set of HPC applications and support the development of optimizations that can be translated into the original applications. It should be easier to execute, modify, and re-implement than the original full-scale applications, but still be representative of the targeted problems. It represents a key opportunity to simplify the performance analysis and accelerate the decision making process. Barret *et al.* [170] give an overview of successful uses of this approach. For instance, MiniFE is a mini application representing finite element methods on GPUs on which a register spilling issue has been identified and solved. However, the feedback to the end-user has not yet been clearly identified and demonstrated.

The process of building a proto-application is based on an intensive profiling of the original application to be able to localize and characterize the targeted issues to optimize. Then, either we strip-down the original application to the essence of the problem, or we build-up a synthetic benchmark that exposes the same behavior. The idea is to support research on representative use cases of actual applications, instead of generic benchmarks. The message to the application developers is the following: *"if you cannot open your applications and use cases, you can open the problems"*. It will leverage:

- *Community engagement*, by providing up-to-date realistic use cases.

- *Reproducible and comparable results* on a common reference set of applications.
- *Direct valorisation* of the community improvements by providing a close to application code interface between the community and the application developers. They will be able to implement the relevant improvements into their codes.

The proto-application concept is central in the FP7 EXA2CT european project [68]. Our Mini-FEM proto-application, built during this thesis, is a collaboration between UVSQ and Dassault Aviation. It is a strip-down version of the DEFMESH application from Dassault Aviation. The Mini-FEM proto-application consists of the first step of DEFMESH: the FEM assembly. The FEM matrix assembly step has been chosen since it is the first step of many applications, such as seismic simulation, metal forming simulation, or crash test simulation. It consists on building the matrix describing the linear system of equations to solve from a given mesh.

Mini-FEM captures the input data of the DEFMESH assembly step and changes its internal computation to remove the exact physical model while keeping its complexity and data parallelism. The variable's names are also changed to be related to the algorithm and not to the physical problem. To validate our results, Mini-FEM captures the output of the DEFMESH assembly step and compares the results. In addition to the initial MPI domain decomposition version, the algorithm is parallelized with the D&C library. The two open-source use case LM6 and EIB from Dassault Aviation are shared with the proto-application. The optimizations we made have been implemented in the original DEFMESH application and are currently being integrated in the AETHER application.

## 4.4 D&C Library Integration in AETHER

Integrating the D&C library into the AETHER code was straightforward. However, we still encountered difficulties with some incompatible Fortran constructs detailed in a previous publication [24]. As shown in more details in Section 5.2.6, the original Fortran code has been modified with minimum intrusion. Integrating the D&C library into an application consists in two calls to our C++ library which can be commented out to fall back on the original pure MPI application. The first call triggers the initialization phase required by our D&C library. The second call triggers the parallel execution of the original FEM user function.

Moreover, the D&C library improves the data locality of the target applications by recursively reordering the data as explained in Section 5.2.3. Firstly, inside a leaf, the nodes and elements are permuted in order to place consecutively all to those from the left leave, then the right and at last the separator. Secondly, since the task distribution follows a recursive tree, the neighboring leaves are stored contiguously. Therefore, data locality is improved both inside a domain and between the neighboring domains. By using the D&C library, the L3 cache misses in the AETHER application have been divided by 15.

## 4.5 Conclusion

In this chapter, we have introduced the proto-application concept used to ease the development and the experimentation of new algorithm. We have presented the targeted DEFMESH and AETHER FEM applications from Dassault Aviation and the striped-down Mini-FEM proto-application which we released open-source under the LGPL 3.0 license. Thanks to the positive results obtained on the proto-application, the D&C library is currently being incorporated in the DEFMESH and AETHER industrial applications. Moreover, the D&C library is planned to be used by other applications in the near future.

CHAPTER 5

# NODE LEVEL PARALLELISM

**Contents**


**5.1 Introduction . . . . . . . . . . 67**

**5.2 Shared Memory Divide and Conquer on Unstructured Meshes . . . . . . . . 68**

5.2.1 Notations Used . . . . . . . . . . 69

5.2.2 Recursive Bisection . . . . . . . . . . 69

5.2.3 Locality Improvement . . . . . . . . . . 70

5.2.4 Numerical Stability of the Results . . . . . . . . . . 72

5.2.5 D&C Matrix Storage Format . . . . . . . . . . 73

5.2.6 Precomputation of the D&C Tree . . . . . . . . . . 73

5.2.7 Cilk Plus Implementation . . . . . . . . . . 74

5.2.8 OpenMP Implementation . . . . . . . . . . 75

**5.3 Experimental Results . . . . . . . . . . 77**

5.3.1 Experimental Setup . . . . . . . . . . 77

5.3.2 FEM Assembly . . . . . . . . . . 78

5.3.3 Solver . . . . . . . . . . 83

5.3.4 Comparison Between Cilk Plus and OpenMP 3.0 Tasks . . . . . . . . . . 83

**5.4 Conclusion . . . . . . . . . . 87**


## 5.1 Introduction

Current algorithms and runtimes struggle to scale to a large number of cores and show a poor parallel efficiency. The increasing number of cores and parallel units described in Chapter 1 results in a severe challenge for performance scalability. Relying solely on domain decomposition and distributed memory parallelism limits the performance on current supercomputers. At scale, a larger number of smaller domains can lead to an increased communication volume and to load balancing issues. When using a finer domain decomposition, the ratio of frontier elements which are duplicated and communicated is growing and therefore, the memory and bandwidth consumption rise. This limits the scalability when the communications become predominant. Moreover, the decrease of the memory per core is not compatible with the memory overhead of a finer domain decomposition.

In order to mitigate the node scalability issues, users can modify their application using hybrid process and thread parallelism to take advantage of the full topology of the machine, and enhance data and

synchronization locality. However, efficient parallelization in shared memory is a challenging and error prone task. A common hybrid implementation uses OpenMP loop parallelization in addition to MPI domain decomposition. But while MPI executions are usually 100% parallel, the loop parallelization approach results in many remaining sequential parts. According to Amdahl's law, by increasing the number of cores, the time proportion of the sequential part increases. Therefore, the traditional loop approach using OpenMP for shared memory parallelization fails to scale efficiently. This is illustrated in the Figure 2 of the general introduction.

In this chapter, we propose and evaluate a new approach based on the Divide and Conquer (D&C) principle to efficiently exploit shared memory parallelism on FEM applications working on unstructured meshes. Our implementation relies on the Intel Cilk Plus task-based runtime [12] presented in Section 2.5.2. We compare this hybrid approach using D&C to the pure domain decomposition and to a state-of-the-art hybrid approach using mesh coloring. Our target application is the DEFMESH application parallelized with MPI domain decomposition and presented in Section 4.2.1. As explained in the previous chapter, we build a proto-application, named Mini-FEM, representative of the assembly step of the DEFMESH application to ease the development process. Our D&C approach has been developed from this proto-application and has led to a library, called DC-lib. The D&C library outperforms the coloring version and the original implementation only based on MPI domain decomposition.

In a second step, we used the proto-application as a basis for the development of an OpenMP version of the D&C library, which consists in replacing the Cilk Plus runtime by OpenMP 3.0 tasks [79]. The proto-application allows a quick prototyping, implementation, and debugging of the OpenMP version. We experiment in Section 5.3 this new version in the Mini-FEM proto-application and the AETHER application to measure the optimization portability from the proto-application to a real application. OpenMP tasks are simple to program and provide comparable but sensibly lower performance than the Cilk implementation. As confirmed in the literature [171], this comes from the lower overhead of the Cilk Plus runtime at scale and from the scheduler which provides effective dynamic load balancing with work-stealing. These experiments lead us to the conclusion that proto-applications are a great opportunity to develop and validate code optimization while preserving the portability into large industrial applications.

## 5.2 Shared Memory Divide and Conquer on Unstructured Meshes

As stated above, to exploit shared memory parallelism when dealing with unstructured mesh applications, a common approach consists in using a mesh coloring algorithm with SPMD models of parallelization. However, as described in Section 3.5.1, mesh coloring allows a very efficient vectorization but has a poor locality and requires global synchronizations. Therefore, we focus on replacing this approach by a task-based approach using the D&C principle. As viewed in Section 2.5, shared memory task-based runtimes present many interesting advantages. Their very fine grain parallelism allows weak synchronization and gives slack for better load balancing techniques such as work stealing [11, 12]. With a small number of cores, the scalability of the D&C parallelization is expected to be equivalent to the MPI domain decomposition. Furthermore, contrary to the MPI approach, the D&C algorithm complexity is independent from the number of cores and should continue scaling.

The main idea of our D&C parallelization approach is to enable shared memory parallelism while preserving a good data locality and minimizing the synchronization costs. With a higher number of cores, we increase the number of threads instead of increasing the number of MPI domains. This way, we take benefit from replacing communication by data sharing. Moreover, global synchronizations from MPI domain decomposition are broken into small synchronizations in shared memory at node, socket, and core level. The D&C approach is also particularly interesting for its ability to scale naturally to an increasing number of nodes thanks to its architecture oblivious concept. Each D&C task is responsible of its own

data and there is a very minimal amount of sharing, avoiding costly locks and coherency protocols.

D&C consists of two steps described in following subsections. The first step is the recursive decomposition and permutations within each MPI domain and the second step is the recursive execution of the D&C recursive tree using Cilk. Since the computation may be low in some FEM loops, it is essential to save bandwidth, reduce the CPU idle time, and minimize the runtime overhead to reach maximal performance. This is the goal of our D&C approach. With the experiments presented in this chapter, we want to demonstrate that recursive algorithms coupled to task-based runtime have the potential to overcome this challenge.

### 5.2.1 Notations Used

A D&C tree is called a $(N,h,d)-tree$ where,

- $N$ is the number of nodes in the tree,
- $h$ is the height of the tree,
- $d$ is the maximal number of children of a node, in our case 3.

A node is at tree level $i$ if it is the $i$-th node on the path from the root to the node. The root is node located at tree level 1. The height of the tree is the maximum tree level. The creation of the D&C tree is a recursive process where each node starting from the root of the D&C tree spawns a maximal number $d$ of children. When a node does not spawn any children, the node is a leaf, else it is an internal node.

### 5.2.2 Recursive Bisection

D&C is based on topological recursive bisections of the mesh. As shown in Figure 5.1, the rational is to recursively divide the work in parallel tasks and to synchronize these tasks locally. At each recursion level, three partitions are created. Two independent left and right partitions, and a separator partition composed of the elements on the cut. The left and right partitions created by these bisections do not share any element and can be executed in parallel. However, the separator elements in the middle have edges on both sides and must be processed after the left and right partitions. All the partitions, including the separators, are recursively partitioned providing a large amount of parallelism. In a pure architecture oblivious D&C approach, the mesh would be partitioned until it only remains one element per partition. However, to exploit core level parallelism, the L1 cache size is a better choice for the partitions size.

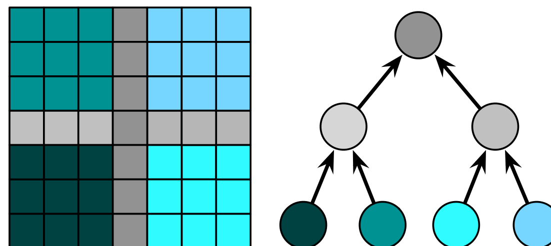


Figure 5.1: D&C recursive tree. The left and right partitions are executed in parallel before their separator elements on the cut.

The resulting global tree is not a simple binary tree as shown in Figure 5.1, but a more complex and unbalanced tree structure. The load balancing between the partitions is important since it influences the

depth of the recursive tree. A balanced tree will minimize the maximum depth and therefore the number of synchronizations on the critical path. Therefore, it is important to use a good partitioner to build equal partitions. In this study, we use the METIS graph partitioner [116] presented in Section 3.4.2 to obtain a good balance between the domains size while keeping a reasonable interface size. The partitioning computed by METIS is topological, cuts are done on edges rather than on geometrical coordinates. Therefore, it is independent from the rest of the computation and allows to compute partitions only once for a mesh. This partitioning is precomputed and the associated permutation is stored with the mesh file. During the run, the application needs to apply the precomputed permutation before executing the recursive FEM routine.

### 5.2.3 Locality Improvement

During the creation of the D&C recursive tree, in order to increase the data locality, we permute the nodes and the elements arrays, as shown in Figure 5.2. Inside each MPI domain, the recursive bisection using METIS results in discontinuous storing of the element and node values inside the D&C partitions. Therefore, after each recursive bisection, the node and element values are reordered according to the execution order of the D&C tasks, i.e. the left and right partitions followed by their associated separator. As a result, these permutations improve the locality between tasks as their distribution is following the recursive domain decomposition. Secondly, the nodes and the elements are consecutively stored in memory inside each D&C partition, improving intra-task locality. Since these permutations are applied to all MPI domains, we renumber the values at the frontier so that exchanged data are consistent.

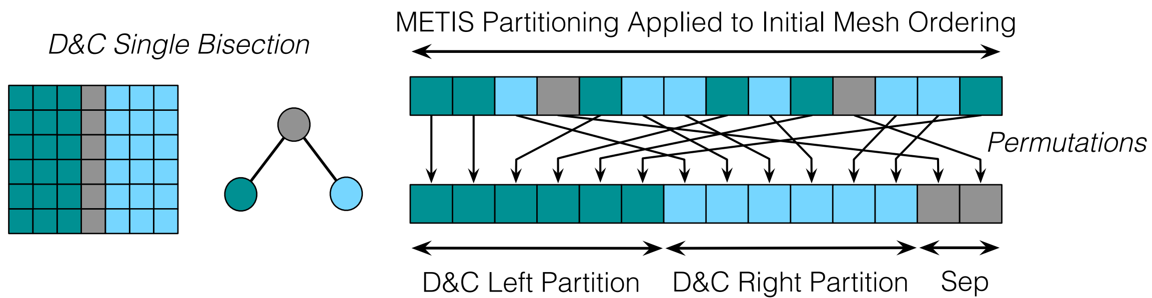
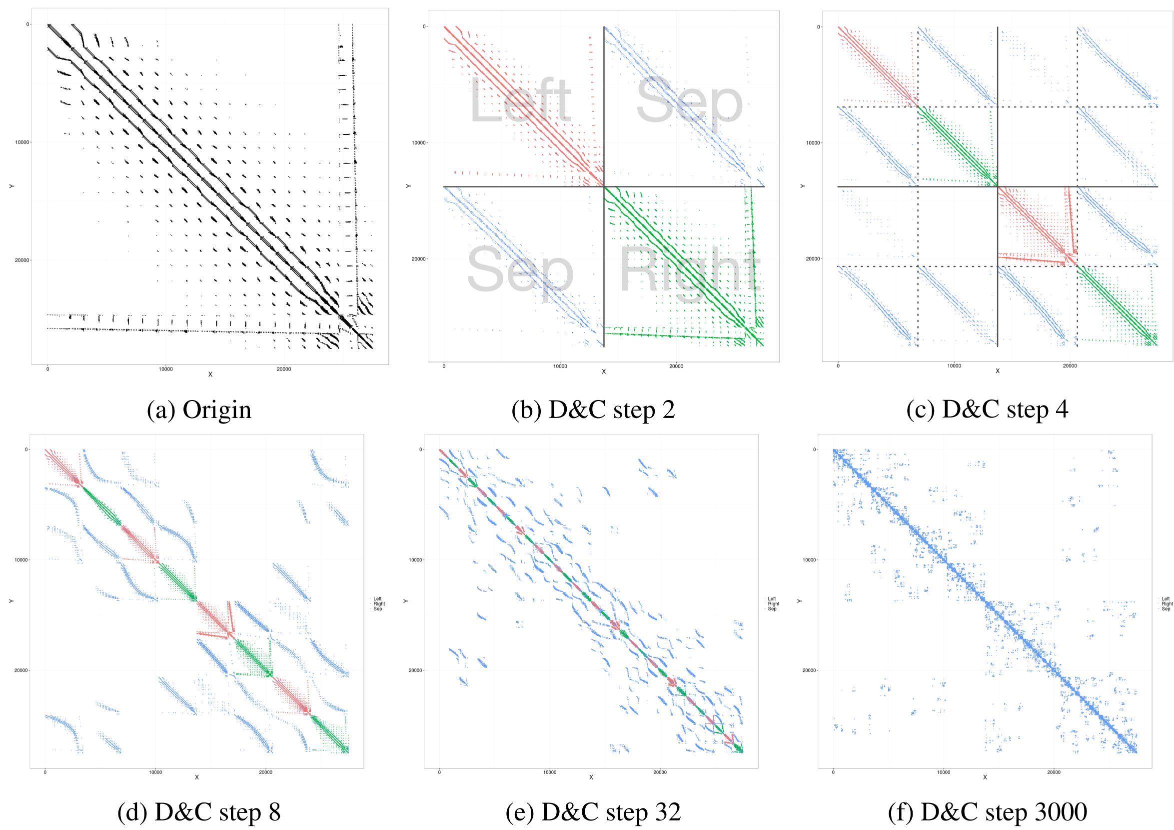



Figure 5.2: D&C permutations. For each cut, elements are reordered in left, right and separator partitions.

The impact of reordering a CSR matrix is illustrated in Figure 5.3. The matrices represent the LM6 small use case provided by Dassault Aviation before and after applying the recursive D&C permutations. Each pixel on the graph represents a nonzero value in the node to node symmetric CSR matrix. The closer to the diagonal are the nonzero values, the closer are stored the elements in memory. Figure 5.3a is the initial matrix. Figure 5.3b corresponds to the matrix after one bisection. The top left corner contains the left partition nonzero values. The bottom right corner contains the right partition values. The interactions between left and right partitions are symmetrically stored on the top right and bottom left corners. Since the separator is a slice splitting the left and right partitions, the values are one order of magnitude less dense in the separator part of the matrix. Figures 5.3c, 5.3d, 5.3e, and 5.3f represent the matrix after further bisection iterations. As expected, the resulting permutation concentrates the elements corresponding to the D&C left and right leaves along the diagonal, with very sparse elements outside which correspond to the separator leaves.

This permutation not only benefits to the assembly step locality, but also to the SpMV kernel in




(a) Origin (b) D&C step 2 (c) D&C step 4

(d) D&C step 8 (e) D&C step 32 (f) D&C step 3000

Figure 5.3: Impact of the D&C reordering on the CSR matrix associated to the LM6 use case.

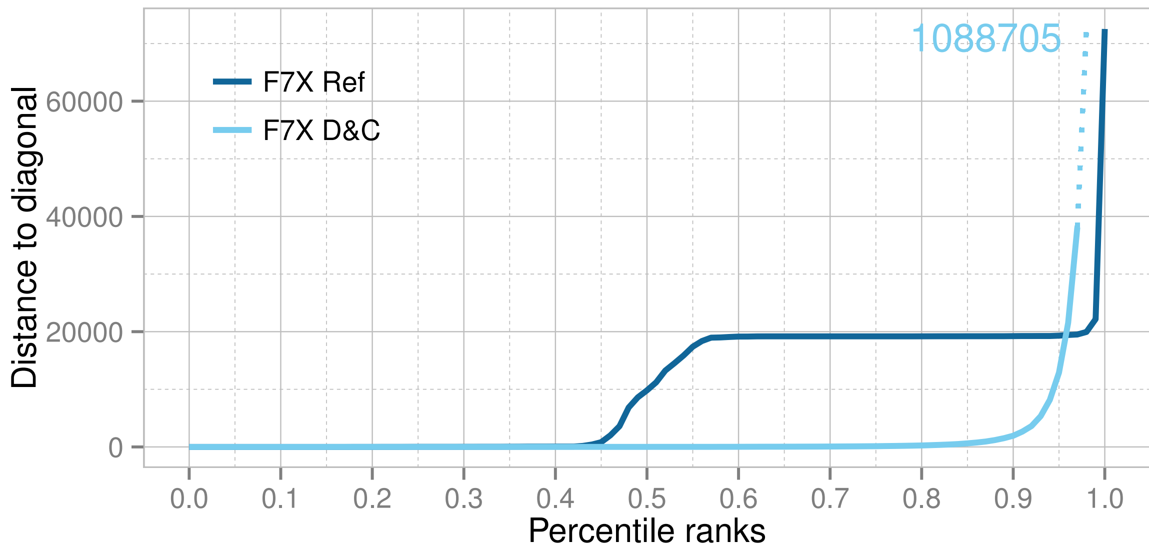



Figure 5.4: Percentiles distribution of the distances to the diagonal with and without the D&C permutations on the F7X use case.

the solver step. We observe an empirical speedup up to 10% depending on the use case. Figure 5.4 highlights the locality improvement of our D&C version on the F7X use case, which originally uses the Cuthill-McKee [159] ordering. It represents the percentiles distribution of the distances to the diagonal. More than 95% of the matrix values are closer to the diagonal in our D&C version than in the reference version. Only the last 5%, mostly representing the sparse separators, are spread far from the diagonal. In comparison, the initial mesh has two parallel lines along the main diagonal which are reflected by a plateau getting further away after the median.

#### L3 Cache Misses Experiment

The locality improvement brought by the D&C library and the bad cache utilization and bandwidth contention of the coloring version are confirmed in Figure 5.5 giving the *L3_miss* hardware counter. We measured on the Anselm cluster, presented in Section 1.6.3, the L3 cache misses of, the pure domain decomposition version *Ref*, the hybrid domain decomposition plus coloring version, called *Coloring*, and our *D&C* version. For the *Coloring* and *D&C* versions, we explored three different configurations of processes and threads. We compared the results running on full processes, full threads, or with a mix of 4 processes and 3 threads per process. As illustrated in Figure 5.5, by running the Mini-FEM proto-application on the EIB use case, all the variants of the *D&C* version have almost 5 times less L3 cache misses than the pure MPI. In the opposite, all the variants of the *Coloring* version have more than 8 times more cache misses than D&C. In the AETHER application running the F7X use case, the impact of the D&C permutations is even more important. The L3 cache misses have been divided by 15 using the D&C library.

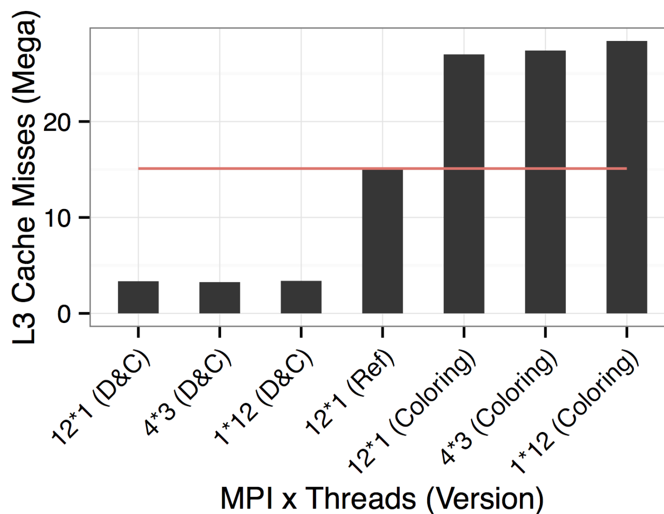



Figure 5.5: Comparison of the L3 cache misses between pure MPI domain decomposition, hybrid MPI + coloring, and hybrid MPI + D&C, using the Mini-FEM proto-application

### 5.2.4 Numerical Stability of the Results

The partitioning of the mesh subdomains using METIS and the resulting D&C permutations slightly impact the numerical accuracy of the initial problem. However, this loss in accuracy also applies to the MPI domain decomposition [126, 127]. Since the D&C recursive bisections and permutations replace the need for further domain decomposition, their impact on the numerical results is equivalent. Moreover, contrary

to MPI domain decomposition, increasing the number of D&C workers does not degrade the numerical stability. The experiments made on the DEFMESH application showed that the global convergence of the iterative algorithm remains unchanged.

### 5.2.5 D&C Matrix Storage Format

As explained in Section 5.2.3, the matrix values are recursively reordered following the left, right, and separator permutation order. To store the matrix nonzero values, we have chosen the CSR matrix storage format presented in Section 3.3. The nonzero values are stored in the CSR according to the D&C leaves accessing them. However, using a separate CSR arrays stored inside each D&C leaf would result in a fragmentation of the CSR matrix. To avoid this memory fragmentation, we contiguously store the nonzero values in a single CSR matrix which is ordered according to the execution of the D&C leaves. The leaves only contain the index intervals of elements, nodes, and edges which are accessed. Therefore, our matrix storage format results in a D&C recursive tree and a CSR matrix. This D&C CSR sparse matrix format is illustrated in Figure 5.6.

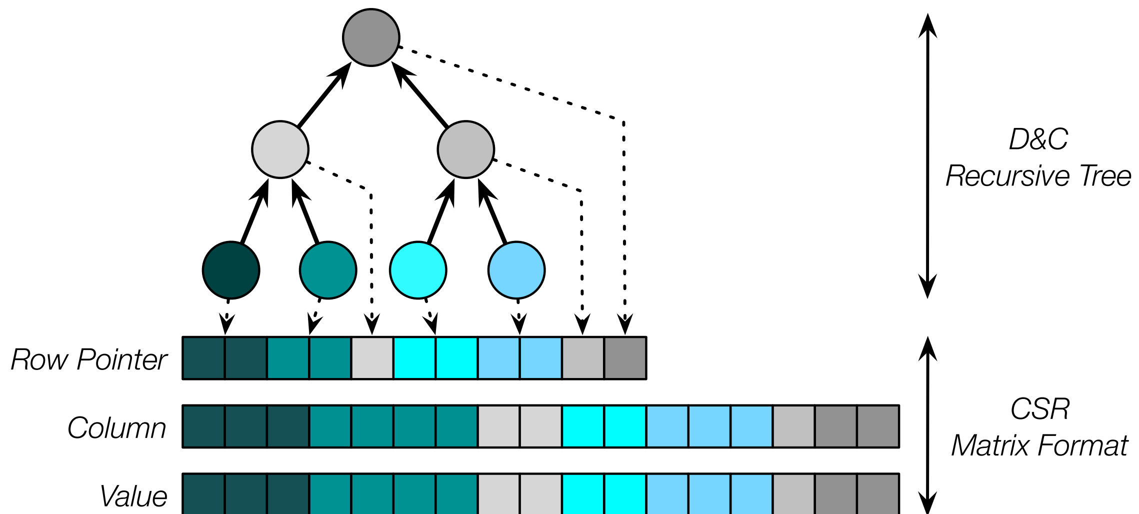



Figure 5.6: D&C CSR sparse matrix storage format. CSR values are contiguously stored following the left, right, and separator partition order.

### 5.2.6 Precomputation of the D&C Tree

The creation of the D&C tree using the METIS graph partitioner and the creation of the permutation tables of elements and nodes are costly. However, since the topology of the mesh is constant, there is no need to recompute them at each iteration. Therefore, as illustrated in Figure 5.7, the creation of the D&C tree and the permutation table can be done only once as a pre-treatment phase.

Since the same use case is used multiple times during the aircraft design process at Dassault Aviation, we offer the possibility to save and reload the permutation table and the recursive tree into a file, to avoid their computation in successive experiments. The only interactions with the original application are the call to the building or loading process, the call to the D&C tree traversal at the assembly step, and the `cilk_for` addition in the preconditioner. The present version of the CG solver used in the DEFMESH

application is an OpenMP version with a low efficiency that will be addressed in future work using our D&C strategy.

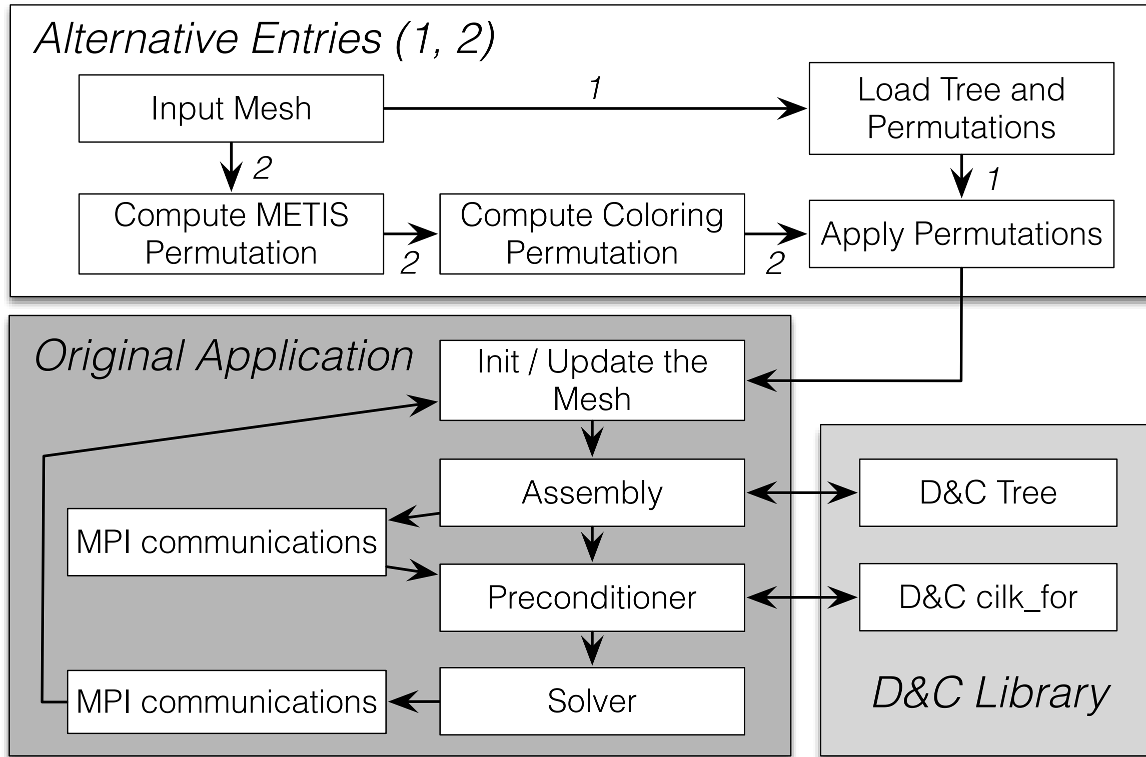



Figure 5.7: Integration of D&C and coloring in the FEM pipeline with constant mesh topology.

### 5.2.7 Cilk Plus Implementation

Cilk Plus [12] is a task-based runtime presented in further details in Section 2.5.2. It allows users to spawn and synchronize efficiently many parallel tasks. All these tasks are organized in queues and executed by *worker* threads. For load balancing, Cilk runtime uses a work-stealing scheduler [85]. When a *worker* completes its queue, it can steal additional tasks from slower *workers*.

Each leaf of the D&C tree is associated with an independent Cilk task which executes the FEM assembly process on its partition. The Cilk Plus implementation of the tree traversal is straightforward and very similar to the pseudo-code given in Algorithm 2. It only uses the two keywords `cilk_spawn` and `cilk_sync`, which are respectively used to spawn a new parallel task and to synchronize all tasks in the current node. Once the D&C recursive tree is created, a Cilk Plus task is executed on each partition. Left and right partitions of each node are split in two parallel tasks, as long as the recursion has not reached a leaf. When the recursion reaches the leaves, the original sequential FEM routine is executed in parallel on each leaf partition. As shown in Algorithm 2, the current worker spawns a new task which will be executed on the right partition and continues its execution on the left partition. Once the left and right tasks are completed and synchronized, the current worker executes the associated separator computation.

The unbalancing between branches of the D&C tree is compensated by the fine grain task-based implementation which produces a very large number of tasks, leveraging the work-stealing of Cilk Plus to offer a near optimal load-balancing. Cilk authors recommend to have a large number of tasks per CPU

**Algorithm 2:** FEM assembly using D&C recursive tree traversal.

```
1  Function Compute (D&C partition)
2  begin
3      if D&C partition ≠ leaf then
4          cilk_spawn Compute (D&C partition → right)
5          Compute (D&C partition → left)
6          cilk_sync
7          Compute (D&C partition → sep)
8      else
9          FEM_assembly (D&C partition)
10     end
11 end
```

thread [12]. On a million nodes domain, we create more than 30,000 parallel tasks to distribute among the 16 cores of a standard Xeon cluster node, or among the 60 cores of a Xeon Phi. When all elements are accessed in a regular loop, there is no need to use a recursive task tree. In this case, similarly to OpenMP, Cilk Plus provides a convenient way to parallelize loops using the `cilk_for` keyword. By substituting the original `for` with the `cilk_for` keyword, iterations of the loop are split recursively into several parallel tasks. For the DEFMESH application, we observe a better scalability with the `cilk_for` than with the OpenMP `parallel for` pragma.

However, Cilk Plus is only supported on x86 architectures and intended to run on Unix-like systems. This is mostly due to the memory model assumptions to handle the dequeue, i.e. spawning, stealing and returning from a spawned function. Nevertheless, it should also run on other systems, provided that GCC, POSIX threads, and GNU autotools are available. In the next section, we propose another implementation based on OpenMP nested sections and tasks. As a future work, we also plan to explore the Intel Threading Building Blocks (TBB) library [14]. The advantage of the TBB library approach is that it can be recompiled by any compiler and adapted to various platforms with minor changes. We have adopted the same approach for DC-lib.

### 5.2.8 OpenMP Implementation

OpenMP is a portable programming interface for shared memory parallel computers supported on many architectures. It provides a platform-independent set of compiler pragmas and directives. The execution model is based on the fork-join principle presented in Section 2.4. OpenMP 2.5 uses thread parallelism through work-sharing. This mechanism of work distribution is enabled by two methods: loop parallelization and parallel sections. In both cases, work units are static and distributed to assigned threads which will execute them from the beginning to the end [79]. As shown in Algorithm 3, OpenMP 2.5 allows recursive parallelism by nested declarations of parallel regions. Unfortunately, this approach leads to overhead due to the cost of creating new parallel regions and to load balancing issues [79].

#### Hierarchical Task-Based Parallelism in OpenMP

OpenMP 3.0 introduces the task proposal to enable dynamic work unit generation [79], irregular parallelism, and recursion. OpenMP 3.0 tasks are introduced in Section 2.5.1. They allow to express the same recursive parallelism replacing the nested parallel regions by tasks spawning other tasks. The *task* construct permits the creation of *explicit tasks*. The tasks are added to a pool of tasks executed by the team of threads from the parallel region. Tasks are guaranteed to be executed at the end of the parallel region,

but it is possible to synchronize them at finer grain using the keyword *taskwait*. The *taskwait* construct suspends the execution of a task until all the children tasks are completed.

In the D&C library, tasks are spawned during the recursive D&C tree traversal. This imposes to authorize nested parallelism through the associated OpenMP system variable. Algorithm 4 shows how the assembly step has been parallelized using OpenMP tasks. Only one parallel region is created and the recursive algorithm spawns the tasks. All new tasks are added to the pool of work and there is only one team of threads.

**Algorithm 3:** Recursion with nested parallel sections.

```
1  Function Recursive_assembly (D&C tree)
2  begin
3  |  #pragma omp parallel section
4  |  begin
5  |  |  #pragma omp section
6  |  |  Recursive_assembly (D&C tree → left)
7  |  |  #pragma omp section
8  |  |  Recursive_assembly (D&C tree → right)
9  |  |  #pragma omp barrier
10 |  |  if D&C tree → sep ≠ null then
11 |  |  |  Recursive_assembly (D&C tree → sep)
12 |  |  end
13 |  end
14 end
```

**Algorithm 4:** Recursion with nested parallel tasks.

```
1  Function Recursive_assembly (D&C tree)
2  begin
3  |  #pragma omp task default (shared)
4  |  Recursive_assembly (D&C tree → left)
5  |  #pragma omp task default (shared)
6  |  Recursive_assembly (D&C tree → right)
7  |  #pragma omp taskwait
8  |  if D&C tree → sep ≠ null then
9  |  |  Recursive_assembly (D&C tree → sep)
10 |  end
11 end
12 Function D&C_assembly ()
13 begin
14 |  #pragma omp parallel
15 |  #pragma omp single nowait
16 |  Recursive_assembly (D&C tree)
17 end
```

## 5.3 Experimental Results

To validate our novel approach, we have applied our D&C library using the Cilk Plus runtime and MPI domain decomposition on the Mini-FEM proto-application presented in the previous chapter. We have also measured the solver step in the DEFMESH application presented in Section 4.2.1 with and without the D&C library to estimate the impact of the D&C data permutations on the rest of the code. Lastly, we have compared the Intel Cilk Plus runtime initially used in our D&C library to the OpenMP 3.0 task-based runtime, on the Mini-FEM proto-application and the AETHER application presented in Section 4.2.2. We perform our experiments on five different platforms.

- The MareNostrum cluster presented in Section 1.6.4.
- The Anselm and Salomon clusters described in Section 1.6.3.
- The KNC nodes of Cirrus presented in Section 1.6.2
- And a cluster of Sandy Bridge from Dassault Aviation.

During these experiments, we compare four different implementations.

- The original pure MPI version using domain decomposition from Dassault Aviation called *Ref (MPI)*.
- The state-of-the-art hybrid version called *Coloring (MPI+Cilk)*, using MPI domain decomposition and mesh coloring at thread level as detailed in Section 3.5.1 This version exploits Cilk Plus to parallelize the loop elements of a same color within each MPI domain.
- Our divide and conquer version, called *D&C (MPI+Cilk)*. This version uses the recursive partitioning of each domain of the mesh. In addition to MPI, it exploits task parallelism using Cilk Plus.
- And lastly, an alternative version of the D&C library using OpenMP 3.0 tasks instead of Cilk Plus. This version is named *D&C (MPI+OpenMP)*.

### 5.3.1 Experimental Setup

The running time of the assembly step is short and slightly fluctuates. As explained in [172], making precise measures is a complicated task. In our case, we run 50 iterations of the algorithm and we repeat the run 4 times. The first iteration is ignored since it is delayed by different initialization processes such as the network card. For each time step, we measure the number of elapsed CPU cycles using the RDTSC counters. The measures correspond to the average time of one iteration. We ensured that the numerical results at each step and the number of solver iterations needed to converge are stable for all versions.

The results are presented both in terms of relative speedup compared to the best sequential time and in terms of parallel efficiency. In the following figures, the $x$ axis represents the number of cores. For speedup experiments, the $y$ axis represents the relative speedup $S$ given by Equation 5.1, where $T_S$ is the sequential time and $T_P$ is the parallel time on $P$ processors.

$$S = \frac{T_S}{T_P} \tag{5.1}$$

Concerning parallel efficiency, the $y$ axis represents the efficiency $E_P$ on $P$ processors, given by Equation 5.2.

$$E_P = \frac{T_S}{P * T_P} \tag{5.2}$$

For each experiment, we vary the number of processes of the reference versions only based on domain decomposition. For the hybrid versions of the code using processes and threads, we use only one process per compute node and we vary the number of Cilk or OpenMP threads up to the node size. According to the Cilk documentation, a minimum of 10 tasks per core is appropriate. In our case, as explained in Section 5.2.2, we downsize the tasks in order that they fit in L1 caches. This results in hundreds of tasks per core depending of the size of the input mesh.

In all the experiments made on the Intel Xeon Phi, the figures are cut in two parts. The left white background part on the left corresponds to the 60 physical cores of the KNC. And the grey background part on the right of the figures corresponds to the 4 hyper-threads available per core, which results in up to 240 threads per KNC.

For each experiment, the OpenMP affinity is set to scatter and the Cilk *worker* threads are not pinned as recommended in the Cilk documentation.

### 5.3.2 FEM Assembly

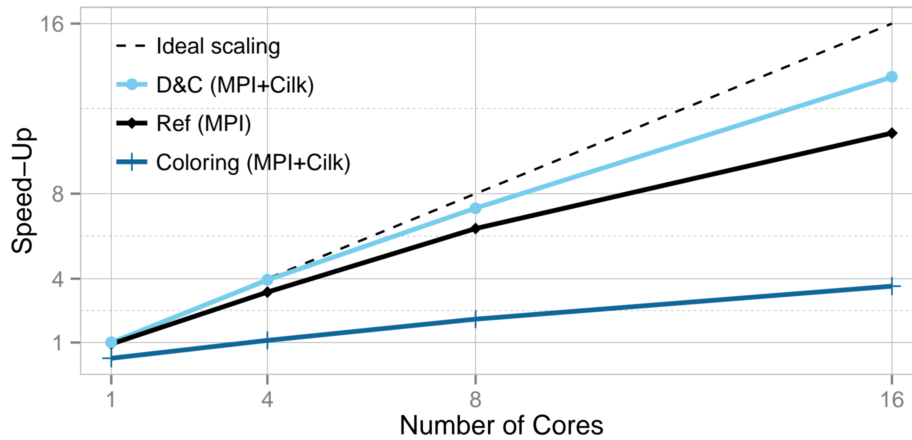
The experiments made in this section are done on the Mini-FEM proto-application running the EIB use case on the MareNostrum cluster. The application is compiled with Intel composers 14.0.2 and Intel MPI 4.1.3. We first compare the state-of-the-art *Ref (MPI)* and *Coloring (MPI+Cilk)* versions to our *D&C (MPI+Cilk)* version on a single node composed of 16 Sandy Bridge cores detailed in Section 1.6.1. Then we extend our comparison to a strong scaling experiment on up to 32 nodes totalizing 512 cores. We also compare these versions on up to 4 Intel Xeon Phi 7120P.

#### Single Node Experiment

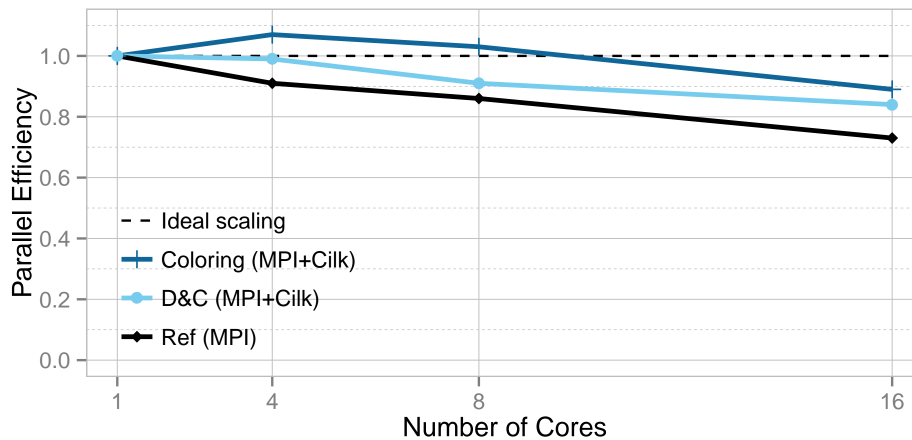
Figure 5.8 presents a speedup and parallel efficiency comparison on a single node of 16 Sandy Bridge cores from MareNostrum. As expected, the *Ref (MPI)* version efficiently scales on a single Sandy Bridge node while the *Coloring (MPI+Cilk)* version has poor performance and ends far behind the other versions. The sequential performance of *Coloring (MPI+Cilk)* is 3.9 times slower than *D&C (MPI+Cilk)*. However, it benefits more from the growing parallel resources than the other versions and has a better parallel efficiency, being only 3.7 times slower than D&C at 16 cores. Indeed, the large amount of accessed data for each color has more chance to fit in cache as the number of cores and caches increases. Moreover, the memory accesses benefit from a larger memory bandwidth. As viewed in Figure 5.8b, this leads the *Coloring (MPI+Cilk)* version to a super-linear speedup using 4 cores.

The *D&C (MPI+Cilk)* and *Ref (MPI)* versions are closer to each other but the D&C version overpasses the reference version with a 1.24× speedup at 16 cores. The 16 subdomains and processes used by the pure MPI version are replaced by a single domain and a single process using D&C. The resulting MPI duplications and communications are replaced by data sharing. Furthermore, unlike the original version which makes irregular accesses to the matrix values, D&C packs the matrix values contiguously inside each Cilk tasks. Lastly, for any problem size, by using *D&C (MPI+Cilk)* we can increase the number of partitions in order that the tasks working-set always fit in cache. This makes D&C a competitive solution not only for future exascale systems, but also for current platforms.

This first experiment illustrates the phenomena explained in the general introduction: improving the code performance reduces its parallel time and therefore, increases its proportion of sequential time while decreasing its parallel efficiency. This emphasizes the fact that scalability does not reflect an application performance, it must be completed by relative speedup or raw performance.




(a) Speedup compared to the best sequential performance




(b) Parallel Efficiency

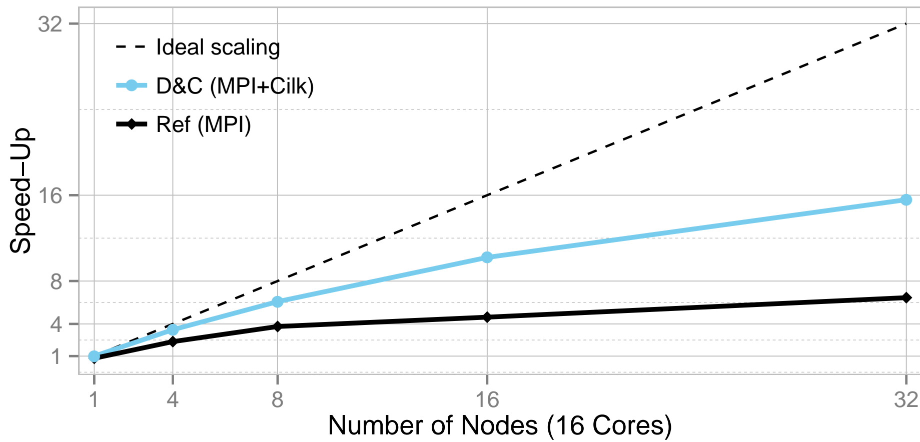
Figure 5.8: FEM assembly speedup and parallel efficiency comparison on the EIB use case running on a single node of the MareNostrum cluster.

### Strong Scaling Experiment

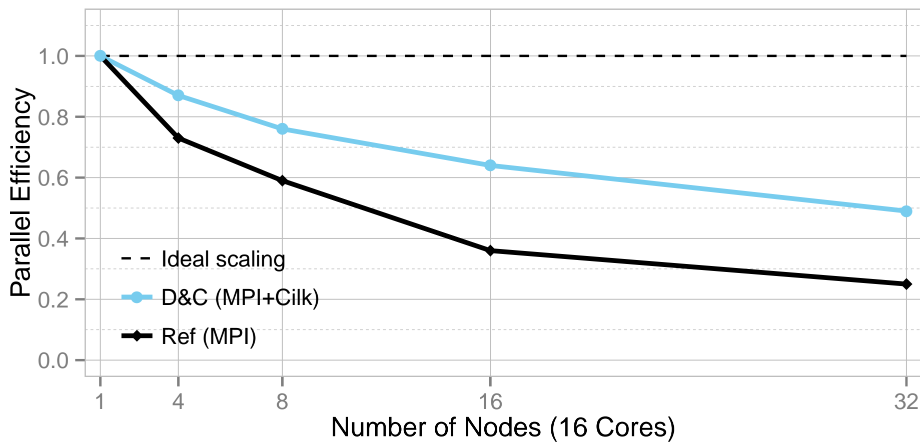
Production runs at the scale of the EIB use case, i.e. around a million vertices, are usually done on 32 cores. In this experiment illustrated in Figure 5.9, we run the *Ref (MPI)* and our *D&C (MPI+Cilk)* versions on a higher number of cores to test the strong scalability in extreme conditions. The objective is to test whether future systems can improve the time to solve problems of current size. We put aside the *Coloring (MPI+Cilk)* version because of its poor performance.

At 512 cores, by using the 1 million vertices EIB use case, there are only 2000 vertices accessed per core. At that scale, the overhead of the runtime has an important impact on performance. Every single serialization in the code represents a higher part of the total execution time as explained by the Amdahl law. Moreover, as shown in Section 3.5, when cut in 512 subdomains, the ratio of duplicated values in the halos of EIB reaches 60%.

These different reasons explain the increasing gap between the *Ref (MPI)* and the *D&C (MPI+Cilk)* version. While the *Ref (MPI)* version ends at only 25% of parallel efficiency, the *D&C (MPI+Cilk)* version




(a) Speedup compared to the best sequential performance



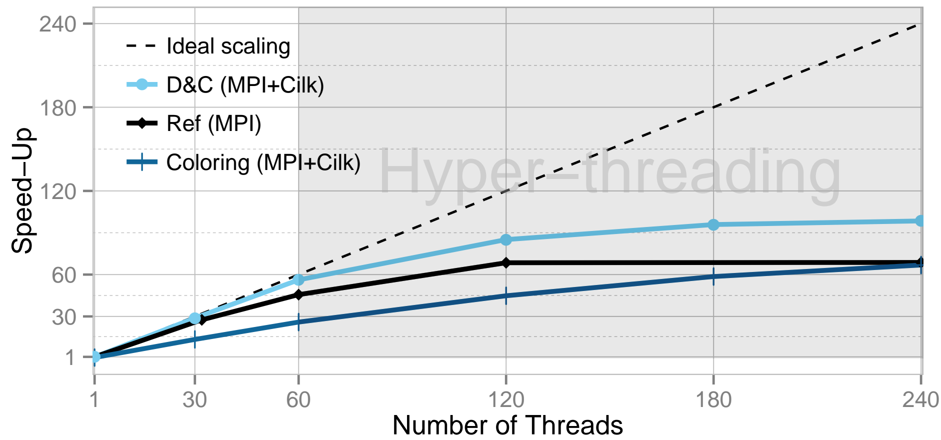

(b) Parallel Efficiency

Figure 5.9: FEM assembly speedup and parallel efficiency comparison on the EIB use case running on up to 32 nodes (512 cores) of the MareNostrum cluster.

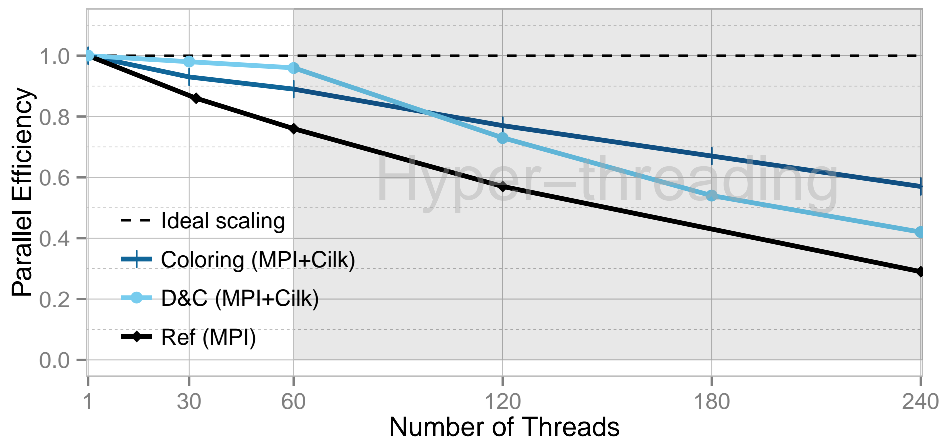
is 2.4 times faster and ends at 50% of parallel efficiency. This is due to the very fine grain parallelism, the lower amount of communications and data duplications, and the improved locality brought by the D&C permutations. However, there is still an important gap compared to the ideal scaling curve which makes room for further optimizations. Even with the lower amount of communications of the *D&C (MPI+Cilk)* version, the remaining ones between the distributed nodes negatively impact the performance. We propose in Chapter 7, a new communication pattern aimed at exploiting the D&C shared parallelism among the communications and enabling communication and computation overlap. Moreover, the vector resources within the compute cores are not exploited in this early version. The next chapter presents our approach to enable vectorization within the small D&C tasks.

**Single KNC Experiment**

The Figure 5.10 presents the performance and the parallel efficiency comparison on an Intel Xeon Phi based on the KNC architecture presented in Section 1.4. Surprisingly, the *Ref (MPI)* version with its




(a) Speedup compared to the best sequential performance




(b) Parallel Efficiency

Figure 5.10: FEM assembly speedup and parallel efficiency comparison on the EIB use case running on a Xeon Phi (KNC) of the Salomon cluster.

message passing model scales pretty well on the 60 physical cores of the shared memory KNC architecture, with 76% of parallel efficiency. But it rapidly stagnates when using the hyper-threads and ends at less than 30% of parallel efficiency. The *Coloring (MPI+Cilk)* version has the lowest sequential and parallel performance. This is mostly due to a bad locality and bandwidth usage.

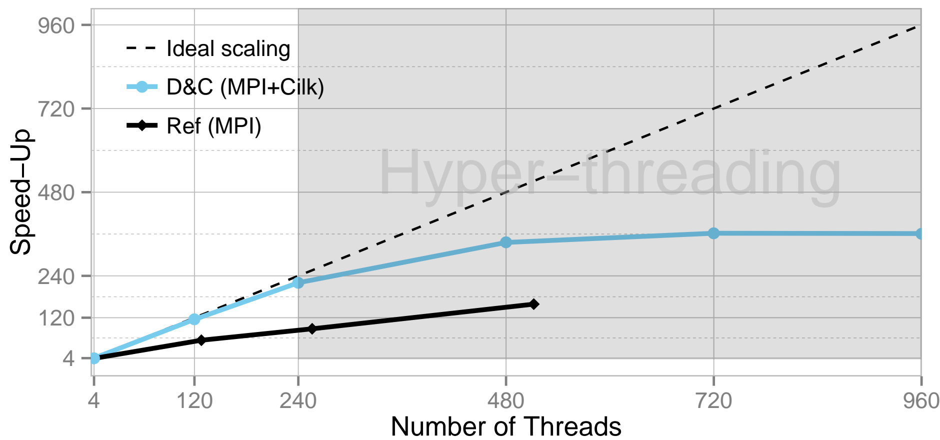
But as stated in the previous single node experiment and in the general introduction, this poor performance leads to a good scalability. By increasing the number of cores, the data per cache and the available bandwidth increase, which benefits to the parallel efficiency metric. Indeed, the coloring version is 2.17 times slower than D&C on the 60 physical cores and only 1.47 times slower when using the 240 hyper-threads. This confirms once again that the scalability and the parallel efficiency do not reflect the performance.

Concerning our *D&C (MPI+Cilk)* approach, it has an impressive 96% parallel efficiency on the 60 physical cores of the KNC and a speedup of 44% over the *Ref (MPI)* version using the KNC hyper-threads. The performance is equivalent to 10 Intel E5-2665 Xeon Sandy Bridge cores.

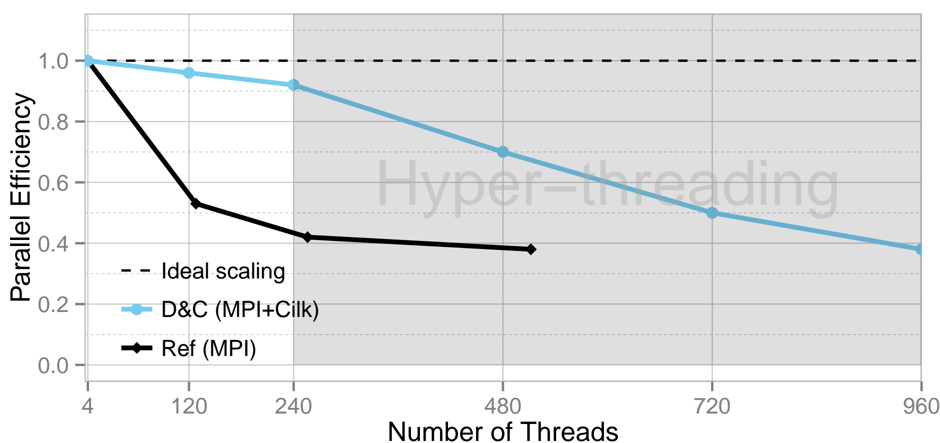
### Multiple KNC Experiment

We extend our Xeon Phi experiment on 4 KNC. Figure 5.11 presents the scalability and parallel efficiency results on the 7 million vertices FGN use case. The experiments done on the *Ref (MPI)* version are limited to 512 cores. This limitation corresponds to the highest domain decomposition provided by Dassault Aviation. In the pure MPI approach, the high numbers of communications and data duplications inside and between the 4 KNC degrade the performance. By increasing the number of processes per KNC, the parallel efficiency stabilizes since the MPI communications are recovered by the hyper-threads. In contrast, our D&C approach only uses 1 MPI process per KNC. Therefore, the number of communications inside and between KNC is drastically reduced. As a result, D&C perfectly scales with 92% parallel efficiency on the 240 physical cores of the 4 KNC and is 2.5 times faster than *Ref (MPI)*. D&C achieves a final speedup of 360× compared to its sequential execution and obtains similar performance to 33 Intel E5-2665 Xeon Sandy Bridge cores.

To conclude, focusing on high concurrency and good locality produces a code which works out of




(a) Speedup compared to the best sequential performance




(b) Parallel Efficiency

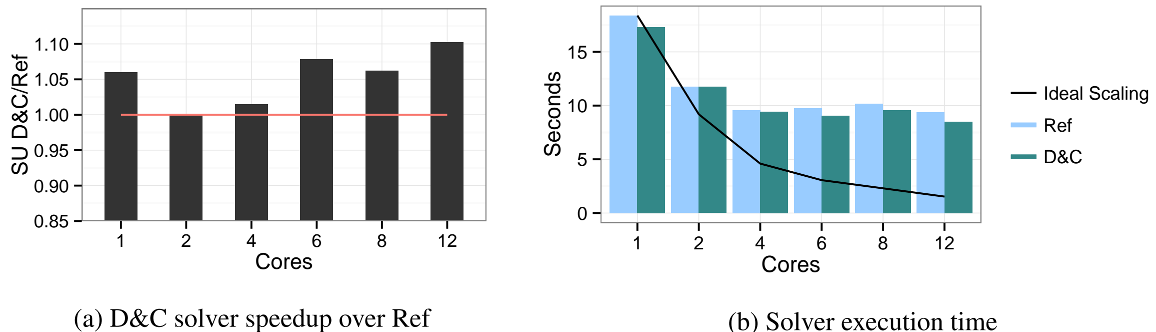
Figure 5.11: FEM assembly speedup and parallel efficiency comparison on the FGN use case running on 4 Xeon Phi (KNC) of the Salomon cluster.

the box when porting from conventional multicore CPUs to manycores. However, when comparing to different generations of Xeon, we notice that a single KNC has equivalent performance to 25 Westmere cores, to 16 Sandy Bridge cores, or to 10 current generation Xeon E5 cores fitting in a single socket. This can be explained by the KNC age. Current Xeon Phi roadmap is slower than standard Xeon and is de facto becoming an experimentation platform to test manycore programming approach. The comparison should be done again with the next generation of Xeon Phi, i.e. the KNL architecture.

### 5.3.3 Solver

In this section, we measure the performance of the solver step with and without the permutations of our D&C library. Although we did not modify the solver part, we measure its execution time to estimate the impact of the locality improvements brought by the D&C library. This experiment was made on the DEFMESH application running the EIB use case. We observe in Figure 5.12 an average 6% speedup for *D&C (MPI+Cilk)* compared to *Ref (MPI)* with a maximum speedup of 10.2% on twelve cores. This improvement is only due to the better data locality enabled by the permutations explained in Section 5.2.3. The L3 cache misses occurring in the solver are 23% lower with *D&C (MPI+Cilk)* than with *Ref (MPI)*.

Additionally, to show the impact of locality, we randomly permute the data in the *Ref (MPI)* version of the code. We observe a degradation of 30% in performance between this *RandomRef (MPI)* version and *D&C (MPI+Cilk)* and 18% between *Random Ref (MPI)* and *Ref (MPI)*. These results highlight the strong impact of the locality on the solver performance.




(a) D&C solver speedup over Ref

(b) Solver execution time

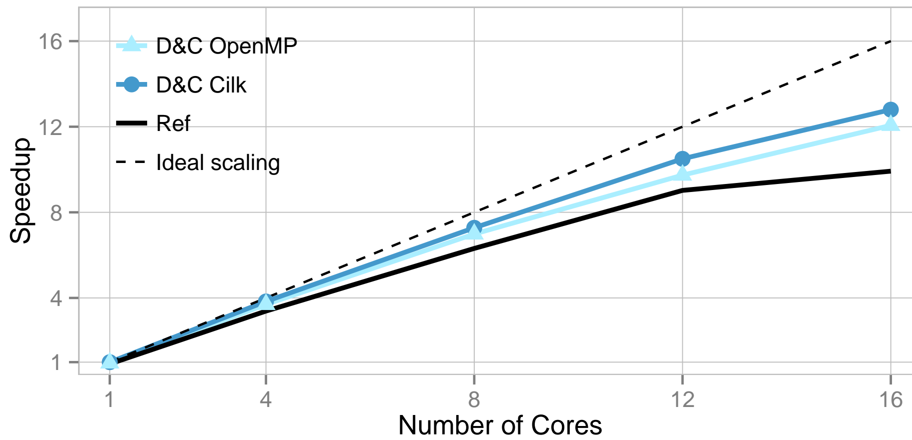
Figure 5.12: Solver execution time and speedup using D&C permutations.

### 5.3.4 Comparison Between Cilk Plus and OpenMP 3.0 Tasks

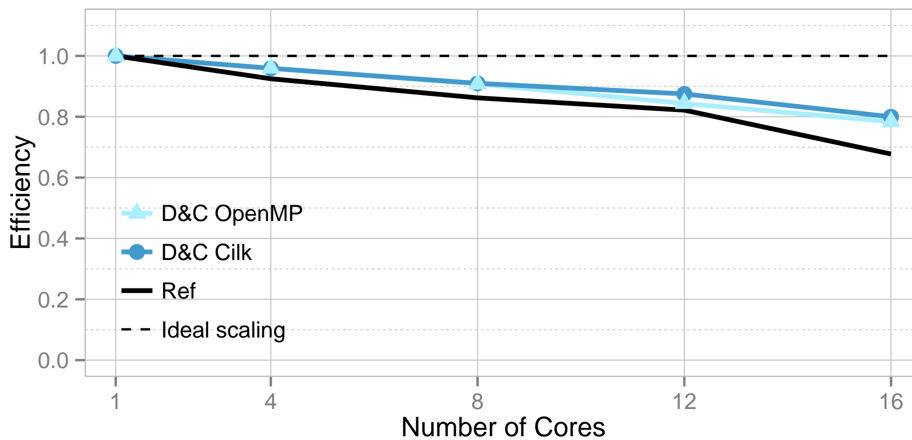
In the following experiments, we compare the performance of the Intel Cilk Plus runtime to the OpenMP 3.0 runtime based on task parallelism. We have integrated these two versions in our D&C library. The experiments are made on a single Sandy Bridge node and on 4 Intel Xeon Phi based on the KNC architecture using the Mini-FEM proto-application. And on up to 1024 Sandy Bridge cores using AETHER, the industrial CFD code from Dassault Aviation presented in Section 4.2.2.

#### Sandy Bridge Experiment on the Mini-FEM Proto-Application

At first, we experiment the Mini-FEM proto-application running the EIB use case on the Anselm cluster. The Figure 5.13 shows the comparison between the OpenMP and Cilk versions of the D&C library on a single compute node. These two versions are also compared to the *Ref (MPI)* version. The Cilk Plus




(a) Speedup compared to the best sequential performance




(b) Parallel Efficiency

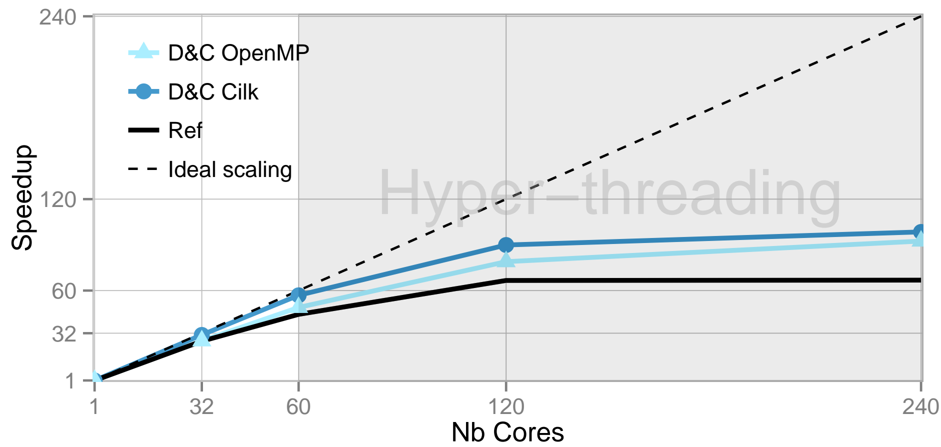
Figure 5.13: FEM assembly speedup and parallel efficiency comparison between Cilk Plus and OpenMP 3.0 tasks. Mini-FEM running the EIB use case on Anselm cluster.

version shows the best performance, both in terms of speedup and efficiency. However, there is only 6% improvement over OpenMP tasks on 16 cores.

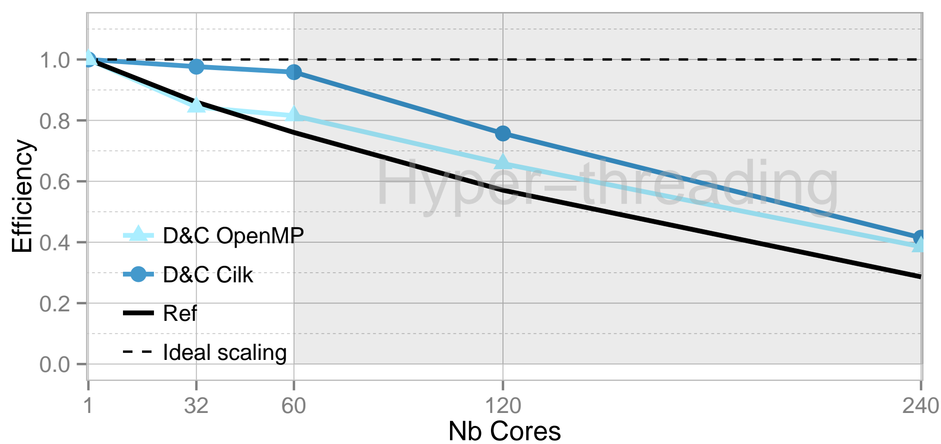
It is also interesting to note that the *D&C (MPI+Cilk)* version using Cilk Plus on the Anselm cluster, similarly behaves on the larger memory nodes of the MareNostrum cluster used in previous experiments. However, this larger memory benefits to the *Ref (MPI)* version which scales slightly better to 16 cores on MareNostrum than on Anselm.

### KNC Experiment on the Mini-FEM Proto-Application

Then, we compare the OpenMP 3.0 tasks to the Cilk Plus runtime on a single Intel Xeon Phi using KNC architecture. This is illustrated in Figure 5.14. By using the 60 physical cores, Cilk Plus scales almost ideally with 97% efficiency. The OpenMP 3.0 version suffers from the runtime overhead and achieves 81% parallel efficiency. At 60 cores, Cilk Plus is 16.3% faster than OpenMP tasks. This falls to 6.5% by using the hyper-threads. Indeed, since the performance of the OpenMP version is not optimal, hyper-threads




(a) Speedup compared to the best sequential performance




(b) Parallel Efficiency

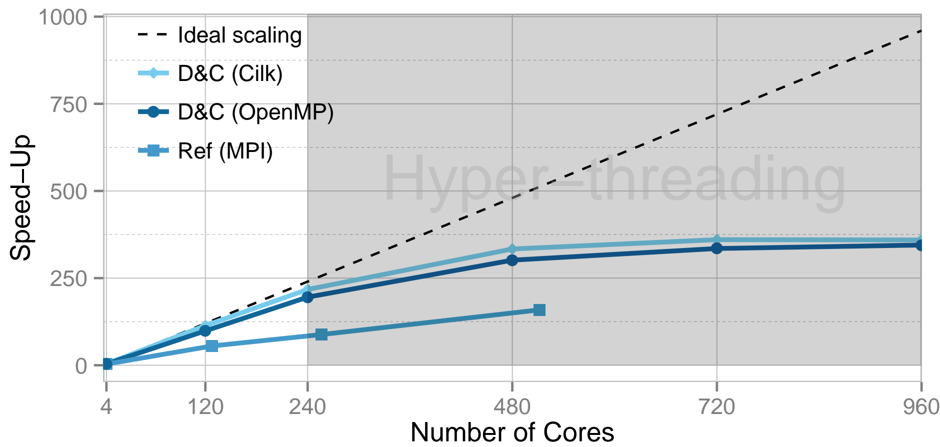
Figure 5.14: FEM assembly speedup and parallel efficiency comparison between Cilk Plus and OpenMP 3.0 tasks. Mini-FEM running the EIB use case on an Intel Xeon Phi KNC.

have more improvement opportunities to hide the latencies and overheads.

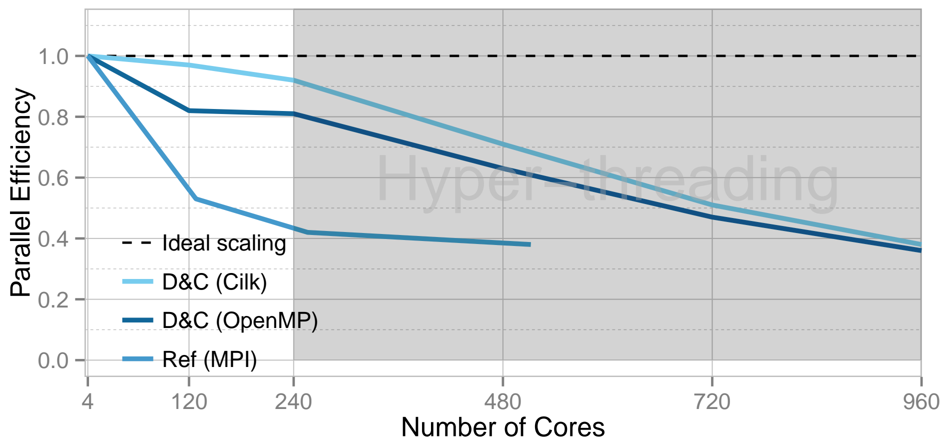
In Figure 5.15, we compare this two runtimes in a weak scalability experiment by using up to 960 threads distributed over 4 Intel Xeon Phi. In this experiment, we use the larger 7 million nodes FGN use case. On the physical cores, we obtain similar results than on a single Phi. But once again, the hyper-threads allows to hide the OpenMP runtime overhead. Its final performance is close to Cilk with only 4.2% less.

### Strong Scaling Experiment on the AETHER Application

To validate the results obtained with our Mini-FEM proto-application developed from the DEFMESH application, we apply the same experiment on the AETHER application. AETHER is compiled with Intel 11 composers, while the D&C library is compiled with Intel 13 due to the Cilk Plus requirements. The experiments are performed with the F7X use case on a Sandy Bridge cluster from Dassault Aviation with similar nodes as Anselm. We fully use the 16 cores of a node and increase the nodes count from




(a) Speedup compared to the best sequential performance




(b) Parallel Efficiency

Figure 5.15: FEM assembly speedup and parallel efficiency comparison between Cilk Plus and OpenMP 3.0 tasks. Mini-FEM running the FGN use case on 4 Intel Xeon Phi KNC.

1 to 64, totalizing a maximum of 1024 cores. To measure and compare the Cilk Plus and OpenMP runtime overhead, we measure the computation time without any MPI communication in a strong scaling experiment. Indeed, since the mesh contains around 6 million elements and is partitioned until all leaves contain 200 elements, we obtain 30,000 tasks spread over 1024 cores. This results in about 30 tasks per core.

Figure 5.16 shows a speedup comparison between *Ref (Comm Free)*, and the two *D&C (MPI+Cilk)* and *D&C (MPI+OpenMP)* versions. At 1024 cores, Cilk speedup overpasses OpenMP by 9%. The Cilk Plus implementation of task parallelism benefits from lighter overhead and dynamic load balancing via work-stealing [11, 12]. As a result, Cilk Plus shows a better scalability than OpenMP, which is consistent with the results previously obtained on the proto-application.

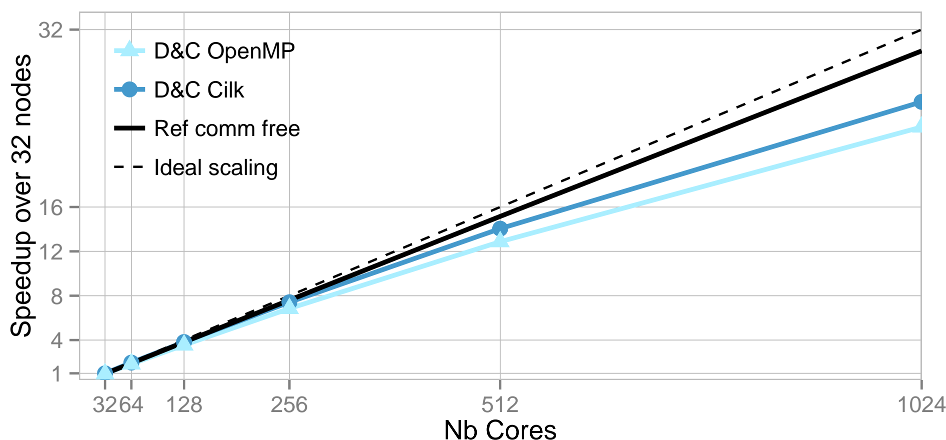



Figure 5.16: Performance comparison between *Ref (Comm Free)*, *D&C (MPI+Cilk)*, and *D&C (MPI+OpenMP)* on AETHER with F7X mesh on 1024 Sandy Bridge cores.

## 5.4 Conclusion

In this chapter, we have proposed and evaluated a new hybrid approach based on MPI domain decomposition and divide & conquer to efficiently parallelize unstructured mesh applications. In this approach, we recursively partition each mesh subdomain in two partitions. For each cut, the partitions are split in a left and a right partition. The values at the frontier between left and right partitions form the separator partitions. These separators are executed after the completion of their corresponding left and right partitions. The recursive bisection continues until reaching the desired partition size which typically corresponds to the L1 cache size.

At each recursion level, the data are permuted to be contiguously stored inside each partition. This enables a good intra-task data locality. Moreover, the contiguous data partitions are stored according to their execution order. The left and right partitions are stored first, followed by their associated separator. This improves the inter-tasks locality.

The contribution presented in this chapter provides efficient utilization of modern shared memory resources. The hybrid parallelization allows to reduce the synchronization needs and the data duplications induced by the classical pure MPI approach. The increasing core count within compute nodes is filled by the large amount of D&C parallel tasks. Lastly the D&C data permutations improve the locality and enable an efficient use of the small cache memories, especially concerning the recent Xeon Phi manycore architecture.

Our results have been validated on the Mini-FEM proto-application. Our D&C library overpasses in performance and scalability the pure MPI domain decomposition and the hybrid approach using mesh coloring on two different Sandy Bridge clusters and on up to 4 Xeon Phi. Even without exploiting thread level parallelism, the permutations brought by the D&C library significantly improve the locality, the scalability, and the execution time. Two versions of the library are available using either the Intel Cilk Plus runtime or the OpenMP 3.0 tasks. The Cilk Plus implementation is significantly faster than OpenMP at scale.

# CHAPTER 6

# CORE LEVEL VECTORIZATION

**Contents**


**6.1 Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **89**

**6.2 Coloring for Efficient Vectorization** . . . . . . . . . . . . . . . . . . . . . . . . . . **90**

6.2.1 Implementing the Vectorization . . . . . . . . . . . . . . . . . . . . . . . 90

6.2.2 Color-Based Vectorization Model . . . . . . . . . . . . . . . . . . . . . . 91

6.2.3 Longest Colors Strategy . . . . . . . . . . . . . . . . . . . . . . . . . . . 92

6.2.4 Bounded Colors Strategy . . . . . . . . . . . . . . . . . . . . . . . . . . . 93

6.2.5 Reduction of Memory Consumption and Synchronization Needs . . . . . 95

6.2.6 Structuredness, Vectorization and Locality . . . . . . . . . . . . . . . . . 96

6.2.7 Vector Length Sensitivity Study . . . . . . . . . . . . . . . . . . . . . . . 97

**6.3 Experimental Results** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **98**

6.3.1 Single Node Experiment . . . . . . . . . . . . . . . . . . . . . . . . . . . 99

6.3.2 Strong Scaling Experiment with Increased Arithmetic Intensity . . . . . . 100

6.3.3 KNC Experiments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101

**6.4 Conclusion** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **104**


## 6.1 Introduction

In 1986, at the crossing between vector machines and multiprocessor supercomputers, Padua *et al.* proposed different code transformations which can be used to optimize compilers auto-vectorization and later to detect parallel constructions [152]. Recently, they evaluate the evolution of auto-vectorization in modern compilers [153]. Although many progresses have been made during these 40 years, the vectorization ratio of compilers on real application loops is still very low. Yet, we have seen in Chapter 1 that the vectorization units integrated within the cores of modern multicores and manycores are getting larger. This makes the vectorization critical to get close to the peak FLOP performance of a system.

In the same time, the cache memories available in these cores tend to decrease. And we have also seen in the previous chapter, the importance of the locality and the positive impact of data blocking. In this context, it becomes challenging to enable a good vectorization ratio while preserving locality. The creation of many data vectors while working on small datasets is particularly difficult, especially when dealing with irregular application working on unstructured meshes.

In this chapter, we extend the D&C approach presented in previous chapter to efficiently handle vectorization of unstructured mesh applications on modern architectures. As explained in Section 3.5.1, the coloring approach was originally designed for vector machine. Considering current CPU cores as vector machines, coloring can be viewed as a good strategy for vectorization. We observe that current coloring strategies [127, 17, 129] are not efficient on the very small data partition size of the fine grain task-based parallelism. In Section 6.2.4, we propose a new coloring heuristic, named *bounded colors*, to reveal data-parallelism in small partitions. The selected runtime, Cilk Plus, provides a useful array notation to implement the vectorization at core level. The effective implementation of the vectorization using coloring is described in Section 6.2.1.

Lastly, we propose a vectorization ratio prediction model as a function of the vector length and the data partition size. We apply it to evaluate the state-of-the-art *longest colors* and our original *bounded colors* strategies described in Sections 6.2.3 and 6.2.4. It reveals the critical influence of the coloring strategy and the new arising trade-off on manycores between structuredness, memory locality, and vector length in upcoming manycore systems. We show in Section 6.3.3 that a better speedup can be achieved by limiting the vector length. The overall objective is to produce an application which makes no compromise on current performing solutions and which is able to scale with current trends in system design.

## 6.2 Coloring for Efficient Vectorization

In previous chapter, we have built a D&C recursive tree containing a large amount of parallel tasks shared among the compute cores. To build vector parallelism and exploit the vectorial units integrated within these cores, the idea is to use mesh coloring inside each partition of the D&C tree. While the coloring strategy is not efficient at system and node level for both data locality and synchronization, there are no such issues at core level. Indeed, as explained in Section 5.2.2, we choose the locality parameter for the D&C partitions to be the L1 cache size, in order to benefit from the highest core level bandwidth. Moreover, synchronizations between colors are local to each partition and therefore affect a single core. By applying mesh coloring on each of these partitions, we build independent and contiguous vectors that all fit in cache.

To reach optimal performances, the vectors have to be large enough to fill the vector instructions. Many coloring technics exist for large domains [129], but an extensive study of coloring for small partitions remains unexplored. However, our current coloring strategy provides sufficient improvement with current cache sizes and vector lengths. We tuned it with a vector length aware strategy presented in Section 6.2.4.

### 6.2.1 Implementing the Vectorization

Exposing data parallelism on unstructured meshes is a challenging task to solve in order to take the best advantage of the increasing vector size of the architectures. To make the approach practical for the developers, we use the Cilk Plus array notation [89] as an entry level. We explore good code constructs and data structures for efficient and portable vectorization. A simple example of the Cilk Plus array notation usage is given in Figure 6.1. Additional examples can be found in [89].

Contrary to the array notation in Fortran, which is equivalent to loop over the index in sequence, the Cilk Plus array notation has a data parallel meaning. In the first loop of the pseudo-code given in Figure 6.1b, the vectors `a` and `b` are summed into `a`. This supposes that vectors `a` and `b` do not alias. As shown in the example of Figure 6.1, indirections and conditional branches are perfectly taken into account. From our experiments, as we checked the generated assembly code, this array notation is efficiently transformed in vectorial loops by the compiler and makes the usage of intrinsics in the source code obsolete. We also find that for unstructured meshes, the Cilk Plus array notation can be used to write

```
for (i = 0; i < ARRAY_SIZE; i++) {
    if (a[i] == LAMBDA)  a[i] += b[i];
    else                 a[i]  = b[c[i]];
}
```

(a) Original C code

```
if (a[:] == LAMBDA)  a[:] += b[:];
else                 a[:]  = b[c[:]];
```

(b) Array notation code

```
cond[:] = (a[:] == LAMBDA);
a[:] = cond[:] ? a[:]+b[:] : b[c[:]];
```

(c) Predicated array notation code

Figure 6.1: Cilk Plus array notation examples.

predicated vectorial code as shown in Figure 6.1c. The generated loops are perfectly fused by the compiler to handle the iteration in SIMD by executing divergent branches in a single instruction flow. It results in an execution model close to the Nvidia GPUs [173]. Coupled with the IMCI mask system of the Intel Xeon Phi, we believe that generalizing such a programing model can mitigate the increasing vector length constraint on irregular codes, as demonstrated in the experiments of Section 6.3.3.

### 6.2.2 Color-Based Vectorization Model

From the coloring description, it is possible to build a model to predict the code vectorization ratio. This model is described as a function of the vector length, *vecSize*, and the colors cardinality, *colorCard*. Let us consider a color $i$ of size $colorCard_i$. The vectorized, $vecCard_i$, and not vectorized, $noVecCard_i$, elements cardinality are given by the following equation:

$$\begin{aligned} vecCard_i &= colorCard_i - colorCard_i \% vecSize \\ noVecCard_i &= colorCard_i \% vecSize \end{aligned} \tag{6.1}$$

The global vectorization ratio is given by the Equation 6.2, where *nbElem* is the total number of elements, *nbColors* is the number of colors used, and $colorVecRatio_i$ corresponds to the vectorization ratio of the $i$-th color.

$$globalVecRatio = \frac{1}{nbElem} * \sum_{i=1}^{nbColors} colorCard_i * colorVecRatio_i \tag{6.2}$$

To compute $colorVecRatio_i$, we consider two distinct models: the first one only deals with full vectors, while the second one supports the masked vector execution, such as the one used in the Xeon Phi. The masks allow to compute on incomplete vectors and are an efficient and generic alternative to padding technics. By using this model, we can vary the vector size even on non available lengths to evaluate their impact.

**Model 1.** The vectorization ratio local to a color is simply given by:

$$colorVecRatio_i = \frac{vecCard_i}{colorCard_i} \tag{6.3}$$

The global vectorization ratio given by Equation 6.2 can then be simplified in:

$$globalVecRatio = \frac{1}{nbElem} * \sum_{i=1}^{nbColors} vecCard_i \tag{6.4}$$

**Model 2.** Using mask instruction does not change the full vectors ratio but it allows to take into account incomplete vectors. Since $colorCard_i \% vecSize < vecSize$, there is a single additional incomplete vector per color containing $noVecCard_i$ elements. Its completeness ratio is given by $noVecCard_i/vecSize$. The vectorization ratio local to a color becomes:

$$colorVecRatio_i = \frac{vecCard_i}{colorCard_i} + \frac{noVecCard_i}{colorCard_i} * \frac{noVecCard_i}{vecSize} \tag{6.5}$$

In the next sections, we use our models to evaluate the two proposed coloring strategies.

### 6.2.3 Longest Colors Strategy

The rational of the longest colors strategy is to preferentially use the colors already allocated. The common way to do this is to look at the color of all the neighbor elements and pick-up the first available one [129]. Therefore, a new color is created only if all the other allocated colors are used by the neighbors. The algorithm of the longest colors strategy is given in Algorithm 5. The variable `mask` is a bit field where each bit represents a color. The `elemToColor[nbElem]` array contains the bit value of the color allocated to each element. In our use case with the optimal leaf size, the maximum number of colors required is lower than 128 and therefore, a simple `uint128_t` integer can be used as a bit field. However, having multiple color sets or a larger bit field is trivial to implement.

The resulting element per color distribution is shown in Figure 6.2a. Each histogram represents the distribution of the population per color for all leaves. The $x$ axis represents the colors size and the $y$ axis represents the number of elements which are part of a color of size $x$. The plots sharing a same row have a fixed partition size and an increasing vector length. The plots sharing a same column have a fixed vector length and an increasing partition size. We compare three vector lengths: the SSE instructions containing 2 elements (e.g. Westmere), the AVX instructions containing 4 elements (e.g. Sandy Bridge), and the AVX512 instructions containing 8 elements (e.g. Xeon Phi). The maximum size of the D&C partitions varies between 50, 200, and 500 elements. The lower part of the bars, in light color, represents the fully

**Algorithm 5:** Longest colors strategy pseudo-code.

```
1  foreach element do
2  |  myColor ← 0
3  |  mask ← 1
4  |  foreach neighbor element NE do
5  |  |  neighborColor |= elemToColor[NE]
6  |  end
7  |  while neighborColor & mask do
8  |  |  neighborColor ← neighborColor >> 1
9  |  |  myColor++
10 |  end
11 |  elemToColor[E] ← (mask << myColor)
12 end
```

vectorized elements and the upper part, in dark color, represents the scalar elements. The repartition is computed by using the first vectorization model and has been empirically checked against the loop trip counters. For instance, with a color of 5 elements and SSE vector instructions of length 2, $4/5$ of the elements will be vectorized and $1/5$ will not. In our use case, the maximum leaf size to fit in L1 is 200 elements.

With SSE instructions, the vectorization ratio for very small leaves of 50 elements, is only 70% but it grows to more than 90% when using bigger leaf size. When the vector length increases, the partition size must increase too, to keep a high vectorization ratio. Considering the AVX512 vector length, we must use at least 500 elements per partition to have a reasonably good vectorization ratio of 66.3%. However, since the L1 cache of 32 KB can only store 200 elements, the vectorization ratio is limited to 43.8%.

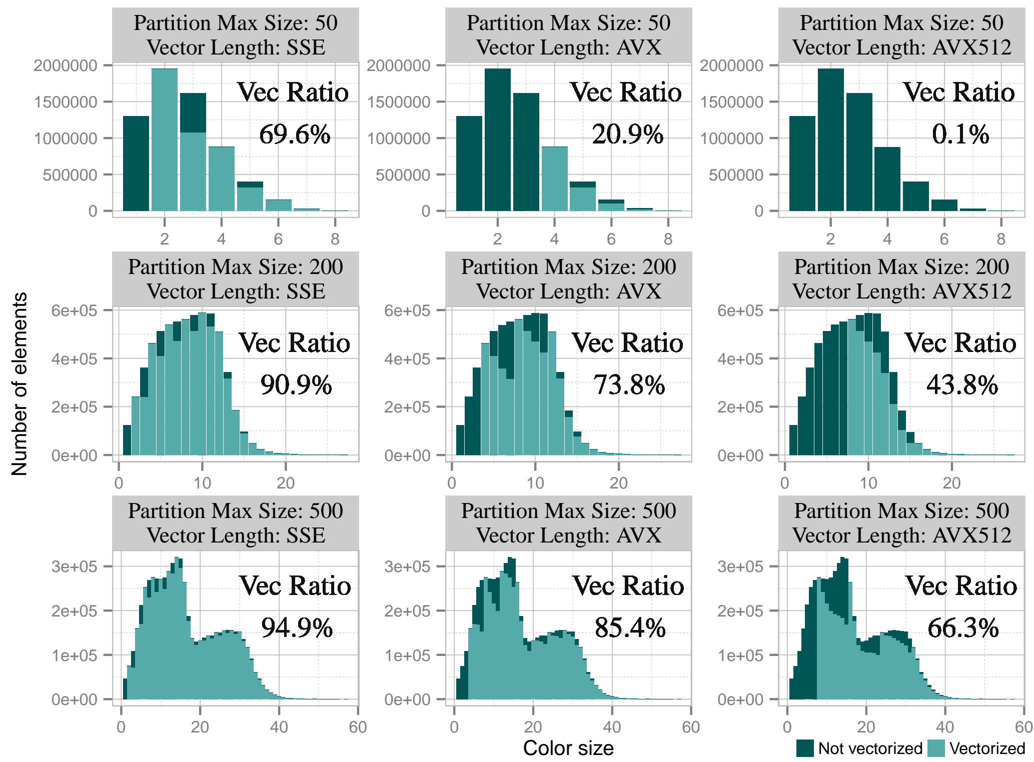
The vector length effect could be mitigated using padding but this requires either profound changes to the global element to node structure, or a local padded copy of the elements resulting in memory and performance overhead. We will not study this solution for two reasons. Firstly, we do not want to make profound architecture-dependent changes to the data and code structure. Secondly, it is not influencing the theoretical conclusion since it just shifts the problem to larger vector sizes. However, to avoid the intrusive padding, it is possible to influence the vectorization ratio by only changing the coloring strategy. Our intuition is that with different choices, the incomplete vectors could be recombined with other elements to form vectors. The next section presents our original coloring strategy for small partitions optimized for a given vector size.

### 6.2.4 Bounded Colors Strategy

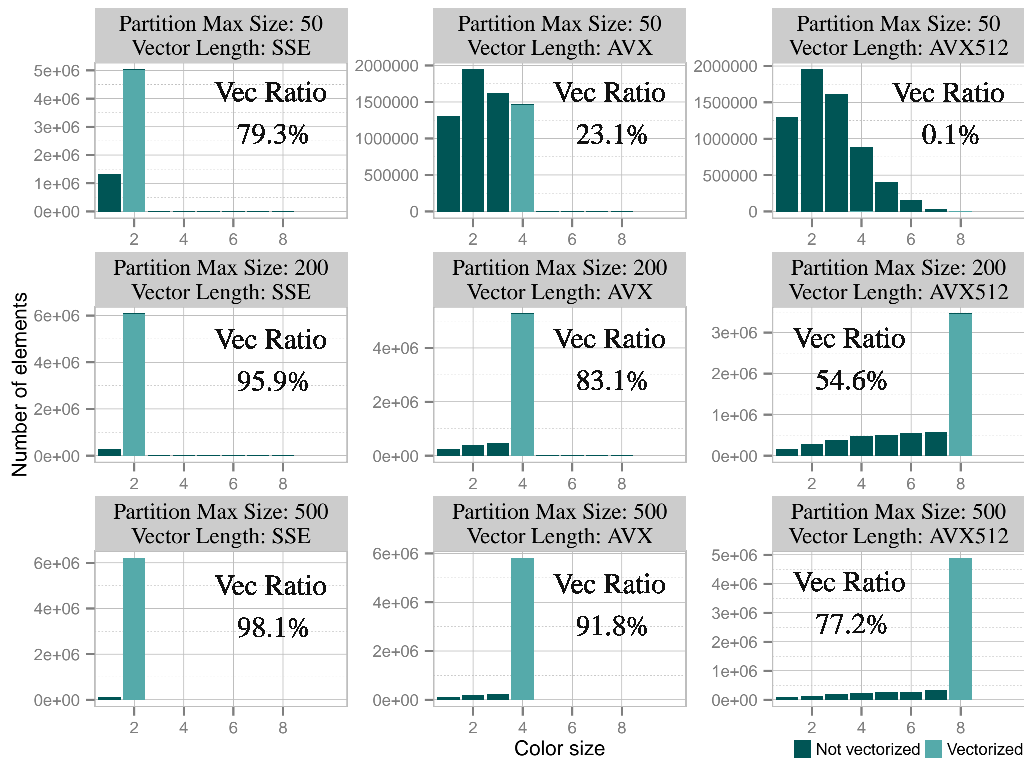
The rational of the bounded colors strategy is to relax the constraint on the number of colors used to fill in priority all the colors up to the vector size. The major drawback of this approach is the increasing number of colors required. However, as explained in the next section, we do not need to store the color of each element and we do not loop and synchronize for each color. Therefore, the number of colors is not a critical factor for our implementation of the vectorization, contrary to standard usage of coloring [129]. Furthermore, in our case, it is still possible to keep the number of colors to 128 when using leaf sizes that fit in L1 caches. The modifications compared to the longest color strategy are minor and detailed in Algorithm 6. They correspond to the additional storage of the color cardinality in `colorCard[nbColors]` and to the modification of the color selection of an element. The new version uses the first available color

**Algorithm 6:** Bounded colors strategy pseudo-code.

```
1  foreach element do
2      myColor ← 0
3      mask ← 1
4      foreach neighbor element NE do
5          neighborColor |= elemToColor[NE]
6      end
7      while neighborColor & mask or colorCard[myColor] >= vectorSize do
8          neighborColor ← neighborColor >> 1
9          myColor++
10     end
11     elemToColor[E] ← (mask << myColor)
12     colorCard[myColor]++
13 end
```




(a) Longest colors strategy




(b) Bounded colors strategy

Figure 6.2: Vectorization ratio for various leaf and vector sizes.

which is not used by a neighbor and with a cardinal lower than the vector length. In order to explore new vector sizes, it is possible to change the `vecSize` parameter to produce the associated distribution and then apply the model described in Section 6.2.2. Such projections are given in Table 6.1 of Section 6.3.3.

The results of our bounded colors strategy are given in Figure 6.2b. On very small leaves size, the median number of elements neighbors is close to the leaves size. Therefore, there is not much data parallelism available and our optimization has a limited influence. For bigger leaves, we benefit from a significant improvement for all vector sizes. The vectorization ratio for a 200 leaf size increases from 73% to 83% for AVX and from 43% to 54% for AVX512.

### 6.2.5 Reduction of Memory Consumption and Synchronization Needs

We have seen in Section 5.2.6 that the coloring is done only once at the beginning of the program, outside the iterative process of the finite element method. Therefore, the cost of the coloring is negligible. However, storing the color of each element is expensive. Furthermore, as shown in Algorithm 7, coloring introduces an extra loop level to sequentially iterate over the colors. For each color, a tail loop handles the elements which are not part of a full vector.

**Algorithm 7:** Loops on elements using longest colors strategy.

```
1 foreach color ∈ leaf do
2     foreach element ∈ [0 : color_size % vector_size[ do vectorially
3         ...
4     end
5     foreach element ∈ [color_size % vector_size : color_size] do sequentially
6         ...
7     end
8 end
```

**Algorithm 8:** Loops on elements using bounded colors strategy.

```
1 foreach element ∈ [0 : offset[ do vectorially
2     ...
3 end
4 foreach element ∈ [offset : leaf_size] do sequentially
5     ...
6 end
```

We solve this issue by using a well chosen permutation of the element array shown in Figure 6.3. Full vectors are stored firstly and the remaining elements are stored afterward. Since the vectors are computed by a single core, the targeted construction is a sequential loop of vectors. By aligning the inter-vectors dependencies and the frontiers of the iterations, there is no need to store the color of the vectorized elements. Ideally, the remaining elements represent a small fraction that will be computed sequentially. Therefore, as shown in Algorithm 8, we can do a loop over vectors of elements on the first part, and the second part can safely be executed as a sequential scalar loop. Despite producing a code aware of the vector length, the interesting side effect is that we only need to store the offset of the end of the vectorial loop for each leaf of the D&C tree.

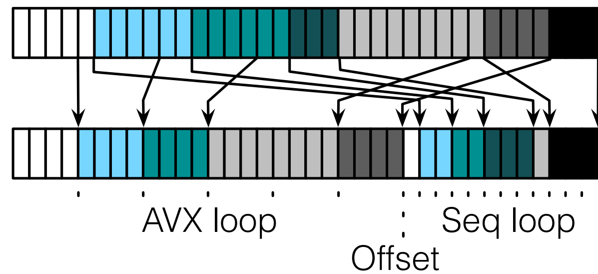



Figure 6.3: Permutation of the elements after bounded coloring. Only the offset needs to be stored to execute the vector loop.

### 6.2.6 Structuredness, Vectorization and Locality

To reach maximal performance and scalability, there must be enough partitions to expose massive parallelism and the D&C partitions have to fit in cache. This implies small partitions. However, the smaller the D&C partitions are, the smaller the colors are, and therefore, as shown in Section 6.2.3 and 6.2.4, the smaller the vectors are. If the domain was more structured, the density of data parallelism would have been higher and we could have used smaller partitions without loosing vectorization potential.

The global trade-off is described in Figure 6.4. A pure D&C strategy with the leaves as small as possible would make impossible to find independent vector operations inside the final partition. Vectorization across partitions would be possible, but at the cost of complex gathering operations and ninja programing skills, which are incompatible with the portability and maintainability of the solutions required by an industrial application. Having a structured mesh would allow to have locality and vectors at the same time with a simple geometrical decomposition and an optimal coloring. A pure coloring version, sacrificing the locality, would produce efficient vectorization with very long colors. Our hybrid approach is tuned to present the best empirical compromise. The structuredness parameter cannot be controlled and is related to the use case.

This small study allows us to draw the pessimistic conclusion that with decreasing memory per core and increasing vector size, the efficiency of computation on unstructured meshes will rapidly hit a

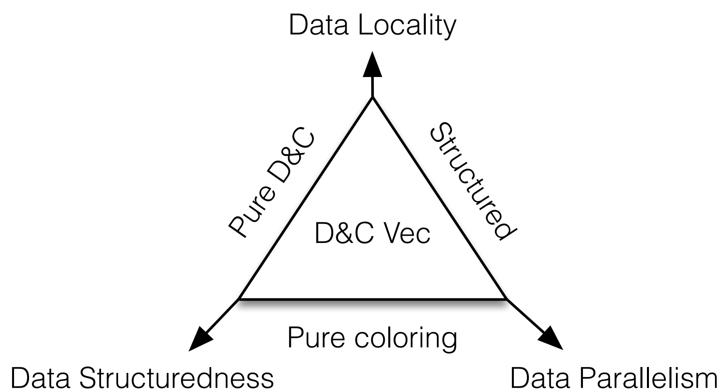


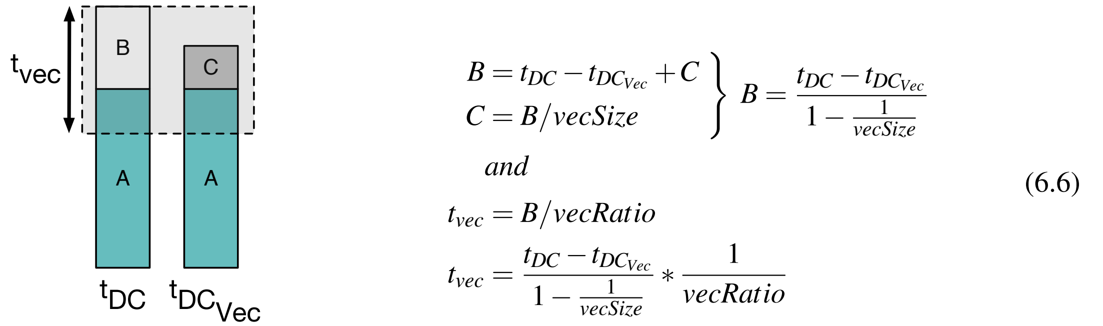

Figure 6.4: Trade-off representation between memory locality, structuredness and vector length.

bottom. Users will have to choose between locality and vectorization, or to consider using more structured meshes. Higher order methods could also be explored to expose other vectorization directions than current element-wise approach.

### 6.2.7 Vector Length Sensitivity Study

We evaluate the vectorization effect on larger leaves to experiment the trade-off between locality and vectorization. To look at the impact of the vector size, we combine our first model of Section 6.2.2 and the Amdahl law. Assuming that the speedup is only due to vector operations and is limited by the vectorization ratio, a simple system of linear equations allows us to deduce the total execution time on which the optimization apply. This is illustrated in Figure 6.5.

In this figure, $t_{DC}$ and $t_{DC_{Vec}}$ respectively correspond to the execution time using the original D&C version and the vectorized D&C version, introduced in this chapter. $t_{vec}$ represents the proportion of time on which the vectorization applied, while $B$ represents the proportion of time which will really be vectorized. $B$ is lower than $t_{vec}$ as long as the vectorization ratio, $vecRatio$, is lower than 100%. Once vectorized, the $B$ part is accelerated by a factor equivalent to the size of the vectors, $vecSize$. This corresponds to $C$. Lastly, $A$ represents the sequential execution time.




$$\left.\begin{aligned} B &= t_{DC} - t_{DC_{Vec}} + C \\ C &= B/vecSize \end{aligned}\right\} B = \frac{t_{DC} - t_{DC_{Vec}}}{1 - \frac{1}{vecSize}}$$
$$and$$
$$t_{vec} = B/vecRatio$$
$$t_{vec} = \frac{t_{DC} - t_{DC_{Vec}}}{1 - \frac{1}{vecSize}} * \frac{1}{vecRatio} \tag{6.6}$$

Figure 6.5: Proportion of time impacted by the vectorization.

Starting from the equations obtained in Figure 6.5, we can deduce the proportion of time, $p_{opt}$, which can benefit from the vectorization:

$$p_{opt} = \frac{B}{t_{DC}} = \frac{t_{DC} - t_{DC_{Vec}}}{t_{DC}} * \frac{vecSize}{vecSize - 1} \tag{6.7}$$

Furthermore, we assume that the gain for other vector instruction sets will be proportional to their length. The vectorization ratio, $vecRatio$, for leaves of size 200 and the studied vector lengths are given by the first model of Section 6.2.2 and summarized in Table 6.1. By applying the Amdahl law, the expected speedup is given by:

$$expectedSU = \frac{B}{C} = \frac{1}{1 - p_{opt} * (1 - \frac{1}{vecSize})} \tag{6.8}$$

The point of this experiment is not to compute exact predictions but to illustrate that we can compute an ideal vector length for a given mesh and locality parameter. Unlike the first thought, trying to generate larger vectors does not necessarily lead to higher speedups. The higher vectorization ratio obtained with smaller vector sizes compensates the loss of data parallelism. In the case of EIB with a leaf size of 200, by using the measured time obtained with the original D&C library and the new vectorized version, the best trade-off for the vector length is 4. This is detailed in Table 6.1. By using the mask instructions of the

Intel Xeon Phi, it is possible to constraint the usage of the vector operation to 256 bits in software to get the maximum speedup. Furthermore, by using our model, we can predict that the upcoming 1024 bits vector size will be problematic for unstructured meshes, even with the same cache size, with an expected speedup of only 1%. Fortunately, these future architectures will still be able to handle smaller vector lengths.

Table 6.1: Vectorization expected speedups for a leaf size of 200.

| *vecSize* | 2 | 3 | 4 | 5 | 6 | 7 | 8 (native) | 16 |
|---|---|---|---|---|---|---|---|---|
| *vecRatio* | 0.96 | 0.90 | 0.83 | 0.76 | 0.69 | 0.62 | 0.55 | 0.02 |
| *expectedSU* | 1.27 | 1.36 | 1.38 | 1.37 | 1.34 | 1.31 | 1.27 | 1,01 |

However, this study only deals with full vectors and does not take into account the remaining incomplete vector sizes generated during the coloring. Indeed, these incomplete vectors could be vectorized using padding or the Xeon Phi masked instructions. While the actual implementation will be addressed in a future work, we can extend the study to handle the incomplete vectors by using our second vectorization model presented in Section 6.2.2.

Let us consider a partition colored with our bounded coloring strategy and running on an AVX512 target architecture. The color distribution is illustrated in Figure 6.6. The colors are sorted by decreasing size. For now, only the colors composed of 8 elements are vectorized and are taken into account in the computed expected speedups. But the colors varying between 2 to 7 elements could be partly vectorized by using padding or Xeon Phi masked instructions.

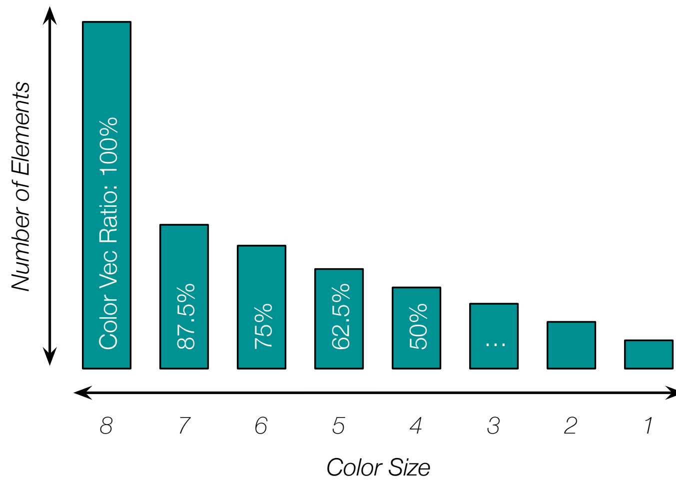



Figure 6.6: Color distribution sorted by decreasing size using our bounded coloring strategy.

## 6.3 Experimental Results

To measure the impact of our new vectorial algorithm using D&C and mesh coloring, we apply and measure it on our Mini-FEM proto-application presented in Chapter 4. We performed our experiments

on the EIB and FGN 3D unstructured meshes from Dassault Aviation presented in Section 4.2.3. In the following experiments, we compare three versions of the FEM assembly step.

- The original pure MPI version using domain decomposition, called *Ref (MPI)*.
- Our D&C version using MPI at distributed memory level and Cilk Plus at shared memory level, presented in previous chapter and called *D&C (MPI+Cilk)*.
- And our new vectorial D&C version, called *D&C Vec (MPI+Cilk)*, using MPI at distributed memory level, Cilk Plus at shared memory level, and coloring at core level to express the vectorization.

We perform our experiments on three different platforms. The first one is the Anselm cluster described in Section 1.6.3. The second platform is a cluster of Intel Xeon Phi from Cirrus presented in Section 1.6.2. And the last one is the MareNostrum cluster presented in Section 1.6.4. The experimental setup is identical to the one used in previous chapter and is detailed in Section 5.3.1. The application is compiled with Intel composers 13.1 and Intel MPI 4.1. Similarly to the experiments presented in Section 5.3.1 of previous chapter, we present the results in terms of relative speedup compared to the best sequential solution and in terms of parallel efficiency.

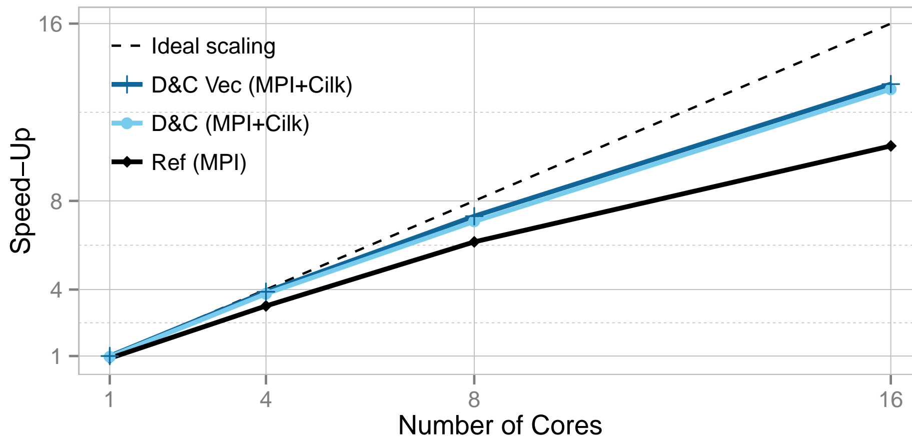
### 6.3.1 Single Node Experiment

We start our experiments on a single node of 16 Sandy Bridge cores, as illustrated in Figure 6.7. While both D&C approaches benefit from the better scalability and efficiency obtained in previous chapter compared to the pure MPI version, the impact of the vectorization is negligible. The *D&C Vec (MPI+Cilk)* version has equivalent behavior than the previous *D&C (MPI+Cilk)*. We checked that the generated assembly code is well vectorized. However, the proportion of vectorized code is negligible compared to the entire assembly time. Indeed, the assembly step has a very low arithmetic intensity and is dominated by the non vectorized CSR traversal and mesh data gathering.

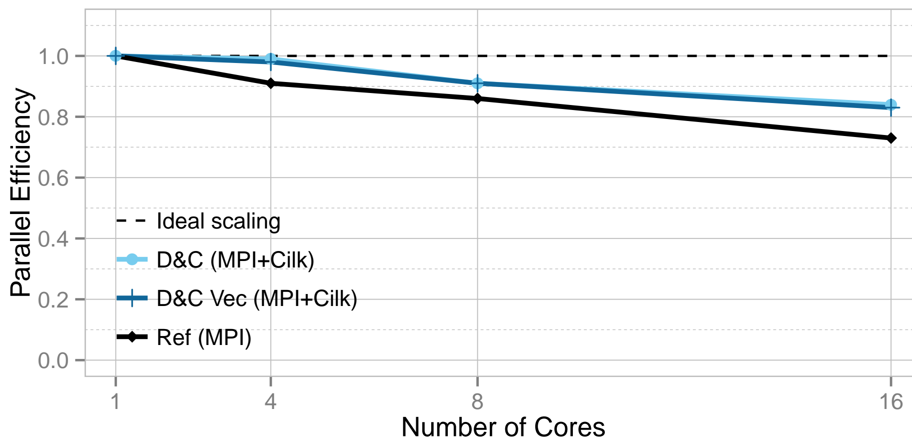
#### Increased Arithmetic Intensity

To give opportunity to the vectorization to pay-off and highlight its benefits, we increase the arithmetic intensity of the assembly step and therefore, the time proportion of vectorized code. The higher arithmetic intensity of the assembly kernel makes it closer to other computation kernels, such as the solver step of FEM applications. It is obtained by looping over the computation of the elements coefficient a hundred times. The speedup and the parallel efficiency of the 100$\times$ arithmetic intensity experiment are presented in Figures 6.8a and 6.8b.

At first, we observe that while the standard D&C version is not much impacted by the higher arithmetic intensity, the pure MPI one scales more efficiently. The higher overhead of the communications and data duplications induced by MPI compared to the Cilk Plus runtime is partly hidden by the higher computation ratio. This way, *Ref (MPI)* and *D&C (MPI+Cilk)* get closer to each other. But this larger computation part mainly benefits to the *D&C Vec (MPI+Cilk)* version since the vectorized part of the computation represents a higher proportion of the overall assembly step. The sequential execution of the vectorized version of the D&C library is 14.8% faster than the standard one. At full node, it ends 13.5% faster since the higher computation benefit is slightly attenuated by the parallelization overhead. In the end, both D&C versions achieve 89% of parallel efficiency.




(a) Speedup compared to the best sequential performance

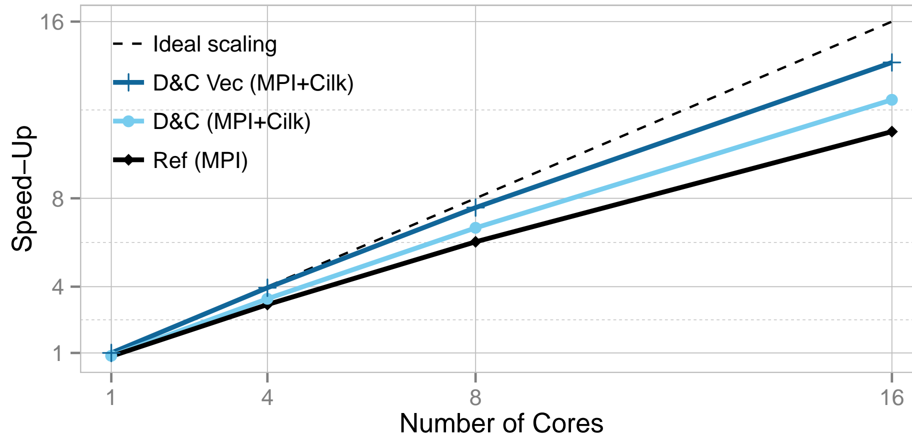



(b) Parallel Efficiency

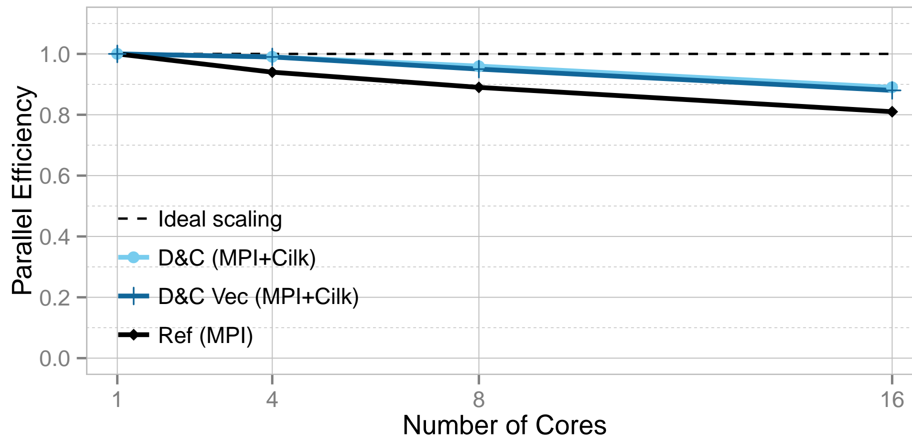
Figure 6.7: FEM assembly speedup compared to the best sequential time and parallel efficiency on a single Sandy Bridge two-socket node.

### 6.3.2 Strong Scaling Experiment with Increased Arithmetic Intensity

Similarly to previous chapter, we extend the vectorization experiments in a 512 cores run to test the strong scalability of the new *D&C Vec (MPI+Cilk)* version. This experiment is presented in Figure 6.9. The increased arithmetic intensity of the assembly operator improves the scalability of all the experimented versions. This is due to the higher proportion of computation compared to the runtimes overhead and to the amount of communications. This way, the *Ref (MPI)* version already evaluated at that scale in previous chapter, moves up from 25% of parallel efficiency to 57%. Similarly, the *D&C (MPI+Cilk)* version, which uses MPI communications between the 32 compute nodes, reaches 71% of efficiency compared to 50% with the standard assembly operator. Concerning the new *D&C Vec (MPI+Cilk)* vectorized version, it reaches 77% of parallel efficiency with only 2000 vertices per core, which leads to an impressive speedup of 323× compared to its sequential execution. The largest computation ratio enables a 13.5% speedup at 512 cores compared to the standard D&C version. This is equivalent to the performance gain obtained in previous section on a single node. This makes sense since the vectorization effort applied




(a) Speedup compared to the best sequential performance



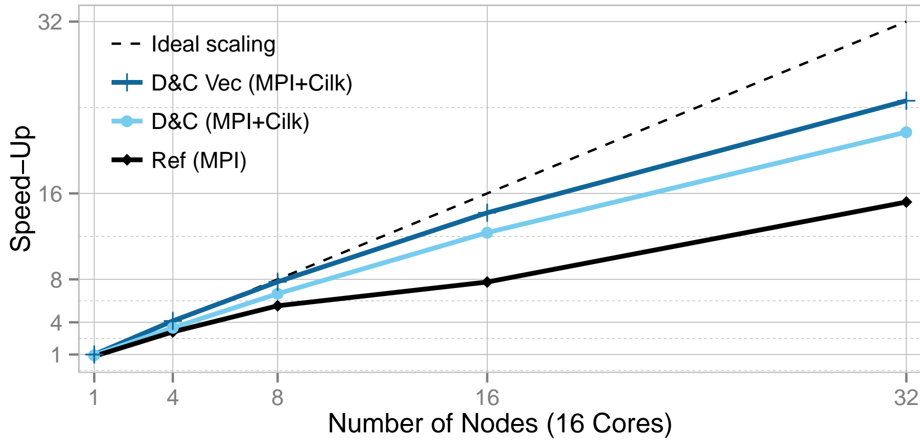

(b) Parallel Efficiency

Figure 6.8: FEM assembly speedup compared to the best sequential time and parallel efficiency with arithmetic intensity increased by a factor of 100×.

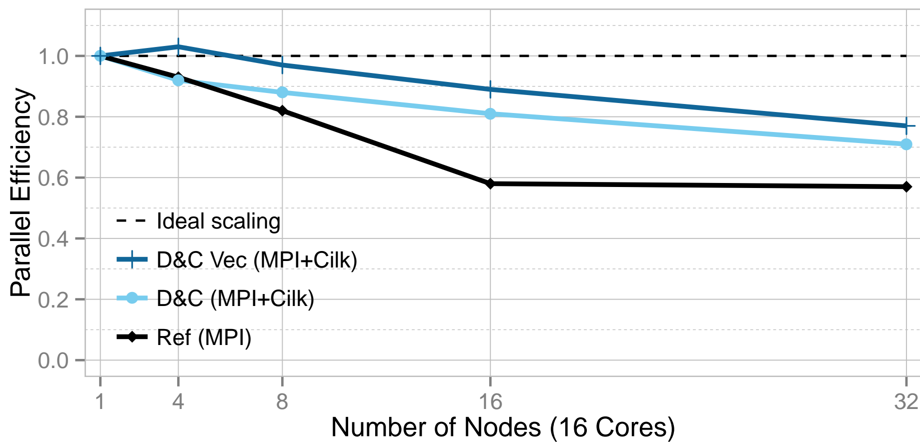
to the D&C tasks allows to take advantage of the vector units within the compute cores and to achieve higher performance. But, it does not change the scalability of the algorithm, which is mainly dragged down by the communications and synchronizations between compute nodes. This last obstacle will be addressed in the next chapter. Nevertheless, results are encouraging for future systems that will probably reach thousands of nodes composed of a thousand cores.

### 6.3.3 KNC Experiments

Similarly to the first experiment presented in Section 6.3.1, we have evaluated the vectorized version of the D&C library on the original assembly kernel with a low arithmetic intensity. But this time, we experiment the *D&C Vec (MPI+Cilk)* on a Xeon Phi KNC composed of larger 512 bits vectorization units and smaller caches, as detailed in Section 1.4. This experiment is illustrated in Figure 6.10. Once again, the two D&C versions have equivalent behavior. Before doing this experiment, we varied the size of the D&C partitions from 50 to 500 elements. We measured that the smaller the D&C leaves are, the better the performances




(a) Speedup compared to the best sequential performance



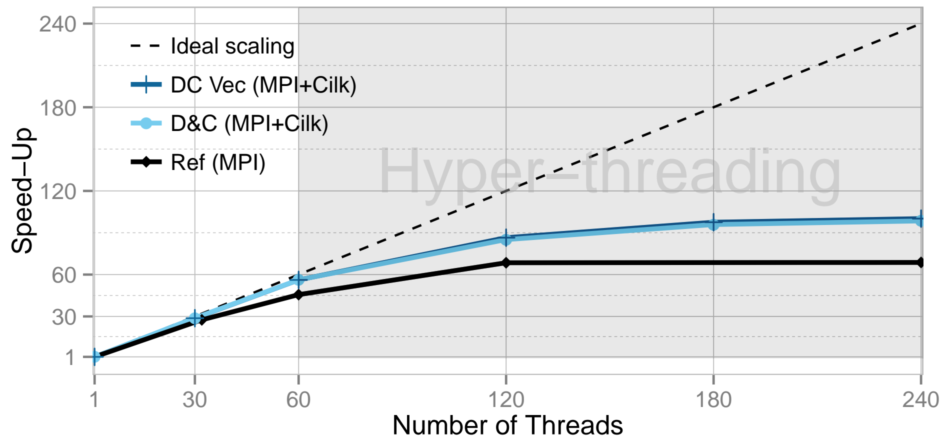

(b) Parallel Efficiency

Figure 6.9: FEM assembly speedup and parallel efficiency using 100× increased arithmetic intensity and running on 512 Sandy Bridge cores.

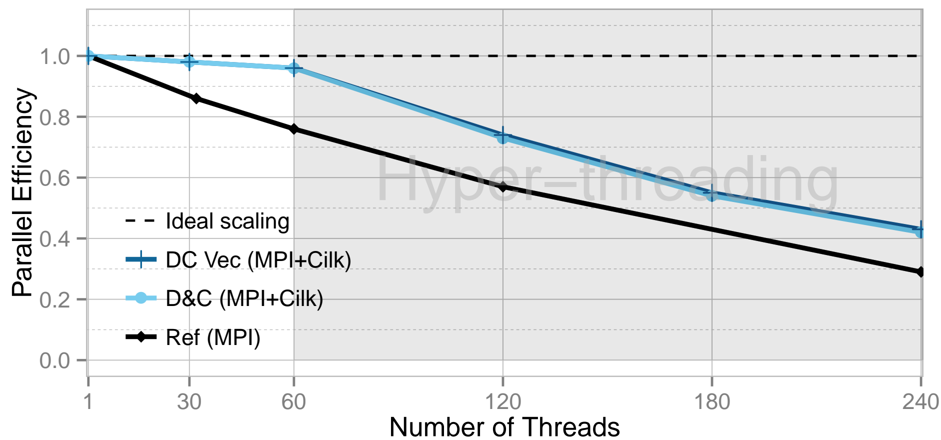
are. As shown in Section 6.2.4, when choosing a leaf size of 50, the vectorization ratio is close to 0. Therefore, vectorization has a limited 1% impact on performance. As a result, although the time spent per element is lower in the vectorial *D&C Vec (MPI+Cilk)* version than in the pure *D&C (MPI+Cilk)* version, the higher vectorization ratio brought by larger partitions does not compensate the loss in locality, because of the low arithmetic intensity of the FEM assembly step.

#### Increased Arithmetic Intensity

To ensure it, we have tested our vectorized D&C version on the more intensive matrix assembly kernel. As illustrated in Figure 6.11, this reveals the difference between *D&C (MPI+Cilk)* and *D&C Vec (MPI+Cilk)*. It is even more pronounced on the KNC than on the previous increased arithmetic intensity experiments made on Sandy Bridge multicores. The KNC architecture with its low power cores, small caches, and large vectorial units benefits more from the higher computation ratio. As a result, the vectorized version of D&C is 1.2× faster than pure D&C on the 60 physical cores of the KNC and 1.19× using the 240




(a) Speedup compared to the best sequential solution

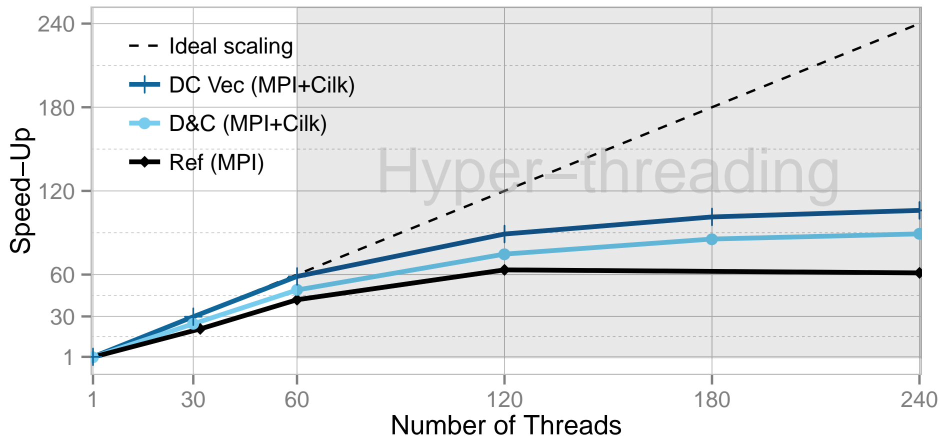



(b) Parallel Efficiency

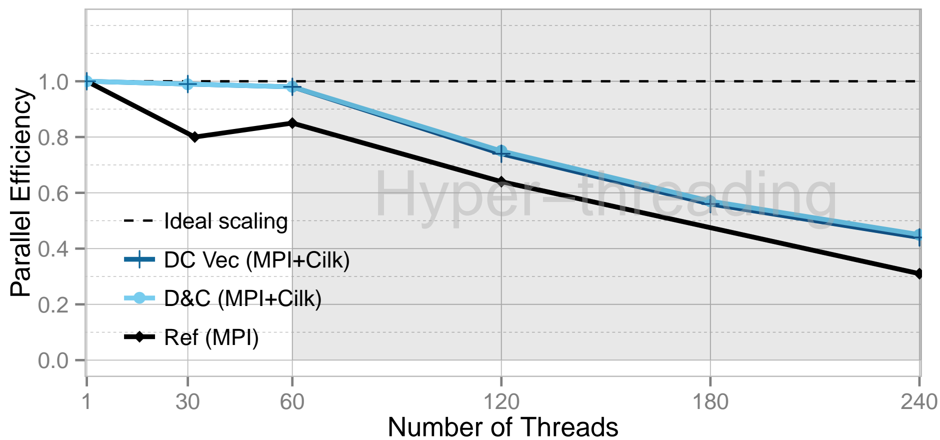
Figure 6.10: FEM assembly speedup and parallel efficiency with standard arithmetic intensity on a single KNC.

hyper-threads. Both of the D&C version achieve exactly the same parallel efficiency of 98% on the 60 physical cores compared to 85% for the pure MPI version. We note that with the increased arithmetic intensity, the *Ref (MPI)* version benefits from the physical core count increase from 30 to 60. This is certainly due to cache or bandwidth saturation effects.

Lastly, we applied the same experiment on 4 KNC. This is illustrated in Figure 6.12. This results in only 1000 vertices per threads when using the 960 hyper-threads. The difference between *D&C Vec (MPI+Cilk)* and *D&C (MPI+Cilk)* on the single thread execution is even more pronounced by running on 4 KNC with a 1.52× speedup in favor of the vectorized version. Indeed, by running the same use case on 4 KNC instead of a single one, the memory bandwidth requirements per KNC decrease. This enables higher performance gains brought by the vectorized version which is more bandwidth greedy. On the 240 physical cores, the vectorization brings a 1.44× speedup compared to the standard D&C version and a 1.22× speedup using the 960 threads. However, the vectorization decreases the time proportion of the parallel part of the code. This makes the sequential part more important and therefore, decreases the parallel efficiency of the code. While the pure D&C version has a parallel efficiency of 94% on the 240




(a) Speedup compared to the best sequential solution




(b) Parallel Efficiency

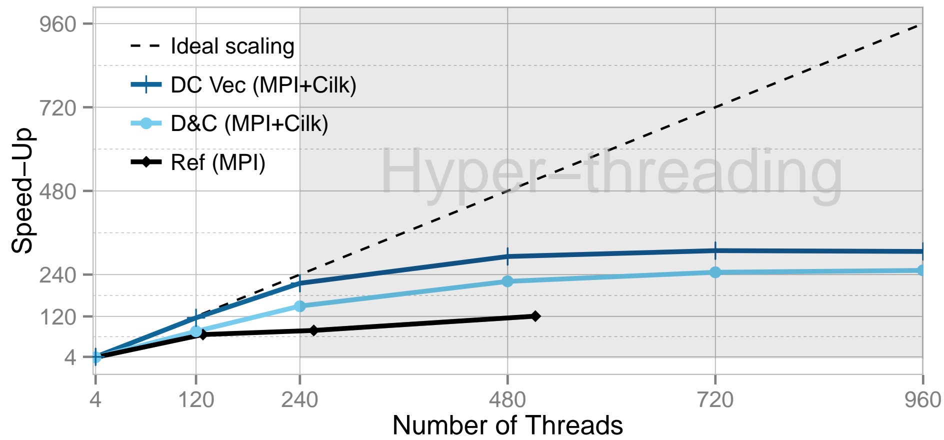
Figure 6.11: FEM assembly speedup and parallel efficiency with 100× increased arithmetic intensity on a single KNC.

cores, the vectorized version has dropped to 90%.

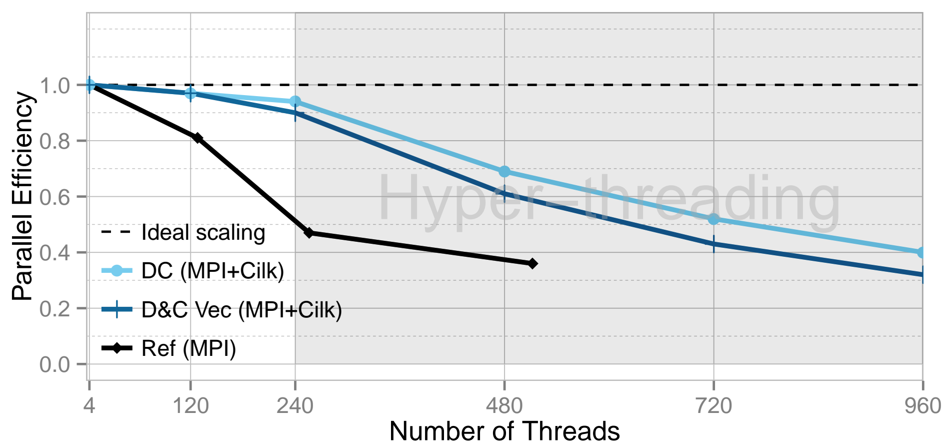
## 6.4 Conclusion

This chapter presents our vectorized version of the D&C library. It proposes a new strategy for unstructured mesh assembly computation which takes fully advantage of the modern multicore and manycore resources. Our strong scaling experiment is very encouraging for an efficient support of upcoming systems. The original bounded colors strategy exposes most of the data parallelism available in cache. However, the current trend which consists in limiting the memory per core and increasing the vector size, is not compatible with the complexity and precision of the new physical models which requires more unstructured meshes. This is even more true on low intensive compute kernels, such as the FEM assembly step.

Using more intensive compute kernels reduces this problem since it decreases the proportion of




(a) Speedup compared to the best sequential solution




(b) Parallel Efficiency

Figure 6.12: FEM assembly speedup and parallel efficiency with 100× increased arithmetic intensity on 4 KNC.

communication and the runtime overhead. But this is not enough to perfectly scale on an important number of compute nodes. Moreover, the vectorization brings significant speedups, especially on the Xeon Phi manycore, but it does not improve the scalability. In the strong scaling experiment made on 4 KNC with 960 hyper-threads and only 1000 vertices per thread, the scalability is reduced by a few percents compared to the non vectorized D&C execution but is still over 90% of parallel efficiency.

The major remaining obstacle to enhance the scalability of our library is the communication layer between distributed nodes. For now, we have proposed an efficient approach to address the shared memory parallel resources within compute nodes and to exploit the cores vectorial resources on small partitions of unstructured meshes. In the next chapter, we propose a new communication pattern aimed at overlapping the communications and reducing the synchronizations.

# CHAPTER 7

# DISTRIBUTED LEVEL PARALLELISM

**Contents**


**7.1 Introduction . . . 107**
**7.2 Potential Gain of Communication Optimization . . . 108**
7.2.1 Benefits and Limitations of Communication Overlap . . . 109
**7.3 Bulk-Synchronous Communications Using GASPI . . . 110**
**7.4 Asynchronous and Multithreaded Communications Using GASPI . . . 111**
7.4.1 Communication Level Version . . . 112
7.4.2 Bounded Communication Size Version . . . 114
**7.5 Experimental Results . . . 118**
7.5.1 Impact of the Number of Processes per Node . . . 118
7.5.2 Bulk-Synchronous Comparison on a Single Node Experiment . . . 119
7.5.3 Strong Scaling Experiment Using Standard Arithmetic Intensity . . . 121
7.5.4 Multiple KNC Experiment Using Standard Arithmetic Intensity . . . 123
7.5.5 All-in-One D&C Version . . . 124
**7.6 Conclusion . . . 127**


## 7.1 Introduction

In Chapter 5, we have built a parallelization scheme optimized to exploit shared memory parallelism inside each MPI distributed domain, using the divide and conquer approach. Then, in Chapter 6, we have addressed the vector units embedded in modern compute cores using a tuned coloring heuristic for small D&C partitions. Our last contribution focuses on replacing the standard MPI two-sided communication model, presented in Section 2.2.1, by a multithreaded and asynchronous communication model based on PGAS, detailed in Section 2.3, and on one-sided communications. We use the GASPI API [6, 4, 5] viewed in Section 2.3.2 which enables asynchronous one-sided transfers using RDMA interconnect.

A study on the potential performance gain brought by the optimization of the communication model is given in Section 7.2. Then, we have developed a bulk-synchronous GASPI implementation mimicking the MPI approach, but using one-sided communications to compare GASPI one-sided performance to MPI two-sideds. This experiment is detailed in Section 7.3. Lastly, we propose two different approaches both aimed at exploiting the multithreaded parallelism between D&C tasks and overlapping communication by computation. As soon as a D&C task achieves the computation of a part of the interface, it can

directly send it without waiting for the rest of the computation. However, a trade-off between earlier communication and their number must be explored.

We propose two different approaches. The first one consists in fixing a communication level at a given height of the D&C tree below which no communication is done. This version is explained in more details in Section 7.4.1. However, by using this approach, all the descendants of the tasks located at the communication level must be ended before they can start to communicate. This reduces the overlap possibilities between communication and computation. Our second approach presented in Section 7.4.2, consists in buffering the data to communicate according to their remote receiver, and to send the buffer as soon as it reaches the chosen size. This version allows better balancing between communications and increases the communication overlap possibilities. As shown in Section 7.5, both of these two versions benefit from a better performance and scalability, and decrease the memory consumption induces by the MPI model.

## 7.2 Potential Gain of Communication Optimization

The contributions proposed in the two previous chapters have permit to notably increase the performance compared to our reference version only based on MPI domain decomposition. This node performance gain between D&C and the reference version in our strong scaling experiments made at 512 cores using 2000 vertices per core, is illustrated in Figure 7.1. In this chapter, we focus on the impact of the distributed communications on the performance. To estimate their overhead, we comment out the communication calls. The resulting *D&C Comm Free* version represents the best performance that is possible to attain by using our shared memory model. It includes all the node performance improvements brought in the previous chapters, such as the D&C locality gains, the Cilk Plus threading model, and the vectorization of the D&C tasks.

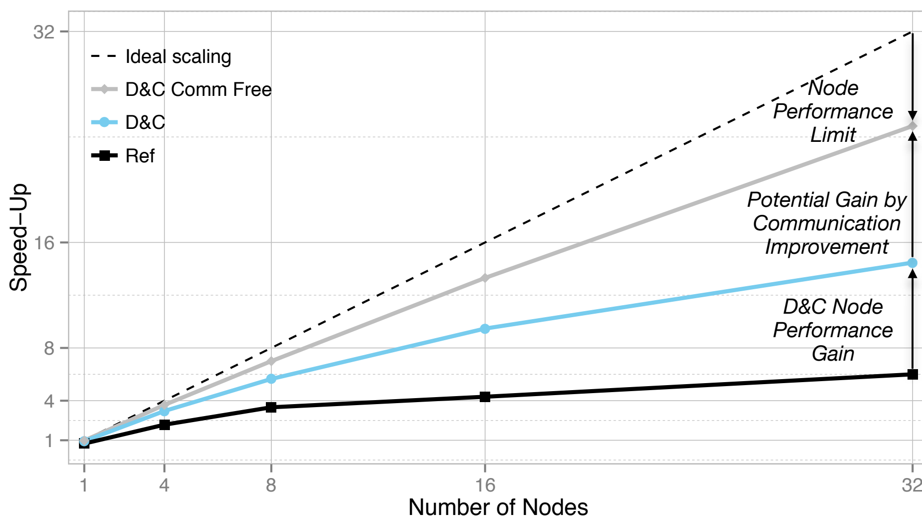



Figure 7.1: Potential gain brought by overlapping communication by computation.

The difference between the communication free version and our D&C version represents the maximal performance gain which can be obtained by optimizing the communication pattern. The threading model is no more a limitation. It scales almost perfectly with 96% of parallel efficiency on the Xeon Phi physical cores. The main bottleneck is the communication model. It could be optimized by changing the numerical algorithm using communication avoiding approaches like the one used in pipelined CG [65]. However, this

is not our research topic. Our optimization opportunities include the parallelization of the communications and of the scatter and gather operations, the removal of synchronization points, and the overlap of the communications by computation. The goal of the contributions proposed in this chapter is to bring D&C closer to the ideal *D&C Comm Free* performance.

### 7.2.1 Benefits and Limitations of Communication Overlap

Recovering the communications by computation may lead to significant performance improvements. However, the performance gains depend on the ratio between communication and computation. The overall execution time is bounded to the maximum between execution and communication time. Therefore, the maximum speedup brought by communication overlapping can be reached if the time spent in the communications is equivalent to the time spent computing and if both of them entirely overlap. This is illustrated in Figure 7.2d. Communicating includes all the remote and network operations while computing includes the local shared memory operations and I/Os.

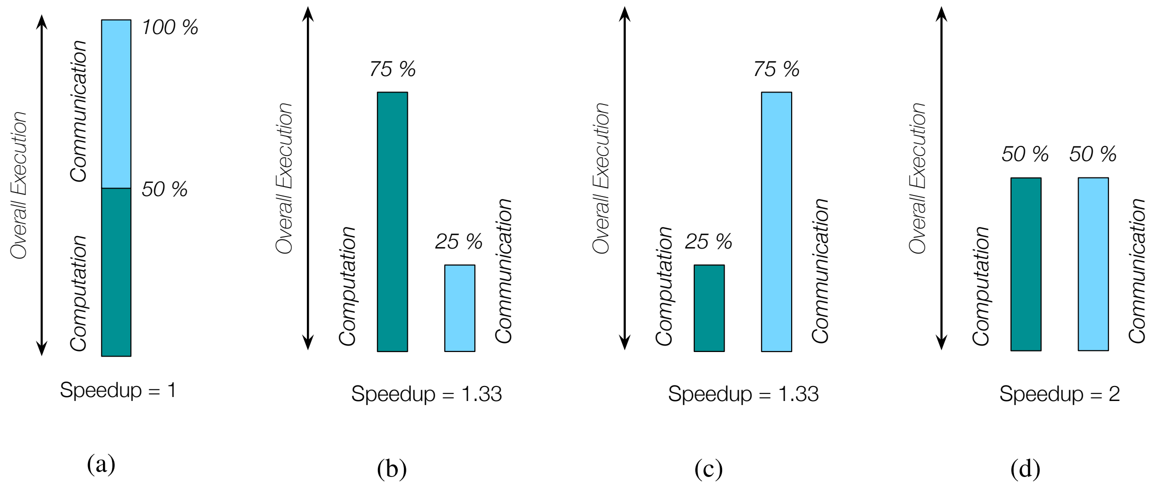



Figure 7.2: Synthetic illustration of attainable speedups with a complete overlap of communication and computation.

More formally, let *comm* be the proportion of time spent in the communications, *comp* the proportion of time spent computing, and *overlap* the proportion of time in which computation and communication are done simultaneously. The performance gain brought by entirely overlapping the communications is given by:

$$speedup = \frac{comp + comm}{max\ (comp, comm)} \leq 2 \tag{7.1}$$

The maximal speedup is obtained if $max(comp, comm)$ is minimal, i.e. if $comp = comm = 50$. Therefore, the maximal speedup that can be reached by doing communication overlap is 2.

However in practice, computation and communication are unlikely to entirely recover with each other. In the context of FEM, the communications involve the halo exchanges between distributed subdomains. In this case, the computation of the interface value has to be achieved, at least partially, before starting to communicate. This way, the maximal expected speedup is lower than 2. It can be computed according to the following equation:

$$speedup = \frac{comp + comm}{comp + comm - overlap} \leq 2 \tag{7.2}$$

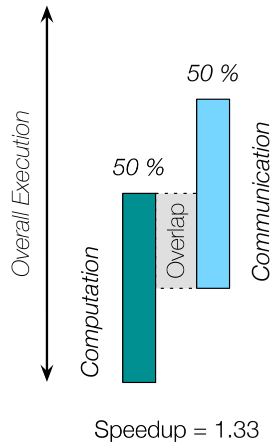



Figure 7.3: Incomplete overlap with equilibrated computation and communication loads.

In Figure 7.3, $comp = comm = 50\%$ and half of them are executed simultaneously, i.e. $overlap = 25\%$. The speedup compared to the bulk-synchronous execution with no overlap is equal to $1.33\times$.

Furthermore, during the strong scaling experiments, the computation part tends to 0% while the communication part tends to 100%. This way, the expected speedup tends to 1. In that case, there is little interest to invest in communication overlapping. At scale, the performance gain gets similar to bulk-synchronous approaches. However, the parallelization of the communications and of the pack/unpack and gather/scatter operations is critical to improve the scalability of the code since it directly impacts the communication time that will become predominant.

## 7.3 Bulk-Synchronous Communications Using GASPI

As discussed in Section 3.5, a pure domain decomposition approach requires to increase the number of subdomains, and consequently the number of duplicated ghost cells and communications in order to increase the amount of parallelism. This way, the more subdomains there are, the smaller these subdomains are, and the higher the proportion of time spent communicating is. The common approach when using MPI consists in executing the computation phase entirely before launching the communication phase. This bulk-synchronous approach is illustrated in Figure 7.4.

To measure the GASPI one-sided communication performances, we built a preliminary bulk-synchronous version of the halo exchange step, called *GASPI Bulk*. This version is really similar to standard MPI two-sided communications. Each process loops over its neighbors in parallel, packs the data to the GASPI

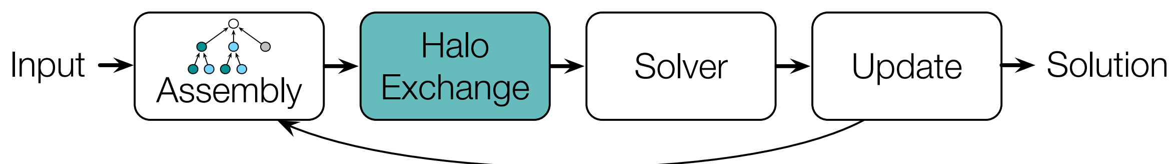



Figure 7.4: *Bulk-synchronous approach* - The halo exchange communication phase starts after the end of the assembly computation phase.

segment, and write data directly to the neighbors. Just after, the process waits for any communication coming from one of its neighbors in any order. As soon as it receives one notification, it will be able to unpack the corresponding data and to wait for the next communication.

In strong scaling experiments, the computation phase tends to zero as the parallelism grows, while the time spent in the communications gets bigger and bigger. As stated in previous section, the communication overlap opportunities are then limited. The major expected performance gain during this experiment comes from the parallelization of the communications exploiting the parallelism offered by the recent networks technologies, such as the RDMA. Moreover, the number of subdomains must be reduced to its minimum, i.e. one per distributed computing unit, and we need to exploit shared memory resources inside each distributed subdomain. Section 5.2 explains how we use the D&C approach to enable shared memory parallelism by creating a large amount of parallel tasks. This allows to create more concurrency by increasing the number of threads instead of increasing the number of distributed domains, and to replace communications and halo duplications by data sharing inside each NUMA node. However, the communications between the distributed NUMA nodes still represent a non negligible part of the execution time on large clusters.

## 7.4 Asynchronous and Multithreaded Communications Using GASPI

To take full advantage of GASPI one-sided asynchronous communications, we replace our bulk-synchronous implementation of the halo exchange by an *asynchronous and multithreaded* version illustrated in Figure 7.5. The idea is to exploit the parallelism of the D&C tasks when packing the communication buffers and sending them. Moreover, to minimize the impact of communications, it is also crucial to overlap them with computation. Therefore, communications have to start as soon as a piece of data is ready to be sent. Our actual standard MPI two-sided halo exchange needs to wait for the end of the D&C parallel region before starting the communication process. This process consists in packing all required data into the communication buffers, sending to all neighbors the corresponding interface, and finally receiving and unpacking the data from all neighbors. Conversely, by using GASPI asynchronous one-sided communications, it is possible to directly start this process inside the D&C tasks running in parallel. This way, once a piece of data handled by a D&C task is ready, it can be directly packed and written to the appropriate neighbor. Then, the computation of the other D&C tasks will overlap this communication. Moreover, since the communications are handled by the InfiniBand cards, the cost of the write is low. Once the D&C parallel region comes to its end, most of the communications are completed and the data unpacking can start with minimal waiting times.

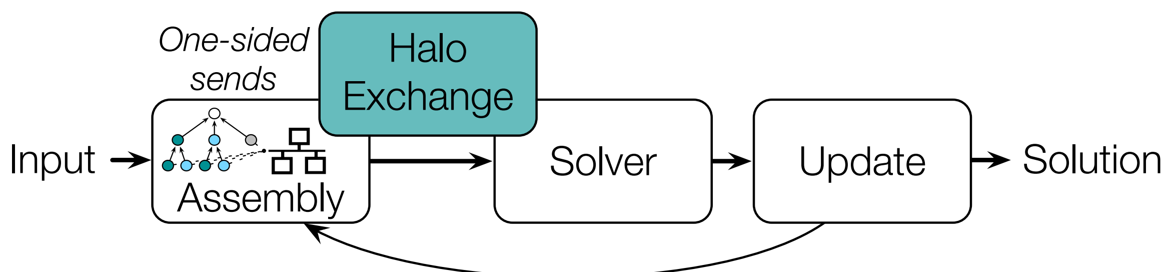



Figure 7.5: *Asynchronous approach* - As soon as a piece of data to communicate is ready, it is sent. Once the D&C parallel region comes to its end, most of the communications are completed and the data unpacking can start with minimal waiting times.

The aim of the D&C decomposition is to build a large number of very fine grain tasks. This granularity is incompatible with the latency of remote operations. Therefore, we have to limit the number of communicating tasks. We have built two asynchronous one-sided versions implementing different approaches to reduce the amount of communications. They are explained in the following sections.

### 7.4.1 Communication Level Version

To control the number of communications outgoing from the D&C tree, we firstly define a communication level. This communication level corresponds to a customizable depth of the D&C tree below which no communication is done. Only the tasks upper this level can handle communications. Let us call these D&C tasks, the *communicating tasks*. All the D&C tasks located under the communication level only deal with computation. We call them the *standard tasks*. Therefore, the interface values owned by the *standard tasks* have to be exchanged via other *communicating tasks*. The D&C nodes located at the communication level handle the interface values of their whole subtree. The *communicating tasks* corresponding to the leaves of the D&C tree just have to pack and send their local interface values.

This first version is called *GASPI Async v1* and is illustrated in Figure 7.6. The internal D&C nodes *a* and *b* have to communicate the interface values contained in their respective subtree while the D&C leaf *c* only sends its local interface values. With this communication level version, it is possible to configure the number of D&C tasks involved in the communications and therefore, the number of communications and indirectly their size. The upper the communication level is, the less communications there are, and the bigger they are. The lower it is, the more parallelism there is to handle communications, and the more overlapping possibilities there are between communication and computation. We empirically found, as shown in Table 7.1, that a third of the D&C tree is a good compromise between the number of communications and the amount of parallelism.

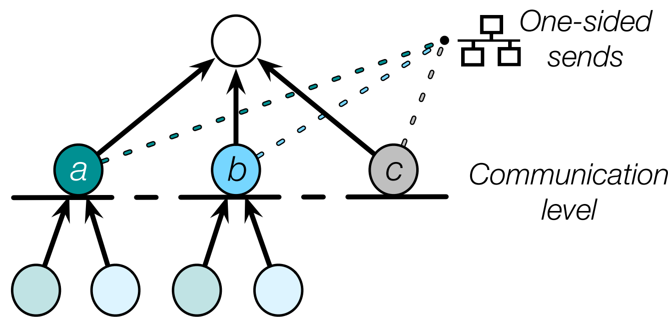



Figure 7.6: *GASPI Async v1* - Illustration of the communication level version using GASPI one-sided in the D&C tasks located above the level.

However, this approach has two main drawbacks. Firstly, it leads to unbalanced communication sizes. Indeed, there may be a huge difference between the D&C nodes handling a large subtree and the leaves handling sometimes only one value to communicate. Lastly, increasing the size of the communications

Table 7.1: Communication level impact between a quarter, a third, and half of the D&C tree.

| *Height of the communication level* | 1/2 | 1/3 | 1/4 |
|---|---|---|---|
| *Speedup compared to sequential at 128 cores* | 5.85 | 6.31 | 6.06 |

to better exploit the network bandwidth and reducing the number of tiny communications, implies to bring the communication level closer to the root of the D&C tree. This results in a lower amount of parallelism available to pack the data. And more importantly, this reduces the amount of computation which can be used to overlap the communications since the main part of the computation is located under the communication level and has already been executed. In the opposite, if the communication level is too low, there are too many communications. The best trade-off at $1/3$ is limited by this large number of communications.

Moreover, in practice the development of this first asynchronous and multithreaded version using GASPI one-sided communications was not straightforward. It requires several modifications in the code and additional data structures. A pseudo-code of the additional pre-computations required for this version is given in Algorithm 9. We have to split the original interface index into as many parts as

**Algorithm 9:** Required pre-computations for the communication level approach.

```
1  Function D&C_tree_second_traversal (D&C tree)
2  begin
3  |  Compute_offset_in_GASPI_segment (D&C tree → partition)
4  |  cilk_spawn
5  |  D&C_tree_second_traversal (D&C tree → left)
6  |  D&C_tree_second_traversal (D&C tree → right)
7  |  cilk_sync
8  |  if D&C tree → sep ≠ null then
9  |  |  D&C_tree_second_traversal (D&C tree → sep)
10 |  end
11 end
12 Function D&C_tree_first_traversal (D&C tree)
13 begin
14 |  Compute_list_of_interface_nodes (D&C tree → partition)
15 |  if D&C tree → level = communication level then
16 |  |  Compute_list_of_subtree_interface_nodes (D&C tree → partition)
17 |  |  Compute_number_of_communications (D&C tree → partition)
18 |  end
19 |  cilk_spawn
20 |  D&C_tree_first_traversal (D&C tree → left)
21 |  D&C_tree_first_traversal (D&C tree → right)
22 |  cilk_sync
23 |  if D&C tree → sep ≠ null then
24 |  |  D&C_tree_first_traversal (D&C tree → sep)
25 |  end
26 end
27 Function D&C_communication_level (D&C tree)
28 begin
29 |  D&C_tree_first_traversal (D&C tree → head)
30 |  D&C_tree_second_traversal (D&C tree → head)
31 |  Exchange_number_of_incoming_GASPI_notifications ()
32 end
```

the number of D&C tasks handling values on the interfaces. We also need to compute the number of incoming communications for each process and their size. And lastly, we compute the offset of each *communicating task* to give them a unique emplacement in the GASPI segments, and we exchange the number of communications between neighbor processes. This requires an additional tree traversal in the pre-computation phase in order to at first, reduce to the root the number of interface values handles by each left, right, and separator D&C task, and secondly, to spread to the *communicating tasks* the number of interface values handled before them.

### 7.4.2 Bounded Communication Size Version

To solve the communication balancing and overlap issues of the previous version, we have developed a new asynchronous and multithreaded version still using GASPI one-sided communications. The idea of this new version is to balance the size of the communications while increasing the amount of parallelism available to pack the data and increasing the overlap possibilities. To this end, we remove the communication level and replace it by a customizable communication size parameter. In this version, all the D&C tasks handling values on the interface are eligible to append interface elements into the GASPI segment. Only the tasks attaining the communication size and the tasks handling the last value on the interface send the values appended in the buffer from the previous communication.

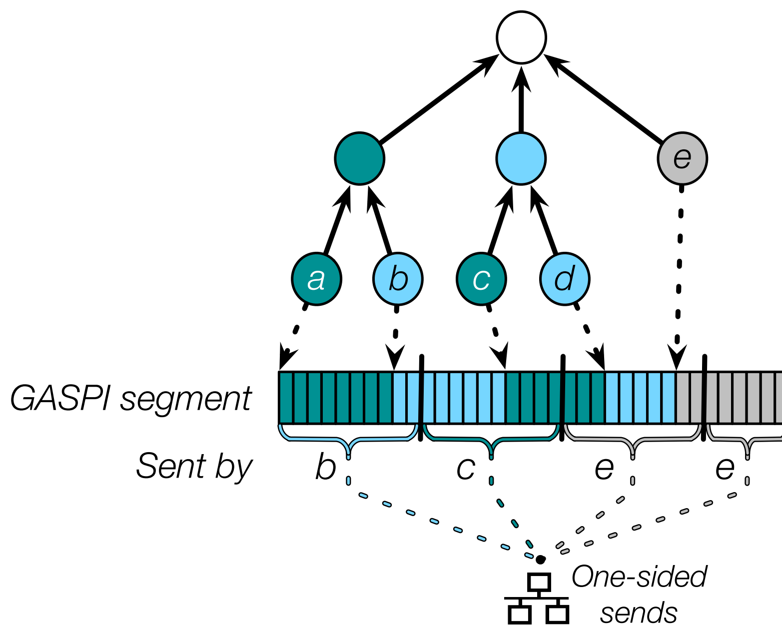



Figure 7.7: *GASPI Async v2* - Illustration of the bounded communication size using GASPI one-sided inside any D&C tasks attaining the communication size.

This new version, called *GASPI Async v2*, is illustrated in Figure 7.7. The D&C task *a* is the first task to pack data in the GASPI segment. The task *b* appends its data to those of *a* and reaches the first communication, therefore it has to send the first chunk of the GASPI segment containing a part of its data, and the data of *a*. Then, the task *c* reaches the second communication and sends the beginning of its data and the end the data of *b*. Lastly, *e* sends the last part of *c* data, the data of *d* and the beginning of its data, and then makes a second communication when it attains the last interface value even if the communication size is not reached. However, this version requires two additional counters per interface.

For each interface, the first counter indicates how many values have been packed into its communication buffers and the second one indicates until which offset the values have already been sent. In the previous version, only the D&C tasks upper the communication level containing themselves or in their subtree, values on the interfaces, handle the communications. Moreover, they systematically send them if they are eligible, no matter if there are 1 or 1000 values to communicate. In this new version, any D&C task, provided that it contains values on the interface, packs its data to the communication buffer of the appropriate neighbor and increments the first counter of values contained in the buffer. If the task does not reach the communication size, it is done. However, if it reaches the size, it sends all the data packed in the current interface buffer since the last communication, coming from different tasks, and increments the second counter to the last sent value. In the same way, the task which access the last value to communicate, will send all the data packed since last communication, even if the communication size is not reached.

As the different tasks run in parallel, the accesses to the counters have to be protected by a lock in order to avoid race conditions. Since all interfaces have their own communication buffers and their own counters, they are protected by independent locks. These locks still negatively impact the performance by delaying the tasks accessing to the same interface buffers at the same time. But in order to minimize the time spent in the locks, the pointers values are only read and updated if necessary, and then the locks are immediately released. The lock contention could be reduced again by splitting each interface into several buffers, using independent locks or even by using double-ended buffering. However, these locks are not the bottleneck for now. The rest of the execution is done with local copies of the counters.

This new version with fixed communication size requires less modifications in the code than the previous version using the communication level. Indeed, the number of communications can simply be computed knowing the number of values on the interface and the size of the communications. This is illustrated by a pseudo-code given in Algorithm 10. The size of the incoming communications are even more trivial to know, except for the last one which requires to be explicitly sent. There is no more offset to compute the position of the data in the GASPI segment since the values are appended in the order of execution of the D&C tasks. This allows us to save the additional tree traversal required in previous version.

**Algorithm 10:** Required pre-computations for the bounded communication size approach.

1 **Function** D&C_tree_traversal (*D&C tree*)
2 **begin**
3 | Compute_list_of_interface_nodes ($D\&C\ tree \rightarrow partition$)
4 | Compute_number_of_communications ($D\&C\ tree \rightarrow partition$)
5 | cilk_spawn
6 | D&C_tree_traversal ($D\&C\ tree \rightarrow left$)
7 | D&C_tree_traversal ($D\&C\ tree \rightarrow right$)
8 | cilk_sync
9 | **if** $D\&C\ tree \rightarrow sep \neq null$ **then**
10 | | D&C_tree_traversal ($D\&C\ tree \rightarrow sep$)
11 | **end**
12 **end**
13 **Function** D&C_bounded_communication_size (*D&C tree*)
14 **begin**
15 | D&C_tree_traversal ($D\&C\ tree \rightarrow head$)
16 | Exchange_number_of_incoming_GASPI_notifications ()
17 **end**

**Impact of the Size of the Communications**

In order to have an estimation on the way the size of the communications impacts their number, we vary the communication size from 100 to 2000 values and measure the total number of communications. The numbers of GASPI processes and of Cilk threads remain fixed. We use 8 nodes of 16 cores using 1 process and 16 Cilk threads per node and we run the EIB use case. This experiment is illustrated in Figure 7.8. Contrary to our first thought, the size of the communications is not linearly correlated to their number. Indeed, at the scale of our experiment, i.e. using 128 cores with around 7800 vertices per core and 36% of the mesh values in the interfaces, the interface halos are small and irregular. Most of them are composed of less than 1500 values. This way, the communication count is almost unchanged by increasing the communication size from 1500 to 2000 values. Since the communications are triggered if their maximal size is reached, or, if the last interface value is attained, most of them are smaller than the fixed size. In the opposite, all the interfaces are composed of more than 200 values. Therefore, the number of communications is divided by two, by increasing the communication size from 100 to 200. There is then no need to strongly increase the communication size to maintain a reduced number of communications. In the example of Figure 7.8, using 500 values per communication seems to be a sweet spot between the amount of parallelism in the communications and their number.

In a second time, to measure the impact of the communication size parameter on the performance, we vary it from 100 to 2000 values by using different sizes of D&C partitions. The D&C partitions vary from 50 to 400 elements. This experiment is illustrated in Figure 7.9. Although the curves are not smooths and the differences are not really pronounced, we can observe different behaviors. Firstly, the two parameters, i.e. the size of the communications and the size of the D&C partitions are not strongly correlated. The "bowl" shape of the curves does not change significantly depending on the partitions size. This makes sense since no matter how small the partitions are, the interface values will only be sent when the communication size will be reached. Secondly, the optimal communication size is located around 500 values. Lastly, the bigger the D&C partitions are, the worst the performance is.

Since the arithmetic intensity during the assembly step is low, we reproduce the same experiment with an artificially increased intensity. The idea is to increase the proportion of work compared to the

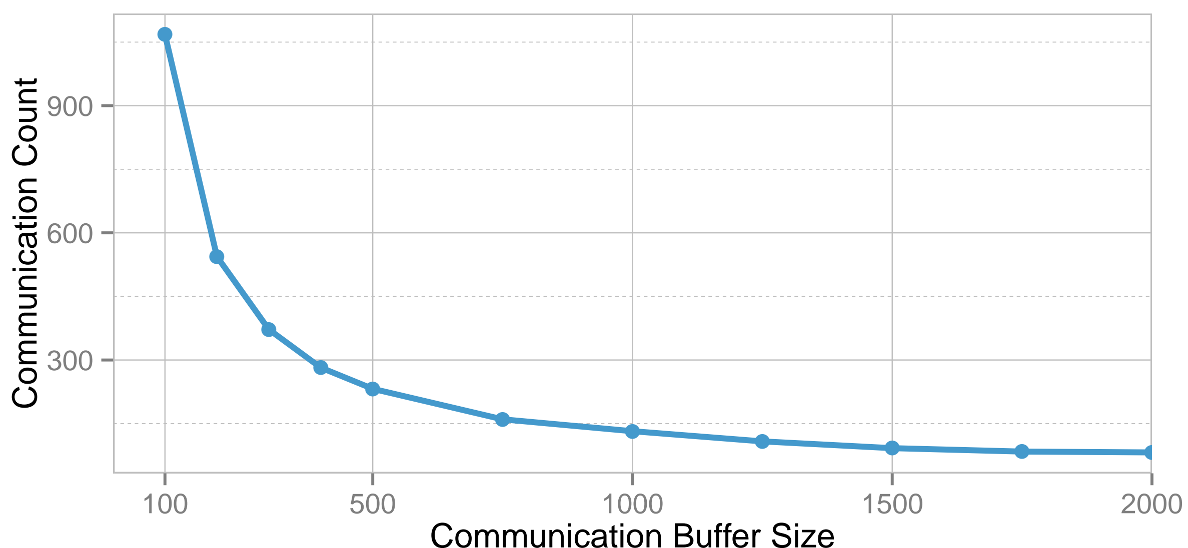



Figure 7.8: Number of communications depending on their size.

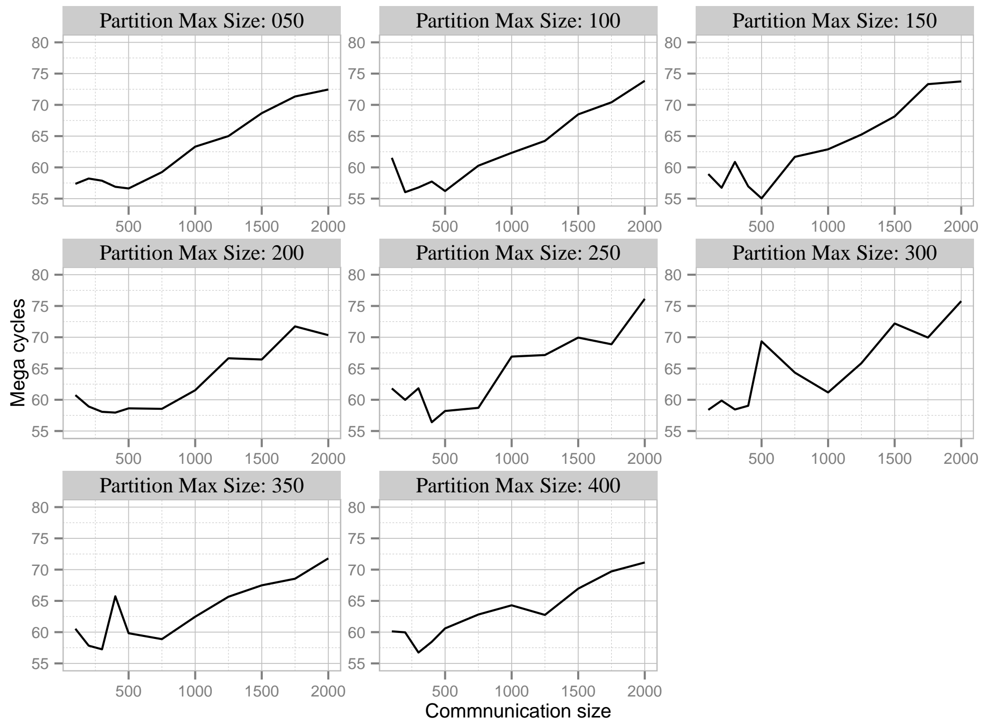



Figure 7.9: Impact of the communication and the D&C partition size on the performance.

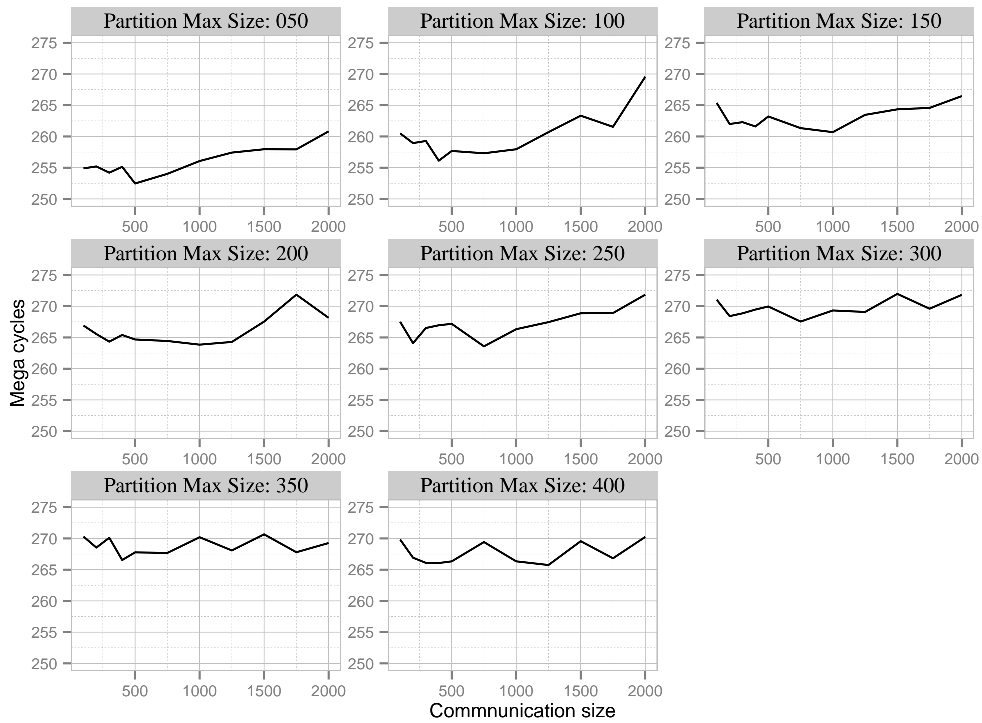



Figure 7.10: Impact of the communication and the D&C partition size with a hundred times more work.

computation, to increase the overlap possibilities. To do so, we compute a hundred times the coefficient of each cell of the mesh. This new experiment is illustrated in Figure 7.10. The impact of the communication size becomes more negligible and it's difficult to determine an optimal. However, the impact of the partition size is more pronounced. Smaller partitions result in more locality, as explained in Section 5.2.3, and in more parallelism without impacting the size of the communications.

## 7.5 Experimental Results

To validate our new approaches, we apply them on the Mini-FEM proto-application, presented in Section 4.3. We compare six versions of the halo exchange implementation on 512 Sandy Bridge cores.

- The original bulk-synchronous version using domain decomposition in addition to MPI two-sided communications, called *Ref (MPI)*.
- The original bulk-synchronous version using domain decomposition with GASPI one-sided communications, called *Ref (GASPI)*.
- The D&C bulk-synchronous version using domain decomposition with MPI two-sided communications and our divide and conquer approach using Cilk inside each MPI subdomain. This version is called *D&C (MPI+Cilk)*.
- The D&C bulk-synchronous version using domain decomposition with GASPI one-sided communications and our divide and conquer approach using Cilk inside each GASPI subdomain. This is the *D&C Bulk (GASPI+Cilk)* version.
- The D&C asynchronous version using domain decomposition and divide and conquer with GASPI one-sided communications inside the D&C tasks above a communication level, called *D&C Async v1 (GASPI+Cilk)*.
- And the D&C asynchronous version using domain decomposition and divide and conquer with GASPI one-sided communications inside any D&C tasks with a fixed communication size, called *D&C Async v2 (GASPI+Cilk)*.

Lastly, we evaluate the all-in-one *D&C Vec Async v2 (GASPI+Cilk)* version. This version combines all the contributions presented during this thesis. The D&C parallelization at shared level using the Cilk Plus runtime, the vectorization using coloring at task level, and our last version of the multithreaded and asynchronous communication pattern using GASPI one-sided. The experiments are made on the MareNostrum cluster, presented in Section 1.6.4, and on Salomon cluster's KNC described in Section 1.6.3. The experimental setup presented in Section 5.3.1 remains unchanged. The application are compiled with the Intel composers 14.0.2. We use the Intel MPI 4.1.3 library and the GPI-2 version 1.3 of GASPI.

### 7.5.1 Impact of the Number of Processes per Node

As a first experiment, we tried to determine which of the two options between using one process per node, or one process per socket, results to the best performance. We compare these two options on the *D&C MPI*, the *D&C GASPI Bulk*, and the *D&C GASPI Async* versions. The experiment was made on 8 distributed nodes. It is illustrated in Figure 7.11. For all hybrid versions, using either MPI or GASPI, the best performance is achieved by only using one process per node. We observe a 10% speedup using only one process per node on both the MPI and GASPI bulk-synchronous versions and a 20% speedup on the GASPI asynchronous version. Indeed, having one process per node instead of one per socket involves less

communications per node and bigger subdomains. Therefore, the ratio of communication compared to computation decreases inside each distributed node. Moreover, the smaller number of communications can be more easily recovered by the larger amount of computation per process.

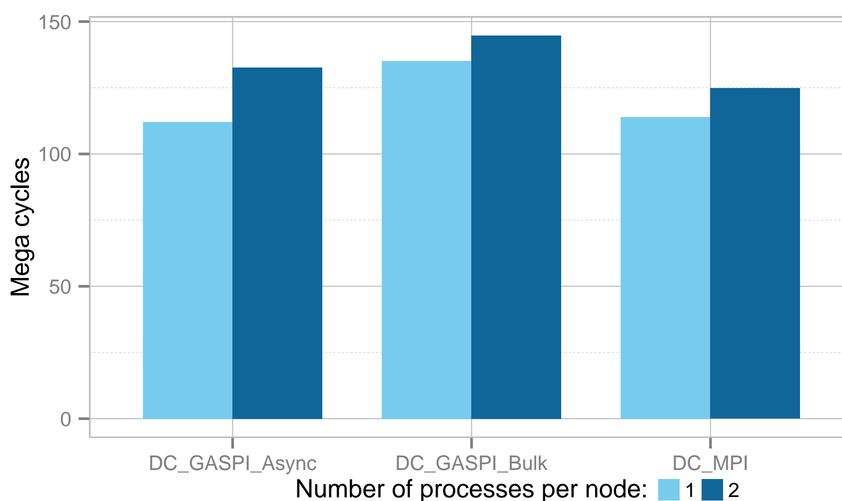


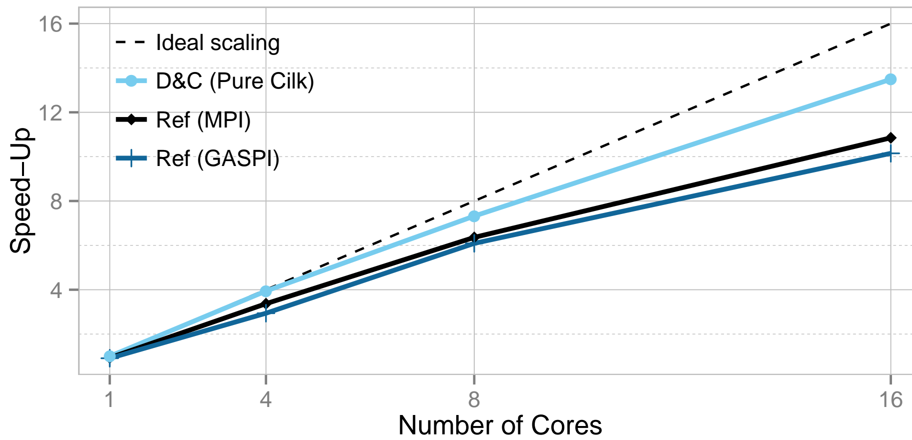

Figure 7.11: Performance comparison between one process per node and one process per socket.

### 7.5.2 Bulk-Synchronous Comparison on a Single Node Experiment

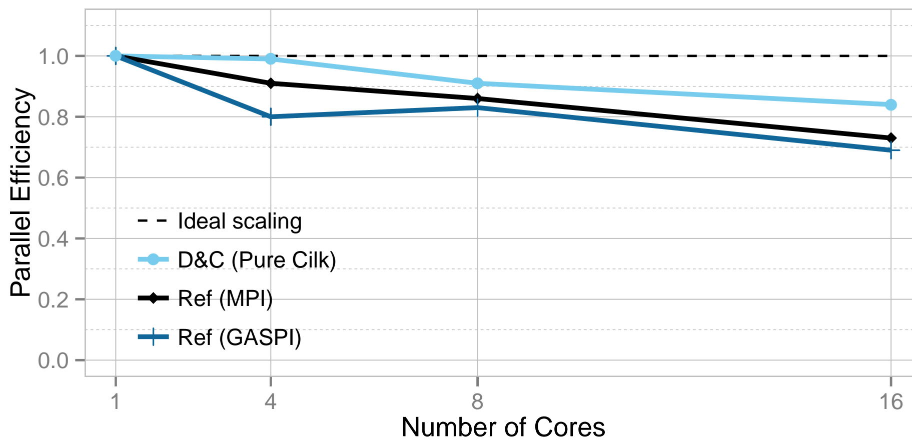
In the first experiment, we compare the performance of the Intel MPI library to the GPI-2 GASPI library using the shared memory resources of a single node. The MPI and GASPI bulk-synchronous implementations are also compared to the Cilk version of the D&C library. Since this D&C version only uses a single process, whether we choose MPI, GASPI, a bulk-synchronous, or an asynchronous version, has no influence on the performance. We reproduce the same experiment twice. The first time by using the original assembly operator, and the second time by using the 100× higher computation intensive operator presented in Section 6.3. In this bulk-synchronous single node experiment, the increased arithmetic intensity is expected to improve the scalability by decreasing the communication ratio and so reducing the communication latencies impact.

#### Standard Arithmetic Intensity

The first experiment using the standard arithmetic intensity kernel running on the EIB use case is illustrated in Figure 7.12. While as expected, the Cilk version of the D&C library scales better than both of the communication libraries, we note that the Intel MPI library exploits more efficiently the shared memory resources of a single compute node. This is not surprising since MPI has been optimized to replace the distributed communications by data movement when running on several processes bound to a same NUMA node. At the opposite, GASPI is mainly designed for distributed communications. Even if two GASPI processes are on a same processor, the communications between them will be handled by the network card. Still, this only results in a 6.9% speedup in favor of MPI compared to GASPI since there are not many communications at that scale.




(a) Speedup compared to the best sequential solution

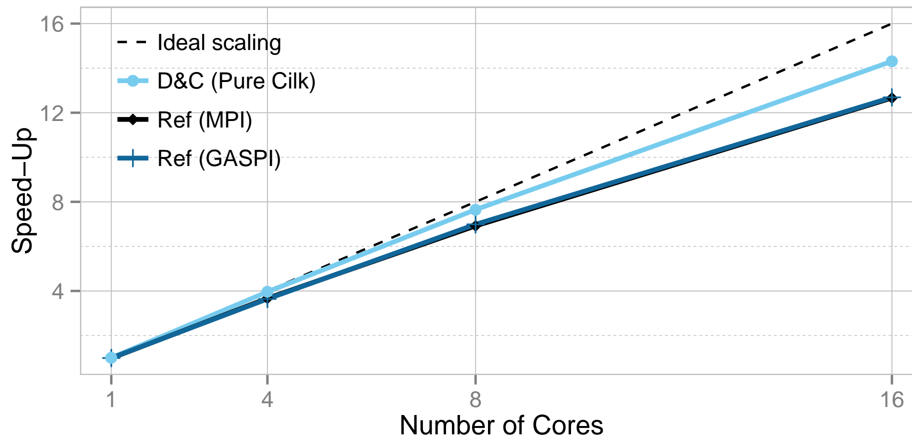



(b) Parallel Efficiency

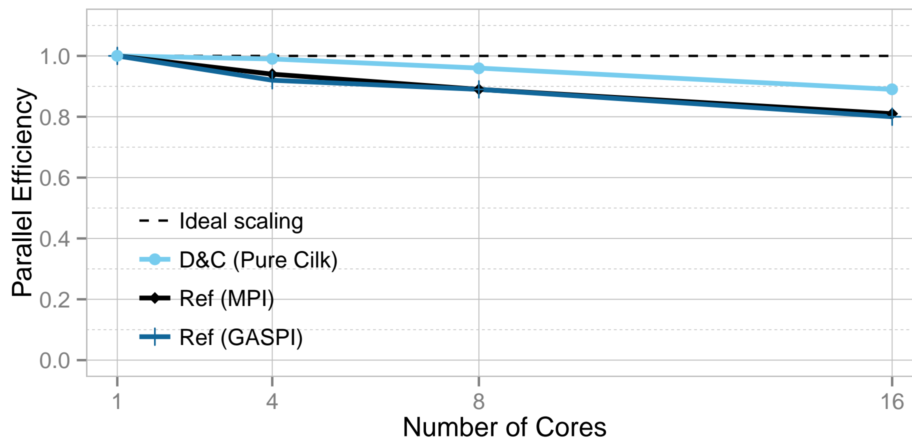
Figure 7.12: Single node experiment with standard arithmetic intensity using the EIB use case.

#### Increased Arithmetic Intensity

In this second experiment, we increase the arithmetic intensity by a factor of 100. As shown in Figure 7.13, the difference between MPI and GASPI completely disappears. The higher ratio of computation compared to communication on a single node execution, hides the performance advantage of MPI compared to GASPI when running on shared memory resources. We can also note that the *Ref (MPI)* and *Ref (GASPI)* versions benefit more from the higher arithmetic intensity than the pure D&C version. Indeed, on a single node, the D&C version execution has no communication, only data sharing. Since D&C already scales almost perfectly by using the original assembly operator with a lower arithmetic intensity, the higher computation ratio is more negligible. The final 12.8% performance advantage of the D&C version compared to pure GASPI or pure MPI is due to the absence of communication and to the improved data locality brought by the D&C permutations.




(a) Speedup compared to the best sequential solution




(b) Parallel Efficiency

Figure 7.13: Single node experiment with 100× increased arithmetic intensity using the EIB use case.

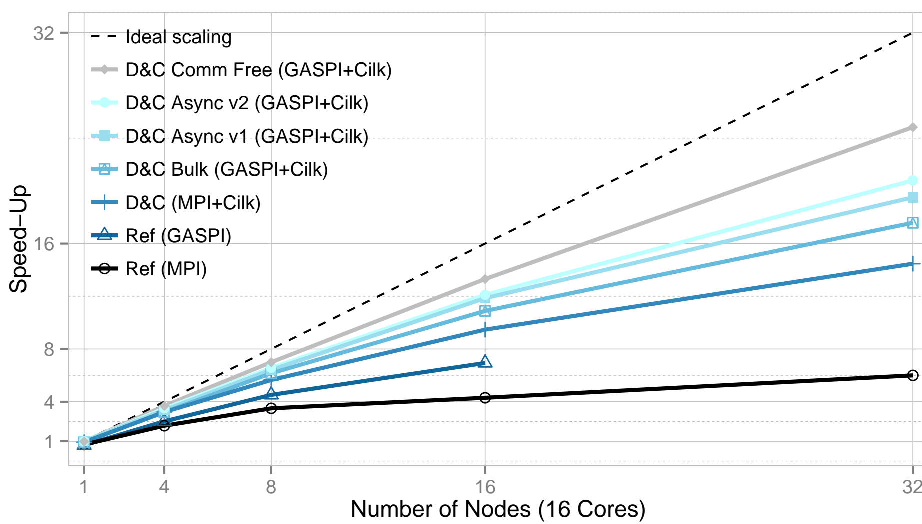
### 7.5.3 Strong Scaling Experiment Using Standard Arithmetic Intensity

The next experiment consists in comparing the six different versions previously described running on up to 512 Sandy Bridge cores. Contrary to the previous experiments made to compare the behavior of the GASPI and MPI libraries on shared memory resources, the following experiments reflect the standard setup for hybrid parallelized applications. This setup consists in a single process per distributed node and a thread per core inside each node.

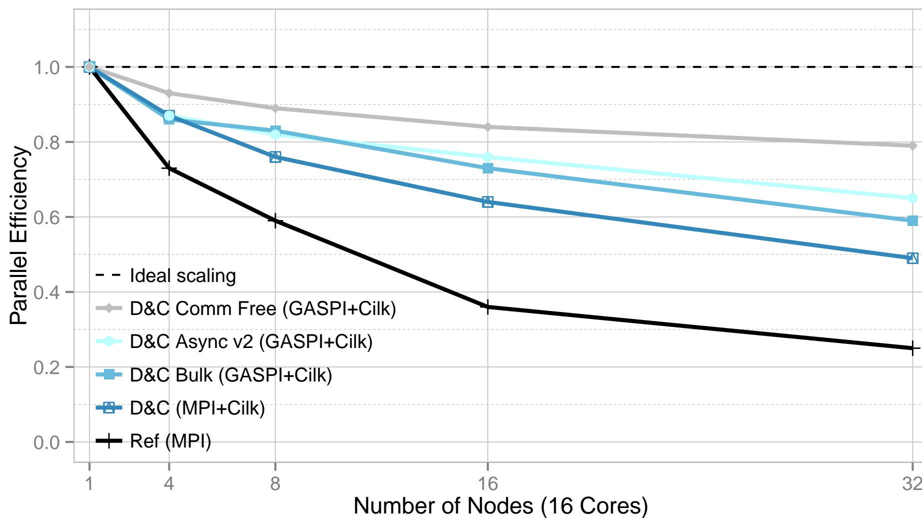
As illustrated in Figure 7.14a, the two pure domain decomposition *Ref (MPI)* and *Ref (GASPI)* versions have the poorest performance. Indeed, at 32 nodes, i.e. 512 processes, there are only 2000 vertices per subdomains and a non negligible part of the mesh is located in the ghost cells: around 60% as seen in Figure 3.8 of Section 3.5. This leads to important data duplications. The *Ref (GASPI)* version has failed to execute up to 512 processes because of its too high memory requirements. The four other versions only use one process per node, therefore 32 subdomains instead of 512. This drastically reduces the data duplication and the number of communications, which leads to better performances.

As seen in Chapter 5, the *D&C (MPI+Cilk)* version is 2.4 times faster than *Ref (MPI)* and ends at

50% of parallel efficiency compared to 25%. However, without changing the communication pattern, this *D&C (MPI+Cilk)* version is overpassed by 21.5% by its counterpart bulk-synchronous version using GASPI one-sided communications. With a simple bulk-synchronous communication scheme using a single GASPI process per distributed node in addition to the D&C library, it is possible to reach 60% of parallel efficiency at 512 cores with only 2000 vertices per core. Our two asynchronous versions with a new multithreaded communication scheme at D&C task level permit to go even further. The *D&C Async v1 (GASPI+Cilk)* version gains additional 11% speedup compared to *D&C (GASPI+Cilk)* with a very close parallel efficiency. And lastly our *D&C Async v2 (GASPI+Cilk)* version brings 18.3% speedup compared to *D&C (GASPI+Cilk)* and attains 65% of parallel efficiency. In the ends, it is 3.47 times faster than the reference version only parallelized through MPI domain decomposition. The better scalability




(a) Speedup compared to the best sequential solution



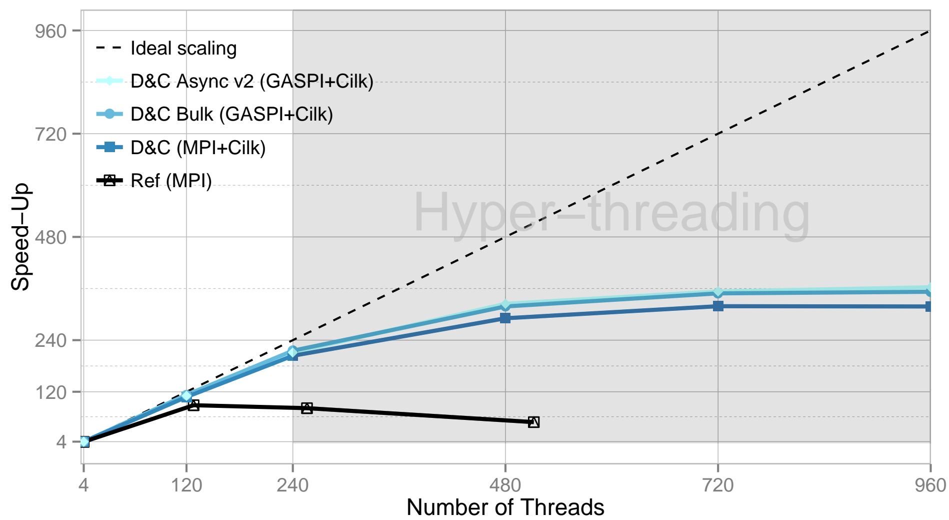

(b) Parallel Efficiency

Figure 7.14: Strong scaling experiment with standard arithmetic intensity using the EIB use case on 512 Sandy Bridge cores.

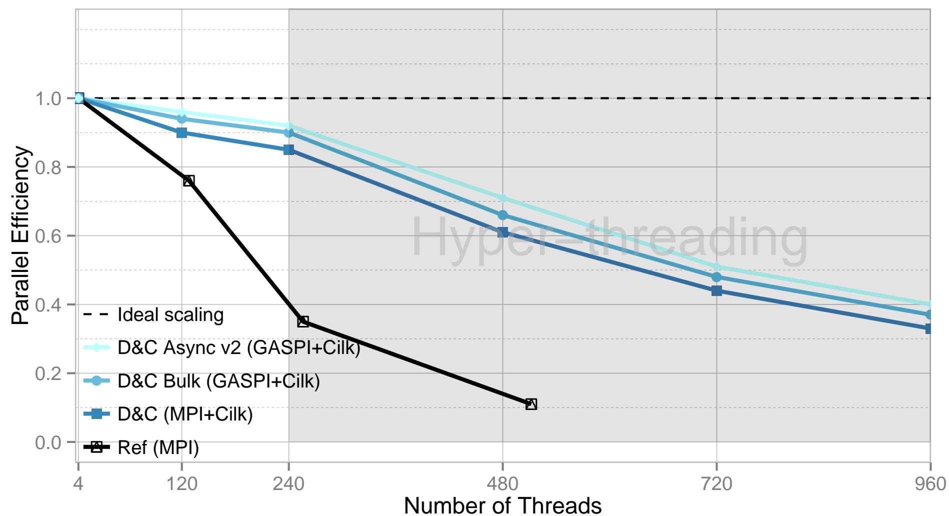
of the second asynchronous version using fixed communication sizes is due to the higher overlap of communication by computation. However, further investigations could be led to increase the proportion of overlapped communications. There is still 19.4% to gain to reach the theoretical maximal speedup represented by the *D&C Comm Free (GASPI+Cilk)* version. This version corresponds to our *D&C Async v2 (GASPI+Cilk)* version in which the communications have been disabled.

### 7.5.4 Multiple KNC Experiment Using Standard Arithmetic Intensity

This experiment compares the four main different approaches used on a very strong scaling experiment using 960 KNC hyper-threads on the 1 million vertices EIB use case, i.e. 1000 vertices per thread.




(a) Speedup compared to the best sequential solution




(b) Parallel Efficiency

Figure 7.15: Strong scaling experiment with standard arithmetic intensity using the EIB use case on 4 KNC manycores.

Moreover, in this case we use the original assembly operator with the low arithmetic intensity to bring out as much as possible the bottlenecks of the different implementations.

The reference implementation *Ref (MPI)*, only parallelized through MPI domain decomposition, still scales up to around half of the 240 physical cores of the 4 KNC with 76% of parallel efficiency. Beyond, the scalability drastically dropped and ends 6.56 times slower than the best D&C version with only 11% of efficiency using 512 subdomains, i.e. the maximal decomposition provided. The three other approaches based on hybrid parallelization scale way better. The standard *D&C (MPI+Cilk)* implementation achieves 85% of parallel efficiency on the 240 physical cores and is 2.5 times faster than *Ref (MPI)*. The similar version *D&C Bulk (GASPI+Cilk)* using GASPI one-sided communications gains additional 5.8% speedup. The performance advantage brought by the asynchronous *D&C Async v2 (GASPI+Cilk)* version is reduced compared to the previous strong scaling experiment made on Sandy Bridge multicores. It is still the fastest version but by only a few percents. Nevertheless, it obtains an impressive 92% of parallel efficiency on the 240 physical cores running with few elements per core and using a very low compute intensive kernel.

### 7.5.5 All-in-One D&C Version

To sum up all the contributions presented during this thesis, we evaluate an all-in-one version containing all the advantages of the different presented versions.

- The D&C recursive tasks creation using the Cilk Plus runtime, bringing concurrency and improving locality as explained in Chapter 5.
- The vectorization of the D&C tasks using our bounded colors strategy tuned for small unstructured mesh partitions presented in Chapter 6.
- And lastly, our asynchronous and multithreaded communication pattern using GASPI one-sided communications detailed in this chapter. We choose the second implementation presented in Section 7.4.2 based on fixed communication size.

We compare in two strong scaling experiments this *D&C Vec Async v2 (GASPI+Cilk)* version to the *Ref (MPI)* reference version and to our three versions implementing the contributions described above. The *D&C (MPI+Cilk)* version corresponds to the first contribution of Chapter 5. *D&C Vec (MPI+Cilk)* corresponds to our second contribution of Chapter 6. And *D&C Async v2 (GASPI+Cilk)* implements the contribution presented in this chapter.

The first experiment is made on the MareNostrum cluster and uses 512 Sandy Bridge cores. The second one is made on 4 KNC of the Salomon cluster. For both of these experiments, we use the 100× increased arithmetic intensity assembly operator to enhance the vectorization performance gains. Moreover, the higher computation ratio should improve the balancing between computation and communication loads and increase the overlap opportunities. This is discussed in further details in Section 7.2.1.

**Parameters Setting**

Before doing the two strong scaling experiments, we analyzed the performance evolution depending on the size of the D&C partitions and of the communication buffers. The Figure 7.16 illustrates this preliminary experiment made on the 32 nodes of MareNostrum. The maximal performance on the 512 Sandy Bridge cores is reached with D&C partitions composed of 150 elements. However, the variation of the communication size does not truly impact the performance.

We applied the same analysis on the 960 threads of the 4 KNC experiment. We empirically found that the size of the communications has once again no strong impact on performance. In the opposite, the size of the D&C partitions is more sensitive. This is illustrated in Figure 7.17. As shown in Chapter 6, the

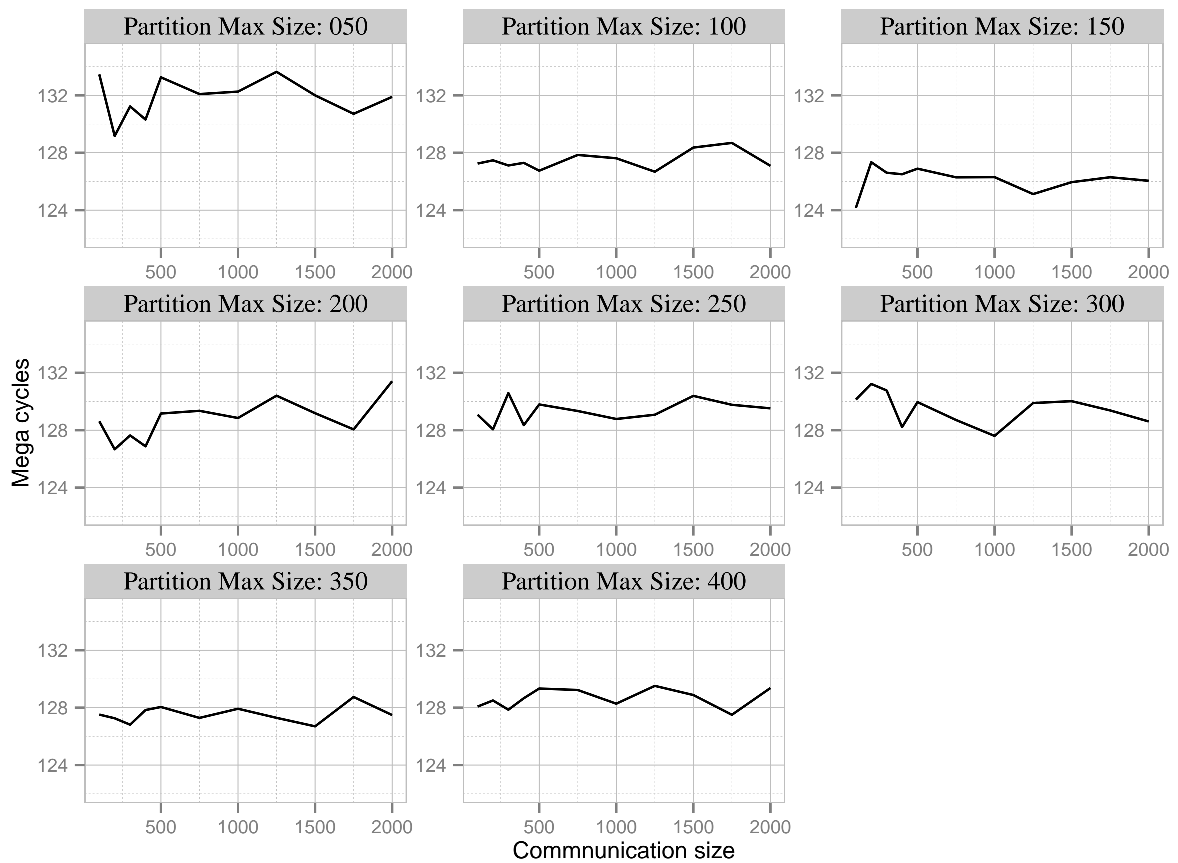



Figure 7.16: Impact of the communications and the D&C partitions size on the performance with 32 Sandy Bridge nodes.

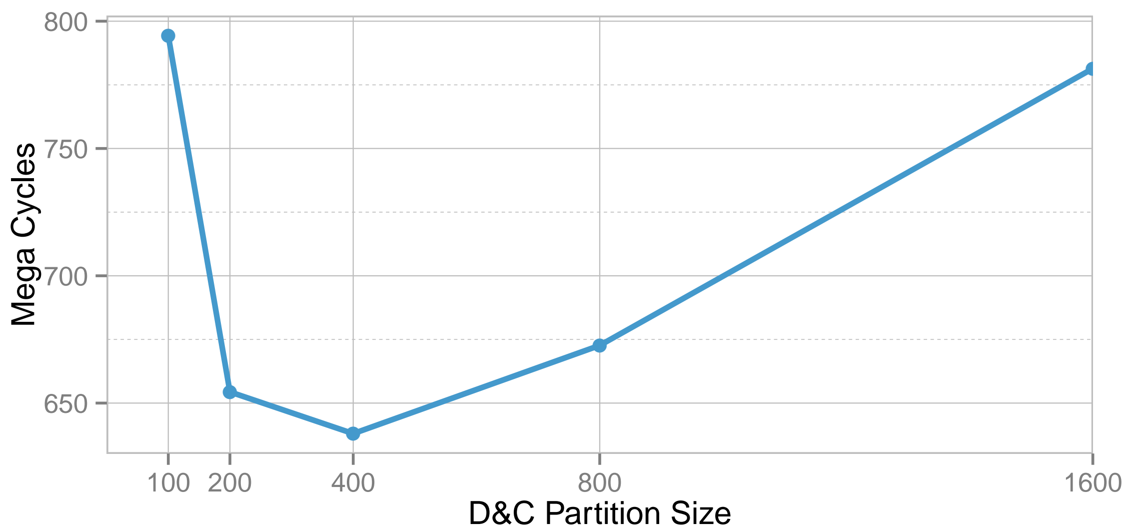
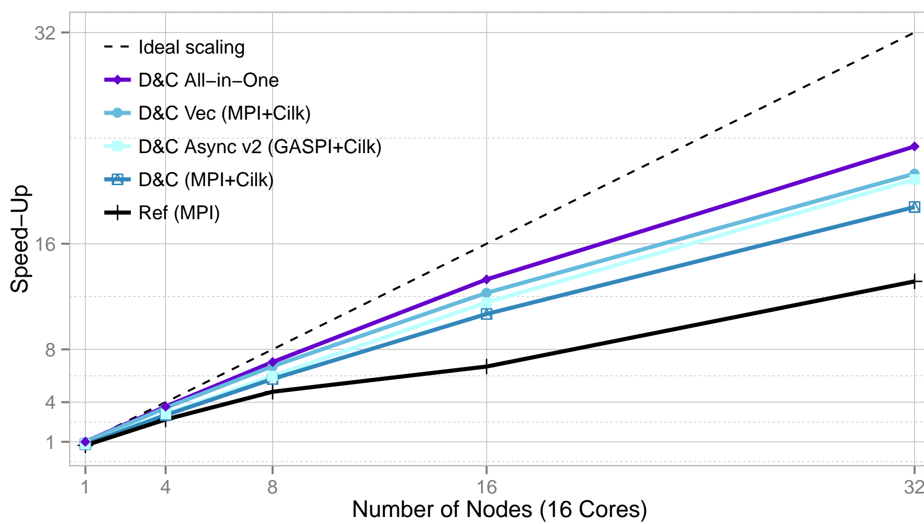



Figure 7.17: Impact of the D&C partitions size on the performance with 4 KNC.

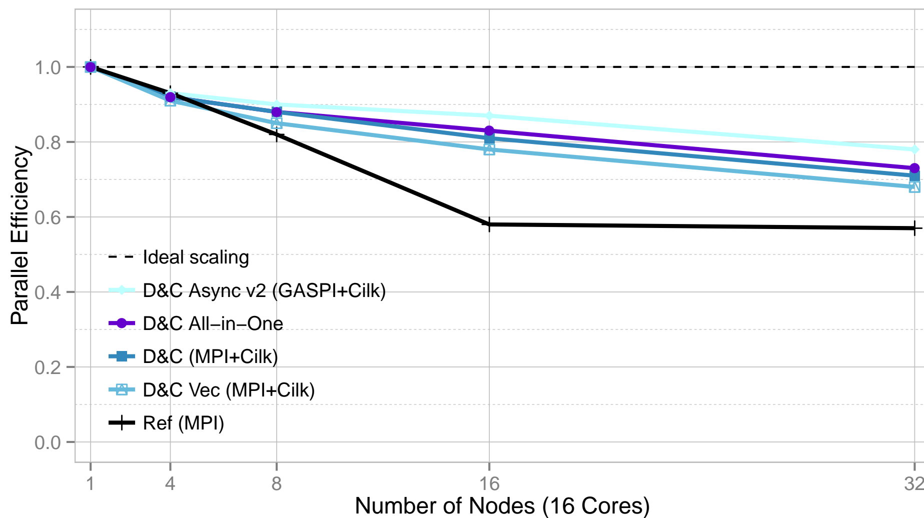
impact of vectorization is more pronounced on the Xeon Phi than on standard Xeon multicores. The D&C partitions composed of 400 elements represent the best trade-off between vectorization opportunities, data locality, and communication overlap.

### Strong Scaling Experiment on Sandy Bridge multicores

Figure 7.18a presents the performance gain provided by each of the presented contributions on the 32 Sandy Bridges nodes. The first contribution, implemented in the *D&C (MPI+Cilk)* version, benefits first of all from the replacement of the 512 processes of *Ref (MPI)* and the resulting communications and data duplications. Moreover, it brings an improved locality and load balancing. This results in 1.43×




(a) Speedup compared to the best sequential solution




(b) Parallel Efficiency

Figure 7.18: Strong scaling experiment with 100× increased arithmetic intensity using the EIB use case on 512 Sandy Bridge cores.

speedup compared to the common implementation only parallelized through MPI domain decomposition. The second contribution, corresponding to *D&C Vec (MPI+Cilk)*, brings additional 13.5% speedup by exploiting the 256 bits AVX vectorial units. Similarly, the asynchronous GASPI one-sided communication pattern represented by the *D&C Async v2 (GASPI+Cilk)* version provides a 11.1% speedup compared to *D&C (MPI+Cilk)*. Merging the previous contributions in the *D&C Vec Async v2 (GASPI+Cilk)* version results in 24.4% speedup compared to *D&C (MPI+Cilk)*. In the end, our all-in-one version overpasses the reference pure MPI version by 77.7%, running on standard Xeon multicores.

As illustrated in Figure 7.18b, the *D&C Async v2 (GASPI+Cilk)* version benefits from the higher scalability with 78% of parallel efficiency at 512 cores and only 2000 vertices per core. The higher scalability is due to the parallelization of the communications and their overlap with computation. However, the vectorization of *D&C Vec (MPI+Cilk)* improves the performance but does not benefit to the scalability. Indeed, the additional memory consumption and control flow overhead start to degrade the performance at that scale of strong scaling experiments. Moreover, the acceleration of the parallel part of the code increases the proportion of sequential code and degrades the parallel efficiency. The all-in-one version is logically located half between the *D&C Vec (MPI+Cilk)* and *D&C Async v2 (GASPI+Cilk)* versions with 73% of parallel efficiency.

#### Strong Scaling Experiment on KNC manycores

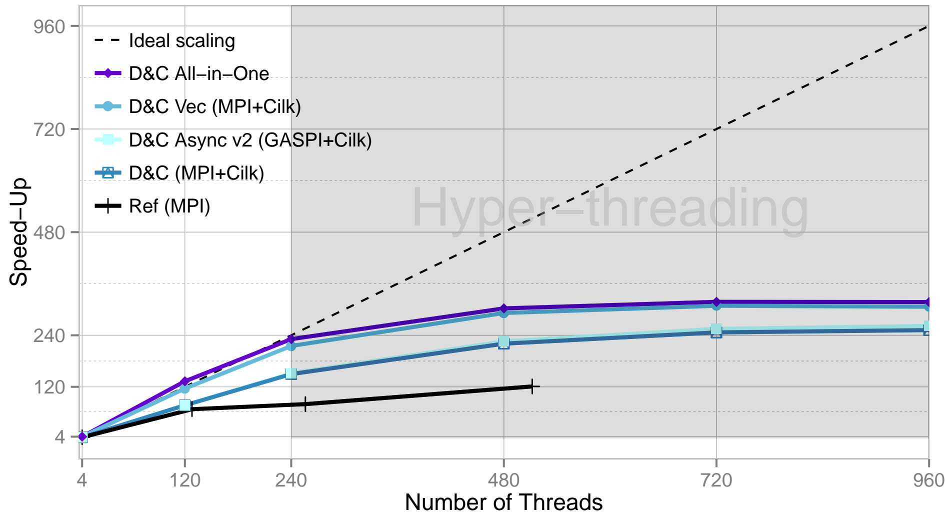
We observe in Figure 7.19 similar behavior on the KNC manycore architecture, excepted that the vectorization has a stronger impact on KNC than on Sandy Bridge multicores. The GASPI one-sided asynchronous communications bring only 4% improvement compared to *D&C (MPI+Cilk)* while *D&C Vec (MPI+Cilk)* is 21.6% faster. Bringing them together, the all-in-one version is 25.9% faster than the original D&C implementation using the 960 hyper-threads and 54.4% on the 240 physical cores. The reference *Ref (MPI)* version, limited to 512 subdomains, is 2.9 times slower than *D&C Vec Async v2 (GASPI+Cilk)* on the physical cores.

Concerning the scalability metric, the two vectorized versions reach an impressive 96% of parallel efficiency running on the 240 physical cores with 4000 vertices per core. This higher scalability compared to the 87% obtained on 256 Sandy Bridge cores is due to the poor sequential performance of the KNC and the strong needs of memory and bandwidth of the vectorized computation kernels. This results in a 1.11$\times$ super-linear speedup moving from 1 to 30 cores per KNC with the *D&C Vec Async v2 (GASPI+Cilk)* version.

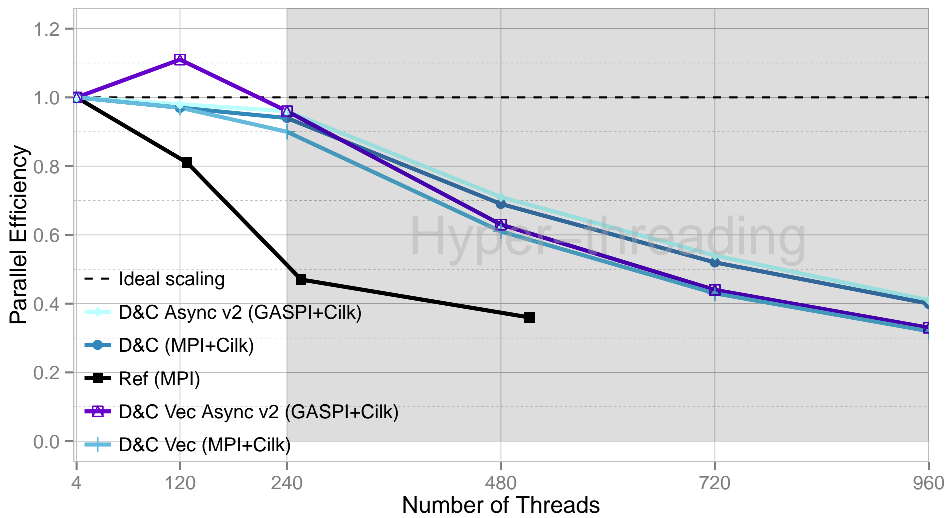
The maximal performance is reached with 180 hyper-threads per KNC by using our all-in-one version. The 4 KNC performance is roughly equivalent to the one achieved by using 6 nodes of the MareNostrum cluster, which corresponds to 96 Sandy Bridge cores.

## 7.6 Conclusion

This chapter presents our last contribution, a tuned communication pattern aimed at exploiting the D&C task parallelism to communicate and to overlap communication by computation. We replace the original bulk-synchronous halo exchange using MPI two-sided communications used in most FEM applications, by an asynchronous version using GASPI one-sided communications and taking advantage of recent RDMA interconnects. The use of GASPI communications forces us to rethink the communication pattern in the right way. The interface values computed by the D&C tasks are appended to the communication buffer which corresponds to the appropriate neighbor process. The task that reaches the chosen communication size, directly writes the values packed in the communication buffer, starting from previous communication to the remote process memory through RDMA. The write is non blocking and handle by the network




(a) Speedup compared to the best sequential solution




(b) Parallel Efficiency

Figure 7.19: Strong scaling experiment with 100× increased arithmetic intensity using the EIB use case on 4 KNC.

card. Therefore, the thread involved in the communication can after the write call, directly continues its execution of the remaining D&C tasks in parallel to the data transfer.

This approach enables many additional sources of parallelism. The gathering and scattering operations are handled in parallel. The communications are also triggered in parallel and handled by the network parallel resources releasing the constraints on the CPU. Lastly, the communications are executed in parallel to the computation. However, in strong scaling experiments, the overlap opportunities are reduced since the computation part highly decreases. In the opposite, the parallel communications and gather/scatter operations become critical.

The contributions proposed in this chapter, results in an impressive 3.47× speedup compared to the state-of-the-art MPI domain decomposition approach used in many FEM applications. This speedup is

obtained on a low intensive computation kernel launched on 512 Sandy Bridge cores with only 2000 vertices per core. Running the same experiment on 4 Xeon Phi manycores, our asynchronous D&C approach achieves 92% of parallel efficiency on their 240 physical cores and is 2.58 times faster than the pure MPI approach.

Lastly, we have merged the different contributions proposed during this thesis in a single all-in-one version, which combines the D&C approach proposed in Chapter 5, the vectorization detailed in Chapter 6, and the new communication pattern described above. We experimented it with a more intensive computation kernel and concluded on the performance improvement brought by the different contributions. This all-in-one version reaches 96% of parallel efficiency on the 240 cores of 4 Xeon Phi KNC and a 2.9$\times$ speedup compared to pure MPI. The 4 KNC execution provides performance similar to 96 Sandy Bridge cores.

# Conclusion and Perspectives

## Thesis Contributions

For years, applications had relied on the ever growing sequential performance of CPU cores. We believe that nowadays and in the future, it will be necessary to modernize the applications for a better parallel efficiency, concurrency, and locality, to subscribe to the next *free lunch* provided by the core count increase. This modernization is a costly process, but the difference of performance observed on current multicores and manycores, e.g. Xeon Phi, already makes it worthwhile.

In this thesis, we propose an holistic approach, detailed in Figure 8.1, to efficiently parallelize irregular applications based on finite element methods and dealing with highly unstructured meshes. We have demonstrated that the parallelization technics developed during this thesis efficiently scale on modern multicore CPUs and modern manycore accelerators. We have also built a proto-application, named Mini-FEM, representative of the assembly step of FEM real world applications. Mini-FEM has eased the development, the debugging, and the experimentation of the algorithms developed during the thesis. Our different contributions have been implemented in the D&C library, used on top of Mini-FEM. We have released our D&C library and the Mini-FEM proto-application open-source, under the LGPL 3.0 licence, to disseminate widely to other users working on similar unstructured meshes. The current implementation is already in used in the production version of DEFMESH and AETHER, two industrial CFD softwares from Dassault Aviation. The integration in other parts of the very large computer assisted aircraft design framework at Dassault Aviation is in progress, with the objective to be prepared for the next generation of systems.

From a software engineering point of view, the lessons learned from our experiments are twofold. At

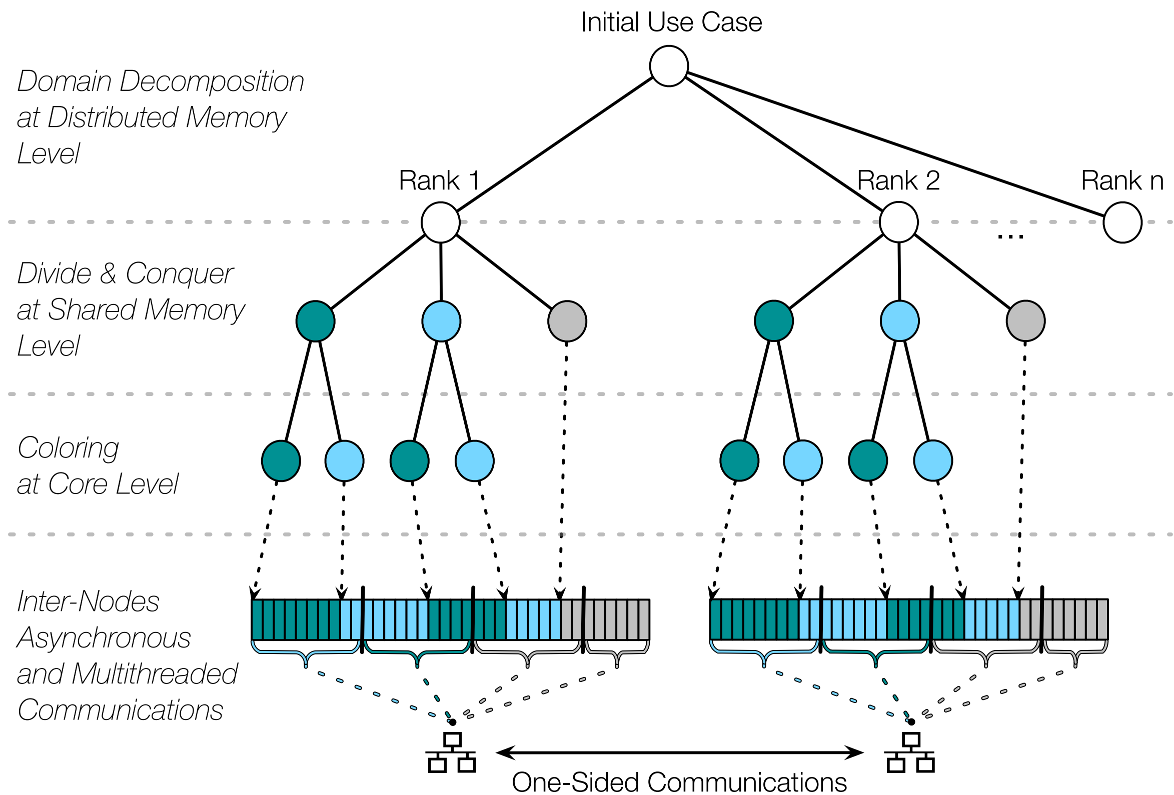



Figure 8.1: Summary of the contributions proposed during the thesis.

first, from an algorithmic point of view, focusing on architecture oblivious design and then, introducing architecture aware parameters, produces easily re-targetable solutions with tuning parameters. Secondly, it is possible to separate the physic modeling concerns and the parallelization implementation in different units, using different languages. In our case, the original Fortran code is interfaced to our C++ and Cilk Plus implementation of the parallelism with minor intrusion.

The main ideas developed during this thesis were to recursively partition meshes used by irregular FEM applications using the divide and conquer approach, and to use these partitions to parallelize the communications at distributed level, to exploit shared parallel resources at node level, and lastly to take advantage of the vectorial resources at core level. All these contributions are summarized in Figure 8.1. The top part of the figure corresponds to the domain decomposition approach used between processes mapped to the distributed compute nodes. This approach is used in most FEM applications and we have built our contributions on its top. Despite the fact that the pure MPI domain decomposition approach scales well on current scale of data-set and node count, our D&C library proves to be more efficient and performant, paving the way to future manycore systems.

At scale, the final speedup brought by our D&C hybrid approach on 512 Xeon cores reaches 3.47× compared to the state-of-the-art domain decomposition method using MPI two-sided communications. The Xeon Phi manycore architecture enables even larger speedups with a 6.56× compared to MPI domain decomposition on 4 KNC, a performance similar to 96 Xeon cores, and 96% parallel efficiency on the 240 physical cores.

### Divide & Conquer at Shared Memory Level

The second part of Figure 8.1 corresponds to our first contribution and refers to the Chapter 5. It consists in recursively partitioning each mesh subdomain in several partitions using the divide and conquer approach. This D&C approach brings many advantages and provides all the characteristics necessary to scale on manycore systems.

- *Concurrency.* The recursive sharing naturally exposes high concurrency. As long as the data-set is large enough, it is possible to produce a deeper recursive tree to get more concurrency and match the higher requirements of manycore systems.

- *Load-balancing.* Using a large number of recursive decomposition levels, leverages efficient work-stealing for dynamic load-balancing and offsets the irregularity of METIS partitioning.

- *Local synchronization.* Unlike the coloring approach, no global barriers are needed. There is only one local synchronization per task between its two local children. This results in $log(n)$ synchronizations on the critical path, which becomes quickly negligible with the increasing mesh sizes.

- *Data locality.* The data locality is improved by recursively reordering the data. At first, the nodes and elements are permuted in order to consecutively place them inside the left, right and separator leaves. Secondly, since the task distribution follows a recursive tree, the neighboring leaves are contiguously stored. Therefore, data locality is improved both inside a domain and between the neighboring domains. This leads to a better locality than the original ordering using the Reverse Cuthill-McKee approach [159]. Moreover, leaves dataset can be downsized to fit into caches.

The experiments made with the D&C library on 4 Intel Xeon Phi based on KNC architecture result in a perfect scaling on the 240 physical cores. D&C achieves 92% parallel efficiency and is 2.5 times faster than the pure MPI approach. It attains a final speedup of 360× compared to its sequential execution and obtains similar performance to 33 Intel E5-2665 Xeon Sandy Bridge cores. Moreover, even without

exploiting thread level parallelism, the permutations brought by the D&C library significantly improve the locality, the scalability, and the execution time.

### Coloring at Core Level

The third part of Figure 8.1 illustrates our second contribution presented in Chapter 6. This contribution targets the large vectorial resources contained in modern multicores and manycores. We use the coloring approach to generate independent data vectors. The coloring is applied to each D&C parallel task of the first contributions. Since the D&C tasks consist of very small partitions of unstructured meshes, the resulting vectorization ratio is low. We propose a new coloring heuristic especially tuned for small unstructured partitions. Our first criteria is to increase the vectorization ratio. Therefore, we relax the constraint of classical coloring approaches, which consists in minimizing the number of colors used. This helps us to increase the number of vectors dimensioned to the target vector length since as soon as a vector is full, it is no longer extended. The remaining values can be reused to fill other vectors.

Full vectors are contiguously stored at first and are followed by the remaining values which cannot form a full vector. This allows to not store the color of the vectorized values, but only the offset between full and partial vectors. Moreover, the full vectors are aligned to the loop iteration frontiers. This way, the loop over colors can be removed. The first part of the loop iterates by chunk dimensioned to the vector size until reaching the offset. Then, the remaining values are sequentially executed.

We also propose a vectorization model used to predict the code vectorization ratio depending on the target vector size, the number of values contained in full vectors, and the remaining values. This leads us to the conclusion that larger vectors do not necessarily lead to bigger speedups. Smaller vector sizes allow higher vectorization ratios, especially when dealing with small partitions of unstructured meshes. And this higher ratio compensates the loss of data parallelism. Therefore, we can predict that future 1024 bits vector size will be problematic for unstructured meshes, even if the cache size is not reduced. Given our experiment, this contribution has led to a 13% speedup over our non-vectorized D&C version on actual Xeon multicores and a 20% speedup on the KNC manycore.

### Inter-Nodes Asynchronous and Multithreaded Communications

Lastly, the bottom part of Figure 8.1 concerns our last contribution dealing with one-sided asynchronous communications. This refers to the Chapter 7. In this chapter, we have replaced the state-of-the-art MPI bulk-synchronous halo exchange based on two-sided communications. Instead, we use the GASPI library based on the PGAS model. GASPI provides a collection of asynchronous one-sided operations using RDMA interconnect and remote completion.

We use it to build two different communication patterns both using the D&C parallel tasks to parallelize the communications and recover them with computation. For these two versions, the idea is to directly write to the neighbors their corresponding interface values as soon as they are ready instead of waiting for the update of the latest interface value before starting to communicate. This way, when the last interface value is ready to be sent, most of the communications are already achieved. However, since there may be lots of interface values, this would led to a large amount of small communications. The differences between the two versions come from the approach used to reduce the number of communications.

In the first version, we fix an empirically chosen level in the D&C tree, called the communication level, below which no communication is done. During our experiments, we determined that a third of the tree height is our optimal. The tasks located above this level, which are the last to update a set of interface values, handle the communications of their whole subtree. However, this approach has two main drawbacks. The first one is the unbalanced between communication sizes. Indeed, the interface values are irregularly spread among almost all the D&C tasks. The second one is the reduction of the overlap

possibilities. The less communications we want, the more computation will be ended before starting to communicate.

Therefore, we have developed a second version in which the fixed communication level is replaced by a fixed communication size. In this version, a communication buffer is created per interface. The last tasks that update an interface value, push it to the appropriate communication buffer and continue their execution. Once a task reaches the predefined communication size, it sends the whole chunk of values and updates the communication pointer. One of the advantage of this approach is that each communication has a balanced size, excepted the communications containing the last remaining interface values. Another advantage are the increased communication overlap opportunities. Indeed, interface data are sent as soon as their communication buffer is full. This last version provides a 1.44× speedup over the MPI two-sided version of the D&C library and an impressive 3.47× speedup compared to the pure MPI domain decomposition approach by using 512 Sandy Bridge cores.

## Perspectives

Our ongoing developments are many folds and consist of two main categories: the improvement and the development of new features in our D&C library, and the integration of the library in other industrial and academic large scale applications.

### Evolutions of the D&C Library

One of the targeted evolutions is to implement padding for incomplete vectorization to improve flops performance on Intel Xeon Phi. The idea is to improve the way we store the colors generated by our bounded coloring strategy, in order to enable vectorization of incomplete vectors. The actual execution of the coloring, detailed in Section 6.2.5, consists in vectorially executing at first the full vectors, and then sequentially looping over the remaining values. To reduce this sequential loop, it is possible to store partial vectors in decreasing order and to aligned them on the loop iteration frontiers by using padding. The appropriate padding values still need to be defined in order to not impact the numerical results. Another alternative would be to use the vectorization masks available in the Intel Xeon Phi. However, even by using the vectorization masks, it would be better to use padding in order to align the memory loads. With this approach, we do not have to store the colors used, but contrary to the actual implementation of the bounded coloring, we need to store as much offsets as there are different vector sizes. This approach based on variable vector sizes is illustrated in Figure 8.2.

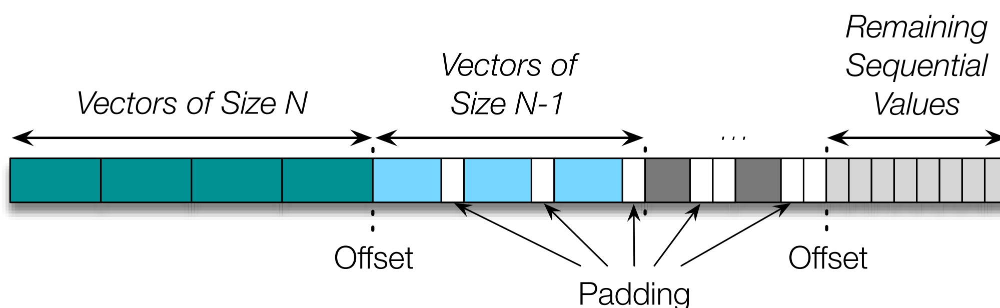



Figure 8.2: Bounded coloring approach using decreasing vector size and memory padding.

We also plan to replace the METIS graph partitioner used during the creation of the D&C tree and to replace it by our own partitioner. The goal is to produce a simpler but quicker partitioner for the shared memory level. Indeed, all the partitions including the separators which contain the elements on the cuts are recursively bisected until reaching few tens of elements. Therefore, we do not need to minimize as most as possible the number of cutting edges. Moreover, the small generated partitions are dynamically balanced through the work-stealing scheduler of the Cilk Plus runtime. A slight imbalance between the partition sizes is then not critical.

Furthermore, it would be possible to directly partition an initial single domain use case. This way, we could handle ourselves the interface data structures between distributed domains and optimize them to be better suited to our communication pattern using GASPI one-sided communications at task level.

Similarly, we want to explore the behavior of the Threading Building Blocks (TBB) library [14] compared to Cilk Plus. Cilk Plus is an extension of the C and C++ languages and is limited to these languages. Moreover, it is limited to x86 architecture and is therefore not compatible with every architecture such as the IBM Blue Genes. In contrast, the TBB library approach allows to be recompiled by any compiler and to be adapted to various platforms with minor changes. This wider compatibility may result in a small loss in performance that we want to characterize.

Another evolution in progress, is the development of a dynamic scheduler at distributed level, named TIny Task Unified Scheduler (TITUS), within the LI-PARAD laboratory of the UVSQ. As a future work, the TITUS scheduler is intended to be used on top of the D&C library in order to equilibrate the unbalancing among distributed nodes, similarly to what is done within the nodes with Cilk Plus.

Lastly, another future work would be to enhance the Mini-FEM proto-application with a representative FEM solver. This would help to estimate the future evolutions of the D&C library and what could be added to handle efficiently shared memory parallelized solvers. We plan to enhance the actual CG solver used in the DEFMESH application by using our D&C strategy. For now, it is based on MPI domain decomposition and OpenMP loop parallelization and it has a low efficiency. Integrating the solver within the D&C library would allow to also integrate the communication layer in the library and to provide a new level of abstraction to the end-user. Indeed, we believe that the future of HPC applications executed on increasingly complex systems is to concentrate on expressing the science. The complexity of the parallelism should rely on finely tuned runtime systems with minimal interfaces.

## Industrial and Academic Collaborations

Alternatively to the enhancement of our D&C library, it is also important to develop new partnerships with application developers. This thesis has been made in collaboration with Dassault Aviation. There are other parts of their applications, such as the solver part, that could be improved using our D&C library.

Additionally, in the next few years, we plan to experiment our D&C library in other CFD applications. One of the targets is YALES2, a large academic application from the COmplexe de Recherche Interprofessionnel en Aérothermochimie (CORIA) [174]. As a first step, we will experiment the D&C library on the proto-application of YALES2 developed by Guillet *et al.* as part of the FP7 EXA2CT european project [68]. Similarly to the work made on the AETHER application from Dassault Aviation and described in Chapter 4, the YALES2 proto-application should allow a quick integration and debugging of the D&C library. The experiments made on the proto-application are supposed to be representative of what would happen on the full size YALES2 application and ease the decision for the final integration.

# Conclusion

In this thesis, we propose strong scientific results which have led to several publications including a top international conference. Two industrial applications have been strongly improved by using our D&C holistic approach implemented in an open-source library. The creation of a proto-application has also been a good time investment to ease the development of our new parallelization strategies and to accelerate the code modernization process. The work produced during this P.h.D. thesis will be continued with two others P.h.D. thesis exploring complementary approaches.

# List of Figures


1 Example of 3D unstructured mesh. . . . 3
2 Speedup according to the proportion of parallelized code on a 256 cores execution. . . . 6
3 Typical speedup and parallel efficiency depending on the level of code optimization. . . . 7

1.1 Representation of the Von Neumann architecture. It is composed of an Arithmetic-Logic Unit (ALU), a control unit, a main memory, and I/O operations. . . . 12
1.2 Cluster representation with two distributed NUMA nodes, two processors per node, and two cores per processor. . . . 14
1.3 Nvidia Maxwell GM200 Architecture. . . . 16
1.4 Memory architecture of the Intel KNC. . . . 17
1.5 Presentation of the Intel Sandy Bridge core architecture from Intel Developer Forum. . . 20
1.6 Presentation of the Curie supercomputer taken from CEA website [41]. . . . 21
1.7 Presentation of the Salomon supercomputer and of its 7D enhanced hypercube topology taken from IT4I website [42]. . . . 21
1.8 Presentation of the MareNostrum supercomputer. . . . 22

2.1 Comparison between MPI standard copy-based mechanism and zero-copy mechanism using remote memory accesses. . . . 25
2.2 MPI two-sided synchronization modes. . . . 26
2.3 MPI two-sided communication protocols. . . . 27
2.4 Set of MPI global communications. . . . 28
2.5 MPI one-sided operations. . . . 29
2.6 MPI one-sided synchronization modes. . . . 30
2.7 *PGAS model* - Each process owns a local part of the segment and has a direct access to the other remote parts of the segment through the RDMA interconnect. . . . 33
2.8 Presentation of *GASPI_write*, *GASPI_notify*, and *GASPI_write_notify* one-sided operations. 34
2.9 Comparison between different communication patterns on 48 two-socket nodes of 12-core Xeon Ivy Bridge EP. With courtesy of Simmendinger *et al.*. . . . 36
2.10 Bulk-synchronous fork-join parallelization model using 3 threads per parallel region. . . 37
2.11 OpenMP 3.0 task workqueuing model. . . . 39
2.12 Cilk Plus work-stealing scheduler. . . . 41
2.13 Comparison between CilkView scalability reports. . . . 42

3.1 2D illustration of a basic iteration over elements in FEM application working on a regular 2D mesh. . . . 44
3.2 Presentation the COO, CSR, and CSC formats on a $4 \times 4$ sparse matrix. . . . 45
3.3 Presentation of the CSB format on a $4 \times 4$ sparse matrix with block size $\beta = 2$. . . . 46
3.4 Presentation the SELL-$C$-$\sigma$ sparse matrix format. . . . 47

3.5 2D illustration of domain decomposition. . . . . . . 48
3.6 *Halo exchange* - At each time step, the halos are exchanged between neighbor subdomains. 48
3.7 2D illustration of the *k*-way partitioning process. . . . . . . 49
3.8 Ratio of the number of interface nodes to duplicate and communicate compared to the number of subdomains on the EIB use case. . . . . . . 50
3.9 2D illustration of mesh coloring. . . . . . . 52
3.10 Example of a vectorized sum over 4 integer values. . . . . . . 55

4.1 Code Modernization Scenario. . . . . . . 62
4.2 Illustrations of the 3D EIB fuel tank position optimization use case. . . . . . . 64

5.1 D&C recursive tree. The left and right partitions are executed in parallel before their separator elements on the cut. . . . . . . 69
5.2 D&C permutations. For each cut, elements are reordered in left, right and separator partitions. . . . . . . 70
5.3 Impact of the D&C reordering on the CSR matrix associated to the LM6 use case. . . . . 71
5.4 Percentiles distribution of the distances to the diagonal with and without the D&C permutations on the F7X use case. . . . . . . 71
5.5 Comparison of the L3 cache misses between pure MPI domain decomposition, hybrid MPI + coloring, and hybrid MPI + D&C, using the Mini-FEM proto-application . . . . . 72
5.6 D&C CSR sparse matrix storage format. CSR values are contiguously stored following the left, right, and separator partition order. . . . . . . 73
5.7 Integration of D&C and coloring in the FEM pipeline with constant mesh topology. . . . 74
5.8 FEM assembly speedup and parallel efficiency comparison on the EIB use case running on a single node of the MareNostrum cluster. . . . . . . 79
5.9 FEM assembly speedup and parallel efficiency comparison on the EIB use case running on up to 32 nodes (512 cores) of the MareNostrum cluster. . . . . . . 80
5.10 FEM assembly speedup and parallel efficiency comparison on the EIB use case running on a Xeon Phi (KNC) of the Salomon cluster. . . . . . . 81
5.11 FEM assembly speedup and parallel efficiency comparison on the FGN use case running on 4 Xeon Phi (KNC) of the Salomon cluster. . . . . . . 82
5.12 Solver execution time and speedup using D&C permutations. . . . . . . 83
5.13 FEM assembly speedup and parallel efficiency comparison between Cilk Plus and OpenMP 3.0 tasks. Mini-FEM running the EIB use case on Anselm cluster. . . . . . . 84
5.14 FEM assembly speedup and parallel efficiency comparison between Cilk Plus and OpenMP 3.0 tasks. Mini-FEM running the EIB use case on an Intel Xeon Phi KNC. . . . 85
5.15 FEM assembly speedup and parallel efficiency comparison between Cilk Plus and OpenMP 3.0 tasks. Mini-FEM running the FGN use case on 4 Intel Xeon Phi KNC. . . . 86
5.16 Performance comparison between *Ref (Comm Free)*, *D&C (MPI+Cilk)*, and *D&C (MPI+OpenMP)* on AETHER with F7X mesh on 1024 Sandy Bridge cores. . . . . . . 87

6.1 Cilk Plus array notation examples. . . . . . . 91
6.2 Vectorization ratio for various leaf and vector sizes. . . . . . . 94
6.3 Permutation of the elements after bounded coloring. Only the offset needs to be stored to execute the vector loop. . . . . . . 96
6.4 Trade-off representation between memory locality, structuredness and vector length. . . . 96
6.5 Proportion of time impacted by the vectorization. . . . . . . 97
6.6 Color distribution sorted by decreasing size using our bounded coloring strategy. . . . . 98

6.7 FEM assembly speedup compared to the best sequential time and parallel efficiency on a single Sandy Bridge two-socket node. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100
6.8 FEM assembly speedup compared to the best sequential time and parallel efficiency with arithmetic intensity increased by a factor of 100×. . . . . . . . . . . . . . . . . . . . . . . . 101
6.9 FEM assembly speedup and parallel efficiency using 100× increased arithmetic intensity and running on 512 Sandy Bridge cores. . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
6.10 FEM assembly speedup and parallel efficiency with standard arithmetic intensity on a single KNC. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
6.11 FEM assembly speedup and parallel efficiency with 100× increased arithmetic intensity on a single KNC. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
6.12 FEM assembly speedup and parallel efficiency with 100× increased arithmetic intensity on 4 KNC. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105

7.1 Potential gain brought by overlapping communication by computation. . . . . . . . . . . 108
7.2 Synthetic illustration of attainable speedups with a complete overlap of communication and computation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
7.3 Incomplete overlap with equilibrated computation and communication loads. . . . . . . 110
7.4 *Bulk-synchronous approach* - The halo exchange communication phase starts after the end of the assembly computation phase. . . . . . . . . . . . . . . . . . . . . . . . . . . 110
7.5 *Asynchronous approach* - As soon as a piece of data to communicate is ready, it is sent. Once the D&C parallel region comes to its end, most of the communications are completed and the data unpacking can start with minimal waiting times. . . . . . . . . . . . . . . . 111
7.6 *GASPI Async v1* - Illustration of the communication level version using GASPI one-sided in the D&C tasks located above the level. . . . . . . . . . . . . . . . . . . . . . . . . 112
7.7 *GASPI Async v2* - Illustration of the bounded communication size using GASPI one-sided inside any D&C tasks attaining the communication size. . . . . . . . . . . . . . . . . 114
7.8 Number of communications depending on their size. . . . . . . . . . . . . . . . . . . . 116
7.9 Impact of the communication and the D&C partition size on the performance. . . . . . . 117
7.10 Impact of the communication and the D&C partition size with a hundred times more work. 117
7.11 Performance comparison between one process per node and one process per socket. . . . 119
7.12 Single node experiment with standard arithmetic intensity using the EIB use case. . . . . 120
7.13 Single node experiment with 100× increased arithmetic intensity using the EIB use case. 121
7.14 Strong scaling experiment with standard arithmetic intensity using the EIB use case on 512 Sandy Bridge cores. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122
7.15 Strong scaling experiment with standard arithmetic intensity using the EIB use case on 4 KNC manycores. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 123
7.16 Impact of the communications and the D&C partitions size on the performance with 32 Sandy Bridge nodes. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 125
7.17 Impact of the D&C partitions size on the performance with 4 KNC. . . . . . . . . . . . 125
7.18 Strong scaling experiment with 100× increased arithmetic intensity using the EIB use case on 512 Sandy Bridge cores. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 126
7.19 Strong scaling experiment with 100× increased arithmetic intensity using the EIB use case on 4 KNC. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 128
8.1 Summary of the contributions proposed during the thesis. . . . . . . . . . . . . . . . . . 133
8.2 Bounded coloring approach using decreasing vector size and memory padding. . . . . . 136

# BIBLIOGRAPHY


[1] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, “A high-performance, portable implementation of the mpi message passing interface standard,” *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.

[2] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. MIT press, 1999, vol. 1.

[3] G. E. Blelloch and B. M. Maggs, “Parallel algorithms,” in *Algorithms and theory of computation handbook*. Chapman & Hall/CRC, 2010, pp. 25–25.

[4] D. Grünewald and C. Simmendinger, “The gaspi api specification and its implementation gpi 2.0,” in *7th International Conference on PGAS Programming Models*, vol. 243, 2013.

[5] C. Simmendinger, M. Rahn, and D. Gruenewald, “The gaspi api: A failure tolerant pgas api for asynchronous dataflow on heterogeneous architectures,” in *Sustained Simulation Performance 2014*. Springer, 2015, pp. 17–32.

[6] C. Simmendinger, J. Jgerskpper, R. Machado, and C. Lojewski, “A pgas-based implementation for the unstructured cfd solver tau,” *PGAS11, Galveston Island, Texas, USA*, 2011.

[7] R. Chandra, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[8] G. J. Gorman, J. Southern, P. E. Farrell, M. D. Piggott, G. Rokos, and P. H. J. Kelly, “Hybrid openmp/mpi anisotropic mesh smoothing,” *Procedia CS*, pp. 1513–1522, 2012.

[9] D. Schmidl, T. Cramer, S. Wienke, C. Terboven, and M. S. Müller, “Assessing the performance of openmp programs on the intel xeon phi,” in *Proceedings of the 19th International Conference on Parallel Processing*, ser. Euro-Par’13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 547–558.

[10] X. Guo, M. Lange, G. Gorman, L. Mitchell, and M. Weiland, “Developing a scalable hybrid mpi/openmp unstructured finite element model,” *Computers & Fluids*, vol. 110, pp. 227–234, 2015.

[11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, “Cilk: An efficient multithreaded runtime system,” *Journal of parallel and distributed computing*, vol. 37, no. 1, pp. 55–69, 1996.

[12] M. Frigo, C. E. Leiserson, and K. H. Randall, “The implementation of the cilk-5 multithreaded language,” in *ACM Sigplan Notices*, vol. 33, no. 5. ACM, 1998, pp. 212–223.

[13] C. E. Leiserson, “The cilk++ concurrency platform,” *The Journal of Supercomputing*, vol. 51, no. 3, pp. 244–257, 2010.

[14] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O’Reilly Media, Inc.", 2007.

[15] C. Cecka, A. J. Lew, and E. Darve, "Assembly of finite element methods on graphics processors," *International journal for numerical methods in engineering*, vol. 85, no. 5, pp. 640–669, 2011.

[16] G. Markall, A. Slemmer, D. Ham, P. Kelly, C. Cantwell, and S. Sherwin, "Finite element assembly strategies on multi-core and many-core architectures," *International Journal for Numerical Methods in Fluids*, vol. 71, no. 1, pp. 80–97, 2013.

[17] C. Farhat and L. Crivelli, "A general approach to nonlinear fe computations on shared-memory multiprocessors," *Computer Methods in Applied Mechanics and Engineering*, vol. 72, no. 2, pp. 153–171, 1989.

[18] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, "Sparse matrix solvers on the gpu: conjugate gradients and multigrid," in *ACM Transactions on Graphics*, vol. 22, 2003, pp. 917–924.

[19] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the dash multiprocessor," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ser. ISCA '90. New York, NY, USA: ACM, 1990, pp. 148–159.

[20] J. H. Kelm, M. R. Johnson, S. S. Lumettta, and S. J. Patel, "Waypoint: Scaling coherence to thousand-core architectures," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 99–110.

[21] L. Thebault, E. Petit, M. Tchiboukdjian, Q. Dinh, and W. Jalby, "Divide and conquer parallelization of finite element method assembly," *Parallel Computing: Accelerating Computational Science and Engineering (CSE), Advances in Parallel Computing 25*, 2014.

[22] E. Petit, L. Thébault, N. Möller, Q. Dinh, and W. Jalby, "Task-based parallelization of unstructured meshes assembly using d&c strategy," in *Proceedings of the 2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS)*. IEEE Computer Society, 2014, pp. 874–877.

[23] L. Thebault, E. Petit, Q. Dinh, and W. Jalby, "Scalable and efficient implementation of 3d unstructured meshes computation: A case study on matrix assembly," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '15, USA, 2015*, 2015.

[24] N. Möller, E. Petit, L. Thébault, and Q. Dinh, "A case study on using a proto-application as a proxy for code modernization," *Procedia Computer Science*, vol. 51, pp. 1433–1442, 2015.

[25] D. Anderson, F. Sparacio, and R. M. Tomasulo, "The ibm system/360 model 91: Machine philosophy and instruction-handling," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 8–24, 1967.

[26] J. R. Goodman, "Using cache memory to reduce processor-memory traffic," *ACM SIGARCH Computer Architecture News*, vol. 11, no. 3, pp. 124–131, 1983.

[27] A. Seznec, "The l-tage branch predictor," in *Journal of Instruction Level Parallelism*. Citeseer, 2006.

[28] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz, *APRIL: a processor architecture for multiprocessing*. ACM, 1990, vol. 18, no. 2SI.

[29] D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[30] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Acm Sigplan Notices*, vol. 37, no. 10. ACM, 2002, pp. 211–222.

[31] J. Lira, C. Molina, and A. González, "Analysis of non-uniform cache architecture policies for chip-multiprocessors using the parsec benchmark suite," in *Proceedings of the workshop on managed many-core systems*, 2009, pp. 1–8.

[32] W. Arden, M. Brillouët, P. Cogez, M. Graef, B. Huizing, and R. Mahnkopf, "Morethan-moore white paper," *Version*, vol. 2, p. 14, 2010.

[33] "Green500 lists november 2015." [Online]. Available: http://www.green500.org/news/green500-list-november-2015

[34] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin *et al.*, "Larrabee: a many-core x86 architecture for visual computing," in *ACM Transactions on Graphics (TOG)*, vol. 27, no. 3. ACM, 2008, p. 18.

[35] L. Chen, P. Jiang, and G. Agrawal, "Exploiting recent simd architectural advances for irregular applications," 2016.

[36] "Top500 lists november 2015." [Online]. Available: http://www.top500.org/lists/2015/11/

[37] E. Saule, K. Kaya, and Ü. V. Çatalyürek, "Performance evaluation of sparse matrix multiplication kernels on intel xeon phi," in *Parallel Processing and Applied Mathematics*. Springer, 2013, pp. 559–570.

[38] J. J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: past, present and future," *Concurrency and Computation: practice and experience*, vol. 15, no. 9, pp. 803–820, 2003.

[39] J. Dongarra and M. A. Heroux, "Toward a new metric for ranking high performance computing systems," *Sandia Report, SAND2013-4744*, vol. 312, 2013.

[40] C. A. Patterson, M. Snir, S. L. Graham *et al.*, *Getting Up to Speed:: The Future of Supercomputing*. National Academies Press, 2005.

[41] "Cea tgcc curie presentation." [Online]. Available: http://www-hpc.cea.fr/fr/complexe/tgcc-curie.htm

[42] "It4i salomon presentation." [Online]. Available: https://docs.it4i.cz/salomon/network-1/7d-enhanced-hypercube

[43] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[44] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 1-3, pp. 66–73, 2010.

[45] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2004, pp. 97–104.

[46] "Intel mpi library." [Online]. Available: https://software.intel.com/en-us/intel-mpi-library

[47] H. Tang and T. Yang, "Optimizing threaded mpi execution on smp clusters," in *Proceedings of the 15th international conference on Supercomputing*. ACM, 2001, pp. 381–392.

[48] C. Huang, O. Lawlor, and L. V. Kale, "Adaptive mpi," in *Languages and Compilers for Parallel Computing*. Springer, 2003, pp. 306–322.

[49] M. Pérache, P. Carribault, and H. Jourdren, "Mpc-mpi: An mpi implementation reducing the overall memory consumption," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2009, pp. 94–103.

[50] F. O'Carroll, H. Tezuka, A. Hori, and Y. Ishikawa, "The design and implementation of zero copy mpi using commodity hardware with a high performance network," in *Proceedings of the 12th international conference on Supercomputing*. ACM, 1998, pp. 243–250.

[51] M. J. Koop, S. Sur, and D. K. Panda, "Zero-copy protocol for mpi using infiniband unreliable datagram," in *Cluster Computing, 2007 IEEE International Conference on*. IEEE, 2007, pp. 179–186.

[52] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced features of the message-passing interface*. MIT press, 1999.

[53] R. Thakur, W. Gropp, and B. Toonen, "Optimizing the synchronization operations in message passing interface one-sided communication," *International Journal of High Performance Computing Applications*, vol. 19, no. 2, pp. 119–128, 2005.

[54] R. Gerstenberger, M. Besta, and T. Hoefler, "Enabling highly-scalable remote memory access programming with mpi-3 one sided," in *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*. IEEE, 2013, pp. 1–12.

[55] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, and K. Underwood, "Remote memory access programming in mpi-3," *ACM Transactions on Parallel Computing*, vol. 2, no. 2, p. 9, 2015.

[56] R. Belli and T. Hoefler, "Notified access: Extending remote memory access programming models for producer-consumer synchronization," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 871–881.

[57] I. Fraunhofer, "Gpi-global address space programming interface," 2013.

[58] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, *Introduction to UPC and language specification*. Center for Computing Sciences, Institute for Defense Analyses, 1999.

[59] A. Aiken, P. Colella, D. Gay, S. Graham, P. Hilfinger, A. Krishnamurthy, B. Liblit, C. Miyamoto, G. Pike, L. Semenzato *et al.*, "Titanium: A high-performance java dialect," *Concurrency: Practice and Experience*, vol. 10, pp. 11–13, 1998.

[60] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," in *ACM Sigplan Fortran Forum*, vol. 17, no. 2. ACM, 1998, pp. 1–31.

[61] B. L. Chamberlain, D. Callahan, and H. P. Zima, “Parallel programmability and the chapel language,” *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[62] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, “X10: an object-oriented approach to non-uniform cluster computing,” *Acm Sigplan Notices*, vol. 40, no. 10, pp. 519–538, 2005.

[63] D. Bonachea and J. Duell, “Problems with using mpi 1.1 and 2.0 as compilation targets for parallel language implementations,” *International Journal of High Performance Computing and Networking*, vol. 1, no. 1-3, pp. 91–99, 2004.

[64] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, “Optimizing bandwidth limited problems using one-sided communication and overlap,” in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International.* IEEE, 2006, pp. 10–pp.

[65] P. Ghysels and W. Vanroose, “Hiding global synchronization latency in the preconditioned conjugate gradient algorithm,” *Parallel Computing*, vol. 40, no. 7, pp. 224–238, 2014.

[66] F. Shahzad, M. Wittmann, M. Kreutzer, T. Zeiser, G. Hager, and G. Wellein, “Pgas implementation of spmvm and lbm using gpi,” in *7th International Conference on PGAS Programming Models*, 2013, p. 172.

[67] R. Machado, S. Abreu, and D. Diaz, “Parallel local search: Experiments with a pgas-based programming model,” *arXiv preprint arXiv:1301.7699*, 2013.

[68] EXA2CT, “The exa2ct european project: Exascale algorithms and advanced computational techniques,” http://www.exa2ct.eu, Jan. 2015.

[69] P. Jarzebski, K. Wisniewski, and R. Taylor, “On parallelization of the loop over elements in feap,” *Computational Mechanics*, vol. 56, no. 1, pp. 77–86, 2015.

[70] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, “Starpu: a unified platform for task scheduling on heterogeneous multicore architectures,” *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[71] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, “Parsec: Exploiting heterogeneity to enhance scalability,” *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[72] C. Augonnet, O. Aumage, N. Furmento, R. Namyst, and S. Thibault, *StarPU-MPI: Task programming over clusters of machines enhanced with accelerators*. Springer, 2012.

[73] O. Delannoy, F. Emad, and S. Petiton, “Workflow global computing with yml,” in *Proceedings of the 7th IEEE/ACM international conference on grid computing*. IEEE Computer Society, 2006, pp. 25–32.

[74] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, “Ompss: a proposal for programming heterogeneous multi-core architectures,” *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[75] P. Gonnet, A. B. Chalk, and M. Schaller, “Quicksched: Task-based parallelism with dependencies and conflicts,” *arXiv preprint arXiv:1601.05384*, 2016.

[76] R. Blikberg and T. Sørevik, "Nested parallelism: Allocation of threads to tasks and openmp implementation," *Scientific Programming*, vol. 9, no. 2, 3, pp. 185–194, 2001.

[77] S. Shah, G. Haab, P. Petersen, and J. Throop, "Flexible control structures for parallelism in openmp," *Concurrency: Practice and Experience*, vol. 12, no. 12, pp. 1219–1239, 2000.

[78] E. Su, X. Tian, M. Girkar, G. Haab, S. Shah, and P. Petersen, "Compiler support of the workqueuing execution model for intel smp architectures," in *Fourth European Workshop on OpenMP, Rome. http://www. caspur. it/ewomp02/PAPERI/EWOMP02-03. pdf*, 2002.

[79] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of openmp tasks," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 20, no. 3, pp. 404–418, 2009.

[80] E. Ayguadé, A. Duran, J. Hoeflinger, F. Massaioli, and X. Teruel, "An experimental evaluation of the new openmp tasking model," in *Languages and Compilers for Parallel Computing*. Springer, 2007, pp. 63–77.

[81] A. Duran, J. Corbalán, and E. Ayguadé, "Evaluation of openmp task scheduling strategies," in *OpenMP in a new era of parallelism*. Springer, 2008, pp. 100–110.

[82] S. L. Olivier and J. F. Prins, "Evaluating openmp 3.0 run time systems on unbalanced task graphs," in *Evolving OpenMP in an Age of Extreme Parallelism*. Springer, 2009, pp. 63–78.

[83] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng, "Uts: An unbalanced tree search benchmark," in *Languages and Compilers for Parallel Computing*. Springer, 2006, pp. 235–250.

[84] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin, "Reducers and other cilk++ hyperobjects," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. ACM, 2009, pp. 79–90.

[85] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.

[86] J. T. Fineman and C. E. Leiserson, "Race detectors for cilk and cilk++ programs," in *Encyclopedia of Parallel Computing*. Springer, 2011, pp. 1706–1719.

[87] Y. He, C. E. Leiserson, and W. M. Leiserson, "The cilkview scalability analyzer," in *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2010, pp. 145–156.

[88] T. B. Schardl, B. C. Kuszmaul, I. Lee, W. M. Leiserson, C. E. Leiserson *et al.*, "The cilkprof scalability profiler," in *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures*. ACM, 2015, pp. 89–100.

[89] C.-K. Luk, R. Newton, W. Hasenplaugh, M. Hampton, and G. Lowney, "A synergetic approach to throughput computing on x86-based multicore desktops," *IEEE software*, vol. 28, no. 1, p. 39, 2011.

[90] Y. Saad, *Iterative methods for sparse linear systems*. Siam, 2003.

[91] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. ACM, 2009, pp. 233–244.

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

[93] A. Buluç, S. Williams, L. Oliker, and J. Demmel, "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 721–733.

[94] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, "When cache blocking of sparse matrix vector multiply works and why," *Applicable Algebra in Engineering, Communication and Computing*, vol. 18, no. 3, pp. 297–311, 2007.

[95] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 135–158, 2004.

[96] M. Martone, S. Filippone, S. Tucci, M. Paprzycki, and M. Ganzha, "Utilizing recursive storage in sparse matrix-vector multiplication-preliminary considerations." in *CATA*, 2010, pp. 300–305.

[97] M. Martone, S. Filippone, M. Paprzycki, and S. Tucci, "On blas operations with recursively stored sparse matrices," in *12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE, 2010, pp. 49–56.

[98] M. Martone, S. Filippone, P. Gepner, M. Paprzycki, and S. Tucci, "Use of hybrid recursive CSR/-COO data structures in sparse matrices-vector multiplication," in *International Multiconference on Computer Science and Information Technology - IMCSIT*, 2010, pp. 327–335.

[99] M. Martone, S. Filippone, S. Tucci, and M. Paprzycki, "Assembling recursively stored sparse matrices," in *Computer Science and Information Technology (IMCSIT), Proceedings of the 2010 International Multiconference on*. IEEE, 2010, pp. 317–325.

[100] K. Kourtis, G. Goumas, and N. Koziris, "Optimizing sparse matrix-vector multiplication using index and value compression," in *Proceedings of the 5th conference on Computing frontiers*. ACM, 2008, pp. 87–96.

[101] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C401–C423, 2014.

[102] J. R. Rice and R. F. Boisvert, *Solving elliptic problems using ELLPACK*. Springer Science & Business Media, 2012, vol. 2.

[103] H. C. Edwards, D. Sunderland, V. Porter, C. Amsler, and S. Mish, "Manycore performance-portability: Kokkos multidimensional array library," *Scientific Programming*, vol. 20, no. 2, pp. 89–114, 2012.

[104] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps *et al.*, "An overview of the trilinos project," *ACM Transactions on Mathematical Software (TOMS)*, vol. 31, no. 3, pp. 397–423, 2005.

[105] M. T. Heath and P. Raghavan, “A cartesian parallel nested dissection algorithm,” *SIAM Journal on Matrix Analysis and Applications*, vol. 16, no. 1, pp. 235–253, 1995.

[106] G. L. Miller, S.-H. Teng, and S. A. Vavasis, “A unified geometric approach to graph separators,” in *Foundations of Computer Science, 1991. Proceedings., 32nd Annual Symposium on*. IEEE, 1991, pp. 538–547.

[107] M. J. Berger and S. H. Bokhari, “A partitioning strategy for nonuniform problems on multiprocessors,” *Computers, IEEE Transactions on*, vol. 100, no. 5, pp. 570–580, 1987.

[108] R. D. Williams, “Performance of dynamic load balancing algorithms for unstructured mesh calculations,” *Concurrency: Practice and experience*, vol. 3, no. 5, pp. 457–481, 1991.

[109] A. Ytterström, “A tool for partitioning structured multiblock meshes for parallel computational mechanics,” *International Journal of High Performance Computing Applications*, vol. 11, no. 4, pp. 336–343, 1997.

[110] C. Lachat, C. Dobrzynski, and F. Pellegrini, “Parallel mesh adaptation using parallel graph partitioning,” in *5th European Conference on Computational Mechanics (ECCM V)*, vol. 3. CIMNE-International Center for Numerical Methods in Engineering, 2014, pp. 2612–2623.

[111] C. Chevalier and F. Pellegrini, “Pt-scotch: A tool for efficient parallel graph ordering,” *Parallel computing*, vol. 34, no. 6, pp. 318–331, 2008.

[112] C. Dobrzynski and P. Frey, “Anisotropic delaunay mesh adaptation for unsteady simulations,” in *Proceedings of the 17th international Meshing Roundtable*. Springer, 2008, pp. 177–194.

[113] H. D. Simon, “Partitioning of unstructured problems for parallel processing,” *Computing Systems in Engineering*, vol. 2, no. 2-3, pp. 135–148, 1991.

[114] S. T. Barnard and H. D. Simon, “Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems,” *Concurrency: Practice and experience*, vol. 6, no. 2, pp. 101–117, 1994.

[115] G. Karypis and V. Kumar, “Parallel multilevel k-way partitioning scheme for irregular graphs, department of computer science,” *University of Minnesota, Minneapolis, MN*, pp. 96–036, 1996.

[116] G. Karypis and V. Kumar, “Metis–unstructured graph partitioning and sparse matrix ordering system, version 2.0,” 1995.

[117] G. Karypis and V. Kumar, “Multilevel k-way partitioning scheme for irregular graphs,” *Journal of Parallel and Distributed computing*, vol. 48, no. 1, pp. 96–129, 1998.

[118] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[119] B. Hendrickson and R. Leland, “A multilevel algorithm for partitioning graphs,” in *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*. ACM, 1995, p. 28.

[120] T. N. Bui and C. Jones, “A heuristic for reducing fill-in in sparse matrix factorization,” Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA (United States), Tech. Rep., 1993.

[121] B. W. Kernighan and S. Lin, “An efficient heuristic procedure for partitioning graphs,” *Bell system technical journal*, vol. 49, no. 2, pp. 291–307, 1970.

[122] D. LaSalle and G. Karypis, “Multi-threaded graph partitioning,” in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 225–236.

[123] G. Karypis, K. Schloegel, and V. Kumar, “Parmetis,” *Parallel graph partitioning and sparse matrix ordering library. Version*, vol. 2, 2003.

[124] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, “Optimization of sparse matrix–vector multiplication on emerging multicore platforms,” *Parallel Computing*, vol. 35, no. 3, pp. 178–194, 2009.

[125] N. Gourdain, L. Gicquel, M. Montagnac, O. Vermorel, M. Gazaix, G. Staffelbach, M. Garcia, J. Boussuge, and T. Poinsot, “High performance parallel computing of flows in complex geometries: I. methods,” *Computational Science & Discovery*, vol. 2, no. 1, p. 015003, 2009.

[126] L. Qu, L. Grigori, and F. Nataf, “Parallel design and performance of nested filtering factorization preconditioner,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC ’13. New York, NY, USA: ACM, 2013, pp. 81:1–81:12.

[127] J. Park, M. Smelyanskiy, K. Vaidyanathan, A. Heinecke, D. D. Kalamkar, X. Liu, M. M. A. Patwary, Y. Lu, and P. Dubey, “Efficient shared-memory implementation of high-performance conjugate gradient benchmark and its application to unstructured matrices,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 945–955.

[128] M. Garcia, “Analysis of precision differences observed for the avbp code,” Technical Report, Tech. Rep., 2003.

[129] A. H. Gebremedhin, D. Nguyen, M. M. A. Patwary, and A. Pothen, “Colpack: Software for graph coloring and related problems in scientific computing,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 40, no. 1, p. 1, 2013.

[130] G. Rokos, G. J. Gorman, K. E. Jensen, and P. H. Kelly, “Thread parallelism for highly irregular computation in anisotropic mesh adaptation,” in *Proceedings of the 3rd International Conference on Exascale Applications and Software*. University of Edinburgh, 2015, pp. 103–108.

[131] A. H. Gebremedhin and F. Manne, “Scalable parallel graph coloring algorithms,” *Concurrency - Practice and Experience*, vol. 12, no. 12, pp. 1131–1146, 2000.

[132] Ü. V. Çatalyürek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen, “Graph coloring algorithms for multi-core and massively multithreaded architectures,” *Parallel Computing*, vol. 38, no. 10, pp. 576–594, 2012.

[133] G. Rokos, G. Gorman, and P. H. Kelly, “A fast and scalable graph coloring algorithm for multi-core and many-core architectures,” in *Euro-Par 2015: Parallel Processing*. Springer, 2015, pp. 414–425.

[134] E. Horowitz and A. Zorat, “Divide-and-conquer for parallel processing,” *Computers, IEEE Transactions on*, vol. 100, no. 6, pp. 582–585, 1983.

[135] C. A. Hoare, “Quicksort,” *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 1962.

[136] R. C. Singleton, “An algorithm for computing the mixed radix fast fourier transform,” *Audio and Electroacoustics, IEEE Transactions on*, vol. 17, no. 2, pp. 93–103, 1969.

[137] M. Tchiboukdjian, V. Danjean, and B. Raffin, "Binary Mesh Partitioning for Cache-Efficient Visualization," *Transactions on Visualization and Computer Graphics*, vol. 16, no. 5, pp. 815–828, 2010.

[138] M. Tchiboukdjian, V. Danjean, and B. Raffin, "Cache-efficient parallel isosurface extraction for shared cache multicores," in *EGPGV*, 2010, pp. 81–90.

[139] M. Tchiboukdjian, V. Danjean, T. Gautier, F. L. Mentec, and B. Raffin, "A Work Stealing Scheduler for Parallel Loops on Shared Cache Multicores," in *Highly Parallel Processing on a Chip (HPPC)*, 2010.

[140] T. Guillet and M. Tchiboukdjian, "Scalable and composable shared memory parallelism with tasks for multicore and manycore," in *Exascale Challenges Workshop, TERATEC Forum*, 2012.

[141] J. Dongarra, V. Eijkhout, and P. Luszczek, "Recursive approach in sparse matrix lu factorization," *Sci. Program.*, vol. 9, no. 1, pp. 51–60, Jan. 2001.

[142] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, no. 1, pp. 38–53, 2009.

[143] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Foundations of Computer Science, 1999. 40th Annual Symposium on*. IEEE, 1999, pp. 285–297.

[144] J. D. Frens and D. S. Wise, "Auto-blocking matrix-multiplication or tracking blas3 performance from source code," in *ACM SIGPLAN Notices*, vol. 32, no. 7. ACM, 1997, pp. 206–216.

[145] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi, "Recursive array layouts and fast matrix multiplication," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, no. 11, pp. 1105–1123, 2002.

[146] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström, "Recursive blocked algorithms and hybrid data structures for dense matrix library software," *SIAM review*, vol. 46, no. 1, pp. 3–45, 2004.

[147] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' guide*. Siam, 1999, vol. 9.

[148] C. Vömel, S. Tomov, and J. Dongarra, "Divide and conquer on hybrid gpu-accelerated multicore systems," *SIAM Journal on Scientific Computing*, vol. 34, no. 2, pp. C70–C82, 2012.

[149] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The plasma and magma projects," in *Journal of Physics: Conference Series*, vol. 180, no. 1. IOP Publishing, 2009, p. 012037.

[150] D. Goudin and J. Roman, "A scalable parallel assembly for irregular meshes based on a block distribution for a parallel block direct solver," in *Applied Parallel Computing. New Paradigms for HPC in Industry and Academia*. Springer, 2000, pp. 113–120.

[151] I.-C. Wu, H.-T. Kung *et al.*, "Communication complexity for parallel divide-and-conquer," in *Foundations of Computer Science, 1991. Proceedings., 32nd Annual Symposium on*. IEEE, 1991, pp. 151–162.

[152] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Communications of the ACM*, vol. 29, no. 12, pp. 1184–1201, 1986.

[153] S. Maleki, Y. Gao, M. J. Garzaran, T. Wong, and D. A. Padua, "An evaluation of vectorizing compilers," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 2011, pp. 372–382.

[154] I. Z. Reguly, E. László, G. R. Mudalige, and M. B. Giles, "Vectorizing unstructured mesh computations for many-core architectures," *Concurrency and Computation: Practice and Experience*, 2015.

[155] G. Mudalige, M. Giles, I. Reguly, C. Bertolli, and P. Kelly, "Op2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures," in *Innovative Parallel Computing (InPar), 2012*. IEEE, 2012, pp. 1–12.

[156] D. A. Burgess, P. I. Crumpton, and M. B. Giles, *A parallel framework for unstructured grid solvers*. Springer, 1994.

[157] P. I. Crumpton and M. B. Giles, "Multigrid aircraft computations using the oplus parallel library," in *Parallel Computational Fluid Dynamics: Implementation and Results using Parallel Computers. Proceedings Parallel CFD*, vol. 95. Citeseer, 1996, pp. 339–346.

[158] R. Löhner, "Cache-efficient renumbering for vectorization," *International Journal for Numerical Methods in Biomedical Engineering*, vol. 26, no. 5, pp. 628–636, 2010.

[159] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th National Conference*, ser. ACM '69. New York, NY, USA: ACM, 1969, pp. 157–172.

[160] L. Oliker, X. Li, P. Husbands, and R. Biswas, "Effects of ordering strategies and programming paradigms on sparse matrix computations," *SIAM Review*, vol. 44, no. 3, pp. 373–393, 2002.

[161] L. Oliker, X. Li, G. Heber, and R. Biswas, "Parallel conjugate gradient: effects of ordering strategies, programming paradigms, and architectural platforms," *Lawrence Berkeley National Laboratory*, 2000.

[162] G. Heber, R. Biswas, and G. R. Gao, "Self-avoiding walks over adaptive unstructured grids," *Concurrency: Practice and Experience*, vol. 12, no. 2-3, pp. 85–109, 2000.

[163] A. Yzelman and R. H. Bisseling, "Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods," *SIAM Journal on Scientific Computing*, vol. 31, no. 4, pp. 3128–3154, 2009.

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

[165] A. Yzelman and R. H. Bisseling, "Two-dimensional cache-oblivious sparse matrix–vector multiplication," *Parallel Computing*, vol. 37, no. 12, pp. 806–819, 2011.

[166] A. George, "Nested dissection of a regular finite element mesh," *SIAM Journal on Numerical Analysis*, vol. 10, no. 2, pp. 345–363, 1973.

[167] I. Brainman and S. Toledo, "Nested-dissection orderings for sparse lu with partial pivoting," *SIAM Journal on Matrix Analysis and Applications*, vol. 23, no. 4, pp. 998–1012, 2002.

[168] P. Cicotti, L. Carrington, and A. Chien, "Toward application-specific memory reconfiguration for energy efficiency," in *Proceedings of the 1st International Workshop on Energy Efficient Supercomputing*. ACM, 2013, p. 2.

[169] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, vol. 3, 2009.

[170] R. F. Barrett, S. Borkar, S. S. Dosanjh, S. D. Hammond, M. A. Heroux, X. S. Hu, J. Luitjens, S. G. Parker, J. Shalf, and L. Tang, "On the role of co-design in high performance computing," *vol*, vol. 24, pp. 141–155, 2013.

[171] S. L. Olivier and J. F. Prins, "Comparison of openmp 3.0 and other task parallel frameworks on unbalanced task graphs," *International Journal of Parallel Programming*, vol. 38, no. 5-6, pp. 341–360, 2010.

[172] P. Langlois, D. Parello, B. Goossens, and K. Porada, "Less hazardous and more scientific research for summation algorithm computing times," 2012.

[173] S. Collange, M. Daumas, D. Defour, and D. Parello, "Barra: A parallel functional simulator for gpgpu," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 351–360.

[174] YALES2, "Yales2 academic application," https://www.coria-cfd.fr/index.php/YALES2.

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

**Titre :** Algorithmes Parallèles Efficaces Appliqués aux Calculs sur Maillages Non Structurés

**Mots clefs :** HPC, FEM, maillage non structuré, D&C, vectorisation, PGAS

**Résumé :** Le besoin croissant en simulation a conduit à l'élaboration de supercalculateurs complexes et d'un nombre croissant de logiciels hautement parallèles. Ces supercalculateurs requièrent un rendement énergétique et une puissance de calcul de plus en plus importants. Les récentes évolutions matérielles consistent à augmenter le nombre de noeuds de calcul et de coeurs par noeud. Certaines ressources n'évoluent cependant pas à la même vitesse. La multiplication des coeurs de calcul implique une diminution de la mémoire par coeur, plus de trafic de données, un protocole de cohérence plus coûteux et requiert d'avantage de parallélisme. De nombreuses applications et modèles actuels peinent ainsi à s'adapter à ces nouvelles tendances. En particulier, générer du parallélisme massif dans des méthodes d'éléments finis utilisant des maillages non structurés, et ce avec un nombre minimal de synchronisations et des charges de travail équilibrées, s'avèrent particulièrement difficile. Afin d'exploiter efficacement les multiples niveaux de parallélisme des architectures actuelles, différentes approches parallèles doivent être combinées. Cette thèse propose plusieurs contributions destinées à paralléliser les codes et les structures irrégulières de manière efficace. Nous avons développé une approche parallèle hybride par tâches à grain fin combinant les formes de parallélisme distribuée, partagée et vectorielle sur des structures irrégulières. Notre approche a été portée sur plusieurs applications industrielles développées par Dassault Aviation et a permis d'importants gains de performance à la fois sur les multicoeurs classiques ainsi que sur le récent Intel Xeon Phi.

**Title :** Scalable and Efficient Algorithms for Unstructured Mesh Computations

**Keywords :** HPC, FEM, unstructured mesh, D&C, vectorization, PGAS

**Abstract :** The growing need for numerical simulations results in larger and more complex computing centers and more HPC softwares. Actual HPC system architectures have an increasing requirement for energy efficiency and performance. Recent advances in hardware design result in an increasing number of nodes and an increasing number of cores per node. However, some resources do not scale at the same rate. The increasing number of cores and parallel units implies a lower memory per core, higher requirement for concurrency, higher coherency traffic, and higher cost for coherency protocol. Most of the applications and runtimes currently in use struggle to scale with the present trend. In the context of finite element methods, exposing massive parallelism on unstructured mesh computations with efficient load balancing and minimal synchronizations is challenging. To make efficient use of these architectures, several parallelization strategies have to be combined together to exploit the multiple levels of parallelism. This P.h.D. thesis proposes several contributions aimed at overpassing this limitation by addressing irregular codes and data structures in an efficient way. We developed a hybrid parallelization approach combining the distributed, shared, and vectorial forms of parallelism in a fine grain task-based approach applied to irregular structures. Our approach has been ported to several industrial applications developed by Dassault Aviation and has led to important speedups using standard multicores and the Intel Xeon Phi manycore.

**Université Paris-Saclay**
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France