Industrial Code Modernization of High Performance Computing simulations on modern supercomputer architectures

Nathalie Möller

▶ **To cite this version:**

Nathalie Möller. Industrial Code Modernization of High Performance Computing simulations on modern supercomputer architectures. Distributed, Parallel, and Cluster Computing [cs.DC]. Université Paris Saclay (COmUE), 2019. English. NNT : 2019SACLV088 . tel-02614318

HAL Id: tel-02614318

https://theses.hal.science/tel-02614318v1

Submitted on 20 May 2020

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 : 2019SACLV088

Adaptation de codes industriels de simulation en Calcul Haute Performance aux architectures modernes de supercalculateurs

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'Université de Versailles-Saint-Quentin-en-Yvelines

Ecole doctorale n°580 Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat: Informatique

Thèse présentée et soutenue à Versailles, le 13 Décembre 2019, par

Nathalie Möller

Composition du Jury :

Michaël Krajecki Professeur, Université de Reims	Président du Jury Rapporteur
Anthony Scemama Ingénieur de Recherche, CNRS	Rapporteur
Nahid Emad Professeur, UVSQ	Examinateur
Elisabeth Brunet Maître de Conférences, Télécom SudParis	Examinateur
William Jalby Professeur, UVSQ	Directeur de thèse
Eric Petit Ingénieur Chercheur, Intel	Co-Encadrant
Quang Dinh Ingénieur Chercheur, Dassault Aviation	Membre Invité
Quentin Carayol Ingénieur Chercheur, Dassault Aviation	Membre Invité
Marc Pérache Ingénieur Chercheur, CEA DAM	Membre Invité

Thèse de doctorat

Contents

Introduction **1**

I Partie I - State of the Art **5**

1 HPC Hardware architectures **7**
1.1 Core architecture . . . 7
1.2 Multicore architecture . . . 8
1.3 NUMA . . . 9
1.4 Accelerators . . . 9
1.5 Distributed architecture . . . 10
1.6 Conclusion . . . 10

2 Parallel Programming Models **11**
2.1 Introduction . . . 11
2.2 The message passing model . . . 12
2.2.1 Point-to-point communications . . . 13
2.2.2 Collective communications . . . 13
2.2.3 Non-blocking asynchronous communications . . . 15
2.2.4 One-sided communications . . . 18
2.3 The Partitioned Global Address Space model . . . 20
2.3.1 The PGAS model . . . 20
2.3.2 The GASPI standard and the GPI library . . . 21
2.4 The Worksharing model . . . 25
2.4.1 The OpenMP standard . . . 25
2.4.2 The OpenMP loop parallelization . . . 26
2.4.3 The OpenMP worksharing constructs . . . 28
2.5 The task-based model . . . 28
2.5.1 OpenMP tasks . . . 28
2.5.2 Cilk tasks . . . 31
2.5.3 Task-based schedulers . . . 33
2.6 Conclusion . . . 34

3 Fast Multipole Method **35**
3.1 Introduction 35
3.2 Fast Multipole Method 35
3.3 Parallelization strategies 40
3.3.1 Low-level optimization 41
3.3.2 Fork-join implementation 41
3.3.3 Tasks 42
3.3.4 Load balancing 45
3.3.5 Communications 48
3.3.6 Accelerators 50
3.4 Conclusion 50

4 Dassault Aviation's and UVSQ Applications and Libraries **53**
4.1 Introduction 53
4.2 SPECTRE 54
4.2.1 Use cases 55
4.3 AETHER 57
4.3.1 Matrix assembly 57
4.3.2 Parallelization 57
4.3.3 Use cases 59
4.4 Mini-FEM and D&C-lib 59
4.4.1 The concept of Proto-Application 59
4.4.2 DEFMESH and Mini-FEM 60
4.4.3 The D&C library 61
4.5 Conclusion 62

II Contributions **63**

5 Code Modernization through Mini-Applications **65**
5.1 Introduction 65
5.2 Porting the D&C library from the proto-app to an industrial application 65
5.3 Using the proto-app to support the OpenMP implementation of the D&C library 68
5.4 Performance evaluation 69
5.5 Conclusion 71

6 Load Balancing **73**
6.1 Introduction 73
6.2 Current load balancing strategy 73
6.2.1 Current algorithm 74
6.2.2 Efficiency characterization 75

6.2.3 Objective and Methodology . . . 76
6.3 Histograms . . . 77
6.4 Morton . . . 79
6.5 Conclusion . . . 85

7 Communications **87**
7.1 Introduction . . . 87
7.2 Profiling and analysis . . . 88
7.3 Bulk synchronous versions, with MPI . . . 92
7.4 Introducing asynchronism, with GASPI . . . 96
7.5 Replacing global communications with local communications . . . 101
7.6 Hybrid GASPI combined with tasks . . . 103
7.7 Overall improvement and remaining potential . . . 106
7.8 Conclusion . . . 107

8 Tasks **111**
8.1 Introduction . . . 111
8.2 Profiling and Analysis . . . 111
8.3 Towards asynchronous task-based parallelism . . . 114
8.3.1 Current parallelization scheme . . . 114
8.3.2 OpenMP 4.0 tasks with depend clause . . . 115
8.3.3 Recursive version with scoreboards . . . 117
8.4 Progress report . . . 118
8.5 Conclusion . . . 122

Conclusion and Future Work **122**

Annex 1 - Difficulties encountered with MPI Runtimes **129**

Annex 2 - Summary **133**

Annex 3 - Résumé **135**

List of Figures **137**

List of Tables **141**

Bibliography **143**

Remerciements

Cette thèse est le résultat de plusieurs années de travail. Je souhaite remercier tous ceux qui ont contribué à les rendre mémorables tant sur le fond, que sur la forme.

Je tiens a remercier Michaël Krajecki et Anthony Scemama pour avoir accepté d'être rapporteurs. Merci également aux examinateurs et autres membres de mon jury : Nahid Emad, Elisabeth Brunet et Marc Pérache.

La thèse Cifre présente la particularité de se situer à la croisée de deux mondes: le domaine académique et le monde industriel. J'ai eu la chance de bénéficier de la richesse de ces deux milieux.

Je remercie donc mon directeur de thèse, William Jalby, pour m'avoir accueillie au laboratoire LI-Parad, et pour m'avoir offert la possibilité d'enseigner en école d'ingénieur. Je remercie tout particulièrement Eric Petit, pour m'avoir encadrée tout au long de ces années. Merci pour les conseils et les discussions. Merci à tous les membres du labo pour les moments partagés autour d'un café, d'un repas, d'un jeu de société, d'un verre, d'un concours de prog, d'un barbecue, d'un mariage, de naissances, et j'en oublie sûrement encore : Loïc, Clément, Seijilo, Mihail, Salah, Hugo, Pablo, Youenn, Yohan, la liste est longue ...

J'ai aussi été très bien accueillie chez Dassault Aviation. Merci à Quang Dinh, pour avoir fait tout son possible pour que cette thèse ait lieu et qu'elle se déroule dans les meilleures conditions. Merci à Quentin Carayol, pour ses explications de la MLFMM, et ses multiples réponses à mes multiples questions ;) Merci à Eric Gounot, manager juste et sympathique :), Tran Chanh pour son efficacité et son humour ! Michel Ravachol, Farid Cerbah et Quang bien sûr, pour l'ambiance du bureau, les conseils avisés et les relectures en détail. Merci à Gilles Zalamanski, "l'électron libre", pour l'initiation à la poterie et à la cérémonie du thé, et à Laurent Zimmer pour les pas de danse ;)

Un thèse, ça dure quelques années, et certains nous ont quittés. Je me dois de remercier Mirjean, mon oncle, pour m'avoir encouragée à reprendre mes études. Merci pour ta sagesse, ton empathie et ta joie de vivre. Merci également à ma grand-mère, pour sa tendresse, son humour et sa bienveillance.

D'autres sont toujours là, et quoi qu'il arrive, on peut compter sur eux. Merci à ma maman d'être formidable :) Merci à Sylvain, de me soutenir, encore et toujours, dans ce long parcours d'études depuis l'IUT jusqu'au doctorat.

Enfin, certains arrivent et nous emportent dans le tourbillon de la vie. Merci à Andréa, petite boule d'énergie qui tout naturellement du haut de sa logique de bébé, nous rappelle chaque jour, qu'avec patience et détermination (acharnement ?), rien n'est impossible :)

Introduction

Computational science is at the core of many scientific domains as much in academic research than in industrial research and development. Computer simulations are used in a wide range of domains such as weather forecasting, financial markets, flight simulators, reservoir simulation for the petroleum industry, car crashes and so on. In the domain of aircraft design, computer simulations of physical processes are being increasingly used.

Dassault Aviation is a major aerospace company, specialized in the manufacturing of both military aircraft and business jets. At Dassault Aviation the whole process from design through production is fully numeric. Dassault Aviation develops three main simulation codes. Figure 1 illustrates a typical example of multi-disciplinary aircraft design for the Neuron UCAV (Unmanned Combat Aerial Vehicle) demonstrator. Computer analyses were used to design the Neuron in the following fields: aerodynamics, electromagnetics and infra-red stealth computation.

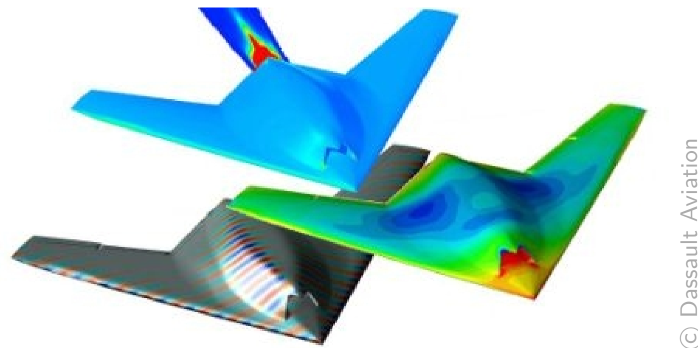


FIGURE 1: Aerodynamic (top) and stealth - in the electromagnetic (bottom) and infra-red (middle) ranges - analyses in the multi-disciplinary design of the Neuron UCAV demonstrator

These simulations perform numerous complex operations and manipulate large amounts of data. Therefore, they necessitate powerful computing facilities. They were mostly written by physicists, starting 20 to 30 years ago, and some of them were recently parallelized using MPI. At Dassault Aviation, most of the current applications benefit from hybrid parallelization relying on MPI at distributed level and OpenMP loop parallelism within nodes.

However, parallel architectures have evolved a lot and are becoming more and more complex. Processors now contain more and more cores with larger vector units and complex cache hierarchies. A typical TOP 500[1] super-computer is composed of hundreds of cores within thousands of nodes. Each core is equipped with larger vector

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

units, and each node eventually contains accelerators like GPUs. This leads to very heterogeneous architectures that are difficult to exploit efficiently. Another limitation is the fact that some resources do not scale at the same rate. Typically, the memory and network bandwidth scale less than the computing power.

In parallel with the architecture evolution, libraries, languages, and tools for parallel programming have evolved to take advantage of these new features. As a result, modernization and optimization large industrial code represents a challenging and tedious task in the middle of heterogeneous architecture and new programming models. In particular, it is necessary to control each parallelism layer and the interaction between them. Furthermore, the gap between the literature in computer science, and the large industrial application's scalability and performance is wide. The academic domain has access to ninja programmers demonstrating ad-hoc solutions on top applications or synthetic benchmarks. This poses challenges both in term of learning and generalization of the technology developed by computer scientists. The vast majority of production codes are written by physicist who do not have the time to investigate and adapt these propositions to their code. The main commonalities between the proposed optimizations are concurrency, locality and asynchronism, which respectively address, or mitigate, the growing parallelism, the memory gap and communication scalability. Nonetheless, applying these principles on large industrial applications turns out to be more difficult to achieve than expected. Interesting methods shifting towards an asynchronous paradigm, while preserving load balancing and locality are demonstrated in the academic field. Among them, very few large scale applications in production today are exploiting asynchronous parallel tasks and asynchronous multithreaded communications to take full advantage of the available concurrency.

In this PhD thesis, we explore methods to modernize large industrial simulation codes. To support our study, we work with two main simulation applications in use in the Dassault Aviation aircraft design process: the AETHER CFD (Computational Fluid Dynamics) simulation code and SPECTRE, the CEM (Computational Electro-Magnetics) application. We experiment two different approaches to bring state-of-the-art progress in HPC into legacy applications.

The first one is the continuation of the work of L. Thébault based on the concept of proto-applications [117]. The critical part to optimize is extracted as a stand alone benchmark that can be shared with the HPC community to demonstrate how to handle the case. In this case, it is representative of the assembly step of a Finite Element Method (FEM) used in CFD codes. This proto-application was then used to support the development of an optimization library, called D&C library. My first contribution consists in porting back and evaluating this optimization in another more complex production code, AETHER, and to document the process. We then reuse the process to extend the D&C library with OpenMP tasks, to propose an alternative to the current implementation based on Cilk tasks. We validated the methodology by demonstrating comparable results on the proto-application and on the target industrial application. This first contribution led to a publication in the ICCS 2015 ALCHEMY workshop [100].

The second, and main contribution, of this thesis aims at optimizing an industrial application for CEM simulations. More specifically, we target the MLFMM (Multi-Level Fast Multipole Method) used in SPECTRE. SPECTRE is developed with similar tools, environment and programming rules than AETHER. For the sake of comparison, and also because of confidentiality constraints, the optimization work is done directly on the application, without

any representative proto-application. Based on measurement and literature, we identify three separate aspects to optimize:

Load balancing the work among the processor nodes directly impacts the global performance. A poor load balancing leads to idle time and bounds the overall performance to the slowest process' execution time. We propose two load balancing methodologies. The first one is based on a Morton space-filling curve and the second on Histograms. We paid great attention on developing a lightweight optimized algorithm so that it can be integrated with minimal overhead. Our load balancing strategies allow for a 5% improvement on communication time. We demonstrate that the effects directly reflect in the communication matrices. Nonetheless, since load balancing efforts may be harmed by a poor synchronous communication model, we chose to prioritize the optimization of the communications.

Communications represent a large and non-scaling part of the SPECTRE's MLFMM algorithm. We start by intensively profiling the communication scheme to identify the bottlenecks. The MPI-based communications are synchronous, two-sided and blocking. They are grouped in a large communication phase, which has to be fully completed before the calculations can continue. This process cannot scale on large number of nodes. To benefit from today's architectures, it is necessary to evolve towards asynchronous, thread-based and local exchanges. Knowing from literature and past experiences that MPI asynchronous communications lack progression, we tried an alternative PGAS-based library called GASPI, and demonstrated its efficiency. Then, we gradually desynchronize the communication scheme, by breaking down large communications into smaller and more local exchanges. We propose different versions progressing from the two-sided blocking synchronous model towards a fully asynchronous task-handled model. These optimizations allow us to divide the communication time by a factor of 6. Along with load balancing, this work was presented in a short paper at Europar 2017 COLOC workshop [99] and published in ICCS 2019 ALCHEMY workshop [98]. To demonstrate our load balancing and communications improvement for MLFMM, we are releasing *FMM-lib* library [4] under LGPL-3 license.

Shared memory parallelism For now, the parallelization of the computing kernels relies on loop parallelization. In irregular applications, the iterations of a same loop carry very different amounts of computational work. Therefore, loop-based parallelization can lead to load imbalance, resulting in idle threads. A first experience with recursive tasks is experienced in the first contribution of this thesis, on the AETHER CFD code [100]. For the CEM/MLFMM algorithm, we target a fully asynchronous task-based version, including the communications. Early analysis and prototyping show great potential for optimizing the shared memory parallelism. We explore two versions that differ in the way of controlling the task dependencies. The first one relies on OpenMP 4.5 tasks, which provides a specific clause to express data dependencies and then schedules the tasks depending on their readiness. The second is a manually handled recursive version, using scoreboards to track the dependencies and spawn the tasks. The task-level GASPI communications are included inside the tasks. The work on task-based parallelization is ongoing. We present the progress achieved so far and give the guidelines for the future work.

This thesis is organized in two parts. The first part presents the state-of-the-art on parallelization of large industrial applications, and also introduces Dassault Aviation's and UVSQ Applications and Libraries. Chapter 1

outlines the current architectural trend, characterized by a growing number of nodes containing large number of cores. Chapter 2 explores parallel programming models. Particular attention is paid to distributed and shared memory parallelism. Chapter 3 presents the Fast Multipole Method. First, we describe the algorithm and then, we focus on the most common parallelizing approaches. Chapter 4 presents the applications and libraries which support the work of this thesis. We introduce Dassault Aviation's industrial electromagnetics and aerodynamics simulation codes and the UVSQ's Mini-FEM proto-application and D&C library. The second part of the thesis focuses on our contributions. In chapter 5, we present our experience with code modernization methodology based on the use of proto-applications. We port the D&C library on Dassault Aviation's aerodynamic application, and we use the Mini-FEM proto-application to support further development of the D&C library. In chapter 6, we present the load balancing strategies we have developed. Chapter 7 focuses on optimizing the communication scheme. We progressively move from a two-sided blocking message model to one-sided asynchronous messages, overlapped with computations. In chapter 8, we focus on shared memory parallelism by the use of tasks. We elaborate an asynchronous and hierarchical strategy based on two possible implementations. We report the progress in developing them and provide the main directions for further work.

Finally, we conclude by summarizing our contributions and propose possible future work for each contribution.

Part I

Partie I - State of the Art

Chapter 1

HPC Hardware architectures

Contents

1.1 Core architecture 7
1.2 Multicore architecture 8
1.3 NUMA 9
1.4 Accelerators 9
1.5 Distributed architecture 10
1.6 Conclusion 10

Current architectures still derive from the Van Neumann model, which was developed in 1945. This model describes a computer architecture as a processing unit, containing an Arithmetic and Logic Unit (ALU) to perform the computations, a control unit to schedule the instructions, a memory unit to contain instructions and data, and input and output devices.

To increase the performance, this basic architecture has evolved over the years. In this chapter, we describe typical architectures encountered in current supercomputers. Computing nodes are described from core level in section 1.1, to multicore in section 1.2, and Non Uniform Memory Architecture (NUMA) in section 1.3. Accelerators are presented in section 1.4, and distributed architectures are detailed in 1.5. In this last section, we also present the clusters which have been used during this thesis for our experiments.

1.1 Core architecture

A core is essentially composed by computing units, cache memory and control units. This cores present different technologies to decrease the latencies, like for example: pipelines, cache hierarchy, branch predicting, out-of-order execution, prefetcher, and simultaneous multithreading [79]. In this section, we discuss the main mechanisms which have to be known in the context of this thesis.

Cache memory Cache memory is used to speed up access to data in the RAM. The central memory is organized in 64-bytes words, called a *cache line*. When the processor tries to access some data, it first checks if the data is available in the cache. If not, it retrieves the complete cache line containing the data. Therefore, if data are accessed contiguously, they may already be available in the cache. The cache memory is hierarchically organized

into cache levels, and current processors generally have three levels: L1, L2, and L3. The L1 cache is the smallest one, from 32kB to 64kB, the fastest, closest to the core, and most expensive. It is split into data and instruction cache. The L2 cache ranges from 512kB to 2MB and the L3 cache can reach tens of megabytes. To accelerate the process of retrieving data, a *prefetcher* unit is used to analyze the access patterns and preload the next cache lines. The anticipation is based on heuristics or statistics. This mechanism is well-suited for regular access to a data structure, but it does not perform well on irregular problems with random accesses. To take benefit of the cache memory mechanism, algorithms must expose locality, both spatial and temporal. Data reuse favors cache reuse and contiguous access favors prefetching.

InstructionpProcessing A computing unit is composed of two parts: the front-end and the back-end. The front-end, also called fetch&decode unit decomposes the instructions into micro-instructions that are placed into a pipeline. The back-end dispatches micro-instructions to the computing units through a scheduler. Independent micro-instructions may be re-ordered to decrease the latency, this is called *out-of-order* execution. *Superscalar* architecture allows for concurrent execution on multiple computing units: ALUs, FPUs (Floating Point Units). In this architecture, independent micro-instructions may also be sent to different computing units to increase the throughput. To further take advantage of this mechanism, current architecture also implements simultaneous multithreading (SMT). In this case, multiple threads are executed on a physical core, and their instructions may share the same pipeline. Since these instructions are guaranteed to be independent, they can be executed on different processing units. However, all these techniques are limited by the pipeline depth and the number of ports. Indeed, instruction level parallelism is limited by the data dependencies within a program. Therefore, it may not be profitable to further increase the number of ports, since they will be difficult to feed.

Vectorization Another possibility to increase the throughput consists in using data parallelism to multiply the number of operations per instruction. Indeed, many kernels perform the same computation on multiple data, for example in linear algebra or image processing codes. The data parallelism is implemented using specific vector units operating on vector registers. In comparison to a scalar instruction which executes an instruction on a single data in a SISD (Single Instruction Single Data) model, vector units apply the same instruction on multiple data, in a SIMD (Single Instruction Multiple Data) parallelism model. Current vector units have the ability to process up to 512 bits of data simultaneously, for example with Intel's AVX-512 instruction set and ARM Fujitsu's SVE (Scalar Vector Extension) implementation. However, using these large units requires more power, which limits their operating frequency. For example, using AVX-512 units on the Skylake 8180 decreases its base frequency from 2.5 GHz down to 1.7 GHz[1]. Moreover, using larger vector units requires fetching more data from the memory and therefore increases the pressure on the bandwidth.

1.2 Multicore architecture

Another way to increase the performance of processors consists in multiplying the number of cores on the same die. Typically, in this architecture, the last level cache (usually L3 for Intel architectures) is shared among all the cores, enabling fast communication between them. Below that last level cache, each core owns its private cache memories.

[1] https://en.wikichip.org/wiki/intel/xeon_platinum/8180

Current processors contain up to 56 cores for the Intel Xeon Platinum 9282[2] and 64 for the AMD EPYC 7742[3]. These multicore architectures allow for either MPMD (Multiple Program Multiple Data) or SPMD (Single Program Multiple Data) parallelism models. In the first case, different independent programs are assigned to different cores, whereas in the latter case, a single program is executed by the cores on a different piece of data. However, the memory bandwidth does not scale proportionally to the number of cores. Therefore, the pressure on the bandwidth increases and limits the parallel efficiency.

1.3 NUMA

Non Uniform Memory Architectures (NUMA) consist in assembling several blocks of cores and memories, also codes NUMA nodes, on the same motherboard. They are connected with a fast interconnection bus: Quick Path Interconnect (QPI) or Ultra Path Interconnect (UPI) for Intel , and Infinity Fabric (IF) for AMD. All memories share the same address space, and are therefore accessible by all nodes. However, the access cost depends on the distance between the core and the memory. To benefit from the accumulated bandwidth provided by these architectures, it is important to take care of the data locality, so that each core mostly accesses the memory in its NUMA node. Moreover, small asynchronous parallel tasks allow spreading the communications along the computation, consequently avoiding bottlenecks and improving the bandwidth usage. Shared memory programming models are introduced in sections 2.4 and 2.5.1. In this thesis, we propose a strategy using a hierarchical task-based algorithm which takes advantage of asynchronism and locality, presented in section 8.

1.4 Accelerators

Accelerators can be used to further improve computation capacities, like NVIDIA's or AMD's Graphics Processing Units (GPUs), Intel Xeon Phis[4], NEC SX-Aurora TSUBASA vector engine[5], Intel FPGA[6], and the upcoming Intel Xe. With the recent announcement for the next generation of pre-exascale and exascale systems, the trend of reaching the required high peak flops using heterogeneous hardware seems to solidify. This will open a new challenging phase for HPC applications that are usually not a good fit for current accelerator design. The characteristics of these accelerators consist in exposing a very large number of simplified computing cores or large vector units, combined with dedicated high speed memories. For example, the Nvidia Telsa V100 GPU contains 5120 cores and 640 tensor cores[7], combined with up to 32GB HBM2 (High Bandwidth Memory) memory delivering a bandwidth of 900GB/s. In comparison, Xeon Phi Knights Landing[8] offers 72 x86 compatible cores, with 4-way SMT, AVX-512 vector units, and 16GB of HBM configurable MCDRAM (Multi Channel DRAM) providing a bandwidth of 400GB/s.

[2] https://ark.intel.com/content/.../194146/intel-xeon-platinum-9282-processor-77m-cache-2-60-ghz.html
[3] https://www.amd.com/fr/products/cpu/amd-epyc-7742
[4] https://ark.intel.com/content/www/fr/fr/ark/products/series/92649/intel-xeon-phi-x100-product-family.html
[5] https://www.nec.com/en/global/solutions/hpc/sx/vector_engine.html
[6] https://www.intel.fr/content/www/fr/fr/products/programmable.html
[7] https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf
[8] https://ark.intel.com/content/.../products/95831/intel-xeon-phi-processor-7290f-16gb-1-50-ghz-72-core.html

1.5 Distributed architecture

Distributed architectures consist in assembling multiple nodes and connect them with networks. They can be arranged following different topologies: trees, rings, stars, fully connected or not. On supercomputers, the network often relies on dedicated network technologies such as Infiniband, Omni-Path or Aries interconnect, with different latencies and bandwidths. Data exchange through the network can be handled by message passing or Partitioned Global Address Space (PGAS). In the first case, nodes communicate data by explicitly sending and receiving information, whereas in the latter case nodes can directly access to distant memories. These programming models are presented in chapter 2. In this thesis, we develop and compare different communication strategies using both message passing and one-sided PGAS communication, detailed in chapter 7.

The different clusters used for our experiments are detailed in table 1.1 hereafter:

Cluster	Processor	Used cores	Network
Dassault Aviation	Sandy Bridge E5-2670	32 nodes x 2 sockets x 8 cores (512)	Infiniband FDR
INTI CEA	Sandy Bridge E5-2680	32 nodes x 2 sockets x 8 cores (512)	Infiniband
Anselm IT4I	Sandy Bridge E5-2665	64 nodes x 2 sockets x 8 cores (1024)	Infiniband
Anselm IT4I	4 KNC	4 x 60 (240)	Infiniband

Table 1.1: Clusters used

1.6 Conclusion

Current distributed architectures are becoming more and more complex, with lots of different levels of parallelism: vector units, multiple cores, NUMA nodes, accelerators and distributed memories. Taking advantage of all these levels requires deep knowledge of the application's parallelism opportunities, the parallel programming models, and available programming libraries. In the next chapter, we present different programming models used in HPC applications to benefit from shared and distributed memory architectures.

Chapter 2

Parallel Programming Models

Contents

2.1 Introduction **11**

2.2 The message passing model **12**

2.2.1 Point-to-point communications 13

2.2.2 Collective communications 13

2.2.3 Non-blocking asynchronous communications 15

2.2.4 One-sided communications 18

2.3 The Partitioned Global Address Space model **20**

2.3.1 The PGAS model 20

2.3.2 The GASPI standard and the GPI library 21

2.4 The Worksharing model **25**

2.4.1 The OpenMP standard 25

2.4.2 The OpenMP loop parallelization 26

2.4.3 The OpenMP worksharing constructs 28

2.5 The task-based model **28**

2.5.1 OpenMP tasks 28

2.5.2 Cilk tasks 31

2.5.3 Task-based schedulers 33

2.6 Conclusion **34**

2.1 Introduction

In chapter 1, we have presented an overview of the current supercomputer architectures. The trend is to ever more complex designs, multiplying the number of nodes and cores, and eventually adding diverse accelerators.

To take advantage of the multiple layers of parallelism, algorithms must be ever more elaborated. It is necessary to combine different parallelization models to profit from each parallel layer. In this chapter, we focus essentially on distributed and shared memory parallelism, since the work of this thesis concentrates on these two levels of parallelism.

However, depending on the architecture, vectorization, NUMA-aware, GPGPU and accelerators programming may be necessary to fully exploit the available parallelism potential.
In order to adapt to the growing size of clusters, applications are distributed across several computational nodes. In numerous HPC applications employing mesh-based spatial discretization, a common way to dispatch the work among processors consists in using domain decomposition, as illustrated in figure 2.1. The mesh is divided into independent subdomains attributed to different processors, and necessary data exchanges between neighbors are done through communications. In Sections 2.2 and 2.3, we will present the two main communication models based

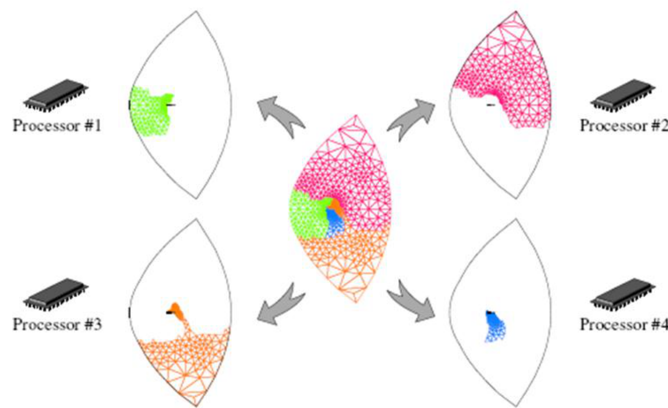


FIGURE 2.1: Domain decomposition

on message passing and global address spaces, and their libraries.
Inside a node, the application has to take profit from the shared memory parallelism. Combining distributed and shared memory parallelism is commonly referred to as hybrid parallelism. Processes are used in the coarse grain to communicate between nodes and threads are used in the finer grain inside a computational node. Multithreading models based on worksharing and tasks are presented in sections 2.4 and 2.5.

2.2 The message passing model

The Message Passing Model is strongly related to distributed memory architectures. As presented in chapter 1, computation nodes are linked together through a network. They do not share any memory space, and the only way to communicate consists in sending data through the network.In the message passing model, communications are cooperative: the sender and the receiver explicitly participate in the process. A communication is complete only once:

- the sender has sent the data to its destination,
- and the receiver has fully received the data.

If a communication never arrives, the receiver will wait indefinitely. Therefore, sends and receives need to match perfectly. The Message Passing Model is well-suited for Single Process Multiple Data (SPMD) parallelism, where

each processor executes the same computations on multiple data and sometimes carries out some communications. But it also supports Multiple Process Multiple Data (MPMD), where different processors may execute different code but still need to communicate or synchronize.

The MPI API MPI (Message Passing Interface) is the de-facto standard for message passing on distributed memory architectures. It defines a standardized and portable API (Application Programmer Interface) for Fortran, C and C++ languages. Processors are abstracted into ranks belonging to a virtual topology and communications take place through communicators.
The MPI standard has been used for over twenty years, and been extended several times. The development started in 1992, with regular meetings of the MPI Forum [13], which brings together over 40 organizations from industrial and academic research. The first release, MPI-1 [65], was published in 1994. Initially, MPI [66] offers two-sided communications with point-to-point message passing and collective operations, in blocking and non-blocking modes. In 1998, MPI-2 [75] [113], the first major extension, introduces dynamic processes and one-sided communications. Collective operations are extended to be used on inter-communicators, an interface is developed to allow third-party applications like debuggers and profilers, language bindings are improved for C++ and Fortran, and parallel I/O are introduced. In 2012, the MPI-3 [72] standard contains significant functionality extensions. It introduces non-blocking and neighboring collective operations, improves one-sided communications with the support of Remote Memory Access (RMA), see 2.2.4, refines the interface to enable performance analysis tools to get access to internal variables, like counters and states, and extends the Fortran bindings to 2008 Fortran standard.
Several libraries implement the MPI standard, the most used being OpenMPI [69], MPICH [74] and Intel MPI [7].

2.2.1 Point-to-point communications

The MPI model relies on message passing between two nodes: sending and receiving are the basic communication mechanisms. MPI provides the *MPI_Send* and the *MPI_Recv* primitives. A simple example of their use is illustrated in figure 2.2. In this example the MPI process identified by the process id 0 sends a message to the MPI process 1. Both have a buffer of size 20 characters. Rank 0 sends the data explicitly to rank 1 with a communication tag 99 and using the MPI_COMM_WORLD communicator. Rank 1 waits for a maximum quantity of 20 characters from rank 0, with the same tag, on the same communicator and with an additional variable status, which contains information about the communication. Since the communications are blocking, late sends or late receive may slow down the complete communication process.

2.2.2 Collective communications

It is possible to involve more than two processes through collective operations. They are of two kinds: data movement with, for example, broadcast, gather and scatter operations, and computation operations with reductions. Reductions can use standard addition, minimum, maximum, logical or operations as well as user defined operations. Performance of collective algorithms can vary depending on the message size, the number of processors and even the network topology. Therefore, different algorithms are developed to implement them. When using Intel MPI, it is possible to select them at runtime through an environment variable. For example for the broadcast, 14 algorithms are available, as listed in the Table 2.1

```
MPI_Init(&argc,&argv );
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank );
char buffer[20];
MPI_Status status;

if (rank == 0){
  strcpy(buffer,"send something")
  MPI_Send(message, strlen(buffer)+1, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
}
else if (rank == 1){
  MPI_Recv(buffer, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
}

MPI_Finalize();
return 0;
```

FIGURE 2.2: Basic MPI communication based on blocking send and receive calls

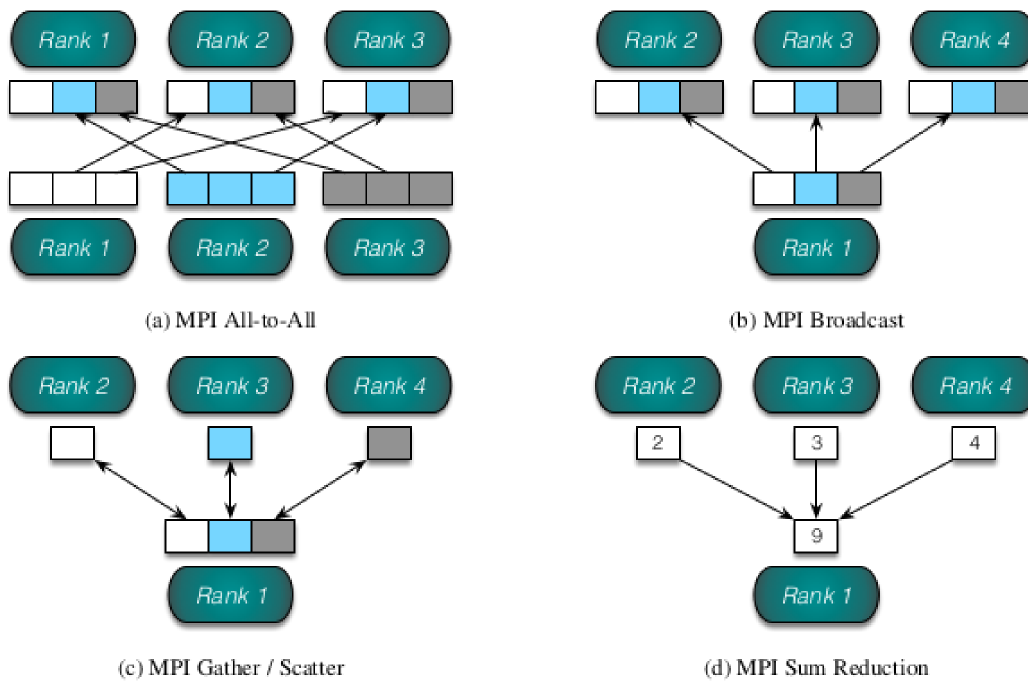


FIGURE 2.3: MPI collective communications

Environment Variable	Collective Operation	Algorithms
I_MPI_ADJUST_BCAST	MPI_Bcast	1. Binomial 2. Recursive doubling 3. Ring 4. Topology aware binomial 5. Topology aware recursive doubling 6. Topology aware ring 7. Shumilin's 8. Knomial 9. Topology aware SHM-based flat 10. Topology aware SHM-based Knomial 11. Topology aware SHM-based Knary 12. NUMA aware SHM-based (SSE4.2) 13. NUMA aware SHM-based (AVX2) 14. NUMA aware SHM-based (AVX512)

TABLE 2.1: MPI Broadcast algorithms list, available in the Intel MPI Library

OpenMPI offers similar tuning possibilities through its MCA (Modular Component Architecture) framework. It is now designed to allow customized MPI built from component choices at runtime [69]. To enable overriding the default behavior it is necessary to set the *mca_coll_tuned_use_dynamic_rules* variable. Then, it is possible to specify the selected algorithm when launching the program:

```
mpirun -np N -mca coll_tuned_use_dynamic_rules 1 -mca coll_tuned_barrier_algorithm 4 a.out
```

The performance not only depends on the application and the use cases but also on the network, the number of processes, and hardware characteristics. Indeed, aligning memory segments to cache lines or taking care of NUMA domains also impacts the latency, and some algorithms consider these hardware properties. Moreover, a complete application may use different MPI communications on different data sets. As a result, determining the optimal algorithms for all communications remains a difficult problem [62] [82] [122].

2.2.3 Non-blocking asynchronous communications

Aiming at overlapping Blocking communications can be time consuming. Therefore, it is interesting to hide the communication latency behind some computations; this process is called overlapping and illustrated in figure 2.4. The left side shows standard execution: the computation phase is followed by a communication phase. During the communication phase, the involved processors wait for the communication completion. The waiting time is equal to the transmission time. On the right side, the communication takes place in the background of the computation. Waiting time is considerably reduced. The communication latency is hided, and the total application benefits from speedup. To this end, the MPI standard provides non-blocking communications. Two-sided send and receive operations can be initiated with *MPI_Irecv* and *MPI_Isend* primitives. These calls return immediately and let the communication progress in the background. It is then possible to wait explicitly or to test for completion. Since MPI-3, there are also asynchronous versions of the collective communications. The use of non-blocking primitives is illustrated in figure 2.5. On the left side, blocking primitives are used, and both processes wait for the

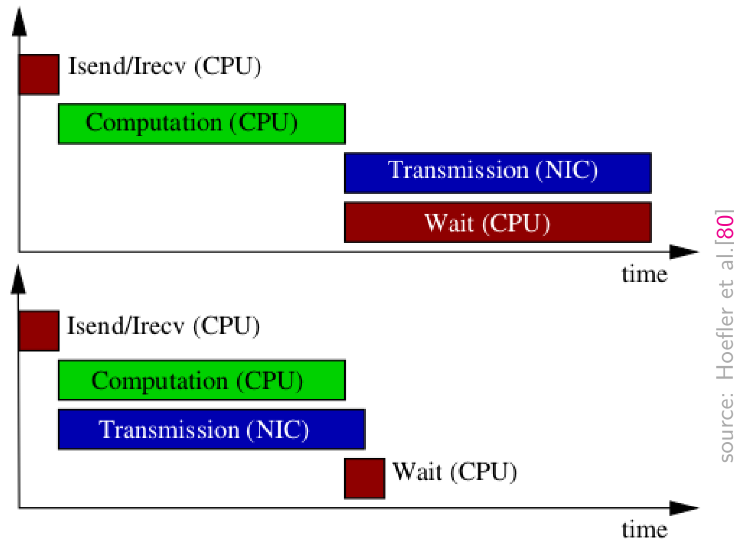


FIGURE 2.4: Overlapping communication with computation

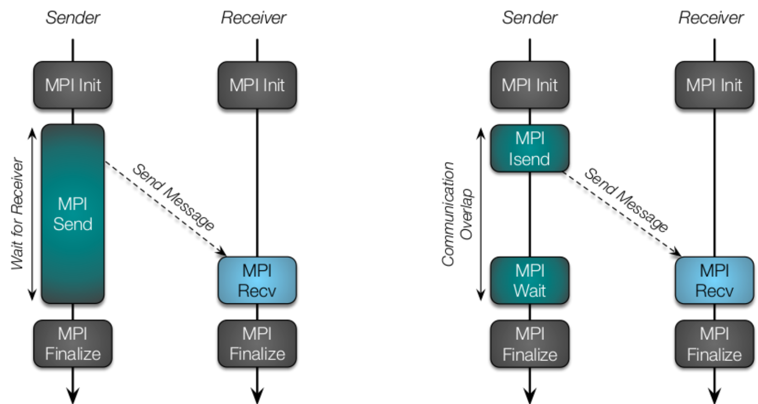


FIGURE 2.5: Blocking and non-blocking two-sided communications

completion,Whereas, on the right side, both processes use non-blocking calls. Thus, both can overlap their waiting time with computation and wait only when necessary: for the sender, when it needs to reuse the send buffer, and for the receiver, when it needs the incoming data.

Making non-blocking communication progress A communication is asynchronous if it progresses in the background of some computation. To this end, MPI proposes non-blocking primitives. Nonetheless, the use of those

primitives does not guarantee asynchronism. First, it is necessary to identify overlapping opportunities, and second, the non-blocking communications have to be efficient. The lack of progression of non-blocking messages is a known problem detailed in [80]. The strategies to send the messages are left out to the MPI implementation, and mainly based on operating system polling and interruptions. Polling consists in waiting on the CPU while retrieving information from the hardware. In that way, the operation remains at the user level. Interruptions consist in making a system call, which is assumed to be expensive. MPI implementations relying on polling, thus bypassing the OS (Operating System), are known to have slower point-to-point latency. For non-blocking communications all these algorithms become more complex. Three methods to enforce the completion are proposed in [80]: The first one, *manual progression*, relies on user level progression. It uses the fact that calls to the *MPI_Test* primitive triggers the communication. The difficulty with using *MPI_Test* consists in choosing the right time: not too soon because there is nothing to progress and not too late when the transfer is already terminated. The second solution is to handle the full asynchronous progression at the hardware level. For example, Infiniband provides RDMA (Remote Direct Memory Access) capabilities which allows for directly writing into recipient's memory without any intermediate copies. [114] The third one consists in using thread level progression, directly in the MPI implementation [126].

Eager and Rendezvous protocols MPI blocking and non-blocking transmissions are handled through communications protocols. For each outgoing message, an *envelope* containing metadata is created: it specifies the source, the destination, the tag, the length and the communicator. A complete message consists in an envelope followed by the data. Depending on the size of the communications, either the *eager* or the *rendezvous* protocol is used to complete the communication.
The *eager* protocol sends the message directly to the receiver, with no prior consent. On the receiver's side the incoming message is saved in a buffer. It is only when the receiver executes the reception operation, that the message is copied from the buffer into the local memory.
The *rendezvous* protocol eliminates the reception buffer with the use of a handshaking model. At first, only the envelope containing the metadata is sent. The receiver prepares the reception, and informs the sender of its readiness. Then, the data can be received and directly stored into the dedicated memory place.

Pros and Cons Both protocols are illustrated in figure 2.6. The *eager* protocol reduces synchronization delays. The sender only waits for the acknowledgment that the data has been copied into the receiver's buffer. But this protocol is space and time consuming: data copies are costly. Moreover, it relies on the assumption that the receiver has enough memory. The MPI norm specifies that the receivers should be able to handle expected messages as well as unexpected ones. Therefore, buffering the incoming communications permits to keep data until a matching reception call occurs. But the receiver does not know how large the reception buffer should be, and may eventually run out of memory and cause program termination. Or, on the contrary, it may allocate too large buffers and waste memory. In comparison, the *rendezvous* protocol is more robust and makes a more efficient use of memory. It avoids unnecessary copies and allocates only the necessary space. Nonetheless, it necessitates more synchronization. Complexity studies [113] show that the *eager* protocol is more rapid for small messages, but at the price of larger memory usage and increased copy. For larger messages, the *rendezvous* protocol is more efficient, but at the cost of additional synchronizations. However, the efficiency of these protocols also depends on their implementations.

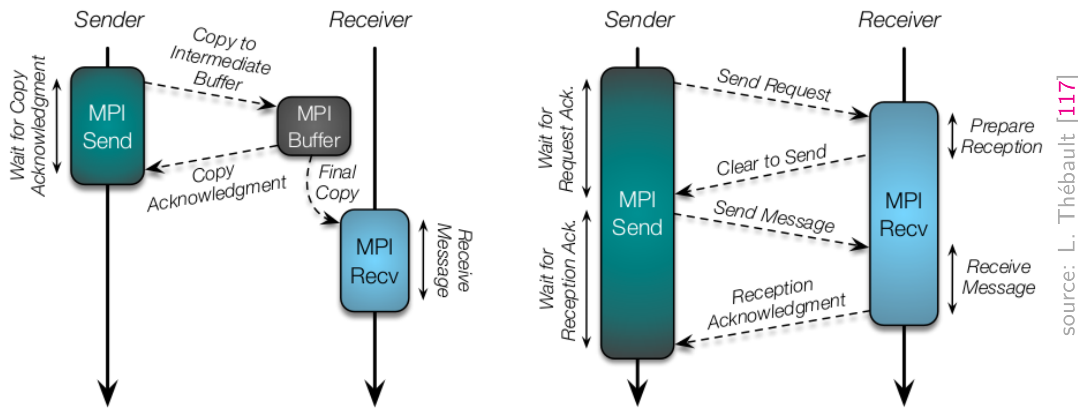


FIGURE 2.6: Eager and Rendezvous Protocols

2.2.4 One-sided communications

One-sided communications have been introduced with MPI-3 RMA. They are largely detailed in [73]. Distributed memory is abstracted as a global memory. Each node exposes a part of its local memory, through a *window*. This piece of local memory is remotely accessible to the other MPI nodes both for reading and writing, through the *MPI_Put* and *MPI_Get* primitives. Atomic calls are also available like *MPI_Accumulate*, *MPI_Get_Accumulate*, *MPI_Compare_and_swap* and *MPI_Fetch_and_op*.
Figure 2.7 illustrates the mechanisms of the remote accesses. On the left side, the source processor remotely writes into the target's memory, and on the right side the target directly retrieves data from the source.

RMA synchronizations Even if these operations directly occur in the shared memory places, synchronization remains mandatory. For example, it may be necessary to ensure that a remote window is accessible, or that a written data is available for the receiver. Therefore, MPI uses the notion of *epoch*. One-sided communications and synchronizations take place inside these epochs. An *access epoch* is created at the beginning of the communication, and its accessibility is defined by the *exposure epoch*. Epoch statuses are defined by synchronization calls. One must distinguish between active and passive synchronization.

Active mode In active communication mode, the target is involved in the data transmission. Available synchronization models are: Fence and Post-Start-Complete-Wait(PSCW), both illustrated in figure 2.8. *MPI_Win_fence* is a collective synchronization call: all processes of the window group start and end an exposure epoch simultaneously. After creating a window, a first collective call to the blocking *MPI_Win_Fence* ensures accessibility. Once this call returns it is possible to remotely write or read from or into a window. But a remote writes and reads are non-blocking. The communications are initiated, but not necessary terminated. It is necessary to wait for the return of a second call to *MPI_Win_Fence* to guarantee that all outstanding memory operations are completed. In comparison, the *PSCW*, mechanism is less costly: it can be used by all or just a subset of the MPI processors. The

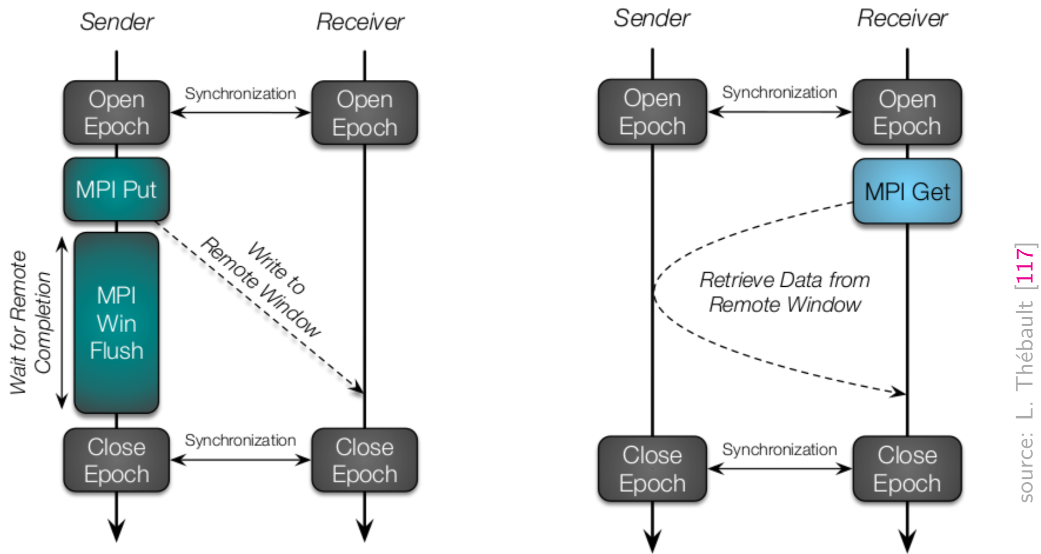


FIGURE 2.7: MPI One-sided Communications

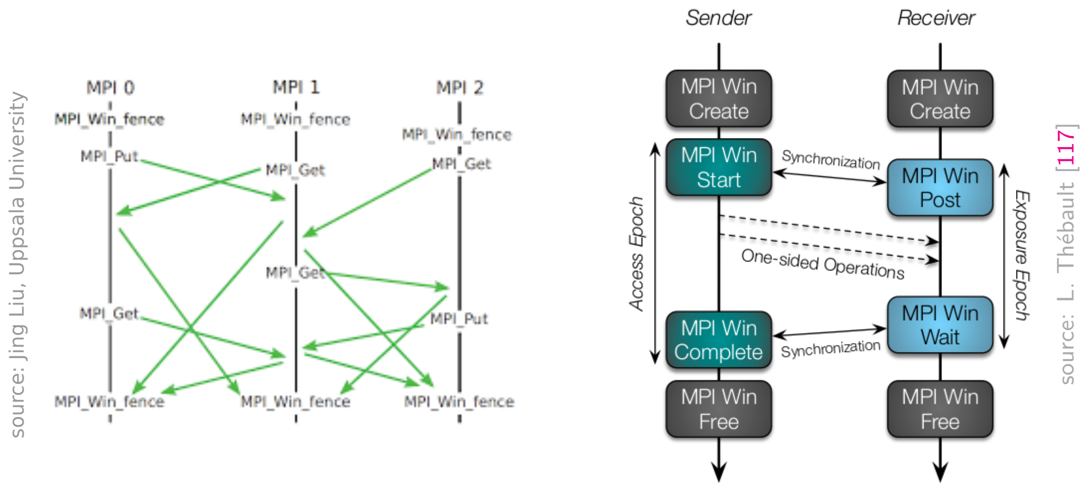


FIGURE 2.8: Left: MPI_Win_Fence, Right: PSCW

origin process defines an *access epoch* with the calls to *MPI_Win_Start* and a *MPI_Win_Complete*. The target defines an *exposure epoch*, with the *MPI_Win_Post* and a *MPI_Win_Wait* primitives.

Passive mode In the passive mode, the target does not participate, neither in communication nor in synchronization. It's the source who uses the Lock/Unlock [116] model. *Lock/Unlock* synchronization is similar to the semaphore semantics: a processor requests a lock, which can be exclusive or shared with other processors. Once the lock is acquired, it executes the read or write operation, and then releases it. When the unlock call returns, the communication is complete on both sides.

Limitations RMA synchronizations are blocking. They can suffer from different kind of latencies. On the sender's side, a *late post* occurs when the target delays the MPI_Win_Post. This can cause an *early transfer*: the RMA communication blocks, since the target exposure window is not ready. On the receiver's side an early wait can occur when the MPI_Win_Wait is called as the RMA communication is not completed. If there is an important difference between the time when the target posts its MPI_Win_Wait and the origin of its MPI_Win_Complete, since both calls synchronize together, it can result in a *late complete*. Fence latencies impacts are even more critical. Since this synchronization is collective, it only takes one of the processes to delay its call to impact the global performance. All these synchronizations add significant overhead. Non-blocking RMA synchronizations are still the object of research, as for example in [136].

Towards truly asynchronous communications MPI communications, even-one sided and relying on hardware progression, induce a lot of synchronization overhead. They don't seem well-suited enough to enable truly asynchronous communications and to maximize profit from overlapping opportunities. At the time of writing this thesis, the MPI-4 release is under development. The MPI-forum indicates that focus will be put on extensions for hybrid programming models, support for fault tolerance and RMA one-sided communications [12].
Nonetheless, in the meantime, some very interesting efficiency oriented communication model, has arisen in the PGAS sphere.

2.3 The Partitioned Global Address Space model

2.3.1 The PGAS model

The model PGAS stands for Partitioned Global Address Space. It provides a global address space which is logically partitioned among the computation nodes. Each processor shares a part of its memory with the others. Therefore, it has a local private piece of memory piece, and another one it shares in the global space. Figure 2.9 illustrates this abstraction with four processors.
The abstraction level of this model can be placed between the shared and distributed models. From the shared memory model it inherits the idea that global address space is shared, and can be accessed by any processor. From the distributed model it inherits the communication costs: accessing local data is cheap, whereas accessing remote data is expensive. A key concept of the PGAS model is the support of one-sided communications. Efficiently implementing them by taking profit of the network RDMA feature increases the performance.

PGAS implementations To exploit the PGAS model, very different and eventually opposite implementations exist [57]. A short list of the main languages is available at [18]. UPC [59], Coarray Fortran (CAF) [102] and

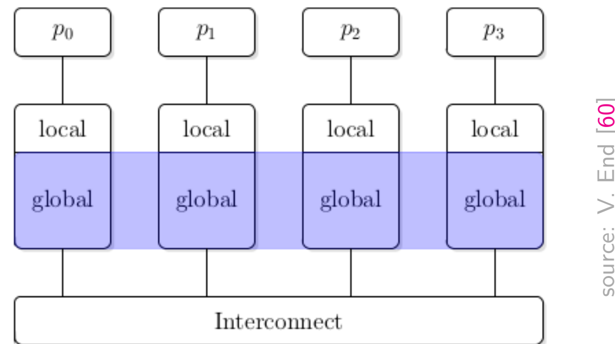


FIGURE 2.9: The PGAS model

Titanium [124] are extensions to respectively C, Fortran and Java Languages. The user can define affinities between the data and a piece of memory, but the underlying communications to access these data are implicit. DPC++ is an extension to the C++ language which defines distributed objects with also implicit communications [137]. Chapel [1], X10 [52] have been explicitly designed for respectively Cray and IBM architectures. OpenSHMEM [51] and GASPI [77] are PGAS APIs. They offer explicit communication routines. The main difference between both lies in the memory management and notifications system. OpenSHMEM requires symmetric memory, which means exactly the same layout on all processors. GASPI offers more flexibility, since it permits to organize the memory differently on each processor. GASPI also provides a more complete system of notifications, which allows for message ordering. In comparison, OpenSHEM only offers synchronization between pairs of processors.
The PGAS implementations ecosystem is complex and still evolving. The performance depends on the runtime, the compiler and even the architecture. Implicit communications simplify the adoption of the PGAS standard, but they may lead to poor performance. Explicit communications are more difficult to handle, but they can be finely tuned to minimize the costs. A similar reasoning can be applied to cache-coherency support. Another difficulty relies in the fact that switching for the PGAS model necessitates complete rewriting of the communications of the entire application.
Among various PGAS API's, the GASPI specification, presented hereafter, is a more recent approach which exposes the communication to the programmer while being inter-operable with the state of the art MPI library. This allows for inserting GASPI communications with no needing to rewrite the complete application. Therefore, it is possible to concentrate on the critical communication phases where significant gain in scalability and speedup is possible. Although being complex to use, the implementation is flexible. Efficient use for asynchronous applications has been demonstrated, see section 2.3.2.

2.3.2 The GASPI standard and the GPI library

The GASPI standard GASPI, for Global Address Space Programming Interface is a PGAS API. It is aimed at three main objectives: extreme scalability, flexibility and fault tolerance. GASPI has been developed with the objective of shifting from the bulk synchronous paradigm to a complete asynchronous model. As GASPI is intended for modern

and heterogeneous supercomputers, memory segments can be mapped on various architectures like: cache-coherent NUMA nodes, NUMA partitions, GPGPU, MIC, non-volatile RAM, SSDs. Communications are one-sided, they consist of writing or reading to or from a remote memory segment without involving the target. Communications are not buffered, no data is copied, and data is directly transferred from a segment memory to another. GASPI is thread-safe: memory segments are accessible by any thread from any process, so that communications can be handled at thread level. GASPI is fault tolerant: it offers a timeout mechanism which can be used to detect failure [112]. It allows growing and shrinking of the number of nodes.

The GPI library GPI is the library implementing the GASPI standard, developed by Fraunhofer ITWM [77] since 2005. GPI is supported by the GASPI project [5] with the aim of developing a robust and reliable library for the HPC community. The library is based on the Infiniband [6] architecture and its verbs. GPI uses the *ibverbs* library, which is available on almost all clusters, to call underlying communication functions using RDMA. Therefore, it is possible to rely on the network layer to completely handle the communications, without intervention, neither from the CPU nor from the OS. Nonetheless, it also support Ethernet network, for the sake of portability during the development phase.

Interoperability with MPI GPI is interoperable with MPI. This allows progressively replacing performance critical parts of large existing codes. It is possible to introduce GASPI communications in a portion of an existing code already parallelized with MPI. The application must be initialized and started with MPI. GPI detects the MPI environment at runtime, and builds its own setup, conserving the numbering of processes. It is possible to call an external library to handle the communications with GASPI. It is also possible to switch between MPI and GASPI sections, for example in an iterative code. Memory segments can be used by MPI and GASPI simultaneously. The only restriction is that GASPI and MPI communications cannot coexist in the same code portion. This has to be ensured by the use of barriers: MPI_Barrier to end an MPI section and switch to GASPI, and respectively, GASPI_Barrier to switch back to MPI.

GASPI_write_notify example Effective handling of the communication is left out to the application developer. Memory is managed by the application. Segments are dynamically allocated and eventually binded to existing buffers. Communications define the source and target segments, the offsets, the quantity, the communication queue, and the timeout, as illustrated in figure 2.10. In this example, the communication blocks until completion. All GASPI calls are encompassed in the *success or die* function. In case of error, this function stops the program and prints error information containing the file, the line and the error id.

Overlapping GPI provides one-sided non-blocking asynchronous communications, atomic operations and collective primitives. Figure 2.11 illustrates an example of asynchronous one-sided communication with GASPI and the overlapping opportunity. Node 1 is the source while Node 2 is the target. The source node initiates a non-blocking write operation that does not involve the target. The non-blocking call returns immediately, even if the communication may not be terminated. GPI provides remote completion via a notification mechanism. The source notifies the target with a *notifyID* and a *notifyValue*. These two tags are usually used to provide information characterizing the source rank and the transmitted data. At some point, when the target needs to know if the

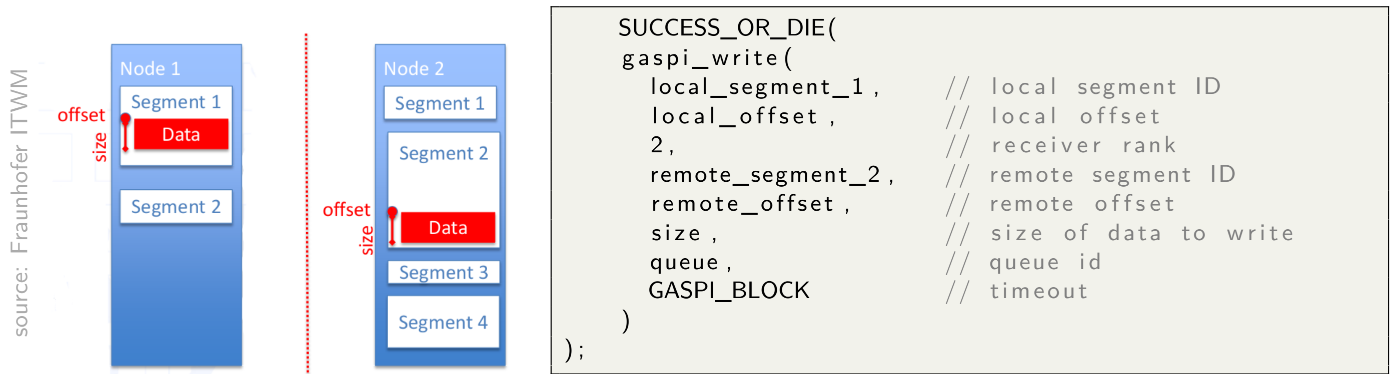


FIGURE 2.10: Communication example with Gaspi

information is available, it can check for the corresponding notification. If the source node needs to know that the target has received the data, the target can send a notification back to the source. In the meantime, the source node progresses its computation, and this ways, overlaps the communication, which permits to hide the communication latency.

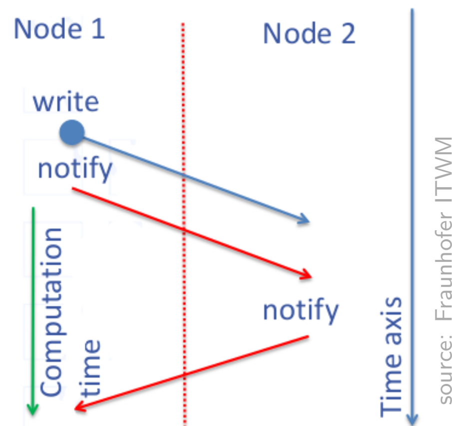


FIGURE 2.11: GASPI asynchronous and overlapped communication example

Queues GASPI specifies communication *queues*. Management of these queues is also left out to the application. It is possible to use multiple queues in an application. This provides different channels for different requests, which can be used, for example, for separation of concerns. GASPI ensures fairness between the queues: one queue will not be waiting indefinitely until other queues are treated. Queues have a finite capacity and need to be purged to avoid overflowing. Therefore, it is necessary to check the remaining room and eventually flushing, before posting a

new request. Communication requests and their corresponding notifications have to be posted in the same queue. This way, the arrival of a notification, guarantees that all preceding communications requests from the same queue, are completed.

Atomics GASPI provides atomic integers which can be modified through atomic procedures. These variables are shared over the global memory and can be used as shared global counters. The atomic procedures are *fetch and add* and *compare and swap*. *Fetch and add* retrieves a variable and increments it with a value. The *compare and swap* procedure, compares a remote variable to another one and if equal, replaces it.

Collectives Finally, GASPI also offers collective operations. It is possible to use all ranks or to define a subset. *Gaspi_barrier* and the *gaspi_allreduce* are built-in functions. The reduction can combined with classical *min*, *max* or *sum* operations, or with a user defined one.

Performance Different experiments comparing GASPI and MPI have been published.
In [110], Shahzad et al. compare the two communication models with two algorithms: the SpMVM (Sparse Matrix Vector Multiplication) and the LBM (Lattice Boltzman Method). For both algorithms they develop five different versions: blocking MPI, the non-blocking MPI, non-blocking MPI supported with APSM (Asynchronous Progress Support for MPI), synchronous GPI and asynchronous GPI. APSM [126] intercepts the non-blocking MPI calls and uses a thread for message progression. Of course, it requires a thread-safe library. The experiments show that the GPI gain depends on the possibility to replace global synchronization by smaller more relaxed synchronizations. Therefore, on the SpMVM test case, the gain depends on the matrix and the number of nodes, whereas on the LBM test case with GPI 95% of communications are overlapped, compared to 50% in the best case and with the help of APSM.

In [76] Grünewald et al. lead a similar experience on the BQCD (Berlin Quantum Chromo Dynamics) [101] application. BQCD is a hybrid Monte-Carlo algorithm that simulates lattice QCD. The compute kernel is a four-dimensional stencil. The GPI version performs complete overlapping of the communications. Compared to MPI, the speedup is of the order of 20 to 30%.

In [111], Simmendinger et al. use the unstructured CFD solver TAU [87]. The obtained speedup is of order of magnitude higher than the standard MPI. Similar studies are also presented in [93], where the authors obtain a gain of factor two by parallelizing an Adaptative Search algorithm.

[103] presents an experience of using GPI with GPUs. As long as the communications can be overlapped, which means on small number of nodes GPI, outperforms MPI.

To sum up, GASPI is an efficient and flexible communication library. Switching for GASPI requires deep application modification. But thanks to the inter-operability with MPI, it possible to make changes progressively. Compared with traditional two-sided MPI, the learning curve is lower and the programming effort is higher. The GASPI model

has been developed with the objective of extreme scalability, and recent experiments demonstrate its effectiveness.

In order to take full advantage of the parallelism exposed by hybrid architectures, it is necessary to combine distributed parallelism with shared memory parallelism, which is the topic of the next two next sections.

2.4 The Worksharing model

To take advantage of the multiple cores of a node, it is necessary to make efficient use of concurrent programming. If the application is compute intensive, it may be possible to decompose the work and to distribute it among the cores. Sometimes, processes, introduced in the last section, are also used at node level. But this may not be the right choice, as creating a process generates overhead by duplicating the address space and communicating between processes is complicated and costly. On the contrary, threads, also called light weight processes, are ten to thousand times faster to create than processes, since they duplicate less memory. Each thread has its own stack for the private variables and shares the same memory space with the other threads, making the communication process easier and cheaper. Threads are perfectly suited for fine grain shared memory parallelism. In this section we explore different multithreading models dedicated to parallelize code at node level.

2.4.1 The OpenMP standard

In 1997, OpenMP has been adopted as an industrial standard by industry researchers and manufacturers. It aims at simplifying multithreaded programming in Fortran and C/C++ [48]. Still largely used in industry and academia, it has been continuously improved and extended. It has evolved with the modern heterogeneous architecture trend and now also targets accelerators, integrated systems and real-time systems. OpenMP specifications are now handled by the ARB, Architectural Review Board, composed by major computer hardware and software actors[16].
The main features from major releases are summarized hereafter: OpenMP 2.0, published in 2000, adds extensions to parallelization possibilities for new Fortran 95 language. In 2008, OpenMP 3.0 adds mainly the possibility to handle explicit tasks. In 2013, OpenMP 4.0, brings offloading possibilities to accelerators, thread placement, portable SIMD vectorization depending on the target architecture, and tasks dependencies management. Finally, the OpenMP 5.0 version, released in 2018, is considered as a major leap forward [107]. It deepens the support of accelerators, updates the language support to Fortran 2008, C++11 and C++17, improves the loop construct to give more freedom to the compiler to make the right choice depending on the target, enhances the memory allocation mechanisms for multi-level memories, and also improves debugging and performance analysis with new tool interfaces to support the development of third party tools..
OpenMP offers compilation directives and clauses by the way of configurable pragmas. Those pragmas are interpreted by the compiler, to produce the parallelized code. During the execution, the program calls functions available in the OpenMP library. It is also possible to set some environment variables to configure the execution at runtime. At the beginning, an OpenMP program is sequential, with only one master thread. A *parallel region* defines a portion of code that can be executed in parallel. At the first parallel section, a pool of threads is created. On the following parallel regions, the work is allocated to the existing thread from the thread pool. Each thread executes an assigned implicit task, concurrently with the other threads. An implicit task is an instance of executable work combined with a piece of data. A standard OpenMP program alternates sequential and parallel regions, according to

the fork and join model, in a bulk synchronous way, see figure 2.12. Creating a parallel region and ending it induces runtime overhead, as the master thread has to create new threads at the beginning of the region and synchronize them at the end. Iterating on this pattern accumulates the delays.

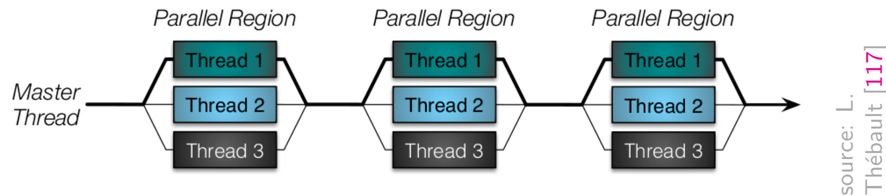


FIGURE 2.12: Bulk Synchronous parallelism.

For multithreading, OpenMP offers two parallelism models relying on worksharing and tasks. Worksharing is described in section 2.4.2 and 2.4.3, and tasks are introduced in section 2.5.1.

2.4.2 The OpenMP loop parallelization

HPC applications often use large loops dealing with very large datasets. If each iteration is independent, it is possible to parallelize these loops by distributing the iterations among the threads. Loop parallelism is also a form of worksharing, but it relies on distribution of the data, in a SPMD (Single Program Multiple Data) way: the same portion of code is applied to different pieces of data.

Load balancing issues Nonetheless, some iterations can be much shorter to execute than others. This leads to an uneven amount of work between threads, generating load imbalance and performance degradation. Figure 2.13 shows an example of load balancing results obtained with static and dynamic scheduling. This example focuses on a case where iterations become shorter along the iterations. On the left side, statically distributing the iterations in a cyclic mode leads to work imbalance, whereas on the right side, using a dynamic strategy compensates the imbalance.

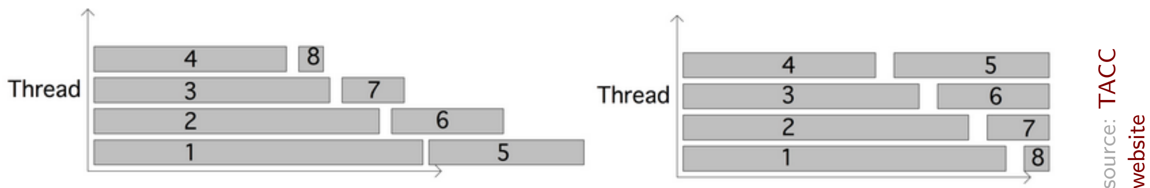


FIGURE 2.13: Static vs Dynamic scheduling.

Loop scheduling OpenMP provides the *OMP_PARALLEL_FOR* directive for loop parallelization, along with the SCHEDULE clause and three scheduling strategies, illustrated in figure 2.14 and detailed hereafter:

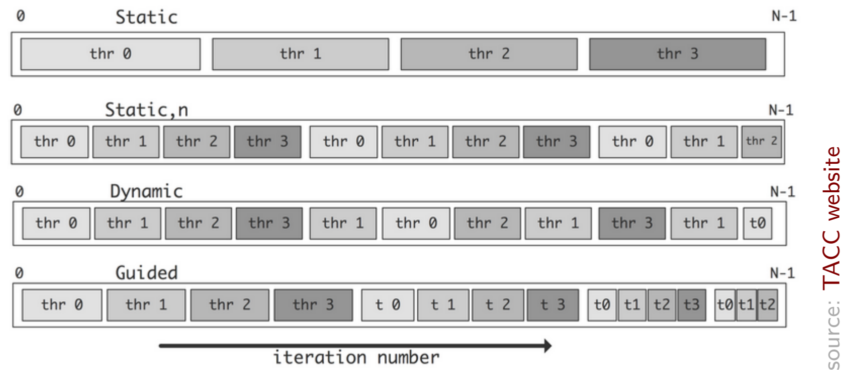


FIGURE 2.14: OpenMP loop scheduling.

Static Static scheduling is handled at compile time. Iterations are divided into chunks of a same given number of iterations (except the last one) and assigned to the threads in a round robin order. The size of the chunk can be configured in the scheduling clause. This strategy would be efficient if the chunks would be of equal time. The size of the chunk is often a compromise between load balancing and runtime overhead. On one hand, the more tasks there are, the more rooms there is for load balancing, but on the other hand, creating and scheduling a large number of tasks may be costly.

Dynamic In dynamic mode, iterations are also divided into equal sized chunks. This size can also be configured, by default it equals one. At runtime, the chunks are handled by a scheduler which maintains a pool of tasks. When a thread has finished its current task, it checks the pool for another one. This strategy can result in much better load balancing, but the default size may be too small and thus generate overhead. Moreover, the use of the scheduler also implies some overhead.

Guided Iterations are also divided into chunks, but with exponentially decreasing sizes. The bigger chunks are distributed at first, and the smaller chunks at the end. The objective is to diminish the overhead with larger chunks in the beginning, while load balancing the work with smaller chunks in the end.

Auto Finally, the auto option leaves the choice to the compiler and the runtime library.

Data sharing attributes Whether it is for loop parallelism or any other form of parallelism, OpenMP provides Data Sharing attributes. Finely managing the scope of each piece of data is important to avoid race conditions and correctness. *Shared* variables are visible by all threads. They can be read or written by any thread. *Private* variables are hidden from the other threads. Each thread owns a local copy on its stack. Two additional clauses can modify the value of a variable: *Firstprivate* initializes the private variable with the value before the parallel construction, and *Lastprivate* updates the variable at the end of a parallel construct.

Reduction Parallel loops can also be completed by *REDUCTION* clause. In this case, each thread computes a part of the result, and when all the iterations are completed, the final result is aggregated. Available operations are arithmetical, logical or intrinsics: with the +, -, x, logical and, or, equal, not equal, min, max, iand(bitwise and) ior(bitwise or) and exclusive or.

Loop fusion Finally, in order to increase the iteration space, it is possible to collapse independent nested loops with the *COLLAPSE* clause.

2.4.3 The OpenMP worksharing constructs

Work-sharing control enables the construction of different parallel sections in a portion of code.
The *workshare* construct is similar to loop parallelization. Instead of distributing the iterations of a loop, it distributes, for example work on an array. In this case, the work is divided into units of the size of one element. Unfortunately the size cannot be modified, and this parallelization mode is known to lead to significant overhead.
Work-sharing also enables the construct of different independent portions of codes, executing different operations, and running concurrently, in a MPMD (Multiple Program Multiple Data) way. To that end OpenMP introduces parallel sections. The *SECTIONS* clause defines a portion of code which will be distributed among the threads with the *SECTION* clause. A section can be executed only by one thread. Independent portions of code can be distributed to different threads.
Sometimes, it can occur that some portions of code, inside a parallel region, have to be kept sequential. OpenMP provides different mechanisms. The SINGLE or MASTER clauses imply that only one thread executes the code: either anyone, or explicitly the master. The CRITICAL directive starts and ends a critical region. All threads will execute it, in a non-deterministic order, but only one at a time. ATOMIC operations guarantee that the atomic variable is accessed by only one thread at a time.

The work-sharing model is thread-centric. It could work well for regular well-structured applications with perfect load-balancing. But it reaches its limits on irregular applications with load balancing issues. Task-level parallelism seems to be better suited for these classes of applications.

2.5 The task-based model

2.5.1 OpenMP tasks

Tasks were introduced in OpenMP 3.0 to overpass the limit of the worksharing model. They are intended to parallelize irregular problems, dynamic problems and recursive algorithms. OpenMP 4.0 introduced the dependency and the taskgroup, and OpenMP 4.5 the taskloops. Finally, OpenMP 5.0 added the possibility of making a reduction among tasks.
Unlike implicit tasks, created by a parallel construct, explicit tasks are created by an explicit *task* construct. When a thread encounters a task construct, it creates a new explicit task containing its code and environment data. A task may be executed by a thread, concurrently with other tasks. Implicit tasks could not be deferred, whereas explicit ones may be executed immediately or handled by a scheduler. In this case, they are inserted in a pool of tasks, to

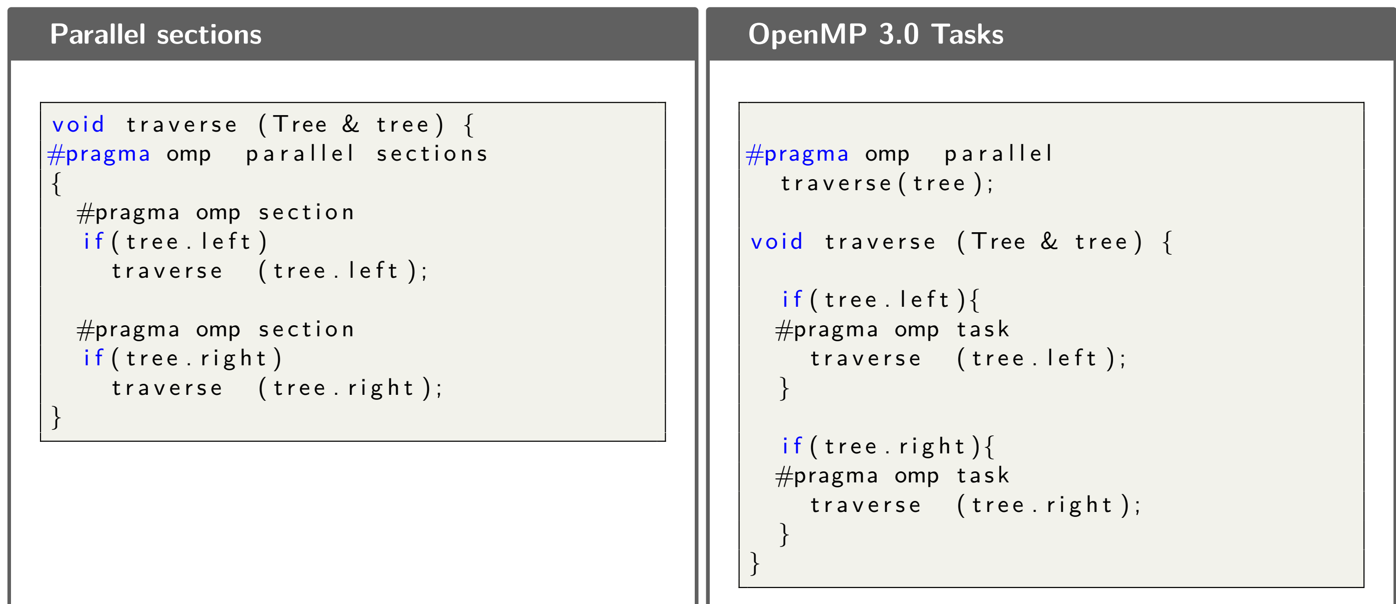


FIGURE 2.15: From parallel sections to tasks

be executed by a group of active threads. Task constructs can be nested, thus allowing for recursive algorithms. Figure 2.15 illustrates two ways of traversing a tree, the first one using parallel sections and the second one using recursive tasks.
Tasks can be generated in a producer/consumer pattern. One thread executes a *single* region and spawns all the tasks, which will be executed by the threads of the same pool. This model is illustrated in figure 2.16.

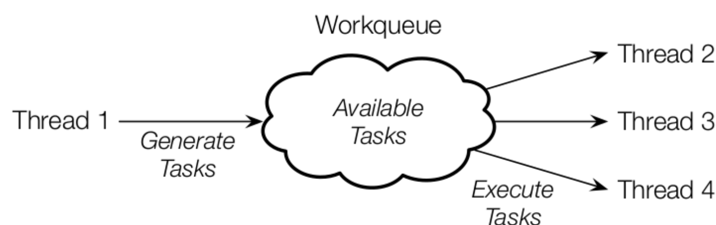


FIGURE 2.16: OpenMP Producer/Consumer model

Dependencies Dependencies management clauses, introduced in OpenMP 4.0, allow for finer synchronization between tasks. The *DEPEND* keyword specifies variables and access modes, which can be set to *in*, *out* or *inout*. This indicates if the variable will be an input, an output or both, and respectively accessed in read, write, or both

modes. These specifications put constraints on the execution order. Thus, at runtime, the scheduler will decide to execute a task or to delay it, depending on its readiness. This model is well-suited to express task flow dependencies. However, the OpenMP 4.0 standard specifies that the dependency clause only applies to sibling tasks. The notion of sibling in a task context means that the tasks are created in the same task region. Therefore, if a task spawns another task, they won't be siblings. In our opinion, this is a very restrictive condition which prevents from using the dependency clause in a recursive parallel algorithm. Moreover, the thinness of the expressed dependencies between tasks may be critical: Too fine dependencies may lead to runtime overhead, while too coarsen dependencies may hide concurrency opportunities [123].

Task synchronization OpenMP provides task synchronization directives. The *taskwait* directive waits for completion of the tasks from the current block. The *taskgroup* directive defines a parallel task region with an implicit task scheduling point at the end: the current task waits for completion of tasks of the region including tasks which were recursively generated. The *taskyield* directive allows for temporarily suspending the execution of a task at a scheduling point. The taskwait, taskgroup, taskyield, task construct, implicit or explicit barriers, and the completion point of a task are scheduling points. At all scheduling points a task can be paused and restarted by another thread. This allows more room for effective scheduling.

Task scheduling While OpenMP provides a model to allow specifying units of work, the way of scheduling is left out to the implementation. Therefore, the efficiency of the execution of task-based code is dependent on the compiler and the runtime library. Task scheduling, load balancing and cut-off strategies can heavily impact performance.
Two main families of task schedulers are identified in the bibliography, [58]: the *work-first* and the *breadth-first* approaches, and in most cases the work-first strategy delivers the best results. Nonetheless, OpenMP introduces the *tied* and *untied* characteristics which add new constraints for the scheduler. A tied task is attached to a thread, and cannot be executed by another one. It can be paused at a scheduling point, but will be resumed by the same thread. An untied task can switch thread, which provides more room for load balancing. The OpenMP default mode consists in creating tied tasks. More specifically, tasks are tied to the first thread that executes them.
The principle of the *Breadth-first* scheduler is to continue the execution of the current task, while adding the created ones in a pool of tasks. This way, the current level of recursion will be terminated before moving to the next one. Two task pools are maintained: a private one and a team pool. The team pool is shared across the threads of the task region, whereas the private one belongs to one thread. Tied tasks are inserted in the private pool, and untied tasks go into the team pool. When a thread needs to pick up a new task, it first checks its private pool. On the opposite, *Work-first* schedulers execute the newly created task first. The rationale is to profit from the data locality from the original sequential algorithm From the data structure point of view: each thread maintains a private pool. The current task is suspended and inserted into to the thread's local pool. When a thread needs a new task, it will check at first its local pool before trying to steal from other threads' pools. Experimental results from Duran et al. [58], show that, with untied task, work-first schedulers obtain the best results. But on the contrary, tied tasks lead to poor performance with this strategy, and obtain better results with breadth-first scheduler.
Task granularity [38] is also an important criteria to consider: while creating a large number of tasks can help the scheduler to load balance the work, the creation of the tasks may induce important overhead. Therefore different *cut-off* strategies exist. In [58], two cut-off strategies are described: the first one limits the total number of existing

tasks and the second constrains the recursion depth. Nonetheless, the choice of an appropriate strategy depends largely on the application and the test case.
Extended research on task scheduling has been carried out. In [86], Korch et al, describe different task pool based schedulers and experiment their efficiency. In [27], Acar et al. demonstrate that taking into account data-locality while stealing delivers best performance. In [68], Frigo et al. focus on reducing the overhead of spawning tasks in the Cilk-5 implementation. In [105], the authors use the UTS (Unbalanced Tree Search) [104] benchmark to compare different OpenMP 3.0 tasks implementation, focusing on runtime overhead and load balancing. Their results show that OpenMP 3.0 tasks struggle to scale on very unbalanced task graphs, and that the best results are obtained with the Intel ICC compiler. Nonetheless, the compiler being commercial, there is no information about the implementation. In [123], Virouleau et al. lead an experimental evaluation of the new OpenMP 4.0 tasks. They demonstrate, with different compilers, libraries and hardware platforms, that dependent tasks achieve comparable results than standard tasks. They even obtain excellent results on a more dedicated data flow application. Nonetheless they also put the emphasis on the fact that the adoption of these new OpenMP features will be largely based on the compilers and runtimes ability to implement the new standard efficiently.
Among several work-first implementation, the Cilk-5 library is often identified as being one of the most efficient. It is described in more details in the next section, 2.5.2.

2.5.2 Cilk tasks

Intel Cilk Plus is a task-based runtime aimed at multithreading parallel programming. Cilk was initially developed in the mid 1990's by the MIT [43]. Then, in 2006, the startup Cilk Arts was launched, and Cilk becomes Cilk++ and benefits from language improvements. In 2009, Intel acquires Cilk Arts. Cilk++ becomes Intel Cilk Plus [91] and is released along with the Intel C++ compiler. It is also supported by the GCC compiler.
Intel Cilk Plus is composed of a language, a compiler and a runtime. It is an extension to C/C++ language with parallel constructs. Three main keywords suffice to express task parallelism: *cilk_spawn* to spawn a new task, *cilk_sync* to locally synchronize between tasks and *cilk_for* to parallelize loops. Figure 2.17 shows how the tree traversal from figure 2.15 would be translated into Cilk tasks.

```
void traverse (Tree & tree) {
  if(tree.left){
    cilk_spawn traverse(tree.left);
  }

  if(tree.right){
    cilk_spawn traverse (tree.right);
  }

  cilk_sync;
}
```

FIGURE 2.17: Recursive tree traversal with Intel Cilk Plus tasks

Cilk also provides loop parallelism with the cilk_for keyword. It parallelizes a loop by recursively dividing the ranges, thus creating a tree of parallel tasks. Intel Cilk Plus *reducers* [67] have been added to allow for reductions among the tasks. The Intel Cilk Plus array notation is inspired by the Fortran array notation and helps the compiler auto-vectorize the code.

Bounded execution time Cilk gives a theoretical view of parallelism and introduces the notions of *work* and *critical path*. Work is the total time needed to compute the parallel region serially. The critical path length is the execution time with an infinite number of processors. The theoretical analyses demonstrate that the parallel execution time is bounded, and therefore predictable. Let T_p represent the execution time on a p number of processors. The work of the computation is T_1, and the critical path is T_∞. The, T_p is the sum of two bounds: parallelized computation and the critical path.

$$T_p = T_1/p + O(T_\infty)$$

Overheads, both in terms of work and critical path, have to be added to this formula. The authors demonstrate that overhead on the critical path, instead of work, has far less impact on the performance, which justifies the choice of a work-first scheduling strategy. In-depth demonstration can be found in [43] and [42].

Work-first scheduling and work stealing As a consequence of the theoretical study, Intel Cilk Plus uses a work-first scheduler with work stealing. The compiler and schedulers are designed to reduce the work overhead as much as possible.
When a new task is spawned, the compiler creates two versions called *fast clone* and *slow clone*. The fast clone is a simple version of the task designated to be run immediately, whereas the slow clone has full support for parallelism. For each spawned task, the compiler will generate both versions, and according to the work-first principle, in most cases, the fast clone runs immediately.
Each processor, called a *worker*, maintains a doubly-ended queue, a *deque*, which two ends are the *head* and the *tail*. Locally, a thread operates on its tail in a *fifo* mode: the last spawned task is executed first, favoring data locality. When it has no work anymore it becomes a *thief*, and will try to steal some work from another thread's deque called *victim*. Remotely, it will work on the opposite side of the deque: the head. The work-stealing is implemented following an optimized protocol, called *THE (Tail, Head, exception)* [68]. When a thief tries to steal from another worker's deque, there are three possibilities, illustrated in figure 2.18. In the first case, there is enough work for both of them, and both workers get a task. In the second case, there is only one task available, and it will be allocated on a first-come first-served basis. In those cases when both try to get the task simultaneously, the owner will be privileged. Finally, if no work is available, theft or pop attempt will fail, and control returns to the runtime. With these work-stealing principles, idle threads steal from busy ones, and conflicts between both should be rare. It is only when a procedure is stolen, that the slow clone resumes and restores it's program counter and data environment. The Intel Cilk Plus scheduler guarantees that when enough parallelism is exposed, fast clones are executed most of the time and the number of steals is small, which contributes to reduce the overhead.

Software development kit Intel Cilk Plus comes with two powerful programming tools named Cilkscreen [63] and Cilkprof [78]. Cilkscreen is a race condition detector. It instruments the code and checks for all possible executions

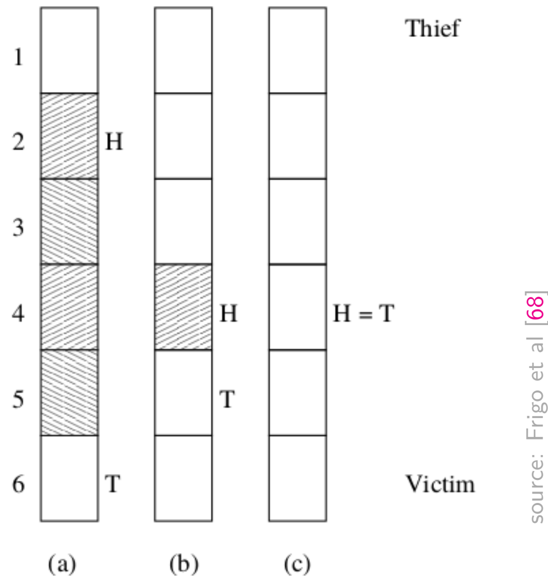


FIGURE 2.18: Intel Cilk Puls work-stealing protocol

to identify possible race conditions. As an output it indicates precisely the concerned variables and the type of accesses. Cilkprof, initially Cilkview [78], is a profiler for multithreaded cilk code. It instruments the code and gives information about parallel regions, which provides insight to prevent bottlenecks.

2.5.3 Task-based schedulers

The task model is evolving and the scheduler's ecosystem is wide. We briefly present those who focus on tasks management. Quark [127] and StarPU [37] dynamically build the task graph. StarPU supports heterogeneous architectures, while Quark does not. Both have their own schedulers. Nonetheless, these are centralized, which can lack scalability. Moreover, their data flow graph is flat, which means that they do not support recursivity. OpenMP 4.0 introduced the depend clause to manage tasks dependencies and is also evaluating towards supporting heterogeneous architectures. However, the depend clause does not support recursivity, and the expression of dependencies is limited to one-dimensional arrays. Kaapi [70] allows recursivity and loop parallelization. It also handles the expression of dependencies on multi-dimensional arrays, which can be useful, for example for sub-matrices dependencies. OmpSs [45], [46] uses the Nanos++ runtime and a master-slave model. It supports a wide range of architectures, but the master-slave model used could lead to scalability issues. Legion [41] is a programming model organized around regions and tasks, and expresses a data flow with dependencies. It uses the low level Realm [121] runtime based on GASNet and the underlying communication layer. It supports shared and distributed memory as well as heterogeneous architectures. PaRSEC [81] managed local and distributed dependencies with a data-flow task-based

runtime. Cilk and TBB [109] create only independent tasks, but they have focused on the task creation costs. Some schedulers support for loop parallelization, like TBB, Cilk Plus, OpenMP and Kaapi. Some others include a cut-off strategy to limit the number of created tasks, like the GCC libGOMP and TBB. Interesting schedulers comparison can be found in [44] and [81].

2.6 Conclusion

In this chapter we have put the emphasis on hybrid parallelism combining internode communications and intranode shared memory parallelization.

At the distributed level, we have presented in detail the MPI and GASPI communication standards. MPI communications are largely dominant in HPC applications. In order to increase performance, asynchronous and one-sided communication primitives are proposed. However, lacks of progression and synchronization overheads subsist. In comparison, GASPI offers efficient hardware-based asynchronous one-sided communications. State-of-the-art experiments show a considerable improvement in performance over MPI. The target application of this thesis, presented in section 4.2, was initially parallelized with MPI. Part of this thesis consists in optimizing the communication scheme by introducing asynchronous one-sided communications with GASPI. This work is presented in section 7.

At the shared memory level, we have introduced worksharing and task-base parallelization with mainly OpenMP and Cilk. On the one side, the work-sharing model is based on parallel sections and parallel loops, and performs well on regular problems. However, on irregular applications, it may suffer from load balancing issues. On the other side, task-based parallelization is intended for irregular, dynamic and even recursive algorithms. Another part of this thesis, presented in section 8, consist in optimizing the intranode parallelism, by moving from a bulk synchronous fork-join model to a dynamic task-based model. Analysis of the different task-based libraries and runtimes, indicates that for our Fortran Application, and despite some limitations, the OpenMP tasks seems the most appropriate. Our final objective consists in mixing both, in order to obtain a totally asynchronous and hierarchical algorithm.

Chapter 3

Fast Multipole Method

Contents

3.1 Introduction **35**
3.2 Fast Multipole Method **35**
3.3 Parallelization strategies **40**
3.3.1 Low-level optimization 41
3.3.2 Fork-join implementation 41
3.3.3 Tasks 42
3.3.4 Load balancing 45
3.3.5 Communications 48
3.3.6 Accelerators 50
3.4 Conclusion **50**

3.1 Introduction

The main contribution of this thesis consists in modernizing Dassault Aviation's electromagnetism SPECTRE application, to be presented in chapter 4. It uses the MLFMM (Multi-Level Fast Multipole Method) to solve the Maxwell equations. In this chapter, we introduce the general FMM (Fast Multipole Method) algorithm in section 3.2 and present state-of-the-art parallelization methods for this algorithm in section 3.3.

3.2 Fast Multipole Method

The FMM (Fast Multipole Method) was introduced in 1987 by L.Greengard and V. Rohklin [71] and is intended to reduce the complexity of N-body problems from quadratic to linear. The FMM is used in a large range of application domains in physics and computational mathematics, like astrophysics, molecular dynamics, fluid dynamics, plasma simulation, acoustics or electromagnetism. It is part of the 20th century top ten algorithms [21].
The N-body problem can be summarized as the simulation of individual motions of a group of objects interacting with each other under a force. The Fast Multipole Method (FMM) is an algorithm which allows fast computation

of the sums:

$$f(x_i) = \sum_{j=1}^{N} P(x_i, y_j)(w_j), \tag{3.1}$$

given a system with N *source* particle positions $y_1, ...y_N$ and M *target* particle positions $x_1, ...x_N$. $f(x_i)$ is the computed potential at target point x_i, w_j is the *density* at the source point j and P is the governing interaction kernel. The summation can also be viewed as a dense matrix-vector product, P being a matrix of size MxN. In most cases, the sources and the targets are identical, thus, assuming $M \approx N$, the complexity of the direct calculation is of order $O(N^2)$. P is induced by the physics of the problem, and depending on its nature, the cost of the matrix-vector products can be reduced from in $O(N^2)$, to $O(N \log N)$ or $O(N)$ in the best case.
The FMM algorithm relies on two key ideas:

- The influence of faraway particles becomes smaller as the distance to the target particle grows. More specifically, the kernel $P(x, y)$ decays as the distance $|y - x|$ increases.
- These interactions can be approximated, while maintaining a user-defined accuracy.

Under these conditions, the influence of a cluster of particles can be approximated by a single representation.

Approximation and distance criteria The FMM uses an accurate approximation of the kernel and a criterion of well separation. The rationale is to model the distant point-to-point interactions by hierarchically grouping the points into single equivalent points. To make the approximation valid, the group of points has to be far enough from the target point: this is the *well-separated* property.

Multipole and Local expansions The FMM introduces two mathematical operations: the *Multipole Expansion*(ME) and the *Local Expansion*(LE). Both are truncated series expansions that converge in different subdomains. The multipole expansion converges outside of a cluster of particles, and after a distance criterion, and the local expansion converges inside a cluster. The multipole expansion determines the influence of a cluster of particles on an evaluation point. The series are centered on the center of the source particles cluster and are valid only outside of the cluster and provided a distance criterion is satisfied. Inversely, the local expansion evaluates, inside a cluster, the influence of particles that are far away. It is centered close to an evaluation point. It is the use of these multipole expansions and local expansions combined with factorization which leads to linear complexity [56].

Distance criteria Distinction between the *near* field and the *far* field is made through a *distance* criteria. Equation 3.1 can be decomposed as:

$$f(x_i) = \sum_{l=1}^{N_{near}} P(x_i, y_l)(w_l) + \sum_{k=1}^{N_{far}} P(x_i, y_k)(w_k), \tag{3.2}$$

where the left sum is computed directly, and the right sum is evaluated using the approximation.
Figure 3.1 illustrates this principle in a two-dimensional space. The quadtree, based on domain decomposition, defines the near field and the far field for the particles located in the green square. The red particles are *well-separated*: they form the far field and their interactions are computed using the far field approximation. On the

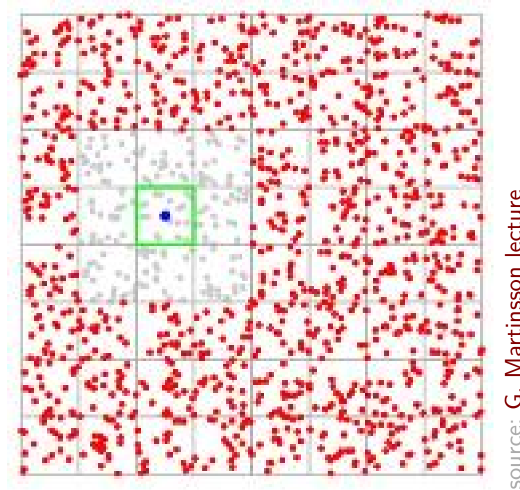


FIGURE 3.1: Near Field and Far Field characterization in 2D.

contrary, the grey particles outside the green square are not well separated, they form the near field and their interactions are computed directly.

Hierarchical domain decomposition To apply the FMM algorithm, the domain is recursively partitioned into subdomains. The recursive subdivision is represented by a hierarchical tree, the type of which depends on the problem dimension. Usually, quadtrees are used for two-dimensional domains and octrees for three dimensions or more.
For each node of the tree, two types of neighboring nodes are defined:

- The commonly named *U-list*, which contains all the adjacent nodes, forming the near field of the current node,
- and the *V-list*, or *interaction list*. It contains the children nodes whose parent is adjacent to the current node's parent, but that are not adjacent to the current node.

Figure 3.2 defines both neighborings on a uniform two-dimensional space represented by a quadtree. Non uniform distributions can be handled either with adaptative trees, or with a standard tree but at the cost of empty nodes. Moreover, they are difficult to balance over a set of processors and can generate heavy communications. An example of adaptative octree is illustrated in figure 3.3. One can also note that two other neighborings are defined: the W-list and the X-list.

Two-phase tree traversal The FMM algorithm is applied using a two-step tree traversal, represented in figure 3.4. The upward phase is illustrated on the left side of the octree and the downward phase on the right side. The upward pass aggregates children's contributions into larger parent nodes, starting from the leaf level. The

source: Chandramowlishwaran's Thesis

FIGURE 3.2: U and V lists of a node B in a two-dimensional space and a uniform quadtree. U-list contains the near field and V-list contains the far field interaction list.

source: Lashuk et al.[90]

FIGURE 3.3: Adaptative quadtree covering a non uniform distribution.

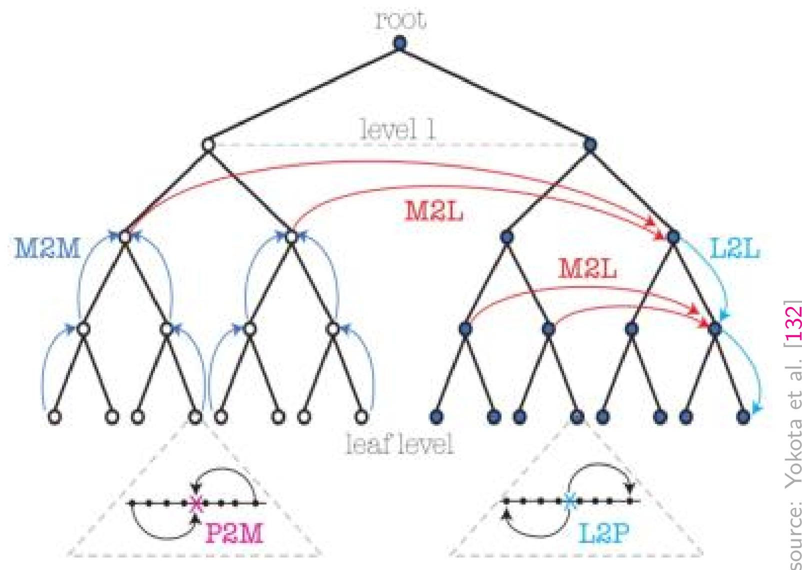


FIGURE 3.4: FMM two-phase tree traversal operators

downward pass collects the contributions from source cells at the same level, and sends the values from parents down to children until the leaf level.

FMM Operators The FMM algorithm is commonly organized into the following operators:

- *Particle to Multipole (P2M)*: The P2M operator is activated at leaf level. It collects the contributions from all particles inside a leaf to form the first multipole expansion.
- *Multipole to Multipole (M2M)*: The M2M operator (see left side of figure 3.5) traverses the tree up. It translates the contributions from the children up to the parents. Children's multipole expansions are accumulated to form the parent multipole expansion.
- *Multipole to Local (M2L)*: The M2L operator translates the multipole expansion of a node into the local expansion of another node from its interaction list. It is illustrated in the middle of figure 3.5. The interaction list from the node B is represented by the nodes marked with the letter V. To compute the local expansion at B, it is necessary to gather and accumulate all the multipole expansions from the interaction list. This step induces communications. If the domain is distributed between different processors, the M2L operator may generate communications through the network to send the contribution.
- *Local to Local (L2L)*: The L2L operator, illustrated in the right side of figure 3.5 traverses the tree down. It sends parent nodes contributions down to the children. There is a data dependency between the M2L and

the L2L operators: the M2L operator of level L has to be completed before the L2L operator can be applied to push the contributions down from level L to level L-1.

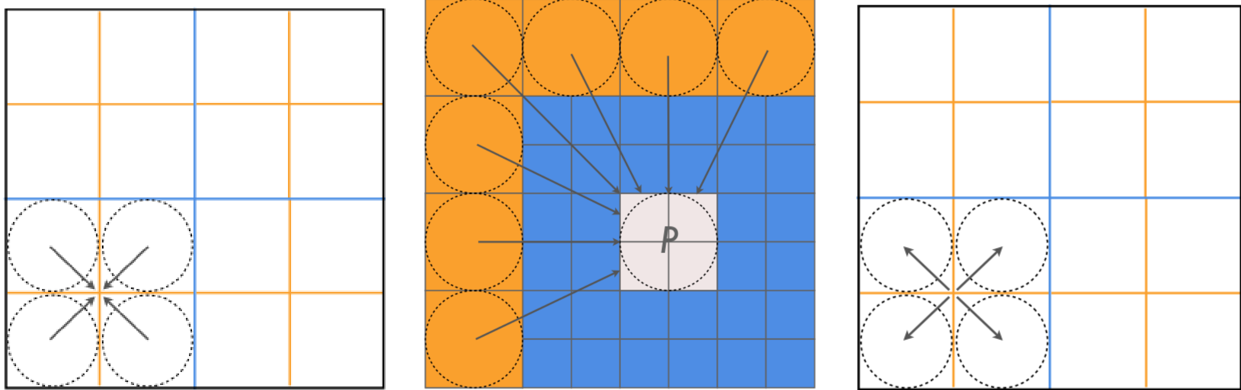


FIGURE 3.5: M2M, M2L and L2L operators

- *Local to Particle (L2P)*: L2P uses the last level local expansions to evaluate the far field contributions from all the particles inside a leaf. They are accumulated to the near field contributions to obtain the total potential.
- *Particle to Particle (P2P)*: P2P is the operator used in the near field. For each leaf, the interactions of the adjacent leaves are evaluated directly, using eq (3.1).

Data dependencies and reductions The FMM tree traversal contains data dependencies and reductions.

- The *data dependencies* impose a definite order for the operations in the tree traversal. During the upward pass, the first M2M has to be computed after the completion of P2M. Then to compute M2M from level L, the former M2M form level L-1 has to be completed. Similarly, during the downward pass L2L can be executed only if the former L2L and the contributions from M2L are calculated. Finally, L2P needs L2L to be terminated.
- Two *reductions* have to be managed carefully. M2L and L2L from the former level access the name node; and similarly L2P and P2P produce contributions to the same particles.

3.3 Parallelization strategies

The Fast Multipole Method is a complex algorithm, which is sensible to load balancing, and involves communication phases, multiple operators and data dependencies. Over the years, the algorithm has been parallelized and optimized at each parallelism layer. An extensive bibliography is available on this subject. In this section, we make a brief overview of the main optimization strategies.

3.3.1 Low-level optimization

Vectorization Chandramowlishwaran et al. present an extensive work on single-node optimization in [50]. They demonstrate their results with KIFMM [129] [128] using a Laplacian kernel. KIFMM, for Kernel Independent FMM, proposes a version relying only on kernel evaluations. Analytic expansions are replaced by equivalent densities computed by solving integral equations on spheres and cubes, and multipole to local translations are accelerated with FFTs (Fast Fourier transformations). Observing that the compiler does not vectorize the code, they apply manual vectorization using Intel intrisics instructions. They also modify the data layout to be vectorization friendly and switch from array-of-structures (AOS) to structure-of-arrays (SOA). However, this modification impacts the tree construction, but the overhead remains acceptable compared to the gain obtained on the computations.

In a similar way, in [132], Yokota et al. also vectorize manually with inline assembly. They use ExaFMM [3], an open source parallelized code with GPU capability, to support the optimizations and demonstrate their results. The P2P kernel obtains a 16x speedup, which results in a 4x speedup on the total application.

Other low-level intrinsic optimizations are explored in [50]. For instance, kernels use square root and divide operations, which are not pipelined. Depending on the architecture and the available instructions for single and double precision, they use inverse square approximations to benefit from pipelining and thus lower latency. Since this approximation is not available in double precision, they even use conversions to single precision and Newton-Raphson refinement, which leads to more operations but lower latency.

Low-level optimization has proven its efficiency, but is architecture dependent. The resulting code is tightly tied to the targeted architecture. Vectors are becoming larger and the intrinsics sets grow consequently. Each architecture evolution necessitates rewriting the manually optimized code. The low-level optimized codes can become less readable and difficult to maintain. We can note that some libraries target these optimizations by proposing a user-friendly abstraction with a higher level programming interface [14]. Moreover, the choice of the data layout is difficult. Choosing an optimized layout for vectorization may harm another more local-sensitive part of the code. For instance, one can also mention hybrid data structures like AoSoA (Arrays of Structures of Arrays) which try to mitigate those effects [85].

Cache misses Reducing data movement is important to optimize cache reuse. In [49] blocking and scheduling strategies are explored to reduce cache misses. The translation vector is blocked in sizes that fit into the cache. OpenMP scheduling is turned to static among the sockets, and cyclic inside. In this way, threads using the same cache should profit from more locality. Optimizations are tested on four platforms with different cache sizes. The authors obtain significant speedups and demonstrate the link with the cache size.

3.3.2 Fork-join implementation

Loop parallelism Standard loop parallelism is widely used to optimize the FMM algorithm among the multiple cores of a computing node [31] [88]. FMM operators are parallelized with OpenMP loop parallelization constructs

on the outermost loops. While this method has proven successful for regular problems, it may be less efficient on irregular algorithms. In the case of the FMM, since the loops iterate on complete tree levels, an implicit barrier is introduced at the end of each level. Furthermore, the iteration step is very small - a leaf or a node - therefore the granularity of work chunks may be very fine. A small number of work units increases the granularity and the concurrency, whereas a larger value limits the concurrency but favors data locality. In [31], the authors propose a parametrized version by grouping the nodes into blocks, following a Morton index, and iterating over the blocks. To identify the best trade-off between concurrency and locality, the authors tune the number of nodes per block. This method allows to mitigate overhead due too fine grain tasks. However, grouping multiple nodes inside a single task also coarsens the dependencies, and prevents from managing the dependencies at node level.

Interleaving near and far fields operations The near field and the far field computations are almost independent. Only P2P and the last FMM operator L2P access the same data. Therefore, it is possible to interleave the computations to reduce idle times.

In [49], the authors study concurrent execution of the P2P and M2L kernels. First, the scalability limit of the M2L kernel is identified through an experimentation phase. Therefore, if the operator scales until a certain number of threads, it is possible to assign a different work to the remaining threads. Concurrency between M2L and P2P operators is achieved via dispatching the threads between both operators. They experiment this strategy with one thread per core and also with hyperthreading. It allows for 60% to 73% improvement on execution time but does not profit from hyperthreading. They assume that the M2L kernel, which is memory bound, competes with the compute bound P2P kernel.

In [31], Agullo et al. interleave almost all the far field computations with P2P, except L2P. They create two parallel *OMP sections*. Two threads walk through these sections and spawn OpenMP tasks which are executed by the pool of threads. An implicit barrier before the L2P operator, also executed by the mean of tasks, ensures the correctness of the code. Tasks are still coarsened by grouping the tree nodes in order to mitigate the scheduler's overhead. The authors point out that this technique is efficient as long as the far field does not become prominent, which is the case at Dassault Aviation. However, when the far field computations equal or dominate the near field, the parallel efficiency is limited by the artificial synchronizations introduced at the end of the fork-join parallel loops.

3.3.3 Tasks

Data-driven models allow for avoiding global synchronizations by replacing them with small local synchronizations. They use schedulers to organize the execution of tasks.

Methods favoring data reuse In [34] Amer et al. propose a way to move towards a task-based algorithm while preserving the data locality. Instead of traversing the tree in a bottom-up manner to spawn the tasks, they traverse it inversely, from top to bottom. They use the massiveThreads [9] library, in which the first tasks in charge of spawning all other tasks may itself be preempted and stolen, thus creating random access to leaves data. However, starting from the top creates tasks that inherit from a complete subtree, which favors data reuse. Moreover, the decentralized scheduler alleviates the pressure on memory bandwidth by spreading the work across the cores.

Schedulers and Tasks Different runtime systems applied to data-driven execution of the adaptative FMM algorithm are explored in [135]. The objective is to examine the runtime's system efficiency to handle the tasks creation, management, and scheduling. The methodology is based on turning the FMM tree into a DAG (Direct Acyclic Graph), where vertices represent tasks and edges represent dependencies, and spawning tasks for computational work. Different versions based on Cilk, C++11, HPX-5 and OpenMP 3.0 tasks are compared using various input data sets. Experiments show that the Cilk version is the fastest, by a factor of two to three over the second fastest version, depending on the input sets. Cilk is an industry-level product, and has less overhead than the other versions. As a compiler-based runtime, Cilk profits from deep static analysis combined to a runtime scheduler that does not involve the OS.

In [31], Agullo et al. use the StarPU scheduler to alleviate the fact that OpenMP 3.0 did neither allow the expression of dependencies between tasks nor the possibility to specify priorities. The authors use a directed acyclic graph (DAG) to express the algorithm with a high level *task-flow* view. StarPu performs the tasks insertions and the computations in parallel while handling the dependencies based on data accesses. In section 3.2, we have presented the FMM algorithm and the two types of dependencies: strict order dependency between M2M operators and possible race conditions between L2P/P2P and L2L/M2L. The authors study the parallelism slackness by trying different combinations inverting the execution order. For instance, executing P2P before L2P enables more concurrency since P2P tasks carry no entry dependency. Similarly the execution of M2L tasks is prioritized over L2L, since their execution cost is much more important. Accordingly, the authors also developed a scheduling strategy which uses priorities. In chapter 8, we discuss a scheduling strategy of asynchronous tasks. It could be interesting to adapt a similar priority strategy to our case.

The work is extended to heterogeneous architectures in [32]. GPU versions of the P2P and M2L kernels are developed, as well a new scheduling strategy. Each task gets two priorities, the first one for CPU scheduling and the second one for GPU. P2P tasks are scheduled in priority on GPU, and M2L are processed on GPUs only if they are not totally charged with P2P. This ensures the maximization of the throughput. At the end of the algorithm, a heuristic is used to avoid imbalance at termination. Finally, in [30], the authors focus on optimizing and analyzing the scheduler's pipeline, for both homogeneous and heterogeneous cases. On homogeneous platforms, the filling of the pipeline is influenced by the size of the tasks, which is tuned by grouping the leaves. On heterogeneous platforms, the strategy depends on the balance between near-field and far-field. By default, P2P kernels are executed on GPUs, while M2L kernels are executed on CPUs. Nonetheless, if the near field becomes predominant, P2P can be scheduled on CPUs and inversely M2L can be executed on GPUs. Since heterogeneous hardware architectures are becoming the mainstream, this on-demand dynamic scheduling allowing for very efficient filling of the execution pipeline is interesting.

In [131], Yokota et al. introduce the concept of dual tree traversal, more used on treecode algorithms, applied to the FMM algorithm. In their ExaFMM code, M2L and P2P operators are computed without storing an explicit interaction list. Instead, the dual tree traversal traverses the octree and computes the interactions recursively. It uses a stack, holding pairs of cells. First, the stack is initialized with a pair of root cells. If the targets and sources are different, the octree can be separated into two octrees, and the root pair contains the source and target roots. Otherwise, the root is simply duplicated to form a pair. Then, the algorithm iterates on three

steps. A pair of cells is popped from the stack. The larger one is subdivided into its child cells. New pairs are created combining the smaller cell with each of the children of the larger cell. If a pair of leaf cells is created, the direct interaction is calculated. If the cells are well separated, the interaction is calculated using the FMM. And finally, if the cells are too near, they are pushed into the stack. The traversal's complexity is of order of $O(n)$. It can be used on adaptative FMM and unbalanced distributions. The author points out that the main drawback is the loss of the outermost parallelizable loop. However, this new algorithm is well-suited to be used with tasks.

The QUARK scheduler [127] creates an implicit DAG by inferring the dependencies between tasks from data access modes. An additional *locality* flag tells the scheduler if the data has to be kept in cache. In this case, tasks accessing the same data will be scheduled on the same thread. Nonetheless, work stealing can still downgrade performance. In [92] the authors use QUARK to schedule M2L and P2P tasks through the dual tree traversal of ExaFMM. The QUARK scheduler is identified to have important overhead, and to overpass this limit it is necessary to coarsen the tasks. Instead of assigning pairs of cells to a task, they use pairs of subtrees. However, experimental results show that overhead grows with the number of tasks. Nonetheless, it can be mitigated by wisely selecting the queue size, or augmenting the computational charge by increasing the order of expansion.

In [97] Pericas et al. use the OmpSs scheduler over OpenMP tasks to handle dependencies. OmpSs implements a task-based dataflow model which allows for additional clauses over the standard OpenMP 3.0 task constructs. Like in the later OpenMP 4.0 release, dependencies are expressed through data accesses. The *ss_clauses* specify the data ranges that are involved in input, output or both modes. Based on this information, the scheduler can build the dependency graph and schedule the tasks. The study focuses on the dual tree traversal of the ExaFMM code, which constructs the interaction lists. The M2L and P2P kernels can be either executed immediately, or stored into a list and executed after the traversal. Since dependencies between M2M and the dual tree traversal are difficult to express, only the downward phase of the tree algorithm is enhanced with dependencies. However, scalability results show improvements over the standard OpenMP task versions.

OpenMP 4 Tasks Using a high performance dedicated scheduler allows for taking profit from advanced features. However, all schedulers are different and choosing one implies to deeply modify the code and tie it to the scheduler. OpenMP is aimed to be portable and user-friendly. Moreover, new features carried by the 4.0 and later releases have improved its efficiency. The task support-based constructs have been extended and the *depend* clause now allows to finely express task dependencies without the use of another runtime system.

Agullo et al. make an extensive study on OpenMP 4.0 tasks and compare the efficiency to native StarPU tasks [28] [29]. Results are experimented on the ScalFMM library [19]. They compare three versions based on tasks: standard OpenMP 3.0 task-based version, OpenMP 4.0 with dependencies, and a version written with StarPU directives. OpenMP 4.5 offers the possibility to prioritize some tasks through the *priority* clause. Thus, in order to favor the progress along the critical path, the authors propose two versions with priority directives: the first one relying on OpenMP 4.5 and the second one on StarPU. Finally, considering the fact that the task insertion order heavily affects the effective execution order, they propose an extension to OpenMP with the *commute* clause. This clause allows the OpenMP scheduler to effectively swap two tasks which access the same data with

no ordering requirements. For instance, L2P and P2P may update the same data. They cannot be scheduled simultaneously but they can be executed in any order. Experiments demonstrate that the new OpenMP revisions allow significant performance improvement, now being able to compete with runtime APIs. They also demonstrate the advantages of OpenMP *depend* tasks over traditional fork-join parallelization, the benefit from being able to commute tasks and, in a minor extent, the possibility to prioritize tasks. These findings favor the use of OpenMP.

In [35], the authors propose to use OpenMP 4.0 *depend* tasks combined with tiling. The size of a tile being an adjustable parameter allows for tuning the task granularity. Particular care is given to the M2L operator which has sparse data accesses. The authors first partition the tree inside the horizontal levels. These partitions results in blocks grouping multiple M2L accesses. They are then used to apply a tiling strategy to both bulk-synchronous and data-driven algorithms. Another difficulty in developing a task-based FMM algorithm relies in the expression of the M2L dependencies. OpenMP 4.0 depend clauses do not handle linked lists, but only array ranges. The authors make the choice to express dependencies on array sections, even if it generates more constraints than necessary. They argue that fine grain dependencies would increase the scheduler's overhead.

However the OpenMP 4.0 and 4.5 releases still carry some weaknesses. In [36] and [28] the authors point out the implicit ordering on task execution originating from the OpenMP implementation. As demonstrated in [28], relaxing this constraint and allowing commuting tasks creates more parallel slackness. Moreover, the fact that the dependency clause does not support parallel traversals may create bottlenecks. Indeed, as presented in section 2.5.1, depend clauses are managed only for sibling tasks. Thus, if two threads traverse the tree to create the tasks, they won't be siblings, and the dependencies will not be managed. Atkinson et al. demonstrate that on manycore architectures, the cost of traversing the complete tree, evaluating the distance between nodes, spawning the corresponding tasks is too heavy for a single thread. They demonstrate on MiniFMM [11], an FMM mini-application, using 256 cores of the KNL architecture, that the remaining threads does not have sufficient work, which results in 22x slowdown.

All these optimization techniques allow for efficiently targeting node-level parallelization. But to take profit from large clusters, it is necessary to distribute the work among the nodes, which implies load balancing the work and optimizing the communications.

3.3.4 Load balancing

In the FMM algorithm, load-balancing consists in evenly distributing the particles among the processes, while minimizing and balancing the size and number of communications of the far-fields. In the literature [84], two main methods are identified for fast N-Body partitioning, and can be classified into Orthogonal Recursive Bisection (ORB) or Hashed Octrees (HOT).

Orthogonal Recursive bisection (ORB) The Orthogonal Recursive Bisection (ORB) divides the space into rectangles while load balancing the number of bodies. It recursively splits the domain in halves alternating the cutting dimension. It stops when the number of partitions equals the targeted number of processors. This method creates a well-balanced binary tree. Figure 3.6 shows how a two-dimensional space can be partitioned using ORB.

Applied to the FMM algorithm, the ORB technique has the weak point of not taking in account the underlying

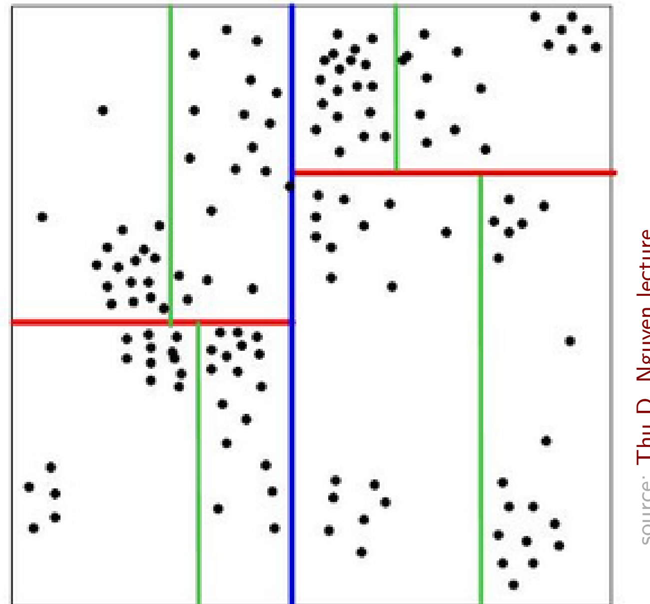


FIGURE 3.6: ORB partitioning of a two-dimensional space

octree. Space separators can be placed anywhere, not matching the octree grid and therefore generating important communications. In [84], the authors propose a composite approach. They use a modified ORB (Orthogonal Recursive Bisection) method, combined with a local Morton key (intriduced hereafter) so that the bisections align with the tree structure. Moreover, the local Morton keys are weighted considering computation and communication, using an ad-hoc formula, which takes into account the local and remote communications. Another limitation is that it requires a power of two numbers of processes. Nonetheless, ORB can be extended with multi-sections. Instead of cutting the space into two sub-spaces, it cuts it into multiple-spaces. This technique is referred to as Hierarchical ORB or multilevel ORB.

Space Filling Curves A space filling curve maps a two or three dimensional domain into a continuous line. The line is then cut into the targeted number of pieces and distributed among the processes. The most used algorithms are Morton and Hilbert [95] [108]. Both can be constructed by recursively dividing the unit square or cube. Figure 3.7 illustrates the first recursion steps of both Morton and Hilbert algorithms.

Spatial continuity Space Filling Curves have mathematical properties, among which the *Hölder condition* which characterizes the function's continuity. Indeed, the Morton curve does not meet the Hölder condition. In figure 3.7, one can see that the domain is divided in two by a long diagonal line. This can lead to spatial discontinuities and generate communication overhead, in particular for non-uniform distributions. In comparison, the Hilbert curve has

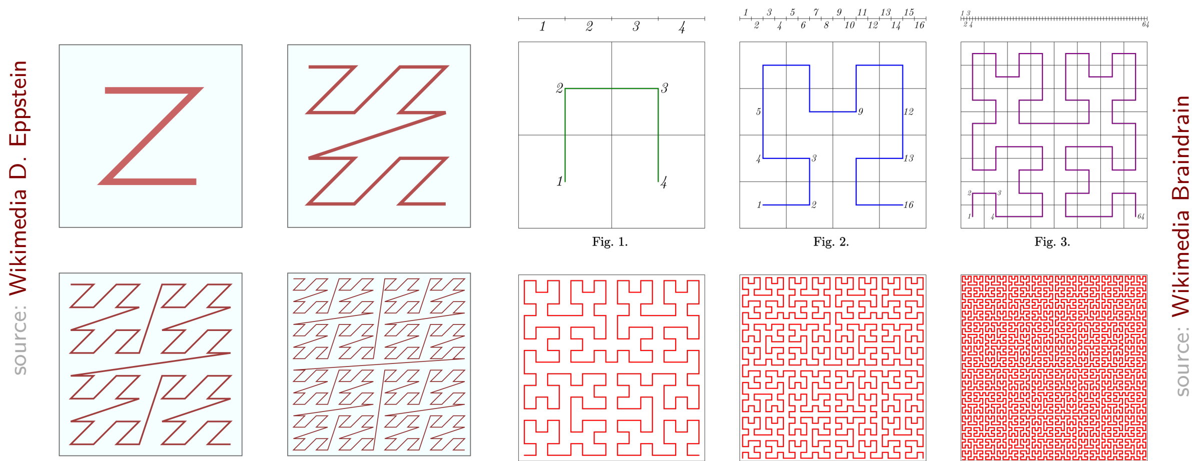


FIGURE 3.7: Left: Four iterations of Morton. Right: Six iterations of Hilbert

better locality characteristics. Sub-intervals are continuous, regardless to the distribution. But this advantage comes with the cost of more computation complexity and memory requirements [95] [108]. In [64], the author compares Morton and Hilbert decompositions on uniform, plummer, cylinder and sphere test cases. He demonstrates no performance gain with Hilbert, particularly on non-uniform distributions. Even if Morton curves have the drawback of possible discontinuities, they are simple to implement, and perform efficiently in practice. Therefore, they are largely used to load balance FMM distributions [96] [90] [55].

Weighted Morton curve A common approach consists in decorating the Morton curve with weights relative to the computational work before distributing it among the processes. In iterative and adaptative cases, weights can also be refined by taking as input the information collected from the last iteration. In [90], Lakshuk et al. load balance the computational work with a weighted Morton curve. Since the M2L operator dominates the algorithm, weights are based on the interaction lists, and assigned to each leaf. In a similar approach, in [96], Milthorpe et al. use a weighted Morton curve with a formula evaluating the computational work of the two main kernels P2P and M2L.

Hashed Oct-Trees (HOT) Hashed Oct-trees have been introduced in [125]. They combine a space filling curve ordering and a hashing function, A key is attributed to each cell. With Morton ordering, the key is computed by interleaving the coordinate bits. Therefore, the tree structure does not need to be stored with pointers anymore, and can be replaced with simple bit arithmetic to compute parent and child keys. A hashing table stores the location of each cell, and various other necessary information. Hashed octrees are very efficient. They offer $O(log(N))$ complexity for tree insertion and $(O(1))$ for access of any cell in the tree. Nonetheless, the tree height is limited to 10 with 32-bit integers and 21 with 64-bit integers.

Partitioning Tools An interesting approach based on graph partitioning tools is presented in [56]. The authors elaborate a strategy to load balance the work between the nodes while minimizing the total amount of communication. They first partition the octree by cutting it horizontally along a certain level. The leaves from the *root tree* then represent the *root* from each *subtree*. An undirected weighted graph is constructed: nodes represent subtrees weighted according to the computational work and edges represent communications weighted with the volume. The partitions are computed using PARMETIS [17], an open source partitioning tool which proposes multi-constraints algorithms. First results show interesting strong scaling results on small clusters, but have not been demonstrated on larger number of nodes.

In this section, elaborated methods to load balance an FMM over multiple processors have been developed. There is no general and optimal solution to this problem. Each application of the FMM algorithm and more specifically each use case, may benefit from different strategies.

3.3.5 Communications

The traditional FMM algorithm uses bulk synchronous communications, alternating computing and communication phases. Moreover, it is target-oriented: it is only when the interaction list is computed that the needed data is retrieved. This process creates bottlenecks in the communication layer and leaves the processors idle while waiting for the incoming contributions.

Optimizing Communications In [88] and [89] the authors introduce fine grain communication overlapping. They invert the communication scheme: Instead of the host processing all the neighboring cells, it is the neighboring cells which transmit their contributions to the host cells. This way, the communications are spreaded during the computation phase. Moreover, to further reduce bottlenecks, they organize their communications in a circular manner. Finally, they use local direct computations to overlap the communications.

In [26], the authors investigate communication reduction for adaptative trees. Observing that direct communication between targets and sources cause network contention, they introduce *HSPX* (Hierarchical Sparse Data Exchange Algorithm). The algorithm is based on relaying the communication through neighboring processes via pairwise exchanges. Communications are asynchronous and initiated by the sender. The algorithm shows good scalability for large adaptative and sparse cases.

PGAS-based FMM In [84], the authors use asynchronous sender-initiated communications with Charm++, and overlap the communications with the load-balancing computation phase. During the dual tree traversal, see 3.3.3, all interactions are computed. Since they are independent, they can be computed in parallel. The downward FMM phase is executed after completion of the dual tree traversal, with a post-order traversal, to cascade down the information from the tree to the particles. Charm++ provides *chares*, which are cooperating message-driven objects. When the application accesses a chare, the runtime sends a message to the invoked object and triggers the execution of the chare's code to handle the message in an asynchronous way. In ExaFMM, the local essential trees are mapped to chares with asynchronous entry functions. Once the needed information has been retrieved,

the downward computations can continue.

Another PGAS experience with FMM is presented in [96]. Milthorpe et al. use the X10 programming language which supports two levels of parallelism: a partitioned global address space for communications and a dynamic task scheduler with work stealing. The X10 programming language offers explicit data locality representation through *places* and asynchronous task parallelism via *activities*. Every object is hosted on a node but may be accessed globally by any processor. Applied to the FMM, each place contains a local essential tree, which contains only the necessary data for locally owned cells computations. Fetching data occurs through distant places via asynchronous activities.

Schedulers and Communications Implementations relying on schedulers usually target multicore architectures, eventually adding GPU support. Extension towards multinode support necessitates managing communications. In [33], different methods to achieve this objective are presented. A first option consists in encapsulating the communication inside a task and to let the scheduler handle the execution. The StarPU-MPI scheduler handles the communications by posting the send and receives to the underlying MPI layer. It then ensures interleaving the communications and the computations. Moreover, it is possible to dedicate one thread for message progression.
Some schedulers also offer the possibility to use the Sequential Task Flow (STF) model. Dependencies are extracted from a sequential valid order. Moreover, this model can also be extended to automatically infer the communications. Since task granularity is identified to generate overhead issue, the authors coarsen the tasks by grouping the nodes of the tree, thus creating a group tree structure. When moving to a distributed algorithm, the drawback is that communications may be larger than necessary. Indeed, the complete group may be sent, even if the recipient only needs one node from the group. The authors propose to extract and communicate only the necessary data. This strategy particularly pays off on non-uniform distributions, where the communication ratio is more important. The authors demonstrate, with the ScalFMM code and the StarPU scheduler, that compared to low-level hybrid MPI+task versions, high-level task-based schemes and the STF model are very competitive. On large number of nodes, and large problem sizes, it is profitable to dedicate one thread to tasks insertion and communication support.

Communication Complexity Communications' complexity has been studied in [83], [134], [90] and [115]. For Yokota et al., the communication complexity, of the ExaFMM application, is of order $O((n/p)^\alpha + \log p)$, n being the problem size, p being the number of processes and α equaling 2/3 in the 3D case. In comparison, Laskshuk et al. have a $O(n/p + \sqrt{p}(n/p)^\alpha)$ complexity, for the KIFMM code. Teng et al. characterizes complexity with $O(p(n/p)^{2/3}(\log n + \mu)^{1/3}))$.
The differences between these formulas are studied in [134]. Lakshuk et al. propose a hypercube reduce and scatter algorithm which changes $O(p)$ to $O(\sqrt{p})$. Teng's $((\log n + \mu)^{1/3})$ factor stands for non-uniform distributions but not for uniform ones. Finally, summing or multiplying the terms ensues from the data structure implementations, the global and local trees, and the possibility to avoid redundant communications. As a result, the communications complexity heavily depends on the application, the underlying kernel, the implementation choices and the data distribution. None of these studies can directly apply to Dassault Aviation's MLFMM for electromagnetic simulations. In the case of MLFMM, even if the number of communications is the same, the volume may be much more important. Indeed, the number of far field terms per node roughly doubles at each level of the octree.

3.3.6 Accelerators

Finally, FMM has also been ported to accelerators as GPUs and Intel Manycore architectures. In [133], the authors develop a GPU based algorithm scaling until 512 GPUs. Detailed explanations on turning the CPU kernels into GPU kernels are provided in [130]. In [53], Choi et al. develop a hybrid CPU-GPU FMM and elaborate a model to predict the execution time. Depending on the prediction, the execution is scheduled dynamically. Optimizations to take profit from the Intel Manycore architectures are presented in [25]. Loop-tiling and handwritten vectorization are used in conjunction with TBB threads. To take benefit from these architectures, an important code modernization is necessary. However, using accelerators allows for an important increase in computational power. Recently, Intel announced the new *Xe* GPU, which seems to integrate a long-term basis trend of heterogeneous computation.

3.4 Conclusion

The FMM algorithm is used in multiple domains. Its linear complexity makes it highly attractive for large-scale simulations. However, it is a complex algorithm in terms of computation work flow and communications.

Multiple parallelization strategies, tackling all parallelism layers are identified in the literature, but with no one-size-fits-all solution. Distributing the work among computational nodes is challenging: computations must be balanced while minimizing communications. The computational load depends on the level of the tree and on the underlying kernel. For instance, in electromagnetic simulations, it roughly doubles at each level of the tree, which is not consistent with the literature exposing higher computational intensity in the lower levels. Moreover, the neighboring scheme is complex. In comparison with a traditional halo exchange, the FMM algorithm carries out different neighboring lists with different interactions. Non uniform distributions even amplify these communications complexity. And finally, the fact that the bodies move or not during the computations also influences the strategy choice. Anyhow, the implementation of the load balancing strategy should be lightweight and efficient. However, if the application is balanced along the computation the load balancing strategy efficiency is crucial. In comparison, if the application is balanced once and for all, one could possibly afford an elaborated offline solution.
It is the case for Dassault Aviation's electromagnetism SPECTRE application, presented in chapter 4. In chapter 6, we propose two load-balancing strategies: the first one is an ameliorated ORB based on multisections, and the second one is a recursive Morton version. Both algorithms are parallel, can handle large amounts of bodies and redistribute them during the computation.

Efforts to optimize the communications have been done in different ways. Turning the algorithm from receiver-oriented to sender-oriented relieves the bandwidth stress. Using non-blocking asynchronous communications allow for overlapping. Relying on global address spaces has also been experimented. However the libraries used are high level user friendly ones: asynchronous communications are handled by the PGAS runtime, hiding the complexity to the user but at the cost of lower efficiency. To our knowledge, effective lightweight low-level PGAS programming has not been studied, and especially not on large industrial applications.
In chapter 7, we analyze the current communication scheme of SPECTRE, and propose an efficient asynchronous one-sided version using GASPI communications.

Moving from a fork-join based model towards a task-based model seems to be the general trend. Replacing large global barriers with fine grain synchronizations allows for more asynchronism, and eliminating idle times. However, using tasks necessitates to be careful about data locality and to tune the algorithm in order to favor data reuse. Moreover, special attention must be paid to task overhead, starting from the task granularity, the task creation and management, until the scheduling and eventual work stealing. Dependencies supported by the OpenMP 4.0 version are an interesting feature for the expression of fine grain dependencies. However, they are still limited to array sections and do not allow the expression of complex dependencies. In addition, recursive tasks do not support dependencies. And finally, dependency tracking could add overhead while still being limited to the task insertion order.
In chapter 8, we study the shared memory parallelism of SPECTRE, and elaborate two task-based strategies. The first one relies on OpenMP 4.0 tasks with dependency clauses, and the second one is a recursive version which handles the dependencies manually through scoreboards.

Schedulers seem to represent a mature option for FMM optimization on different parallelization levels. They target the efficiency of HPC application's. They manage tasks, dependencies, scheduling, load balancing, communications and even accelerators. But despite of being impressive demonstrators opening a lot of opportunities, to our best knowledge, none of these schedulers has been integrated with all their refinements in large production codes to solve industrial use-cases. Moreover, the choice of a scheduler largely impacts the code. Deep modifications are necessary, which tie the application to the scheduler and affects the portability.

In this chapter, we introduced the FMM algorithm and reviewed the main optimization techniques. In the next chapter, we present the applications and libraries which supported this thesis work before concentrating on our contributions in the second part.

Chapter 4

Dassault Aviation's and UVSQ Applications and Libraries

Contents

4.1 Introduction **53**
4.2 SPECTRE **54**
4.2.1 Use cases 55
4.3 AETHER **57**
4.3.1 Matrix assembly 57
4.3.2 Parallelization 57
4.3.3 Use cases 59
4.4 Mini-FEM and D&C-lib **59**
4.4.1 The concept of Proto-Application 59
4.4.2 DEFMESH and Mini-FEM 60
4.4.3 The D&C library 61
4.5 Conclusion **62**

We have introduced the current parallel architectures in chapter 1, the main distributed and shared memory parallel programming models in chapter 2, and the Fast Multipole Method in chapter 3. In this chapter, we present the applications and libraries that are used in this thesis. Dassault Aviation's simulation codes SPECTRE and AETHER are presented in sections 4.2 and 4.3, and the UVSQ's Mini-App proto-application and D&C-lib library are presented in section 4.4.

4.1 Introduction

As presented in the introduction of this thesis, Dassault Aviation's aircraft design process is fully numeric. Dassault Aviation develops its own simulation codes. A typical multi-discipline aircraft design involves aerodynamics, electromagnetics and infra-red stealth simulations. In this thesis, we have worked with SPECTRE, the electromagnetics simulation application and with AETHER, the aerodynamic code.

4.2 SPECTRE

SPECTRE is the in-house Dassault Aviation software for CEM (Computational Electro-Magnetics). Developed since the late 80s, it contains over 120 000 lines of Fortran 90 code, and includes a large number of methods and features. It is used routinely for antenna design and location of civil and military aircraft, and RCS (Radar Cross Section) computation and stealth design of military aircraft. It is also used to compute acoustic propagation. Different methods are available to solve these problems; the most widely used one is surface integral equations in the frequency domain for wave propagation phenomena, i.e. Maxwell equations or Acoustics. With a Galerkin discretization on triangular or quadrangular elements, these equations result in a linear system of equations with a dense matrix: this is the widely known Method of Moments (MoM). In SPECTRE, this linear system can be solved directly by an in-core/out-of-core highly efficient full matrix solver with an $O(N^3)$ complexity, where N denotes the number of unknowns. For problems involving large N, it is more efficient to use a Multi-Level Fast Multipole Method (MLFMM) [47]. In this thesis work, we focus on the optimization of SPECTRE's MLFMM.

MLFMM Compared to general N-body problems using the Fast Multipole Method (FMM), electromagnetic simulation introduces a major difference: at each octree level, the work to compute a multipole is doubling. Therefore the MLFMM's complexity is of order of O($NlogN$) instead of O(N) for the traditional FMM, i-e for N-body problems. Moreover, in the CEM community, FMM is often referred to as MLFMM, for Multi-Level FMM. In the sequel, we use the term MLFMM for CEM applications, and FMM for the general case.
To apply the MLFMM algorithm, the three-dimensional domain is recursively divided using octrees. Figure 4.1 illustrates this decomposition on a Rafale aircraft. The left side of the figure shows the hierarchical partitioning of the domain, and the right side illustrates the building of an octree from a partitioned domain. The obtained octree

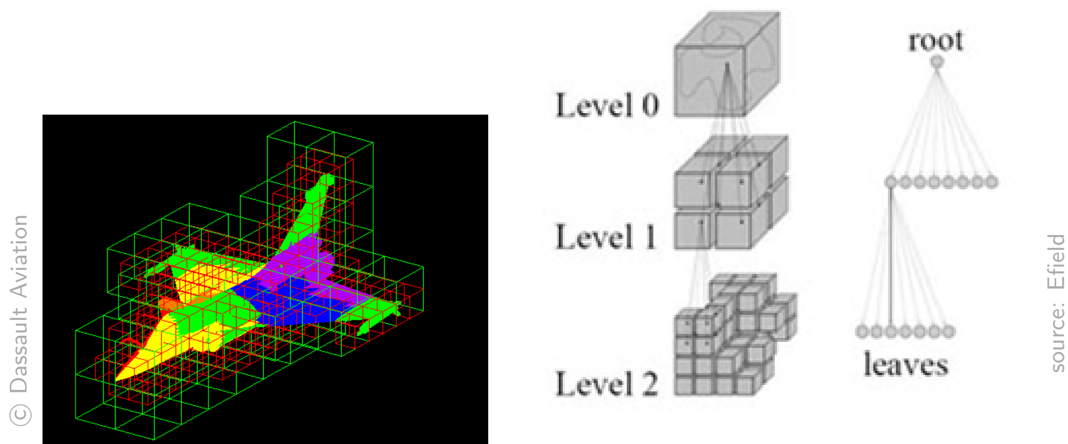


FIGURE 4.1: Left: Hierarchically 3D partitioning, Right: 3-dimensional octree.

is then used to apply the two-phase MLFMM traversal, as introduced in section 3.2.

Parallelization of the MLFMM SPECTRE benefits from hybrid MPI + OpenMP parallelization, which provides state-of-the-art performance compared to similar simulations [24].
The distribution of the octree follows an heuristic which consists in identifying a level of the octree which should contain enough nodes, dividing them into groups and distributing them among the processors. In chapter 6, focusing on the load balancing strategy, we present the current method in detail, analyze the obtained distribution, and identify possible imbalance which may impact the global execution time. Finally, we propose two alternative load balancing strategies, based on Histograms and Morton.

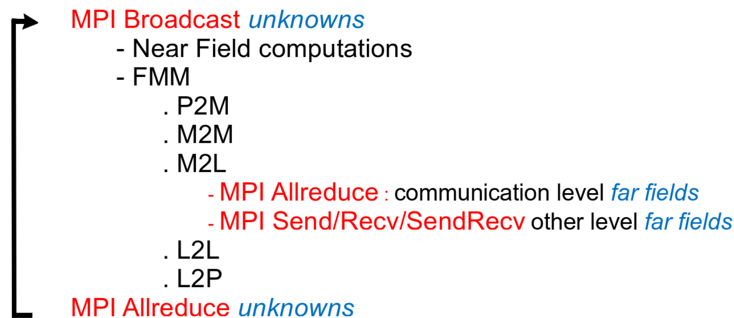


Figure 4.2: MLFMM communications

Communications are handled with MPI. As illustrated in figure 4.2, they consist in exchanging the unknown vector and the neighboring far field cells. The unknown vector is exchanged with global communications, whereas the far fields are communicated with two-sided blocking calls. In chapter 7, after presenting the complete communication scheme along with an in-depth profiling, we propose an alternative communication scheme based on asynchronous one-sided communications with GASPI.
Finally, shared memory parallelization is applied on each MLFMM kernel, through OpenMP loop parallelization. This method is limited by the implicit barriers at the end of each loop. Moreover, it introduces artificial barriers at the end of each octree level. In chapter 8, we present the current parallelization technique in detail and profile it. Then we elaborate two task-based strategies. The first one relies on OpenMP 4.0 tasks using the depend clause, and the second one is a recursive version in which the dependencies are handled through scoreboards.

4.2.1 Use cases

We use several use cases, detailed hereafter, to develop and measure our optimizations.

- *UAV*: The first use-case is a generic metallic UAV (unmanned aerial vehicle), with 95 524 nodes, 193 356 elements and 290 034 unknowns as shown in figure 4.3. This use case is used for RCS computation and radar imaging of a perfectly conducting UAV, equipped with two radar absorbing material (RAM) regions [23].

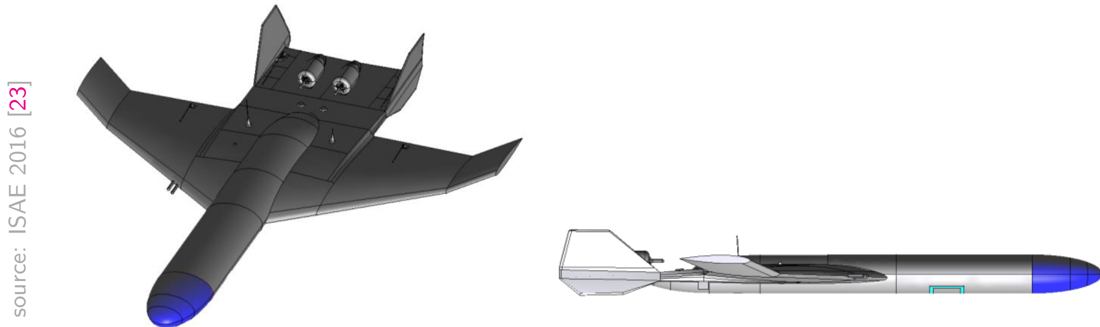


FIGURE 4.3: Generic metallic UAV use case

- *F7X*: The second use-case is extracted from an industrial-grade aircraft design problem: the optimal placement of a conformal phased array antenna on the Falcon F7X, as illustrated in figure 4.4. Several years ago, it used to be a "Grand Challenge" use-case, with runs of several days long on elapsed times on Atos-Bull platforms. It requires the resolution of hundreds of linear systems for the discretized model of roughly 765 000 nodes, 1,530,000 elements and 2,300,000 unknowns.

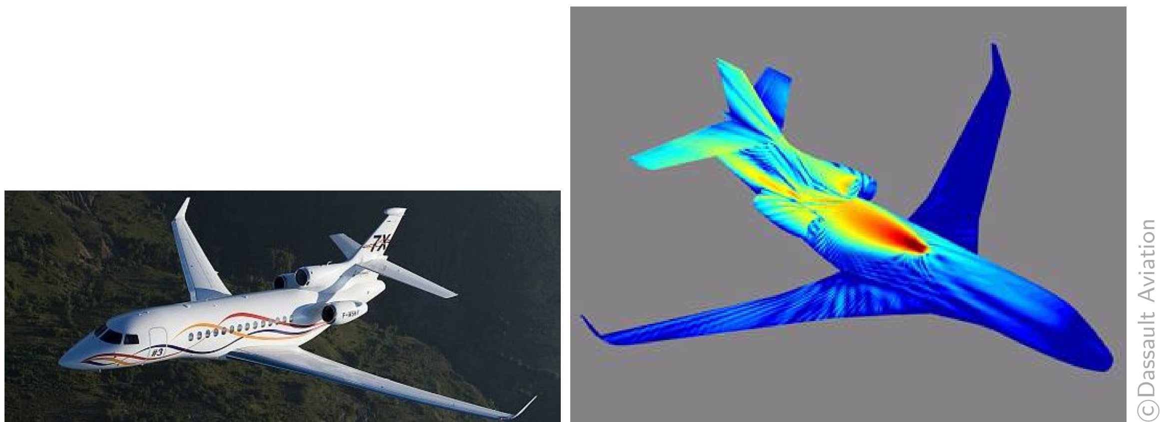
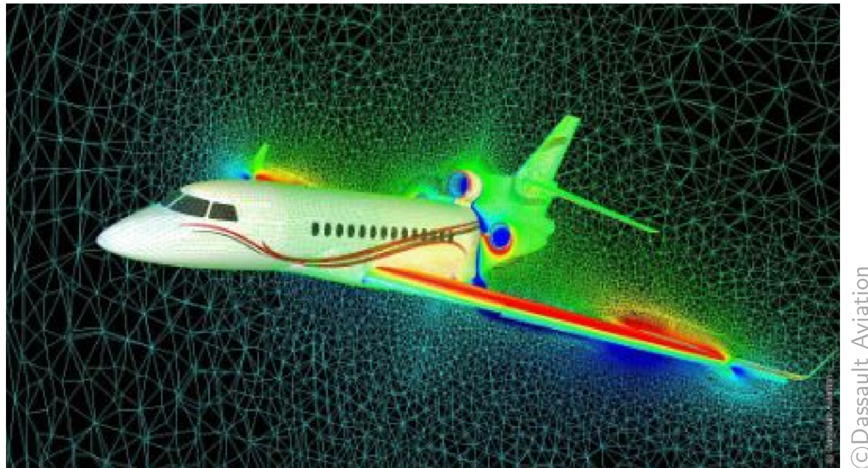


FIGURE 4.4: F7X aircraft use case

- *Sphere 80Ghz*: We also use a medium-sized heterogeneous sphere, with two dielectric parts: 32,300 nodes, 64,952 elements and 188,232 degrees of freedom.

4.3 AETHER

AETHER is an in-house Computational Fluid Dynamics (CFD) code developed at Dassault Aviation for more than 30 years. It solves the Navier-Stokes equations with a finite element method (FEM) on unstructured meshes.



©Dassault Aviation

Figure 4.5: Flow around a business jet aircraft with complex shock-boundary layer interaction, as captured through a section in the CFD unstructured mesh

The FEM is a numerical method which determines an approximated solution to a problem, using a discretizing representing a domain. The problem is generally expressed with Partial Differential Equations (PDE) on a compact domain with boundary limits. An iterative method is implemented, where each iteration is composed of three main steps: *assembling* the sparse matrix representing the equations system, *solving* the system, and *updating* the data with convergence testing.

4.3.1 Matrix assembly

Figure 4.6 illustrates the matrix assembly step for a two-dimensional quadrilateral FEM method. The mesh is presented on the left side, and the associated matrix on the right side. The matrix, representing the system of equations, is built from the mesh. A non-zero value represents an edge shared by two elements. This value is computed with a reduction on the contributions carried by the elements sharing the edge. Thus, the value of the edge $(N1, N2)$ equals $E1 + E2$. The resulting matrix is sparse. In 3D unstructured meshes, the number of elements sharing a same edge can be very large. And since, the reduction must be serialized to avoid race conditions, parallelizing the assembly step is a challenge.

4.3.2 Parallelization

MPI and domain decomposition AETHER is parallelized with MPI, with the use of domain decomposition. The mesh is subdivided into independent subdomains attributed to different processors. The partitioning is done

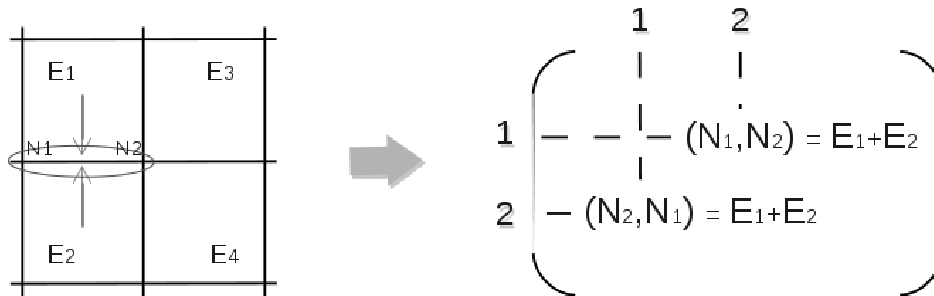


FIGURE 4.6: Illustrations of the assembly step: building the matrix.

on sets of elements. Therefore, subdomains share some nodes which have to be duplicated and exchanged through halo communications, as illustrated in figure 4.7 for two-dimensional triangular FEM. The scalability of this method

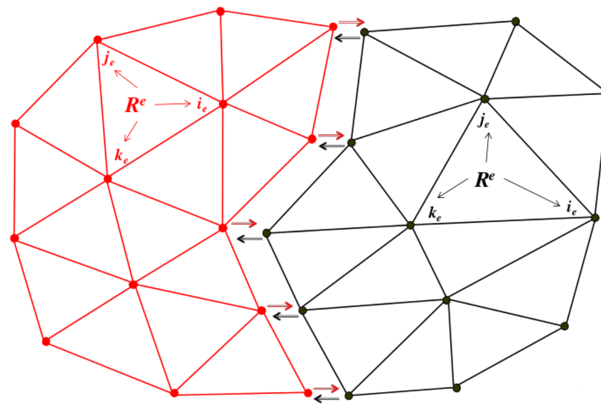


FIGURE 4.7: Halo exchange

is limited by the growing halo exchange. With large number of processes, the communication volume increases. Experiments at Dassault Aviation [117] have shown that with 512 domain partitions for irregular meshes, more than 40% of nodes are in the interfaces, therefore duplicated and communicated at each iteration.

Vectorization AETHER has been vectorized twenty years ago to exploit the vectorization technology of the Cray architecture. It uses the traditional coloring technique, which consist in identifying and grouping independent groups of elements. In other words, elements inside a group do not share any edge, and thus, any reduction variable. They can all be treated in parallel in a vectorial manner. Groups are handled sequentially, resulting in a safe reduction.

Hybrid parallelization: MPI + OpenMP As presented in section 1, current architectures expose multiple levels of parallelism. A largely used hybrid approach consists in using MPI between the computational nodes with OpenMP thread parallelism inside the nodes. The node-level parallelization of AETHER uses OpenMP threads. It is based on the already implemented coloring algorithm. Inside a color, the elements are independent and grouped in blocks. The assembly is completed by iteration over these blocks and parallelized with the *omp parallel do* pragma. This parallelization model is better aligned with current architectures. Nonetheless, it suffers from the coloring method drawbacks. In order to avoid race conditions, colors are serialized and treated sequentially. Colors are large and may cover an important part of the domain. They do not fit in cache and may generate multiple accesses to the main memory. The bandwidth is highly stressed and constitutes a bottleneck. The synchronizations and bandwidth stresses are multiplied by the number of colors. Moreover, coloring does not exploit spatial locality. Neighboring elements, thus sharing a reduction variable are attributed to different colors. Nonetheless, these problems are mitigated by AETHER's high arithmetic intensity at each reduction.

4.3.3 Use cases

Three use cases are provided to experiment our improvements.

- *EIB*, as illustrated in figure 4.8, simulates the displacements of a fuel tank along an airplane fuselage. It contains 1 079 758 vertices and 6 346 108 elements.
- *F7X* uses a 3D mesh around the Falcon 7X, with similar sizes as EIB.
- Finally, *FGN* simulates flows around a generic Falcon. It is a much larger test case with 7 173 650 vertices and 42 574 409 elements.

4.4 Mini-FEM and D&C-lib

4.4.1 The concept of Proto-Application

This section presents the proto-application concept [54, 94], 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. [40] 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

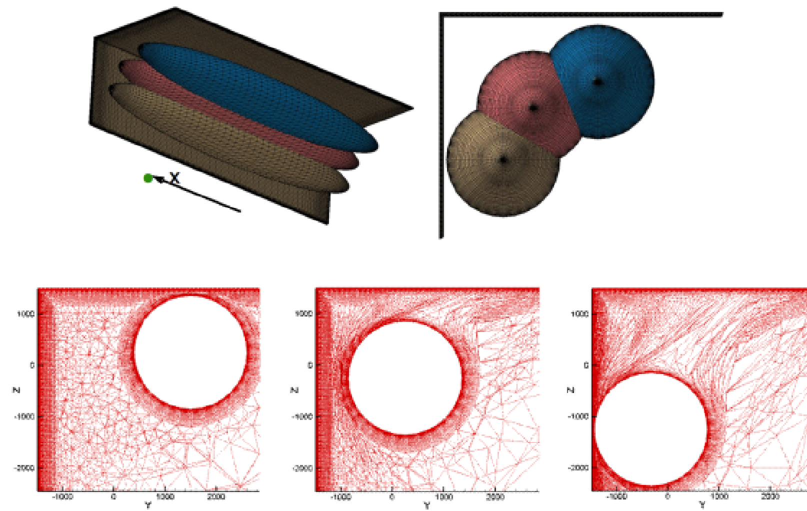


FIGURE 4.8: The EIB fuel tank position optimization use case.

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.

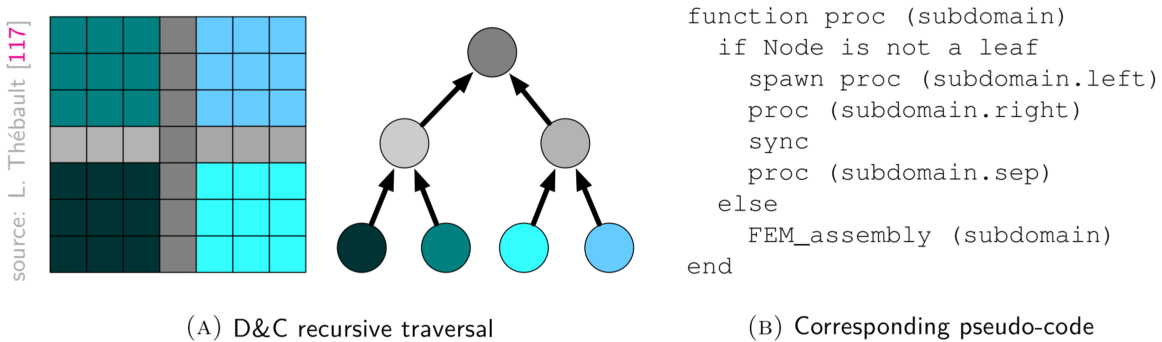
Direct valorization of the community improvements by providing a close to application code interface between the community and the application developers. They will be able to implement the relevant improvements to their codes.

4.4.2 DEFMESH and Mini-FEM

The proto-application concept is central in the european EXA2CT project [61]. The Mini-FEM proto-application [119], was developped in a collaboration between UVSQ and Dassault Aviation. It is a strip-down version of the DEFMESH application from Dassault Aviation. The latter is an unstructured mesh deformation code for CFD applications using Finite Element Method, FEM. The Mini-FEM proto-application consists of the first step of DEFMESH: the FEM assembly. It 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. In addition to the initial MPI domain decomposition version, the algorithm is parallelized with the D&C library. An open-source use-case from Dassault Aviation called EIB is shared with the proto-application.

4.4.3 The D&C library

The main idea of the D&C parallelization approach is to enable shared memory parallelism while preserving a good data locality and minimizing the synchronization costs. When the number of cores increases, the number of threads is increased instead of the number of MPI domains. This allows for replacing communications with data sharing.



(A) D&C recursive traversal (B) Corresponding pseudo-code

FIGURE 4.9: D&C recursive traversal and corresponding pseudo-code.
The left and right parts can be treated in parallel. The edges on the cut form a separator treated after these left and right parts.

As shown in figure 4.9, the rationale is to recursively divide the work in two or more independent tasks and synchronize these tasks locally. The Cilk++ implementation of the tree traversal is straightforward and very similar to the pseudo-code given in figure 4.9. 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.
This recursive approach brings many advantages. First, 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 to match the higher requirement of many-core systems. Using over decomposition leverages efficient work-stealing for load-balancing. Furthermore, no global barrier is needed. Synchronizations are local: only nodes of a same parent in the recursive tree have to be synchronized. Finally, D&C improves the data locality by reordering the

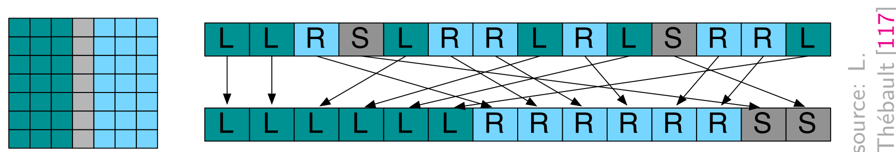


FIGURE 4.10: The elements in left (L), right (R) and separator (S) parts are reordered.

data as shown in Figure 4.10. During the computation of the recursive bisection, the local permutation of the elements of the left, right and separator subdomains are stored in a global array. Thus, during the execution,

the permutation is loaded and applied to the mesh so that all elements in a leaf of the D&C tree are consecutive is memory These locality and concurrency improvements benefit directly to the performance and the scalability.

The gain in locality is illustrated in figure 4.11. This histogram presents the L3 cache misses sum for all sockets for a run of the matrix assembly in different configurations. $x * y$ means x MPI ranks using y threads. As expected, the coloring version using MPI+OpenMP threads have a very poor data locality. On the other hand, all the D&C versions using MPI+Cilk have an average 4x improvement.

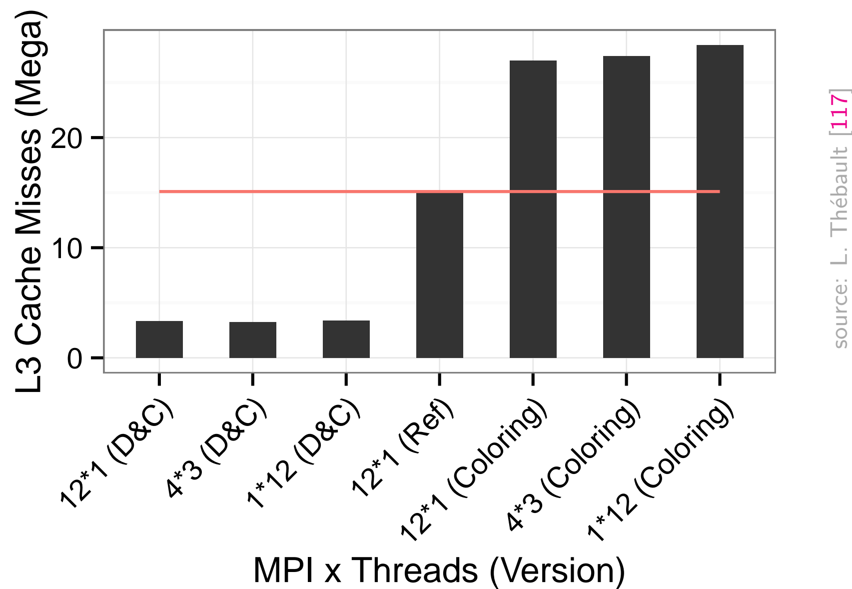


FIGURE 4.11: Last level cache (L3) cache misses measurement on a 12 cores two sockets Ivy Bridge node

The D&C approach has been further improves with vectorization and asynchronous one-sided communications. Further details can be found in [118] and [120].

4.5 Conclusion

Dassault Aviation develops several simulation applications, among which, AETHER and SPECTRE are major ones. Both aerodynamic and electromagnetism codes have been optimized during this thesis. Our first contribution, introduced in chapter 5, consists in porting the D&C approach to the AETHER code. We validate the methodology of using proto-applications for code modernization and compare it to the direct approach used for the work on SPECTRE. Our second, and main, contribution consists in improving the load balancing, the communications and the shared memory parallelization of SPECTRE. It is presented in chapters 6, 7 and 8.

Part II

Contributions

Chapter 5

Code Modernization through Mini-Applications

Contents

5.1 Introduction 65
5.2 Porting the D&C library from the proto-app to an industrial application 65
5.3 Using the proto-app to support the OpenMP implementation of the D&C library 68
5.4 Performance evaluation 69
5.5 Conclusion 71

5.1 Introduction

The first contribution of this thesis consists in evaluating the use of proto-applications as proxies for code modernization. First, in section 5.2, we port the D&C library, presented in section 4.4, to the AETHER industrial code, a very large CFD application from Dassault Aviation, introduced in section 4.3. We discuss key findings regarding efficient task-based parallelization of legacy Fortran codes in shared memory. Second, in section 5.3, we experiment the use of the Mini-FEM proto-application to extend the D&C library with an alternative version using OpenMP tasks instead of Cilk tasks. The proto-application allows quick prototyping, implementation, and debugging of the OpenMP version. Finally, in section 5.4, we focus on the performance portability. We experiment both D&C versions on the proto-application and on AETHER; then we compare the resulting performances.
The work presented in this section was published in [100].

5.2 Porting the D&C library from the proto-app to an industrial application

Motivation The D&C approach aims at improving the assembly step of FEM methods. The code modernization process followed the methodology scenario illustrated in figure 5.1:

1. The Mini-FEM proto-application is developed on the basis of the industrial DEFMESH application. It is a stripped down version of Dassault Aviation DEFMESH application, a mesh deformer. Compared with

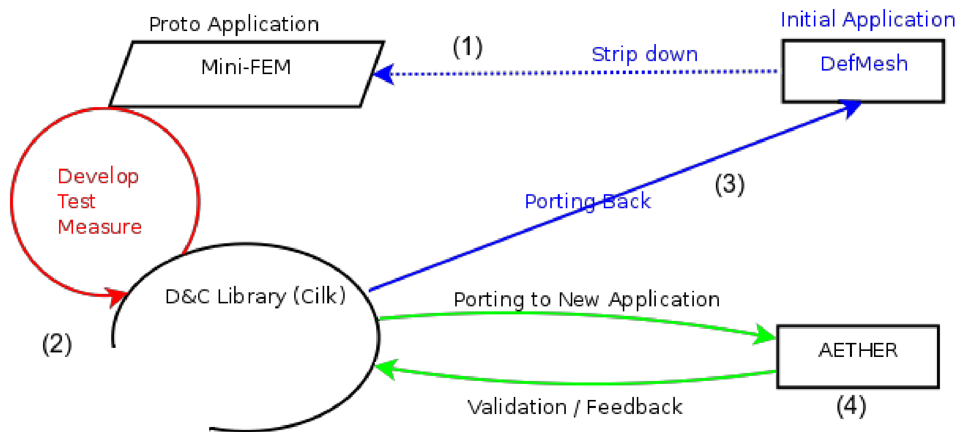


FIGURE 5.1: Optimization scenario: From the development of the proto-application to the integration in the final application

AETHER, the final target, DEFMESH is smaller, less sensitive and therefore shareable. It has been chosen to support the development of the proto-application since its matrix assembly step is representative of the one in AETHER.

2. The D&C library is developed with the support of the Mini-FEM proto-application.

3. The results and efficiency of the D&C library are validated on the original DEFMESH application.

4. Our contribution consists in porting the D&C library to the AETHER application, and to give a feedback on the use of a proto-application as a proxy for code modernization.

Methodology Integrating the D&C library into an application consists in two main steps: calling the *dc_partitioning* and *dc_assembly* functions. First, the *dc_partitioning* function handles the recursive domain decomposition, the node and element permutations, and the D&C task tree creation. Then, the *dc_assembly* function replaces the initial assembly loop by the recursive and parallel tree traversal which calls the initial assembly function on each leaf of the tree.

Integrating the D&C library into the AETHER code is straightforward. However, two main issues relative to the Fortran programming model have been encountered. We propose the adopted workarounds and discuss a future strategy to generalize them.

Recursivity The first issue pertains to parallel programming in Fortran language. Early Fortran was designed to allow the memory requirements of the entire program to be determined at compile time. All variables, including function frames, are static and shared. Thus, if they are instantiated twice using parallel calls, it will cause race

conditions. The *recursive* keyword provides a solution. It informs the compiler that the function may call itself and that the frame has to be dynamically handled on the stack. This solution was initially intended to allow recursion by solving the multiple function instantiation problem. When used in a parallel context, the use of the *recursive* keyword results in the storage of all the variables on the threads private stack.

Common variables The second issue is related to the use of *common* variables. In Fortran, the *common* block is a piece of shared memory widely used for passing information between program units. In AETHER, since dynamic allocation did not exist in Fortran 77, local *common* blocks are used to dynamically allocate memory with a limited scope. In shared memory, the problem is that the temporary *common* blocks are a unique static memory location shared by all the threads and causing many race conditions. Therefore, it is necessary to refactor this local memory allocation. A possible solution consists in using dynamic allocation with the Fortran 90 `allocate` function. Unfortunately, even if the race conditions are eliminated and the computation results are correct, this solution achieves low performance.

Experiment To figure out the bottleneck, we analyze the code scalability with CilkView [78] provided in the Intel Cilk Plus Software Development Kit. CilkView monitors a Cilk Plus binary application and reports parallel statistics and performance prediction on multi-processor systems.

Measure We run our experiment on AETHER with the F7X mesh and compare our results with those previously obtained with Mini-FEM on the EIB mesh. These two meshes, see section 4.3.3, are comparable in size and structure and therefore we expect similar characteristics.

```
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) F7X - AETHER

```
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) EIB - Mini-FEM

FIGURE 5.2: Comparison between CilkView scalability reports.

Figure 5.2 shows the reports obtained on the assembly code section.

Work corresponds to the number of instructions and is much higher in AETHER than in mini-FEM. This is logical given the higher computation ratio. The span indicates the number of instructions on the critical path. The burdened span includes the runtime overhead for scheduling, stealing and synchronization. Due to more complex computation, the runtime overhead is more than 25x smaller in AETHER compared with Mini-FEM. The number of spawns, synchronizations and strands, sequences of sequential instructions, are similar. This makes sense since the meshes have similar sizes and the tree topologies are analogous. The larger difference is mainly due to the

number of atomic operations. The ratio between these atomic operations and the amount of work is 10 times higher for AETHER than Mini-FEM. A rapid experiment putting some local variables on the stack by using local static declarations in conjunction with the *recursive* keyword, drastically reduces the number of atomic operations. We conclude that the high number of atomic operations is the result of the high number of dynamic allocations involved.
Based on this observation, all shared variables have been made private using the threads private stacks. However, increasing meshes imply increasing number of variables to store, which can lead to stack overflows. This problem can be temporarily solved by unsafely adjusting the thread's stack size limit. Nevertheless, estimating the size and setting it at compile time is not a viable solution for long term code maintainability.

Conclusion The global variables commonly used in Fortran have led us to the problem of parallel memory allocation for shared memory parallelization. In the list below, we summarize and discuss the different alternatives considering the lifespan of the variables and the memory efficiency criteria.

- *Global variables*: With thread parallelism, using global variables can lead to race conditions. Thus, they must be protected by costly critical sections. However, they may be used with no additional cost if they are exclusively accessed by read-only operations.
- *Heap*: Using the heap is a solution to privatize variables. However, since it is shared by all the threads, allocating involves costly lock operations.
- *Stack*: The application thread's private stack seems to be a better alternative. However, it is size-limited, and the scope cannot exceed the function boundary. Furthermore, the task-based parallelism model leads to a very large number of independent tasks. An allocation at task level would be repeated as many times as the number of tasks. Since tasks may include multiple functions to execute, allocating at function level would be even worse.
- *Thread Local Storage*: Contrary to the number of tasks, the number of threads is usually not growing with the problem complexity and is bounded by the architecture characteristics. Therefore, allocating at thread level drastically reduces the number of operations. The principle is that each thread owns a private space in memory, and uses it as a scratch pad to execute all its tasks. Future work will address the use of a memory allocator library to handle thread-level dynamic allocation in the Thread Local Storage (TLS).

5.3 Using the proto-app to support the OpenMP implementation of the D&C library

Since Cilk is a commercial library coming with the Intel compiler, we want to propose an alternative and portable version. OpenMP 3.0 introduces the task proposal to enable dynamic work unit generation [39], irregular parallelism, and recursivity.
We experiment the improvement scenario presented in figure 5.3.

5. We use the Mini-FEM application to develop the OpenMP version of the D&C library. Working on the proto-application allows for quick development, prototyping and debugging.

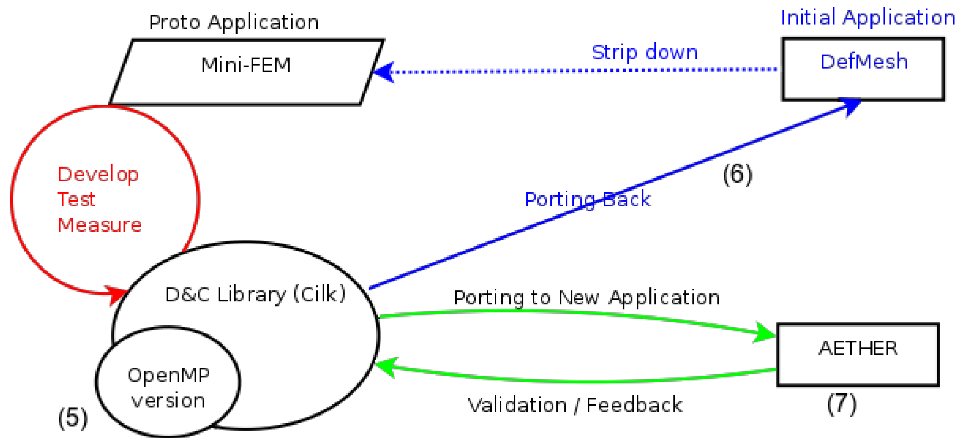


FIGURE 5.3: Optimization scenario: Extension of the D&C library with OpenMP tasks.

6. First confirmation measures are realized on the original DEFMESH application.
7. Finally, at Dassault Aviation, we simply update the already ported D&C library. This enables the AETHER application to take full profit from the new features of the D&C library.

5.4 Performance evaluation

Motivation We have demonstrated the convenience of using proto-applications for development. In this section we focus on performance reproducibility.

Experiment We evaluate D&C on both Mini-FEM and AETHER. We compare three versions of the matrix assembly step: `Ref`, the original pure MPI version, `D&C Cilk`, the hybrid version using the D&C library with Cilk Plus, and `D&C OpenMP`, the alternative version of the D&C library using OpenMP tasks. The results are presented both with relative speedup compared to the best sequential time and parallel efficiency. In the following figures, the x axis represents the number of cores. The y axis represents either the relative speed-up, or the parallel efficiency $E_P = \frac{T_S}{P*T_P}$ on P processors, with T_S being the sequential time, and T_P the time on P processors. The white background part corresponds to the physical cores and the grey one to the hyperthreads.

Measure

Evaluation on the Proto-Application On the Sandy Bridge nodes, we use 1 to 16 MPI processes for *Ref* and 1 MPI process per node with 1 to 16 threads per process for the *D&C* versions. Sandy Bridge experiments take place on the Anselm cluster and the application is compiled with Intel 14 and MPI 4.1. Figure 5.4 shows the comparison between the three versions of the code on the EIB use-case. The *D&C Cilk* version shows the best performance,

both in terms of speedup and efficiency. However, there is only 6% improvement over OpenMP on 16 cores. Since the *Ref* version uses domain decomposition, the domains are getting smaller, and the performance does not scale anymore above 12 cores. This makes a consequent difference with the D&C versions.

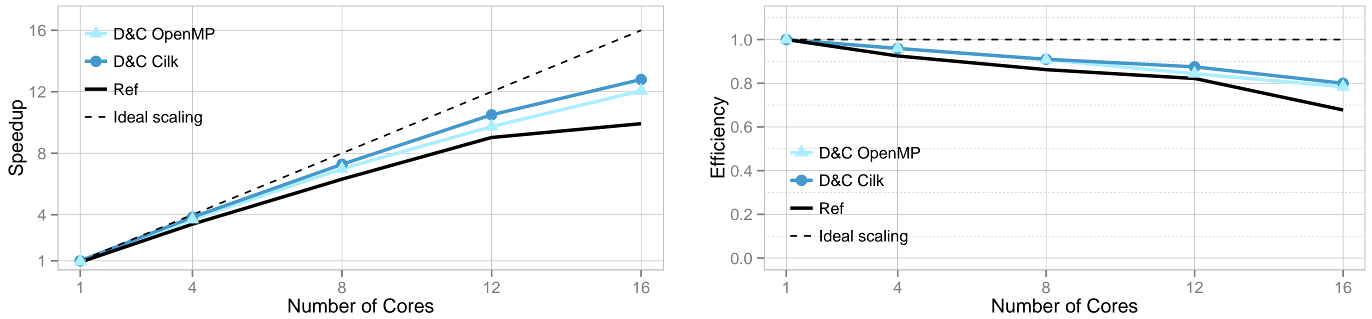
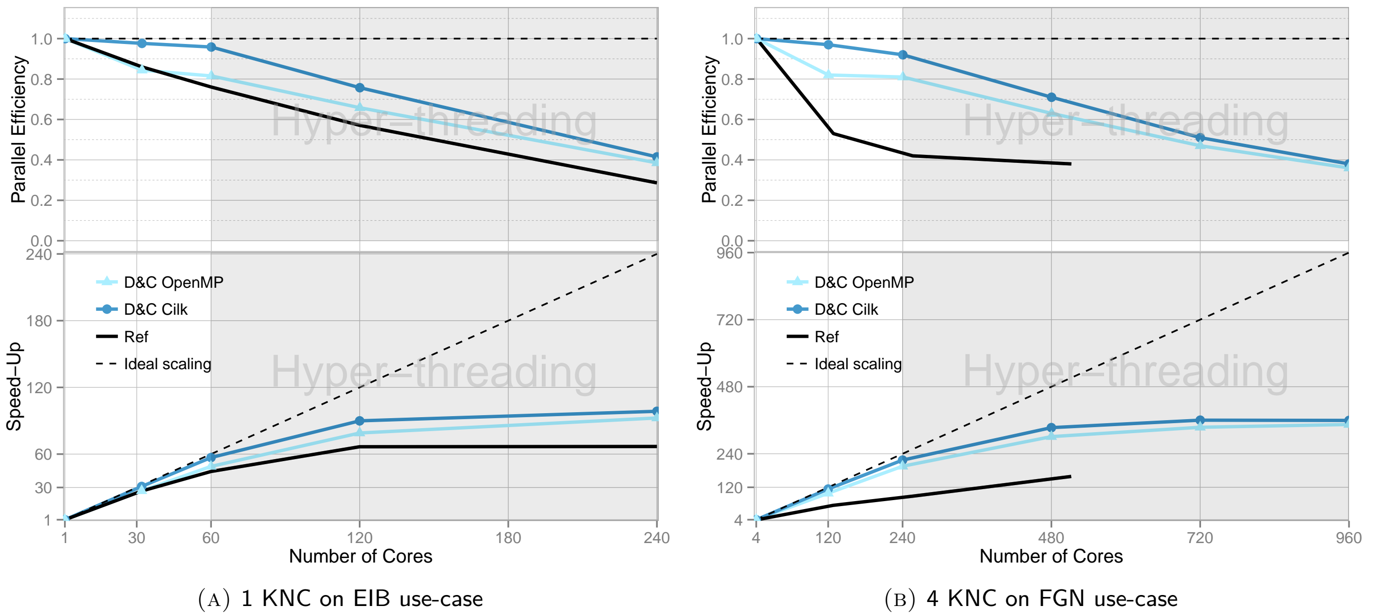


FIGURE 5.4: Proto-application: Intra-node scalabity on the EIB use-case



(A) 1 KNC on EIB use-case

(B) 4 KNC on FGN use-case

FIGURE 5.5: Proto-application: Strong scaling on Xeon Phi KNC

Figure 5.5a represents the results on one Intel Xeon Phi KNC. On the 60 physical cores, represented in the white background part of the graph, Cilk Plus scales almost ideally with 97% efficiency. The OpenMP version suffers from the runtime overhead and achieves 81% parallel efficiency. In figure 5.5b, we evaluate the strong scalability up to 960 threads on 4 Intel Xeon Phi nodes using the larger 7M nodes FGN use-case. On the physical cores, we obtain similar results. However, using hyperthreading, it is possible to mitigate the OpenMP runtime overhead.

Indeed, since the efficiency of the OpenMP version is less optimal, this leaves more improvement opportunities to the hyperthreads.

Evaluation on AETHER The experiments on AETHER are performed on the Sandy Bridge cluster from Dassault Aviation with the F7X use-case. AETHER is compiled with Intel 11, while the D&C library is compiled with Intel 13 due to the Intel Cilk Plus requirements. We use Intel MPI 4.1.0 and Intel 13 OpenMP. We fully use the 16 cores of a node and increase the number of nodes from 1 to 64. 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 1 million elements and is partitioned until all leaves contain 200 elements, we obtain 5000 tasks spread over 1024 cores. This results in less than 5 tasks per core. Figure 5.6 shows a speedup and an efficiency comparison between *Ref comm free*, a version with no communications, and the two *D&C Cilk* and *D&C OpenMP* versions.

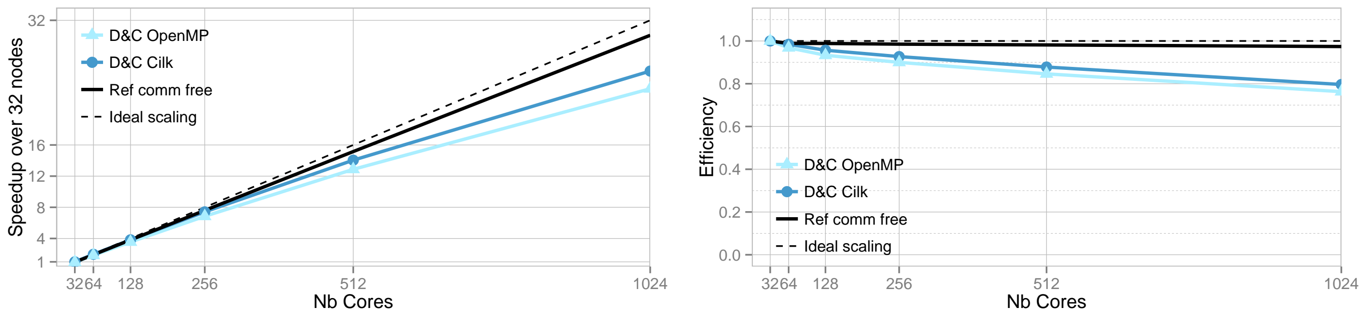


FIGURE 5.6: AETHER: Strong scaling with F7X mesh on 1024 Sandy Bridge cores.

At 1024 cores, `Cilk` speedup outperforms `OpenMP` by 9%. The Cilk Plus implementation of task parallelism benefits of lighter overhead and dynamic load balancing via work-stealing. As a result, Cilk Plus shows a better scalability than OpenMP, which is consistent with the results previously obtained on the proto-application.

5.5 Conclusion

In this work, we experiment the proto-application approach as a methodology for large HPC application modernization on a real use-case scenario. We proceed in two steps:
First, the D&C library development based on the Mini-FEM proto-application has been successfully integrated into AETHER, and the performance results are confirmed on the industrial test case. However, we encountered some difficulties due to differences between the original DEFMESH and the target AETHER applications. Indeed, AETHER is developed in Fortran 77, whereas DEFMESH is developed in Fortran 90/95. The Fortran 77 language did, initially, not allow thread level parallelism. Moreover, AETHER makes intensive use of common variables, which result to be shared between the threads causing race conditions. We propose workarounds for both issues and future work for privatizing the global variables with a TLS.

Second, we used the proto-application as a basis for the development of an OpenMP version of the D&C library and confirmed the results obtained with the porting of the OpenMP version of the D&C library to AETHER. OpenMP tasks are simple to program and provide comparable but sensibly lower performance than the Cilk implementation. As confirmed in the literature [106], this comes both from the lower overhead of the Cilk Plus runtime 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.

Chapter 6

Load Balancing

Contents

6.1 Introduction **73**
6.2 Current load balancing strategy **73**
6.2.1 Current algorithm 74
6.2.2 Efficiency characterization 75
6.2.3 Objective and Methodology 76
6.3 Histograms **77**
6.4 Morton **79**
6.5 Conclusion **85**

6.1 Introduction

In the previous chapter, we discussed the methodology to modernize large industrial simulation codes. In this context, we optimized the assembly step of the CFD application AETHER. In the three following chapters, we focus on the modernization of the MLFMM of the CEM application SPECTRE.
Due to high a confidential level , it is not possible to develop and share a representative proto-application. Therefore, all optimizations are performed directly on the final application. However, in order to facilitate the reuse and whenever possible, we develop our optimization strategies through open source libraries.
In-depth profiling of SPECTRE has demonstrated three separate aspects which are candidates for optimization: load-balancing, communications and shared memory parallelism. This chapter focuses on load-balancing. First, we analyze the current load-balancing strategy, define our objectives, and present our methodology in section 6.2. Then, we present two alternative strategies based on Morton and Histograms, respectively in sections 6.4 and 6.3. This work was published in [100], and the load balancing software is available on GitHub [4] under LGPL-3 license.

6.2 Current load balancing strategy

Depending on the physics of the N-body problem, bodies may move or not. In the case of SPECTRE, the FMM algorithm is used to calculate CEM for solid structures: bodies represent Gauss points are fixed. Therefore, the

initial load balancing is highly important. It impacts the complete execution of the application, from the first iteration to the last one. The load balancing strategy defines the elements distribution among the processes, and influences both computational and communication phases. Indeed, the number of nodes and elements triggers the computational charge. A good strategy balances the work among the nodes, thus reducing the idle times. Moreover, distributing the domain implies cutting it which generates exchanges between neighboring nodes. The less neighboring is generated, the less communication volume is created.

6.2.1 Current algorithm

In the initial version of SPECTRE, the load balancing step takes place after the construction of the octree. It consists in evenly distributing the octree leaves among the processors, taking into account the number of points contained in each leaf. The algorithm, illustrated in figure 6.1 proceeds as follows:

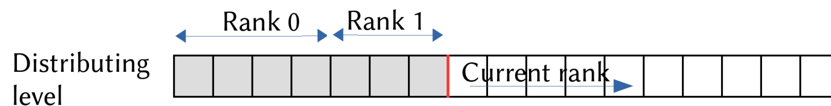


FIGURE 6.1: Original load balancing algorithm

- First, a *distribution level* is defined, which is the octree level that will be traversed to make the distribution. This level is identified by taking into account the height of the tree and the number of MPI domains that are used to execute the run. Experiments, with representative production use cases, have been run to determine the optimal distribution level. For most cases, it is the level 7, which is used. However, if the tree is not high enough, the distribution level will be the leaf level. Similarly, if the number of MPI domains grows, the distribution level is lowered. The rationale is to make a trade-off between the computational charge and the precision of the load balancing strategy.
- Second, the distribution level is traversed in a breadth-first way. Assuming that the octree nodes in gray have already been attributed to the previous ranks, the average number of points to be dispatched to the remaining number of ranks is computed. Then, consecutive nodes are assigned to the current rank, until reaching the target average number of points, exceeding it, or reaching the end of the level.

Although this method works well in practice on current Dassault Aviation use cases, some weaknesses are noticeable. First, the computational costs are reduced at the price of the load balancing precision. Since a distribution level is determined, the algorithm does not benefit from finer information available in deeper levels of the octree. Second, this method is not scalable. It has been validated on a certain class of problems, within a certain number of processors. If the use case or the number of computing nodes grows considerably, the empirically determined optimal distribution level may not be valid anymore.

6.2.2 Efficiency characterization

To analyze the quality of the resulting partitions, we ran some experiments on the UAV test-case. Figure 6.2 displays side by side, the partitions obtained with four processes and the communication matrix generated for 64 processes. The left side shows the partitions for four processes. This representation was obtained using a visualization tool presented in section 6.2.3. Black points represent elements hosted by the rank, and red point represent neighboring elements which need to be exchanged. One can see that the partitioning is not optimal: the top-left processor holds a non-contiguous set of elements, and this disparity increases the number of neighboring particles to exchange.
The right side of the figure displays the communication matrix obtained with 64 domains. We use ScoreP [20] to instrument the code and record the amount of data exchanged, and Cube [2] to explore the results. Then we build a matrix representing the quantity of data exchanged between pairs of processes, with a color scale from blue to red. A gradient from blue to red towards the diagonal translates a good communication locality. On this test case, one can see that the communications are more important around the diagonal. However, in the bottom and on the right part of the matrix, one can see horizontal and vertical lines almost entirely colored in yellow to orange: this means that some processors communicate with almost all the others.

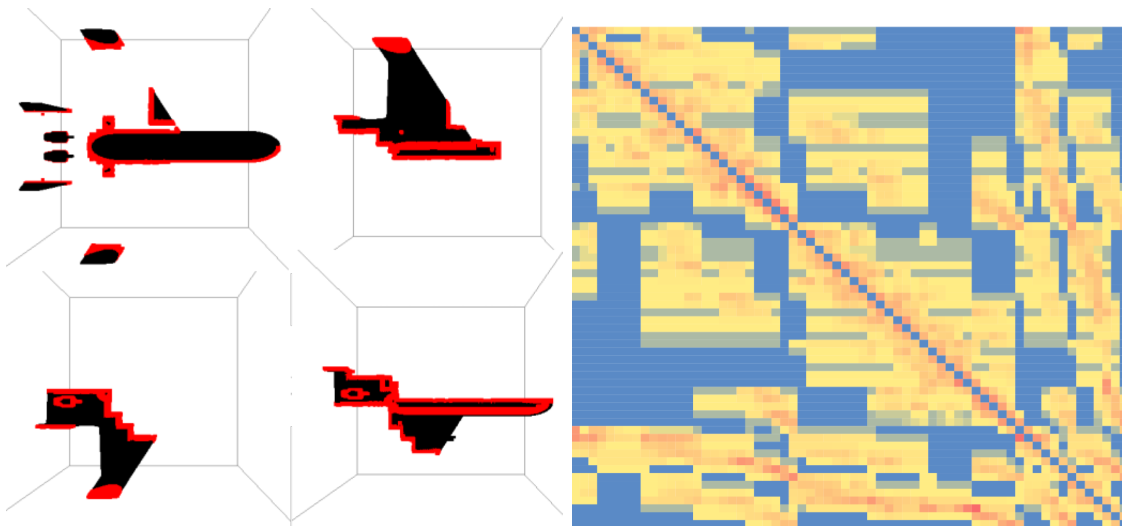


FIGURE 6.2: Original load balancing algorithm applied to the UAV test-case on 4 and 64 processors. On the Left side: visualization of the points hosted by each processor. Neighboring points are represented in red, they require communications. On the right side: the communication matrix generated for 64 processors.

To further investigate the load balancing, we focus on the obtained octrees. We run the experiment with eight processes and still use the UAV test case. For each domain, the Table 6.1 displays the number of nodes per level of the octree. The first levels of the octree are duplicated and are common to all domains since all the processes share the top of the tree. The lower levels represent the distributed subtrees handled by the different domains.

One can notice important load imbalance, emphasized with red color, at leaf and internal node levels.

This load imbalance directly impacts the computations. The near field operates at leaf-level, with a complexity depending on the number of points. Since the leaves roughly contain the same number points, their number should be balanced. However, one can notice an important disparity, in the order of 1.27x, between rank 0 and rank 6. Inside the tree, the complexity of the FMM operator depends on the number of internal nodes, and this gap increases up to a factor of 2.5x.

	Rank 0	Rank 1	Rank 2	Rank 3	Rank 4	Rank 5	Rank 6	Rank 7
Level 1	1	1	1	1	1	1	1	1
Level 2	8	8	8	8	8	8	8	8
Level 3	20	20	20	20	20	20	20	20
Level 4	42	42	42	42	42	42	42	42
Level 5	46	57	59	65	53	61	77	75
Level 6	92	122	141	147	132	126	230	223
Level 7	281	372	486	452	413	381	739	653
Level 8	889	1112	1224	1210	1133	1072	1569	1403
Level 9	3167	3463	3889	3610	3603	3650	4051	3802

TABLE 6.1: Load balancing of the UAV test-case among 8 ranks: number of octree nodes per rank and per level

6.2.3 Objective and Methodology

We have identified some weaknesses in the current load balancing scheme.
Our objective is to propose a strategy that load balances the work, at each level of the octree. In order to favor the reuse of our load balancing strategies we develop them in a distinct library, the FMM-lib library. We focus on developing lightweight parallelized strategies to limit the overhead. Our load balancing strategies can handle large inputs and exchange the bodies during the computation. We propose versions relying on MPI or GASPI communications.

To observe the results obtained with the different strategies, we have developed a visualization tool based on OpenGL (Open Graphics Library) [15], called FMM-viz, shown in figure 6.3. The FMM-viz tool handles octrees, and the corresponding sets of points. It is possible to work with randomly generated lists of points, or to import data from a real application.

We use the FMM-viz tool to support quick development and debugging of the load balancing library, outside of the initial application. Once the list of points is imported, we apply the load balancing strategy and update the scenes with the resulting distributions. Each process displays its own scene with hosted nodes and points. It is possible to switch between the results of the different load balancing strategies. The 3D scenes can be zoomed and rotated around the three axes. This method allows quick visualization and debugging.

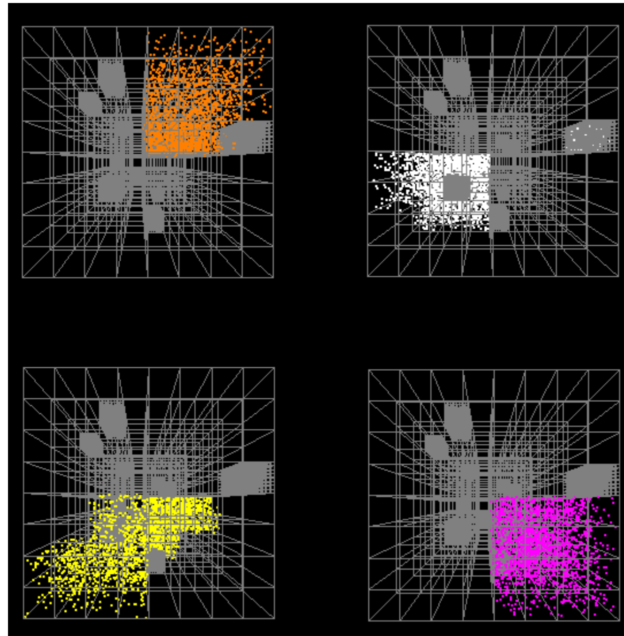


FIGURE 6.3: FMM-viz, our OpenGL-based visualization tool screenshot

In section 3.3.4, we have identified two main methods largely used to load balance FMM algorithms: ORB and HOT. In the following sections we discuss their advantages and limits, and propose our alternative strategies.

6.3 Histograms

Motivation The ORB load balancing method is known to produce well-balanced kd-trees. However, they have two main limits which prevent them to be used in the industrial SPECTRE application. First, they are limited to power of two numbers of processes, and second they do not align on the underlying octree. In the FMM algorithm, the cut-off distance test computation is based on the geometry, therefore it is important that the frontiers match the octree grid. To mitigate these drawbacks, we propose a histogram-like algorithm. It is similar to ORB but it uses multi-sections to overpass the power of two numbers of domains limit. In addition, the octree structure is taken into account to compute matching separators.

Methodology We develop two versions, aimed to be scalable on large number of processes. The first one computes the multi-sections exactly, whereas the second one computes an approximation where the separator is rounded to match the octree grid.

- *Multi-sections* We decompose the target number of domains in prime numbers and compute the multi-sections while alternating the cutting axis. Let's assume we target a cluster composed of 18 compute nodes. The

decomposition in prime numbers results in $3 \times 3 \times 2 = 18$. Therefore, the domain will be cut three times. The first cut is realized along the x axis and cuts the domain in three subdomains. The second cut takes place along the y axis and cuts each subdomain in three, which results in 9 subdomains. Finally, the last cut, along the z axis, cuts each subdomain in two, resulting in 18 subdomains.

- *Parallelization* Our implementation is fully distributed. It can handle large inputs, which do not fit in a single node, by managing the particle exchanges to produce the final sorted set. The number of processes that participate in the computation, grows after each cut. Let's go back to our previous example:
  - At the beginning, one single compute node is in charge of computing the first cut. To that end, it needs a complete knowledge of all the particle coordinates in the designated axis. Therefore, all processes compute a histogram and send it to the root with a reduction communication operation. Then, the root node computes the separator and broadcasts it to all the other nodes.
  - For the next cut, three processes take part in the computation: each one handles one subdomain. The same procedure is repeated: reception of the histograms, computing of the separators and broadcasting the results.
  - Finally, for the third cut, nine processes compute the last separators.
- *Computation Optimization* This algorithm requires lots of histogram computations and collective communications. To reduce the computational load we optimize the histogram computation with bitwise operations and approximation on the octree grid. Histograms on the coordinates are computed progressively. The 64-bits coordinates are split into 5 chunks containing respectively 12 (sign + exponent), 16, 16, 16 and 4 bits. The complete separator is computed chunk by chunk until identifying a matching separator. As illustrated in figure 6.4, only the critical bins are refined:

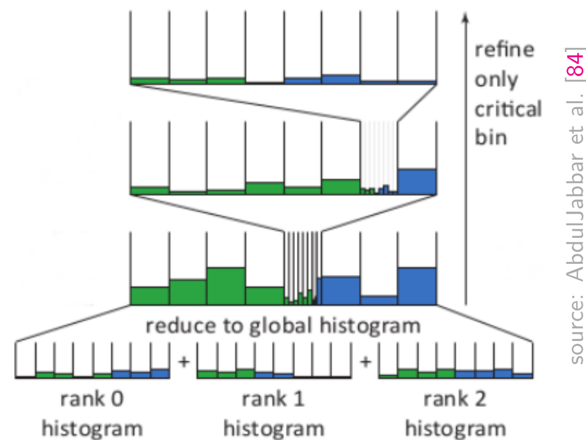


FIGURE 6.4: Optimization of the computation of the separators.

– All processors compute the histograms based on the bits contained in the first chunk, and send it to the root node.
– Based on the complete knowledge of all first chunks histograms, the root node identifies the 12 first bits of the separator and broadcasts them.
– All the processors filter their points based on the first identified chunk of 12 bits. Then, the process is repeated with the remaining points and the following chunks.
– In the approximate case, the computation is stopped when the bin is small enough to match one only octree separator. In the exact case, the computation continues until identifying the complete 64 bits.

Measure Figure 6.5 displays the resulting partitions for 4 and 8 domains. As expected, we obtain regular distributions. For four domains the UAV is cut into four well-balanced subdomains. However, when increasing the number of domains, the results degrade rapidly. With eight domains, the third cut slices the plane along its longest dimension and creates an important neighboring exchange, represented in red.
To gain further understanding, we analyze the generated communications in figure 6.6. The graph on the left side represents the total amount of data exchange, depending on the number of domains. The original version is illustrated with blue bars and the histogram version with red bars. One can see that, starting from eight domains, the communication volume grows considerably. The communication matrix in the right part of the figure confirms that the locality has been worsened. We also measure the corresponding execution time, which results in a 1.7x slowdown in comparison with the original version.

Conclusion This experiment confirms the fact that load balancing strategy has an important impact on the FMM algorithm. Most importantly, it highlights the importance of reducing the communications, while balancing the computations.

6.4 Morton

Motivation Hashed Octrees methods divide the domain using space filling curves. Efficient implementations rely on bitwise computations on the particle coordinates [125]. Therefore they are limited by the coordinate datatype size. The two mainly used space-filling curves are Morton and Hilbert, presented in section 3.3.4. Hilbert curves are known to provide better data locality than Morton but at a significant computation cost. Morton curves are simpler to implement and, in practice, provide good results [95] [26]. We propose a load balancing strategy based on Morton space filling curves and implemented through a depth-first traversal, without limitation by any octree depth.

However, the currently implemented scheme in SPECTRE is very close to a Morton ordering and thus there is little room left for improvement with Morton load balancing. Indeed, the currently implemented scheme results to be very close to a Morton ordering. Once the octree is created, the list of nodes is numbered and stored level by level in an array. As the numbering process boils down to defining a pattern to traverse the nodes, it is very natural to use a Z-shaped pattern. As a matter of fact, in the case of SPECTRE, the chosen order picks the two upper-front cells from left to right, then the two bottom-front cells and repeats the process on the four back cells. As a result,

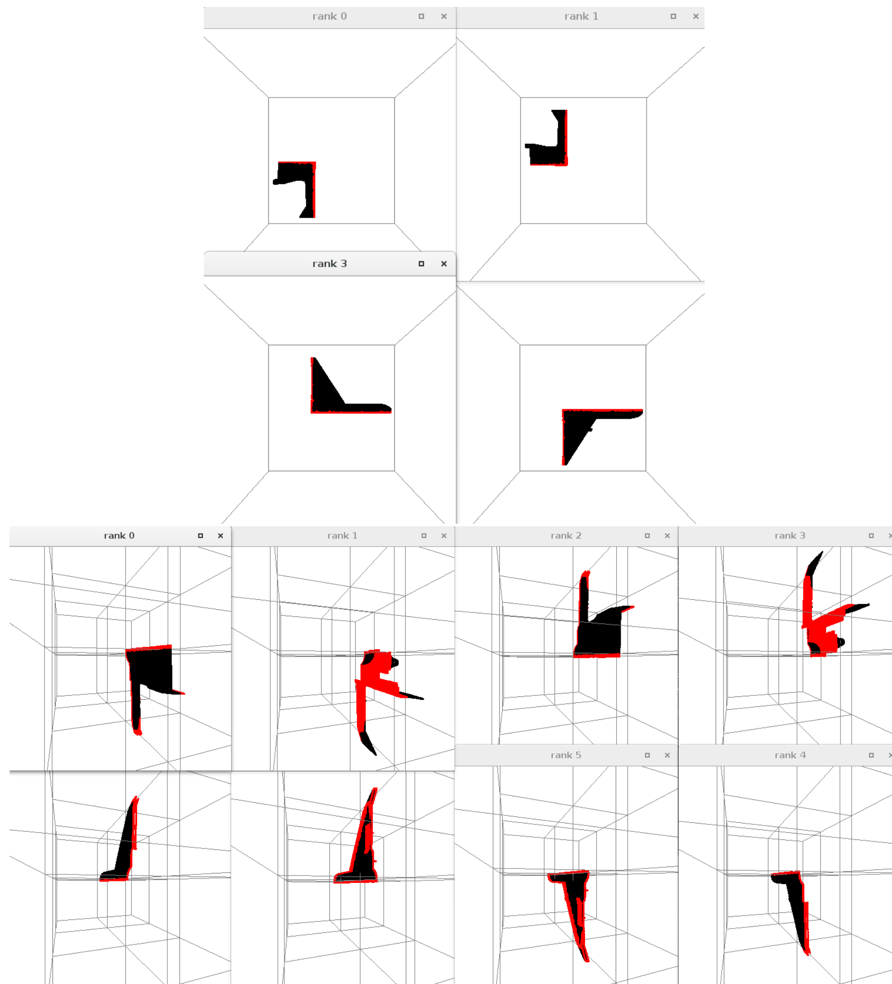


FIGURE 6.5: OpenGL visualization of histogram load balancing on UAV test-case
Top: 4 domains, Bottom: 8 domains

the octree nodes are stored level by level, in a Morton ordering. As illustrated in 2D in figure 6.7, traversing the array in a breadth-first order, is equivalent to applying a Morton ordering.
However, rotating the Z-shaped pattern in a three-dimensional space results in six different possibilities. Moreover, our previous experiments have shown that a misplaced cut can largely impact the communication matrix. Therefore, we are interested in exploring the different possibilities. Furthermore, the current implementation precision is limited to the choice of the distribution level, as explained in 6.2. A more precise strategy could improve the load balancing.

Methodology We describe hereafter the main characteristics of our Morton load balancing strategy:

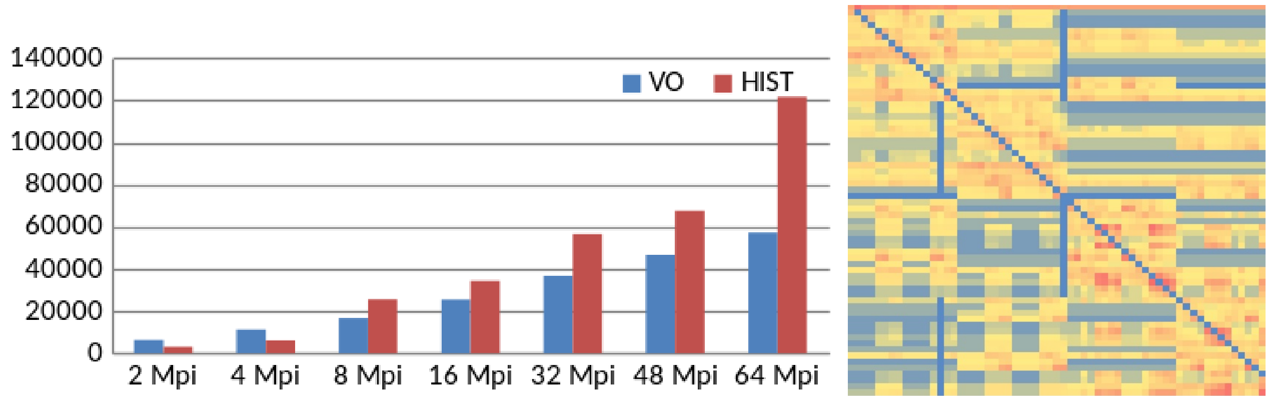


FIGURE 6.6: Side-effect of the Histogram load balancing algorithm: Communication degradation. Left: Communication volume in bytes for 2 to 64 domains, Right: Communication matrix with 64 domains.

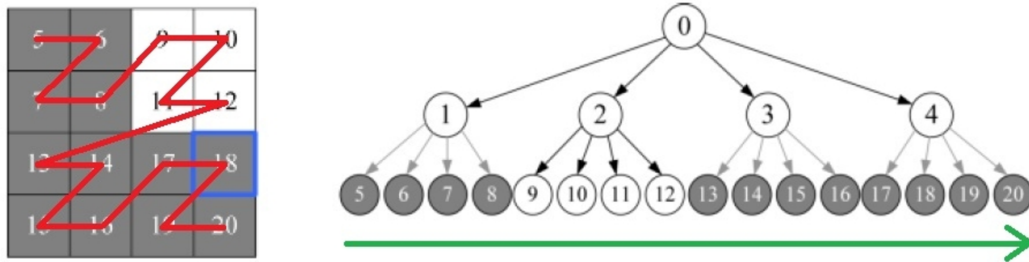


FIGURE 6.7: Morton ordering on a 2D dimensional space represented with a quadtree

- *Depth-first traversal:* A Morton curve can be implemented with a depth-first traversal A first Morton recursion would cut the domain into four subdomains and the Morton Z-shaped pattern would traverse them in order 1-2-3-4, from the upper left cell to the lower right cell. The second division cuts each quadrant recursively into four sub-quadrants, and the resulting ordering traverses the nodes in increasing order from cell 5 to cell 20, as illustrated on the left side of figure 6.7 Referring to the right side of figure 6.7, oen can notice that the first level nodes are accessed in the same order with a depth-first traversal. The same analysis applies in the second level of the quadtree.

- *Tolerance criterion* To avoid over-decomposition, which might bring very low improvements compared to the overhead while computing the FMM iterations, we introduce a tolerance criterion. This adjustable parameter allows for defining a precision range for the expected load balancing result. For example, the user can specify that the particles distribution can allow 5% distance from a perfect balance. While the depth-first traversal algorithm progresses down the octree, nodes become smaller, and the load balancing precision improves. The algorithm stops as soon as a separator meeting the precision threshold is identified.

The algorithm proceeds as follows:

- *Recursive traversal cutting off unnecessary branches* The algorithm arbitrarily starts on level three, and recursively refines only the nodes containing the separator, as illustrated in figure 6.8.

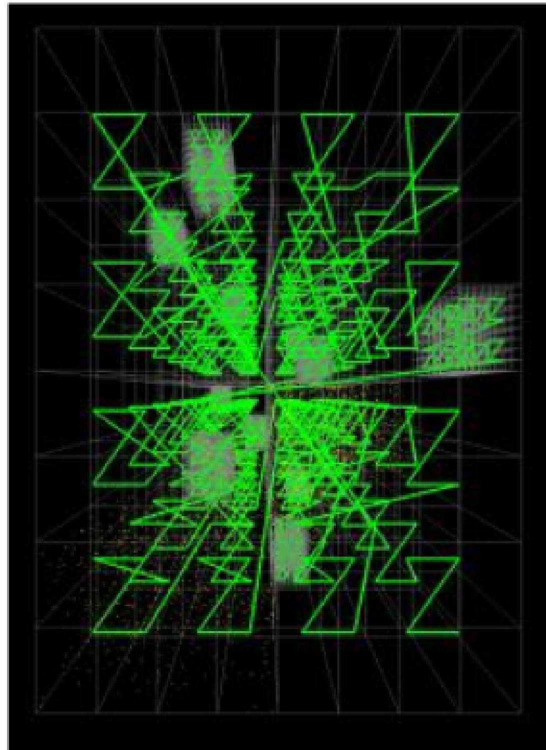


FIGURE 6.8: Morton recursion on a 3D space with cutting off unnecessary refinements.

- All ranks fill an array containing the number of particles per node, at the level three.
- The root nodes receives and sums all the arrays. It computes the medians and the octree nodes containing them, and broadcasts them. If more precision is needed, it informs the other ranks that further refining, only on the nodes holding the separators, is necessary.
- If further Morton recursions are needed, the ranks subdivide the identified node, fill a new array with the number of particles and send it to the root.
- The algorithm continues until a separator is computed.

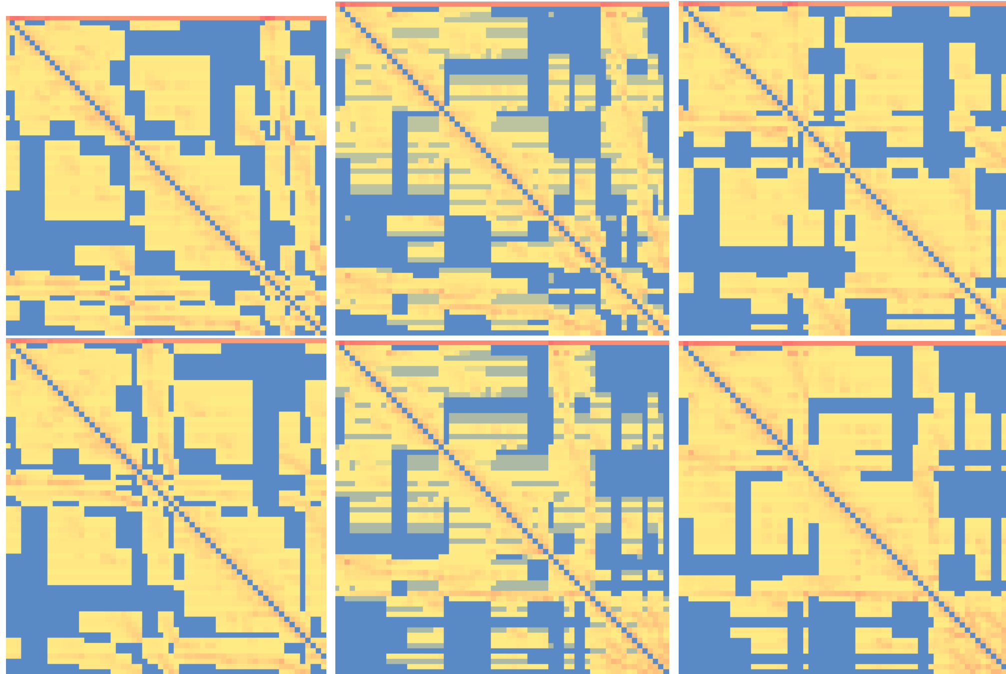


FIGURE 6.9: Axis order influence on communication matrices with 64 ranks. First column: X-axis first: XYZ and XZY - Second column: Y-axis first: YXZ and YZX, - Third column: Z-axis first : ZXY and ZYX.

Measure The standard Morton pattern alternates the cutting dimensions in the order : X, Y and Z. In 3D, rotating the Morton Z-pattern results in six different possibilities, which we have implemented and tested. Figure 6.9 shows the communication matrices obtained for each case, with 64 ranks.
At first, one can see that the Morton-XYZ communication matrix seems identical to the original one, presented in figure 6.2. This confirms the closeness between the original ordering and the Morton-XYZ partitioning. However, it is not exactly the same. In the original algorithm, the distribution takes place on an empirically determined distribution level, whereas in the Morton version, the precision is adjustable. In this experiment, we want to obtain the best balance possible of the computations. Therefore, the precision criterion is set to zero, which implies that the algorithm proceeds down until the leaves.
Second, one notices that rotating the Morton pattern notably impacts the communication matrix. The ZYX and ZXY resulting matrices eliminate the large horizontal and vertical lines previously identified in the original matrix.
Figure 6.10 presents the results in terms of execution time of SPECTRE's solver with the UAV test case. We test each Morton version using from 2 to 64 processes and compare them with the original load balancing, represented by the red dotted line. One can see that the best configuration varies with the number of MPI ranks. However, Morton ZYX ordering obtains a limited 5% to 10% performance improvement.

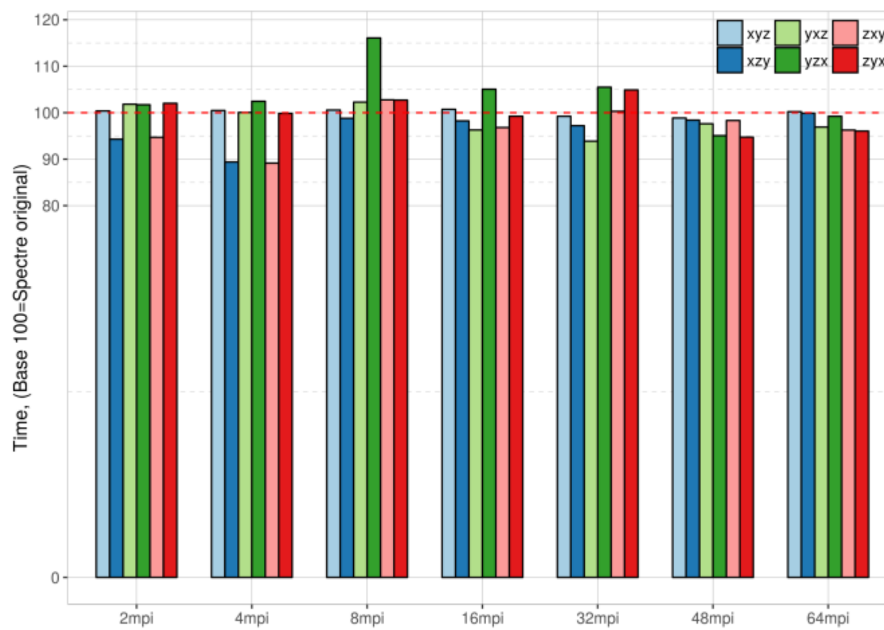


FIGURE 6.10: Execution time spent in the solver, UAV test case, with various Morton space filling curves. Base 100 represents the original execution time without modifying the original load balancing

Conclusion This experiment demonstrates again the impact of the load balancing strategy. The different Morton load balancing versions meet the same load balancing precision criterion. However, the execution times vary. This comes from the communications generated by cutting the domains, and is confirmed by the different communication matrices. This experiment corroborates the idea that reducing the communications should be integrated to the computational load balancing strategy. Finally, we identified that, on the UAV use case, the Z-first Morton ordering obtains the best results. We obtain a small benefit, over the original load balancing strategy, of the order of 5 to 10% depending on the number of domains.

6.5 Conclusion

In this chapter we focus on the load balancing strategy. We start by analyzing the current strategy and its results. This leads us to identify some limits and propose alternative strategies.

We propose two load balancing strategies, inspired by positive results in the literature [84] [55] [90]. The first one is an ameliorated ORB based on multi-sections, and the second one is a recursive Morton algorithm. Both algorithms are parallel. They can handle large amount of input bodies and exchange them during the computation.

With the histogram algorithm we concentrate on balancing the computation, and obtain well balanced domains. But the algorithm cuts generate an important communication volume which slows down the total execution time. We tried another strategy based on Morton ordering. In order to reduce the cost of a depth-first traversal, we propose a recursive algorithm cutting off unnecessary branches. Rotating the Z-shaped pattern produces different distributions, and different communication matrices. We obtain a limited 5 to 10% improvement on the UAV use case. However, the results vary depending on the number of domains, and certainly also on the use case.

Further investigation is required. A good distribution should balance the work but also minimize the communications. However, a blocking bulk-synchronous communication model induces many barriers, which could tear down any load balancing effort. Therefore, introducing asynchronism, overlapping, and fine grain task parallelism inside the nodes may influence our conclusion and must be fixed before exploring new load-balancing strategies.

Future work will investigate a load balancing strategy using a graph partitioner. The octree will be translated into a weighted graph, where weighted nodes represent the amount of computation and weighted edges reflect the remaining non-overlapped communication. This graph can then be partitioned with a mesh partitioner. METIS [10], for example, provides a multi-constraints algorithm which allows for load balancing the work represented by the nodes, while computing partitions that minimize the edge-cut, therefore minimizing the communication volume. A similar approach has been developed in [56] with interesting results. However, they have not been demonstrated on large number of nodes. Moreover, this strategy has not been combined with an asynchronous one-sided communication scheme, nor used with a compute-intensive CEM industrial application.

Chapter 7

Communications

Contents

7.1 Introduction 87
7.2 Profiling and analysis 88
7.3 Bulk synchronous versions, with MPI 92
7.4 Introducing asynchronism, with GASPI 96
7.5 Replacing global communications with local communications 101
7.6 Hybrid GASPI combined with tasks 103
7.7 Overall improvement and remaining potential 106
7.8 Conclusion 107

7.1 Introduction

In chapter 6, we focused on load balancing the work among the computational nodes. Distributing an octree between different nodes generates neighboring, and consequently, communications. In the FMM algorithm, the scalability of the distributed memory is a known difficulty, as presented in section 3.3.5. In this chapter we concentrate on optimizing these communications.
In section 7.2, we analyze the current communication scheme in terms of scalability, elapsed time and volume of data transmitted. In section 7.3, while remaining in the bulk synchronous parallel model, we first develop and compare different versions relying on non-blocking MPI. Then, in section 7.4, we develop two other versions relying on one-sided GASPI, the first one remains bulk and is intended to be compared to the MPI ones and the second one introduces some asynchronism. In section 7.5, we explore the effects of replacing collective communications with point-to-point messages. Finally, in section 7.6 we introduce task-based communications, before reviewing the obtained results and remaining possibilities in section 7.7. Finally, conclusions are drawn in section 7.8. This work has been introduced in the [99] as a short paper and published along with the load balancing strategies in [98]. The FMM-lib library implementing the GASPI communication strategies is available on GitHub [4].

7.2 Profiling and analysis

Motivation Measuring the scalability of our application is the starting point of our optimization process. Therefore, we profile and analyze the current communication scheme.
The objective is to identify potential bottlenecks in order to tackle them in future optimizations. We want to characterize how the application behaves with a growing number of nodes. In this chapter, our interest focuses mainly on the communications.
As presented in section 4.2, the communications consist in exchanging the vector of unknowns, and the required far field terms eventually owned by other MPI ranks. The unknown exchanges are handled with global MPI communications, whereas the far field exchanges are managed with a global Allreduce on the highest level and local point-to-point MPI Send/Receive operations on the other levels. In order to prioritize our optimizations, we need to differentiate the type of communications in order to measure them precisely, with regards to their share in the total communication time.
Since we start with a bulk synchronous scheme, we already foresee optimization possibilities thanks to overlapping. The advantages of overlapping the communications have been presented in section 2.2.3. Nonetheless, it is important to correctly estimate the potential of the planned optimization. In this experiment we evaluate the maximum potential gain obtainable with complete overlapping of all communications.

Methodology We run three types of experiments, detailed hereafter:

- To measure the scalability of the communications, we make *strong scaling* experiments. Strong scaling analysis consists in increasing the number of processing elements while keeping the problem size fixed. In this context, a perfect scaling corresponds to a speedup equal to the number of processors, which means that execution time is exactly divided by the number of processors. In HPC applications, it is difficult to reach linear scaling on large number of nodes, as distributing a problem among a large number of nodes increases the neighboring and thus the communications. We use this experiment to analyze the behavior of the communications in our application.

- In order to evaluate the room for improvement by overlapping, we create an artificial version with no communication at all. To deactivate the communications, we simply inhibit all the calls to the MPI library. Then, we also deactivate all the numerical checks in the code, to prevent it from being stopped because of incorrect results or stagnation. Although numerically meaningless, the execution of this version, with the same number of iterations gives us an insight on the optimization scope. This version is called *Total FMM Comm Free*.

- Finally, we break down the communications depending on what they transmit and how they do it. We differentiate communications involving the unknowns from those transmitting the far fields. We also look at the type of communication: global collective or local point-to-point. We measure the quantity of data transmitted and the elapsed time. Time measurement are done through calls to the MPI_Wtime() primitive, and quantification of the data is achieved via manual instrumentation of the code. Each time a communication is initiated, we add a call to an external library which holds count of the quantities. We indicate the number of items and the size in bytes. At the end of the run, the collected information is aggregated and we obtain

the total amount of data sent by each processor. We are not interested in the amount of received data, since it would just be an inversion of the amount of sent data.

Measures All experiments, for profiling or results, are run on a Dassault Aviation cluster composed by 32 nodes of two Intel Sandy Bridge, introduced in 1.5. Binaries are compiled with Intel 2015 and run with bullxmpi-1.2.9.1.

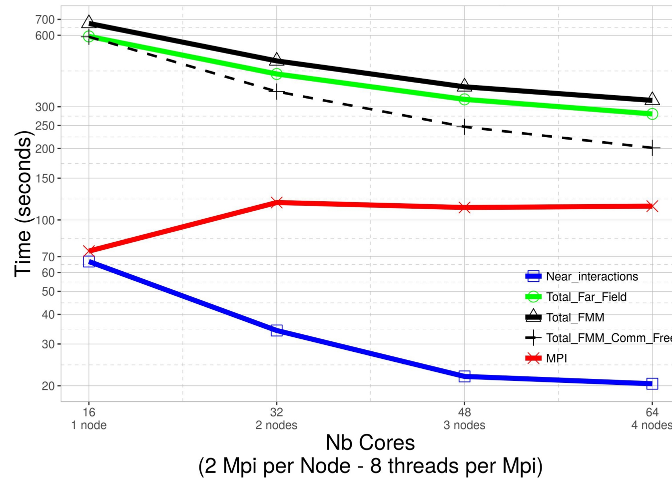


FIGURE 7.1: FMM strong scalability analysis with the UAV test case

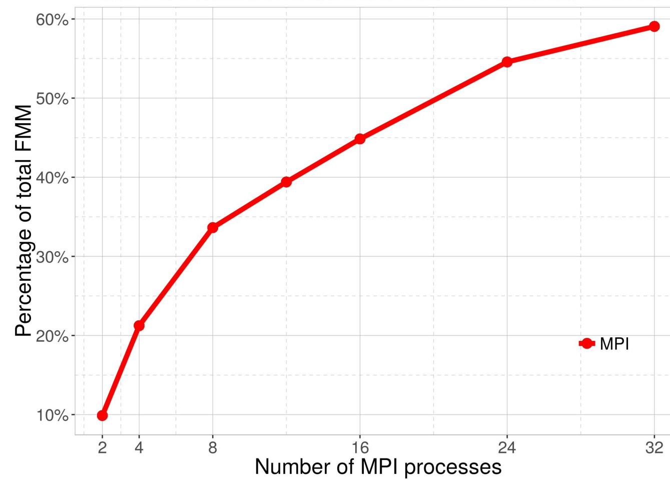


FIGURE 7.2: MPI communication time ratio on total FMM, with larger F7X test case.

Figure 7.1 shows a strong scaling experiment of the MLFMM part of SPECTRE. We use the UAV test case, introduced in section 4.2.1, and execute a complete run until convergence, which takes approximately 1430 iterations, depending on the configuration. In this first experiment, we follow the best performing execution mode used in production: one MPI process per socket with 8 OpenMP threads. Execution time is presented on a logarithmic scale.
Total FMM represents the total execution time, which can be split into far field and near field operations. Near field operations consist of a direct sparse matrix vector multiplication, while the far field operations consist in applying the FMM computational kernels and exchanging the neighboring nodes. These exchanges are measured and represented by the *MPI* line. According to the communication complexity study presented in section 3, one can see that the communication curve has an undesirable shape. *Total FMM Comm Free* is the artificial version with no communication. It is represented by the black dotted line. The gap between the Total FMM and the total *Total FMM Comm Free* depicts the maximum gain we can target by totally overlapping all the communications with computations. In this case, with 8 processes, it reaches 35%.

Further experiments with the larger F7X test case, are illustrated in figure 7.2. The application is run on 32 nodes and with only one MPI process per node. This configuration corresponds better to our target hybrid execution mode, with one process per node and thread parallelism inside the nodes. It also stronger challenges the communication's scalability. This experiment shows that the time spent in the communications grows considerably and reaches 59% at 32 nodes.

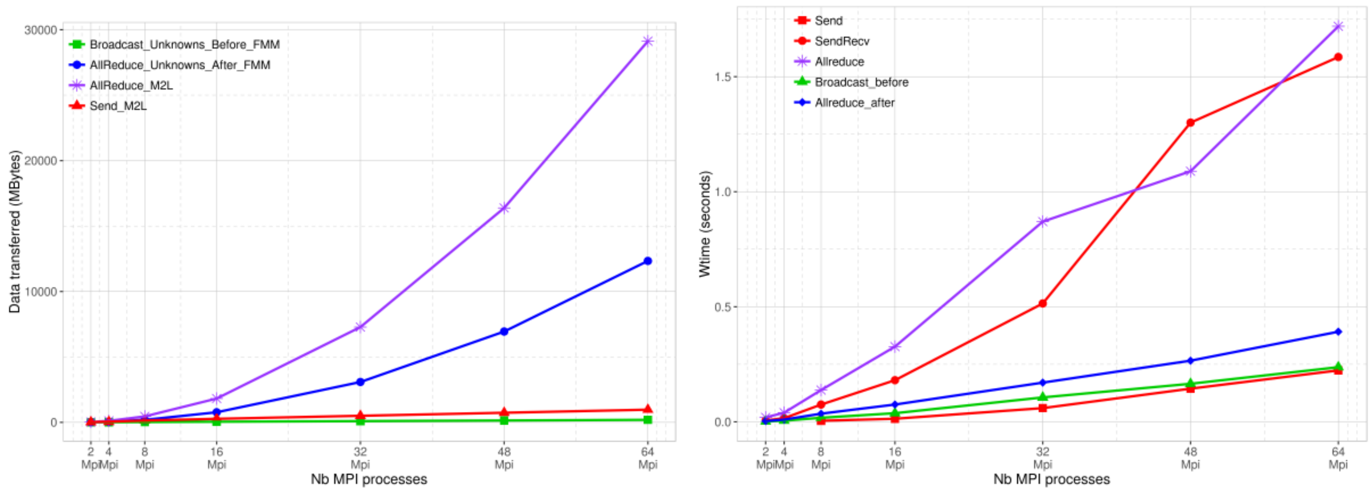


FIGURE 7.3: MPI communication volume analysis.
Left: quantity of data transferred - Right: corresponding elapsed time

Figure 7.3 confronts the amount of data exchanged on the left side, to the corresponding elapsed time, on the right side. For this experiment, we used the 80Ghz sphere test case, see 4.2.1, and four Sandy Bridge nodes. Since we focus only on communication, we make a pure MPI experiment and increment the number of active cores from

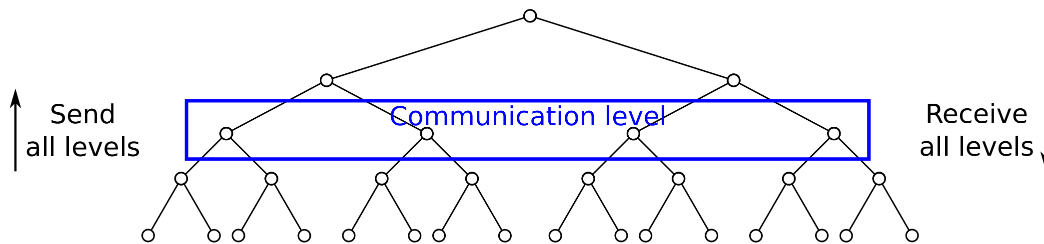


FIGURE 7.4: Bulk synchronous parallel model applied to the two phase FMM octree traversal.

2 to 64. One can see that the quantity of data exchanged is dominated by the two Allreduce operations. Global communications quantities grow fast since they are collective. According to the quantity, we should be focusing on the collective operations. However, the corresponding time costs mitigate the results. The time spent in the communications is mainly composed of the far field term communications equally shared by the global and local communications, with a rather linear growth.

Conclusion This profiling phase allows us to gain a better understanding of the application communication pattern SPECTRE's communications are currently organized in bulk synchronous way, characterized by alternating computation and communication phases. Figure 7.4 illustrates the use of this model in the current fast multipole two-phase octree traversal. The first bottom-up phase traverses the octree from the bottom to the top executing only computations. At the top of the tree, the computations stop and the communications are executed. Once the communications are completed, the second phase of computations starts, and traverses the octree from top to bottom. Generally, strong scaling experiments show that applications relying on bulk synchronous parallelism struggle to scale on large number of nodes. The communication ratio increases against the computations, and becomes predominant.
Our profiling experiments indicate that communications already constitute a bottleneck, which prevents the application from scaling on large clusters. Communications grow consistently until representing more than half of the execution time at only 32 nodes. This is a rather small cluster in comparison to what exists in current supercomputers. We expect them to be even more penalizing on larger configurations. Far field exchanges are identified as being the most time consuming communications.

Objectives Based on the prior findings, we prioritize our communication optimizations on the far field terms exchanges. We aim at proposing a strategy to replace the blocking communications by a hierarchical, asynchronous, more local and multithreaded version of these exchanges. To achieve this objective we will proceed in two steps:

- **Introducing asynchronism and overlapping**
  The first step consists in introducing asynchronous communications to profit from overlapping opportunities. Messages can be sent at the end of each level, during the upward pass, instead of waiting to reach the top of the tree. In the same way, receptions can be handled at the beginning of each level during the downward pass, as late as possible, when the data is really needed. This will permit the computation to continue while the communications progress in the background.

- **Progressing towards a fully asynchronous and hierarchical version, by the use of tasks**
  The second step will consist in extending this model to a full asynchronous version by the use of tasks. The computation will be parallelized with fine grain hierarchical tasks, including the communications. At node level, as soon a task has finished the computational part, it will handle the corresponding send operation. From a hierarchical point of view, without waiting for the end of the current level, it will spawn a new task for the father node, if the dependencies are satisfied. This version will be detailed in section 7.6.

7.3 Bulk synchronous versions, with MPI

Motivation To allow for overlapping, we need to introduce asynchronism and to replace blocking communications by non-blocking ones.
The MPI standard offers non-blocking communications via MPI Isend and MPI Irecv primitives. These calls return immediately, and let the communication progress while continuing the computation. However, as presented in section 2.2.3, the lack of message progression in non-blocking MPI communications, with the implementation available at the time of this PhD thesis, is a known issue.
Among the three methods presented in [80] to make the communications progress we chose to use manual progression. Enabling the progression thread on the MPI library on Dassault Aviation clusters would require recompiling the library. Moreover, to be compatible, all the driver infrastructure and all libraries must be thread-safe. Message progression at hardware level is out of scope, since we cannot modify our network architecture. Manual progression is to be ensured at user level, thus it seems more adapted to our experiments. It relies on calling the MPI Test primitive to trigger the communications. We want to identify the best way to use this feature, and wonder if it is more efficient to push communications from the sender's side or to pull from the receiver's one. We also want to make a fair comparison between the MPI and GPI (GASPI) libraries, to choose the most efficient one for further development. Previous experience with task-based multithreaded communications with GASPI has already demonstrated the efficiency of this library [117].

Methodology To ensure fairness for the evaluation of the efficiency, the original algorithm is altered as less as possible, and only the communications and necessary buffers are modified. Therefore, the communication model remains bulk synchronous.
The original version is illustrated in figure 7.5. The communications are blocking, and organized in rounds of pairs. Before starting the communication rounds, and in order to avoid overwriting during unpacking, each rank copies the current far fields array into a temporary buffer. Messages are large, since they combine, for a given receiver, all the far fields from all the octree levels.
One communication round consists in sending and receiving:

- Send: the data to send is copied from the temporary buffer into the send buffer. Then, it is sent.
- Reception: some data is received into the reception buffer. It is then unpacked and added to the far field array. The far field array is updated by incrementing the already present terms with the incoming contributions.

The new communication scheme remains bulk synchronous, but it relaxes the ordering constraints Each processor can communicate with any other, and in any order. More communications can be executed simultaneously. It is

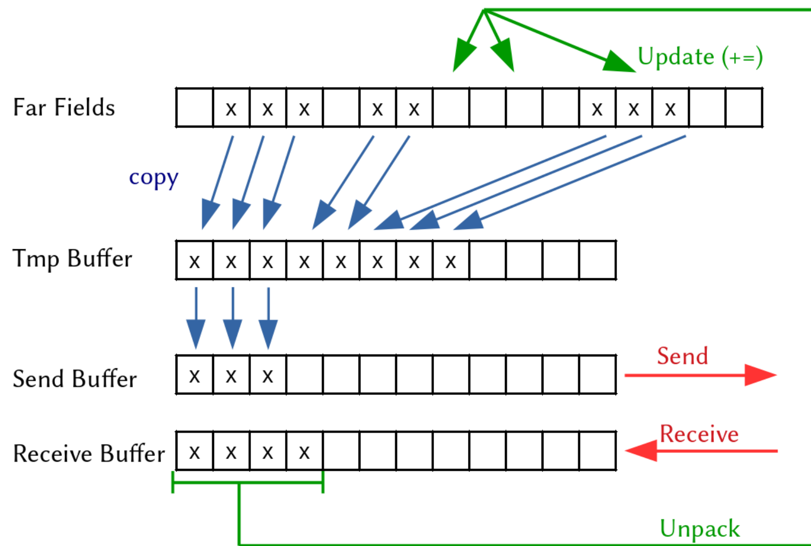


FIGURE 7.5: Original Mpi comunication scheme

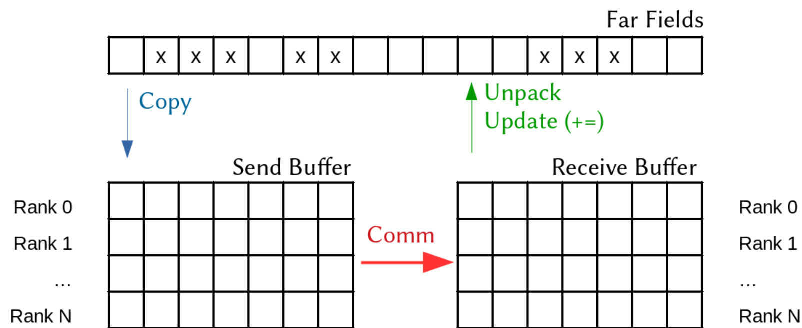


FIGURE 7.6: New Mpi comunication scheme

illustrated in figure 7.6. The local and temporary send and receive buffers are replaced with larger ones. These new buffers are large enough to hold all sending and receiving data. At the beginning of the sending process, the send buffer is initialized with all the far fields to send, for all recipients. Since the initialization is complete once and for all, it is not necessary to preserve the initial far field array anymore, and the temporary buffer is suppressed. The round-organized process is replaced by only one exchange phase, during which all the communications are executed.

Five different versions based non-blocking MPI have been developed. Figure 7.7 summarizes the algorithms with Fortran pseudo-code:

- *MPI waitany* is a naive version: all MPI ranks post all the MPI Irecv calls, followed by all the MPI Isend calls. Then they enter an MPI Waitany call, and thus, wait and receive any message from any source and in any order. Both receive and send calls are non-blocking, and rely on the MPI runtime to make the communication progress.
- *MPI progS* stands for MPI progression on the sender's side. We add a do while loop which iterates on each send operation and exits only once all the Isend operations are fully completed. Inside the loop, all the not terminated send operations are triggered via calls to MPI Test on the corresponding MPI Request.
- In *MPI progR* for progression on Receptions, we use the same principles applied to the receptions, and we unpack the incoming data as it arrives.
- In the *MPI progS progR* version, we trigger the communications first on the send operations, and also on the receptions. Data is unpacked after each arrival.
- And finally, *MPI progSR* is a contracted version: both send and reception communications are triggered, and the received data is unpacked once all communications are completed

There are several difficulties in making the communications progress manually. The first one lies in the opacity of the MPI library. As a user we don't have access to information about the status of the communication. We also do not know about the optimal interval between two MPI test calls. Too short intervals may cause overhead; too long intervals may not be enough. Another difficulty lies in choosing the right moment. Too soon, and there is nothing to progress; too late, and overlapping opportunities are missed.
Figure 7.8 illustrates the experimented triggering mechanisms. In our experiment, we consider the underlying MPI runtime as a blackbox. Nonetheless, we assume that triggering would follow some simple logic, as illustrated in figure 7.8, and detailed hereafter:

- Pushing on a communication takes place on the sender's side. Even if the communication is non-blocking, the sender blocks in a loop in which it pushes the sends. During this time, it cannot return to its computational work, and thus wastes some overlapping opportunities. Nonetheless, we assume that the communication will be effectively sent sooner, but with no guarantee on the behavior of the receiver, which may also delay the reception.
- On the contrary pulling on the communication from the receiver's side relieves the sender earlier. Both processes can compute and overlap the background communication. But during the meantime, the communication may not have progressed efficiently. Thus, part of the progression time may be reported in the pulling phase.
- Combining both schemes to provide the best of the two worlds. But it could be costly to block on both sides in pushing and blocking loops.

FIGURE 7.7: Pseudo code illustrating the five non-blocking MPI versions

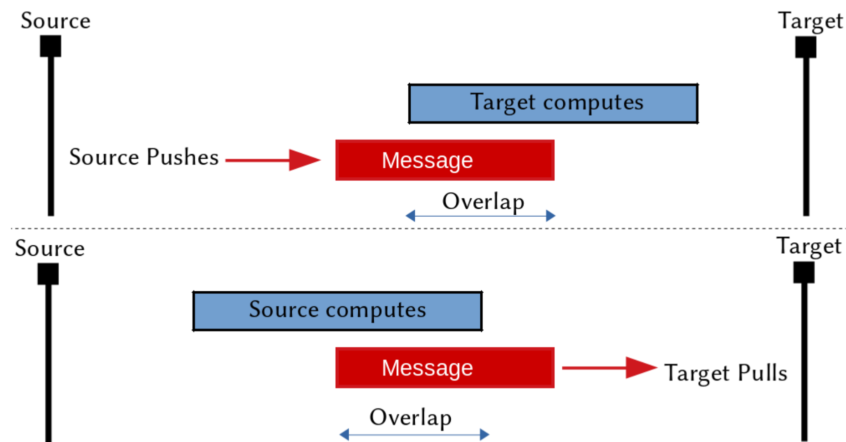


FIGURE 7.8: Expected overlapping opportunities while pushing or pulling the messages

Measure Figure 7.9 compares the execution time of all far field exchanges, of six different versions: the original reference version, and the five bulk synchronous versions, using non-blocking MPI.
The graph shows that, with an increasing number of nodes, the naive version becomes faster than the reference. At 32 nodes, the performance gap reaches 25%. By testing different ways of triggering the non-blocking communications, we want to identify the most effective one. We wonder if it consists in "pushing" the send operations, "pulling" the receptions, both after each non-blocking call, or both in a same loop. Among these versions, *MPI ProgS*, which "pushes" from the sender's side is the most interesting. It is not only more efficient starting at four nodes, but it is also consistently the best version, reaching 32% time gain over the reference at 32 nodes. The three other versions do not provide convincing performance results. Compared to the naive MPI Waitany version, and at some number of nodes (4, 8 and 16), triggering the communications from the receiver's side or combined from both sides, becomes less effective than doing nothing. At the time being, we cannot explain the behavior of these versions. We also have not run this experiment with other MPI libraries than BullxMPI, a tuned OpenMPI for Atos/Bull hardware, which could lead to different results.

Conclusion We have identified, that the most efficient non-blocking version is the *MPI progS* version, relying on pushing from the sender's side. Somewhat surprising is the fact that adding some pulling from the recipient's side degrades the performance. But as previously explained, the underlying MPI runtime is a black box. Maybe the fact of pulling isn't necessary anymore and only bears overhead. In the sequel, this version is retained for further comparison, and referred to as *MPI nonblock*.

7.4 Introducing asynchronism, with GASPI

Motivation As presented in section 7.2, our objective is to introduce communication overlapping with computation, thanks to asynchronous data transmission. In section 7.3 we proposed different non-blocking MPI versions.

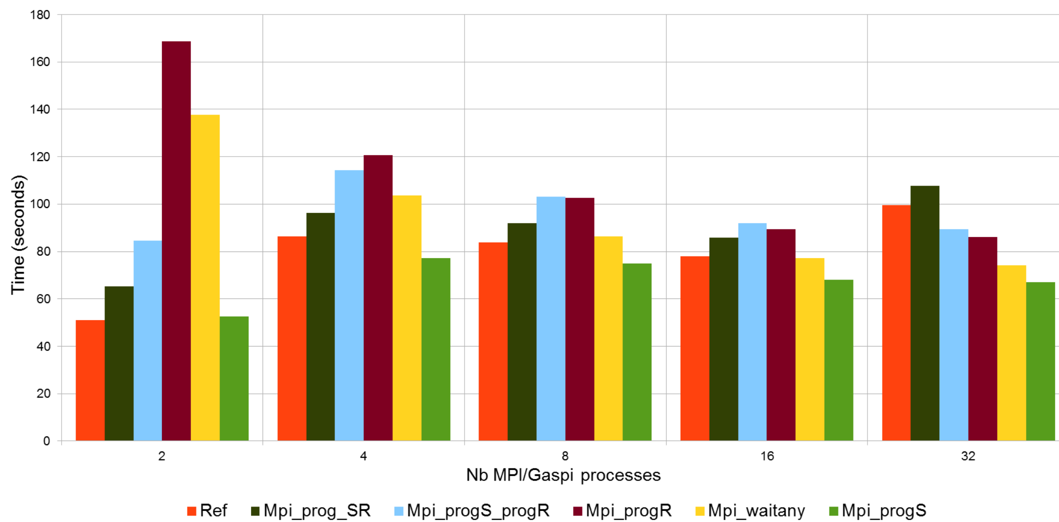


FIGURE 7.9: Far Fields total communication time.
Comparison between the original blocking MPI and five non-blocking MPI versions

In this section, we make a comparison between the best non-blocking MPI version, and a comparable version using GASPI communications. Then, we introduce overlapping by desynchronizing the communication scheme.

Methodology GASPI is introduced in section 2.3, and a simple usage example is detailed in figure 2.10. Thanks to the inter-operability of the GPI library we are able to modify only the target communications.
GASPI communications are entirely handled by the FMM-lib [4] C++ library, outside the original Fortran code. The library is called by the Fortran application, which provides pointers to the structures containing the data to exchange. At the initialization step, the GASPI segments are created and all the necessary information to handle the communications is pre-computed. Since the communications are one-sided, a complete knowledge of which data to exchange and with which process ranks is necessary to organize local and remote offset arrays and counters. Setting up these prerequisites also involves some initial communications.

GASPI Bulk *GASPI Bulk* is a preliminary GASPI version, which handles all the far field communications without modifying the original bulk algorithm. This version is destined to be compared with the non-blocking MPI ones. Accordingly to the precomputed offsets, the data is remotely written into the recipients' GASPI segments. The pseudo-code in figure 7.10 illustrates up this version. To explain how we use the GASPI library, it is more explicit than for the previous MPI versions.

- *On the sender's side*: Depending on the presence or not of data to send, there are two cases:
  - *If there is something to send*: *gaspi_write_notify* is used to write into the recipient's memory and to notify it. *send_buffer_id* and *local_offset* identify the memory segment and the address to read from.

remote_buffer_id and *remote_offset*, designate the recipient's memory segment and the address to write to. The *gaspi_notification_id* and *gaspi_notification_value* are used to indicate the source rank and if or not data is transmitted. We use the *gaspi_queue* 0 and the *GASPI_BLOCK* timeout, which means that the call to *gaspi_write_notify* will return only once the communication is completed.

  – *if not*: If there is nothing to send, we use the *gaspi_notify* call to notify the recipient anyway. We use the same *gaspi_queue* and set the notify value to *NO_DATA*, which is a macro for an integer.

- *On the receiver's side*: The recipient's enters a loop in which it waits for as much notifications as there are other ranks. Since both writes with notifications and simple writes have been sent on the same memory segment, it will receive all notifications with no distinction. When a notification arrives, it handles it by:
  – resetting it, so that it can be reused in the next iteration,
  – reading the notification value, which indicates if there is some data to unpack,
  – updating the counter.
  – if there is some data: it will read the source id, the notification id, unpack the data, and update the far field array. Since the ordering of the far field terms is preserved, there is no need to modify the unpacking process.

GASPI Async The *GASPI Async* version introduces asynchronism and overlapping. Efficient asynchronous communications consist in sending the available data as soon as possible, and receiving necessary information as late as possible, and overlap the transmissions with computations. Figure 7.11 illustrates how we proceed to create some overlapping opportunities: available data is sent and received respectively at the beginning and the end of each octree level. Data sent during the upward phase, at the end of the level *lev*, is only needed during the downward phase at the beginning of this same level. In this way, all the computations from the higher levels *lev-1* until the *communication level*, on both ascending and descending phase, can overlap this communication. When an FMM level is locally terminated, the available information is sent by directly writing into the remote recipient's memory at the pre-computed offset without requiring any action from the recipient. Compared to the GASPI Bulk version, we introduced some modifications:

- GASPI provides only two variables, *notify_value* and *notify_id*, that can be used to specify the containing of a send. Anticipating the future evolutions, we do not use a macro anymore to indicate if there is some data sent. Notifications indicating that no data is sent are suppressed. Instead, we pre-compute the number of transmissions expected from each rank and at each level. Thus, the receiver knows exactly what to wait for.

- *Sending:* On the sender's side, we use the *gaspi_write_notify* call and indicate the source and the level in the notification ID and value.

- *Receiving:* On the receiver's side, and on the beginning of each level we wait until receiving the expected communications. Communications are received from any source and in any order.

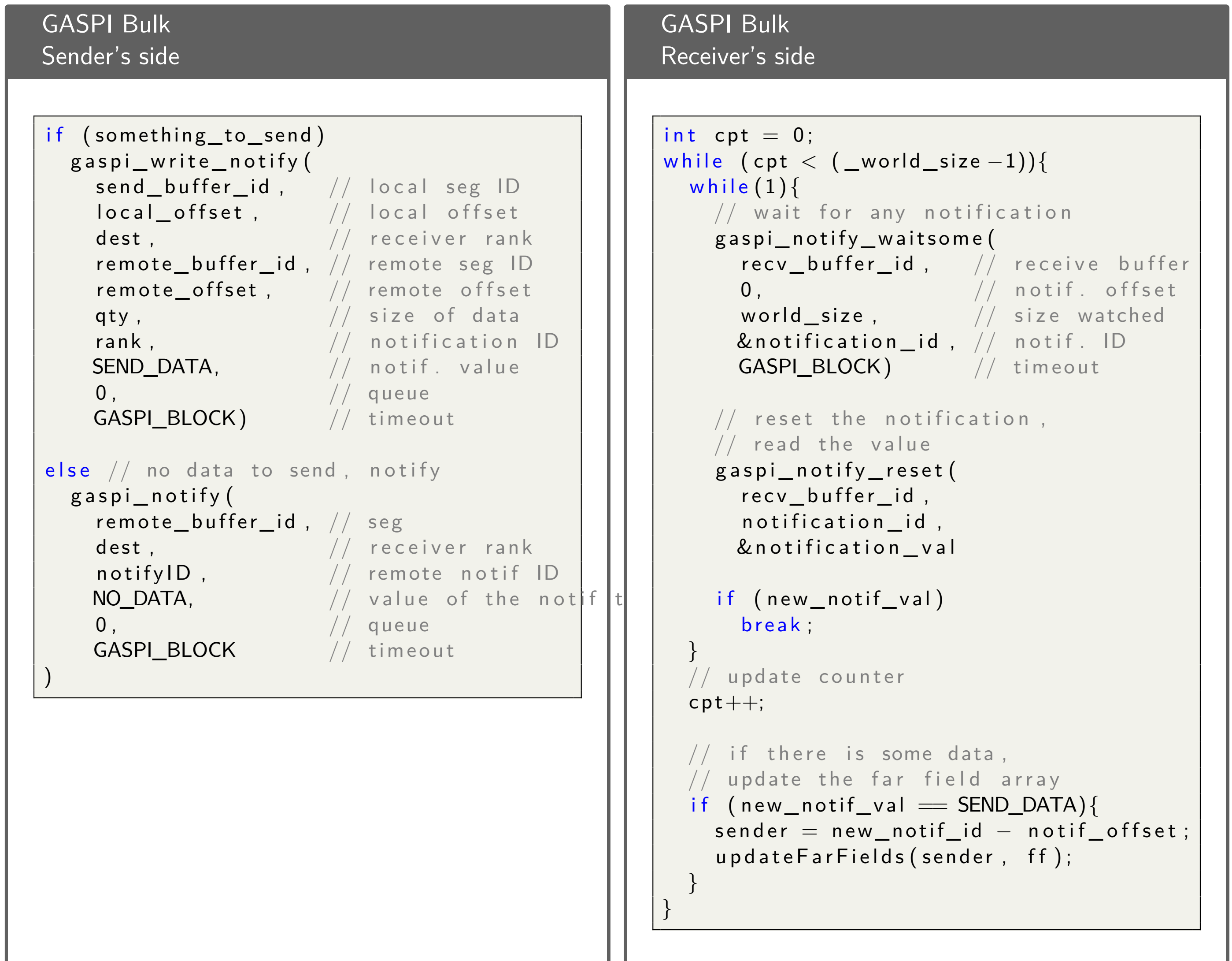


FIGURE 7.10: GASPI Bulk version, from sender and receiver sides.

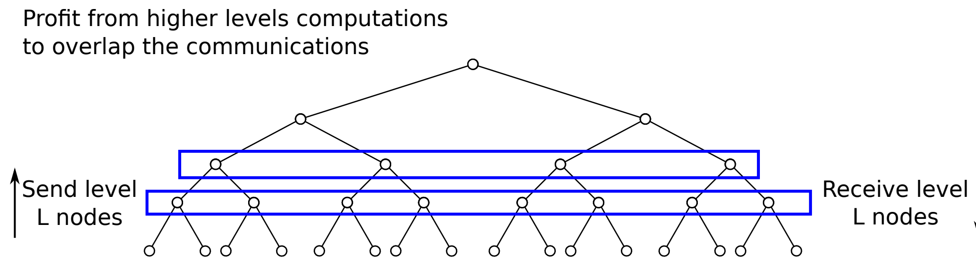


FIGURE 7.11: Introducing asynchronism and overlapping opportunities in the communication scheme

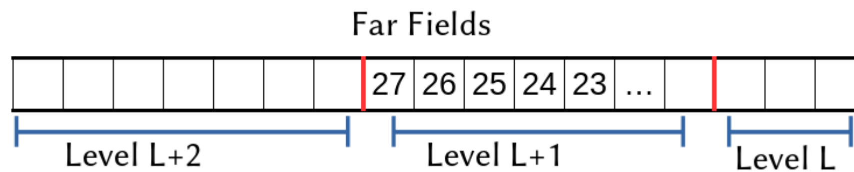


FIGURE 7.12: Far Field terms array organization

- *Unpacking:* Since the communicating process has been modified, the unpacking has to be adapted. One communication does not group all expected data from one rank anymore, but only one level. The far field array is organized as illustrated in figure 7.12.

  Levels are stored consecutively in a decreasing order. Inside a level, nodes are also stored in decreasing order. The size attributed to a level corresponds to the number of phi and theta angles of the level, and doubles with each higher level. Knowing the number of far field terms in each level, we compute an offset corresponding to the address of the incoming level far field terms. Since the order inside the level is still preserved, we keep the original unpacking process combined with the new offset computation.

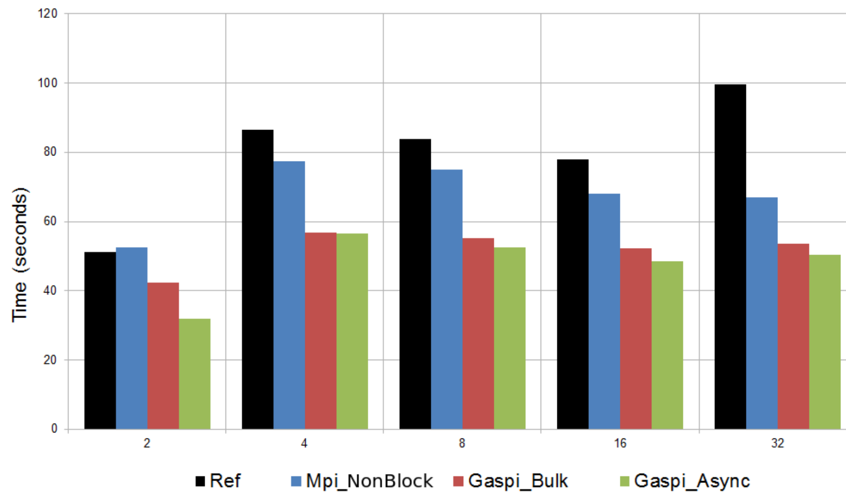


FIGURE 7.13: Far Fields total communication time, UAV test case.
Comparison between MPI and GASPI.

Measure In figure 7.13, in addition to the MPI non-blocking version, and the reference version, we added *GASPI Bulk* and *GASPI Async* to the comparison. Both GASPI versions are more efficient than the MPI non-blocking one, and this is the case for all number of nodes. One can see that, on 32 nodes, the *GASPI Bulk* version already reaches 46% speedup over *Ref*. Without changing the algorithm, and relying only on the efficiency of the communication library, a nice improvement is already attained. While remaining in a bulk synchronous parallel model, this difference must come from the use of different communication schemes: manual progression based for non-blocking two-sided MPI and hardware RDMA-based progression for one-sided GASPI communications. Finally, introducing overlapping enables the **GASPI Async** version to gain three more percentage points over the **Ref** version, with a total of 49% improvement on communications.

Conclusion The objective of this study is to select the most efficient communication model for our future developments. Our early results have been presented in [99]. The experiment results demonstrate the potential gain of moving from the MPI communication standard to GASPI. They are consistent with the results obtained in comparison studies presented in 2.3.2. Improvement thanks to overlapping is not satisfying yet. Further investigation is needed.

7.5 Replacing global communications with local communications

Motivation Figures 7.3 showed that the allreduce at the top of the tree is very large in quantity and important in execution time. Further investigation showed that it is also very sparse. As a blocking collective operation, if in addition contains useless zeros, then it needs to be optimized.

Methodology The collective allreduce operation is broken into small local point-to-point communications:

- We rely on the existing algorithm which initially pre-computes all the exchanges that need to be performed during the communication phase. We increase the iteration loop so that the communication level is included in this preparation phase. Consequently, all required information in order to use point-to-point communications at the communication level is calculated.
- We replace the collective allreduce operation by local point-to-point exchanges for all versions, including the reference MPI.

Each new version is now referred to with the *incLevcom* suffix, for including the communication level.

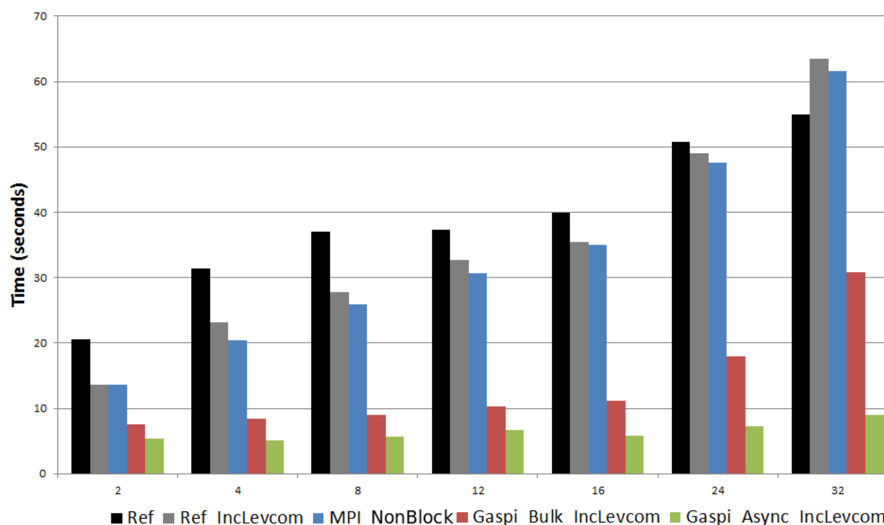


FIGURE 7.14: Far Fields communication time, larger F7X test case.
Top of the tree Allreduce replaced with point-to-point communications, for all versions.

Measure The experiment was first run with the UAV test case, and then confirmed with the larger F7X test case. The F7X results are presented in figure 7.14. One can see that both GASPI versions take more benefit from this modification than the MPI ones. *GASPI Async* outperforms all versions and reaches 83.5% reduction on the total far field communications, which means that the communication time has been divided by a factor of 6x.

Conclusion Communications at the top level of the octree are the most critical ones. Almost as soon as they are sent out, the far fields are needed for the downward phase. Replacing the global allreduce operations by more precise local-to-local communications, permits to suppress the costly transmission of useless zeros. Surprisingly, one can see that, on large number of nodes, the MPI version does not profit from this optimization. Progressively,

the *Ref_IncLevcom* version becomes slower than the original *Ref* one. This leads us to the conclusion that, in this communication scheme, point-to-point exchanges become more expensive than a much larger global communication. By contrast, the improvement obtained with the GASPI version is certain. The comparison between global MPI and local one-sided GASPI, leaves no doubt regarding the most efficient strategy. At this level of the tree there is no overlapping possibility. Communications has to be as small and effective as possible to reduce the bottleneck. We plan to continue refining the sends. The task-based version will enable us to send data by chunks, and to start the second-phase downside computation as soon as some data has arrived. Since it will not be necessary to wait for the entire level to be received,it will be possible to overlap a part of the communication.

7.6 Hybrid GASPI combined with tasks

Motivation In this section we lay the foundation for the inclusion of the communications inside the tasks. First, we want to evaluate the efficiency of multithreaded communications with GASPI. Second, since we aim at a task-based GASPI version, we have to modify the communication strategy accordingly. Until now, the communications take place at the end of each level. The next step will consist of executing them at the end of a task. The small number of large communications will be replaced by a large number of small communications. As this could lead to important overhead, we want to have control on the size of the messages. We also want to determine the best size to optimize the bandwidth usage.

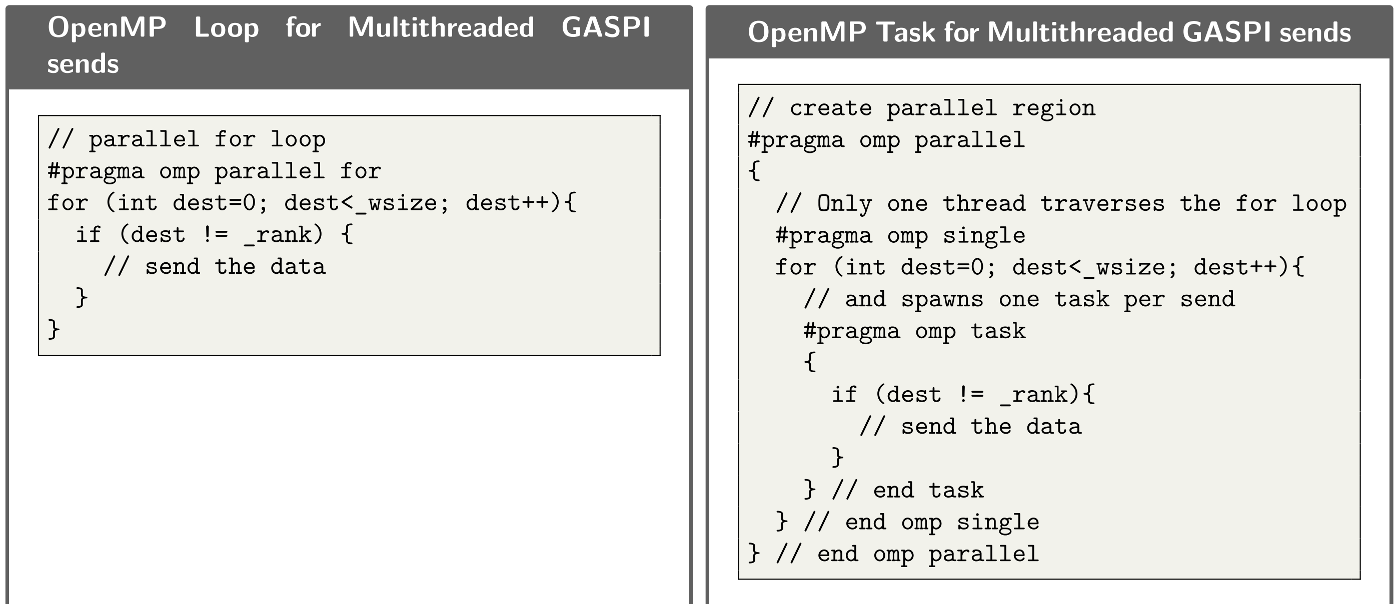
Methodology

Multithreading To evaluate multithreaded GASPI communications, we develop a multithreaded version based on the *Gaspi Async* version. Since each communication involves a different receiver, it is possible to distribute the communications among the threads. We realize two versions, as presented in figure 7.15.

- The first version, *GASPI Async Multi Omp*, uses standard OpenMP loop parallelization achieved via the *#pragma omp parallel for*, both on the sending and the receiving side.
- The second version, *GASPI Async Multi Omp Tasks* uses OpenMP 3.0 tasks. First, we create a parallel OpenMP region. Then, a single OpenMP thread spawns one task per communication. These tasks are inserted in the OpenMP task scheduler and will be handled by the pool of threads from the parallel region. Again, the same process is used on both receiver and sender sides.

Tasks and bounded-sized communication In the current algorithm each octree level is parallelized with a standard OpenMP loop. We replace them with parallel regions spawning a task for each node. Communications are handled by the tasks, at node level. This modification induces important redesign:

- *Task-handled communications* When a task has finished the computation corresponding to a node, it sends the available data. To avoid creating a very large number of small communications, the data is accumulated into a buffer. Since all tasks accumulate their data into the same buffer, we maintain counters, protected by mutexes.



OpenMP Loop for Multithreaded GASPI sends

```
// parallel for loop
#pragma omp parallel for
for (int dest=0; dest<_wsize; dest++){
  if (dest != _rank) {
    // send the data
  }
}
```

OpenMP Task for Multithreaded GASPI sends

```
// create parallel region
#pragma omp parallel
{
  // Only one thread traverses the for loop
  #pragma omp single
  for (int dest=0; dest<_wsize; dest++){
    // and spawns one task per send
    #pragma omp task
    {
      if (dest != _rank){
        // send the data
      }
    } // end task
  } // end omp single
} // end omp parallel
```

FIGURE 7.15: Pseudo code illustrating the two multithreaded GASPI versions

- *A size threshold is introduced*. It is adjustable at execution time through an environment variable. The size threshold is used to parameterize the size of the communications. When the quantity of data ready to be sent reaches the threshold, the communication is executed. The second case when the data is sent, is when the level is completed.
- *Identify what is sent*. In comparison with the original communication scheme, the number, the size and the order of the transmissions are modified. This was not the case in the precedent versions, since we executed the communications at the end of each level. The data was cut into chunks, but the ordering inside a level was kept. Now the size of a chunk varies with the threshold. Moreover, the order of the far fields inside a chunk is not defined. When the recipient receives a communication, knowledge of the source, the quantity of data, and all included nodes IDs is needed. The *notify_value* and *notify_id* are not sufficient to carry all the necessary description. Therefore, we add another GASPI segment, in which we create a buffer which will contain the list of all transmitted node IDs.
- *Know where to write*. On the receiver's side the filling-up of the send buffer is not done in an orderly manner anymore. Since different tasks can write into the same chunk, it is necessary to maintain addresses and counters. In this version we have to add counters, to know where to write *locally* in the send buffer and where to write *remotely* into the recipient's receiving buffer. We maintain different variables, for each recipient and each level:
  - An address into local side buffer. This indicates, for each recipient, and its level, the beginning of the subpart dedicated to this level's data.

  - A corresponding counter. It indicates the quantity of data already written, since the beginning of the level, and is used to compute an offset.
  - The address in the remote buffer. This indicates where to write remotely, into the recipient's buffer.
  - A corresponding counter. It holds the quantity of data already written at this address, also used for offset computation.
  - Finally, we add another counter, which keeps track of the quantity of data that has already been sent.

  Since we created a second send buffer, containing the node IDs, all these counters are duplicated: once for the far fields and once for the node IDs. Finally, counters are also needed to maintain addresses on the receiving buffer, to handle the unpacking.

- *Sending* When a task needs to write, it reserves the necessary place in the send buffer by getting the current address and updating it with the quantity to append. It also checks if it will be responsible for the effective sending of the data, either because a threshold is reached or a level is completed. Since these counters can be simultaneously accessed by multiple tasks, the read and update preparation phase is protected by a mutex. Then, the data is written into the far field and the information segments. If the communication is handled by the task, the process consists in sending two buffers and notifying the recipient. Both sends are executed by the use of *gaspi_write calls*, with the use of the same *gaspi_queue*. Only then, does the sender notify the recipient, using the same GASPI queue. The notify ID is used to indicate the source rank and the notify value to specify the level. Since both *gaspi_write* and the *gaspi_notify* are directed to the same *gaspi_queue*, we have the guarantee that when the recipients gets the notification, both far fields and information are completely transmitted.

- *Receiving* On the receiver's side and at the beginning of a given level, the rank waits for a pre-computed number of incoming nodes. Each time a communication arrives, it reads the source, the quantity and starts unpacking. And finally, from the level and the node id, it computes the offset where to accumulate the incoming data into the far field array.

Measure

Multithreading Execution time measurement with multithreaded communications showed no significant performance improvement for the multithreaded version, neither for parallel loops nor for tasks. We have knowledge of previous experiences [111] [118] where multithreading the communications improved the communication time. At the time of writing this thesis, we have no explanation of this behavior of our version running on Dassault Aviation Sandy Bridge nodes, and we are still investigating.

Bounded-sized communications With this experiment we want to identify the sweet spot in the trade-off between the number of messages and the size of messages, in order to maximize the bandwidth usage. This experiment has been run on 4 nodes and is illustrated in figure 7.16 We obtain a regular curve which decreases with the increasing message sizes, reaches its lowest point, and then increases again. On 4 nodes, and with the UAV test case, the best message size lies around 1 MB.

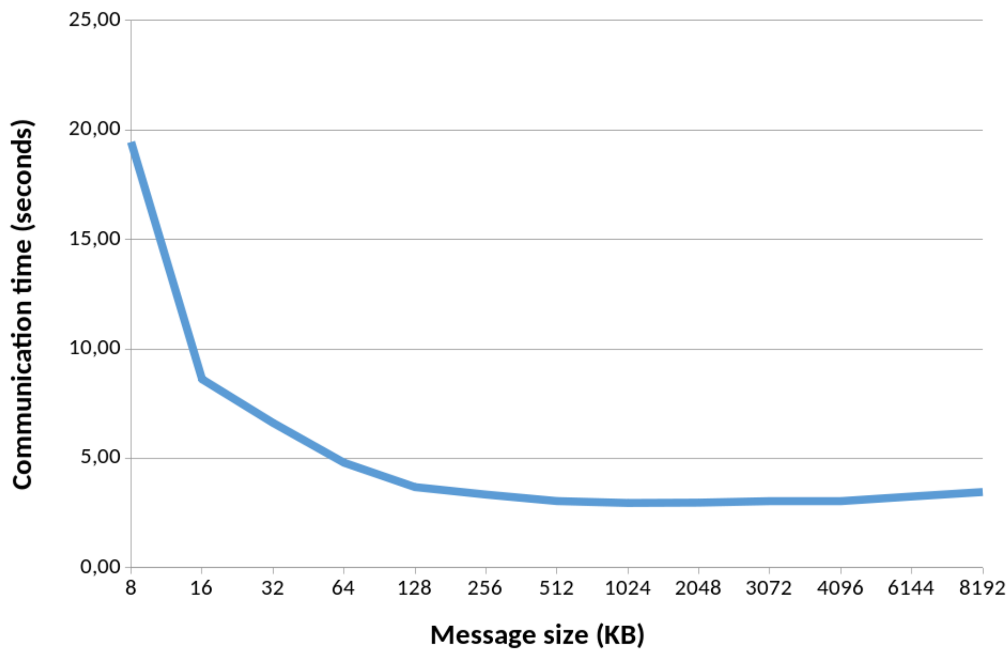


FIGURE 7.16: Message size variation impact experiment with GASPI.

Conclusion In this section we prepare the foundation for task-handled GASPI communications. For the time being, the first experiences with multithreaded communications do not improve the performance. However, they allow us to lay the foundations for future task-based communications. Indeed, we obtain correct numerical results with the task-based version, which validates that the removal of the ordered messages is successful. The second experience demonstrates that the size of the communication heavily impacts the efficiency. Decreasing the message size mechanically increases their number. This experience allows us to identify the best compromise.

7.7 Overall improvement and remaining potential

In this section we want to summarize the gain we have obtained and to identify the remaining room we have for further optimization. We execute our best version, the asynchronous GASPI Async one, and like in the initial profiling phase, see section 7.2, we measure the time spent in communication while differentiating far fields and unknowns. Then we calculate the ratio of each part.
Figure 7.17 shows the evolution of the communication ratio, on the F7X test case. Initial communication ratio, presented in red, was introduced in section 7.2. The green curve presents the new communication ratio. The gap between these two curves represents the gain we have obtained, One can see that, at 32 nodes, the ratio decreased from 59.1% to 33.8%, resulting in a 1.42x overall speedup in the MLFMM with 32 nodes.
or the FMM Matrix x Vector product, the improvement ranges from 10

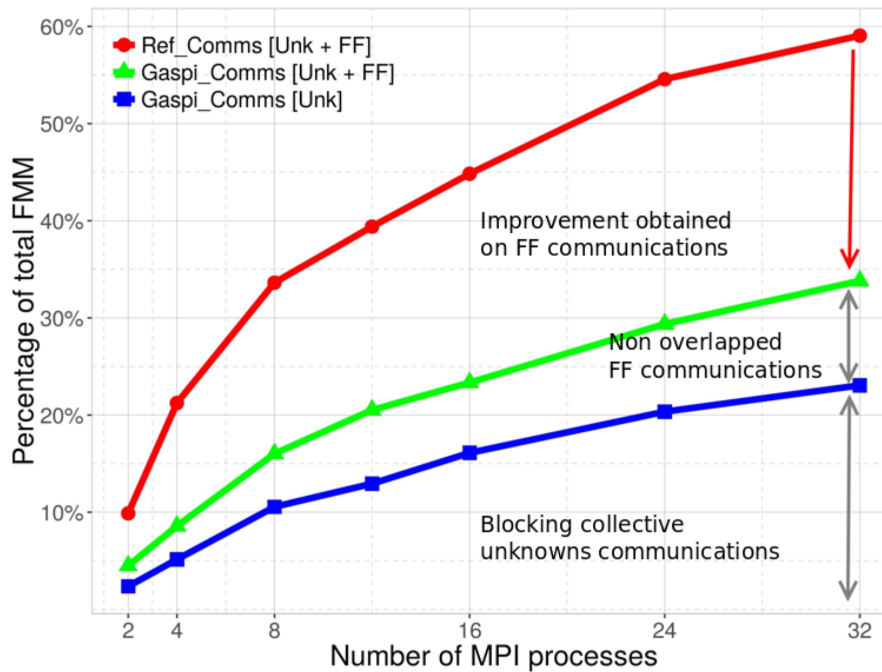


FIGURE 7.17: New Communication time ratio on total FMM, F7X test case. With breakdown between Far Fields and Unknowns

Communications required for unknowns are represented in blue. Since the far field part has been optimized, the unknowns part is mechanically growing. Therefore, it should be a priority for future communications optimizations. Finally, the gap between the green and the blue curve indicates the remaining scope for further improvements on the far field exchanges.

7.8 Conclusion

In this chapter we focus on optimizing the far field communications. We start by profiling our application in order to identify bottlenecks. We use them to specify our optimization target and leverage performance improvement. Therefore, we concentrate on optimizing the far field terms exchanges of the M2L operator.

We make a performance comparison between the largely used MPI library and the new upcoming GASPI standard. We test the non-blocking MPI primitives and try to make the best of them by the use of manual progression. In the context of our application and test cases, pushing from the sender's side is identified as being the most effective strategy. We compare this optimized MPI version, to a comparable bulk parallel GASPI version, and demonstrate the efficiency of GASPI one-sided RDMA messages. This enables us to gain a 46% speedup over the initial MPI version.

To introduce overlapping opportunities we start the move from a bulk synchronous communication scheme towards an asynchronous version allowing for communication overlap. We break down the round-organized blocking communication phase into smaller messages. We replace the large communication phase, initially taking place at the top of the tree, by small communications at each level. The messages are sent at the end of each level, and the receptions are executed, when needed, at the beginning of the octree level. This reveals some overlapping opportunities and allowed us to reach an almost 2x speedup over the original version. We identify that our optimizations are restrained by a large and very sparse collective communication, which we break down into small local point-to-point messages. This allows us for dividing the far field communication time by 6.

To give an insight into the behavior of the application on much larger current supercomputers, 32-node strong scaling experiments are not enough. Courtesy of CEA, Dassault Aviation has access to their INTI Cluster, which hosts a class of more than 200 Sandy nodes. We tried to run our experiments on this cluster. Unfortunately we ran into some problems concerning the MPI runtime. We tested in vain the Intel MPI and OpenMPI libraries. The related experiments are presented in annex 1. At the time of writing this thesis, and due to restriction to run on external clusters, we are not able to make larger experiments to measure the maximum gain in scalability.

To lay the foundations for the future hierarchical task-based version, we are interested in multithreading the GASPI communications with the use of threads and tasks. We progressively move towards task-handled communications. First, we experience multithreading with the simple use of threads and loop parallelization. Then, we introduce tasks, firstly in the sending loop, then at octree level. Important modifications are made to enable the messages to be handled at task level. Among them, controlling the size, and consequently, the number of messages is a priority. Therefore we introduce a threshold which enables us to parameterize the message size at runtime. This allows us for experimenting different message sizes and identifying a sweet spot for our architecture, application and test case combination.

The work on task-handled communications will be extended. For the moment tasks are limited to a parallel region inside a level. In the next section, we will develop a hierarchical parallelization of the FMM algorithm, including task-based communications. The level consistency inside a send buffer will not be maintained anymore, and the messages will aggregate data from different levels. Therefore, a last version of the task-based GASPI communications will be developed. Finally, the analysis of the remaining improvement opportunities shows that the collective communications required for unknowns have grown mechanically and should receive our attention for future improvement.

Optimizing the communications is mandatory to target large supercomputers. Nonetheless, it is also necessary to prepare the application to tackle multicore and manycore architectures which require higher strong scalability. Therefore, intranode optimization will be studied in the next chapter.

Last but not least, during this optimization phase we experience two ways of developing our strategies. MPI versions are coded directly in the original Fortran code, while GASPI development are done through the FMM library. On the one hand developing directly in the original application has the advantage of being quick to start. But it induces manipulations of the large original Fortran code, hence, long compilation times, and therefore tedious testing and

debugging. On the other hand, using a library implies more preparation to set up. All variables must be transmitted to the library, and since it is developed in C++, one has to be very careful about data structure management and indexes computations. Therefore, combining Fortran and C/C++ is error prone. Nonetheless, the advantages are numerous. For the development, the usage of a much more flexible language is appreciable. Since the FMM library is compiled as shared library, the compilation process is much faster. The library is simply linked to the application, which therefore, does not need to be recompiled. Furthermore, the development efforts being capitalized in the library, they could be easily reused for another application. Even better, since the GASPI communication library contains no confidential information about the original SPECTRE application, it can be distributed under the LGPL license. These two points are very important to support an academic work.

Chapter 8

Tasks

Contents

8.1 Introduction **111**
8.2 Profiling and Analysis **111**
8.3 Towards asynchronous task-based parallelism **114**
8.3.1 Current parallelization scheme 114
8.3.2 OpenMP 4.0 tasks with depend clause 115
8.3.3 Recursive version with scoreboards 117
8.4 Progress report **118**
8.5 Conclusion **122**

8.1 Introduction

In chapter 7, we focused on distributed memory parallelism. We broke a bulk synchronous communication pattern into small local asynchronous and one-sided messages. Introducing asynchronism and suppressing unnecessary synchronizations allowed for considerable improvement in time performance and scalability.
In this chapter, we motivate and describe our future work on shared memory. Our objective is to desynchronize the current computation algorithm with tasks, and replace level by level over-synchronization with small local dependencies. Ultimately, both parallelism layers should be combined by introducing the communications inside the tasks, in order to obtain a completely asynchronous algorithm. In section 8.2, we analyze the current parallelism scheme and profile it. In section 8.3, we elaborate two task-based parallelization strategies. Section 8.4 reports the progress and early results, and summarizes the remaining future work.

8.2 Profiling and Analysis

Motivation Before starting the optimization process of the shared memory parallelism, we start by profiling the current shared memory implementation. SPECTRE/MLFMM operators are currently parallelized with OpenMP loop parallelization, introduced in section 2.4.2.

Methodology We run two different experiments detailed hereafter:

- *Strong Scaling:* We measure the scalability of the MLFMM operators with an increasing number of threads while keeping the same problem size. Perfect scaling would be represented by a straight line, with parallel execution time equal to the sequential execution time divided by the number of threads.
- *Hotspots:* We use MAQAO [8], a performance analysis tool developed at UVSQ to identify the hotspots, defined as the most time consuming parts of the application.

Measure

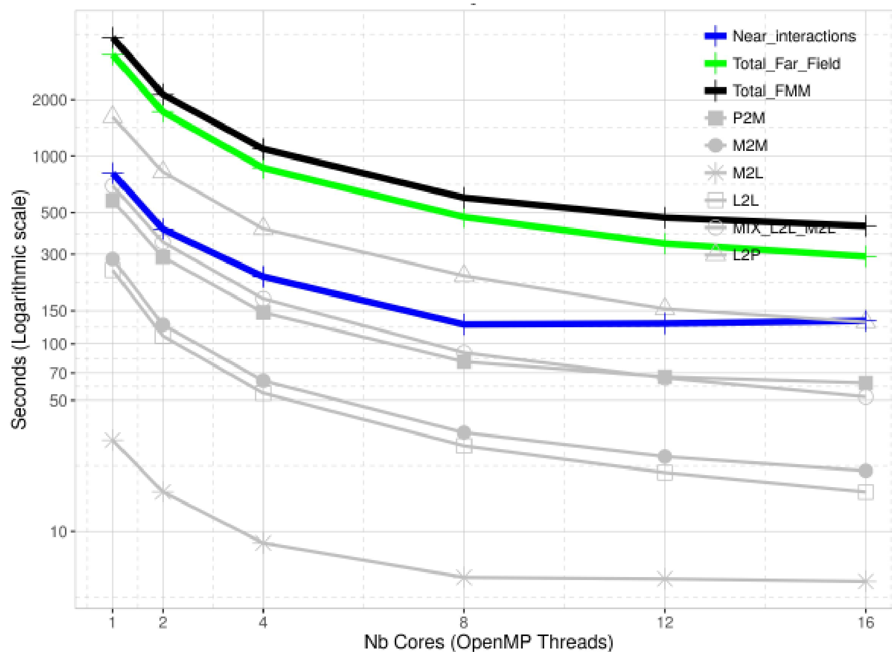


FIGURE 8.1: Shared memory scalability with the UAV test case.

Scalability Figure 8.1 focuses on the scalability of the MLFMM operators. We use the UAV test case and run it on one single Sandy Bridge node while increasing the number of threads from 1 to 16. The graph, with log scales, highlights the lack of shared memory parallelism: over eight threads, the corresponding parallel efficiency falls under 56%. This figure shows that efficient usage of current manycore architectures is a challenge which must be addressed, particularly in light of the future increase in the number of cores available at compute level.

Hotspots Figure 8.2 presents, the result of the hotspot identification with MAQAO: it shows the five most time consuming functions, displayed as ratio of the function execution time over the total application. MAQAO identifies

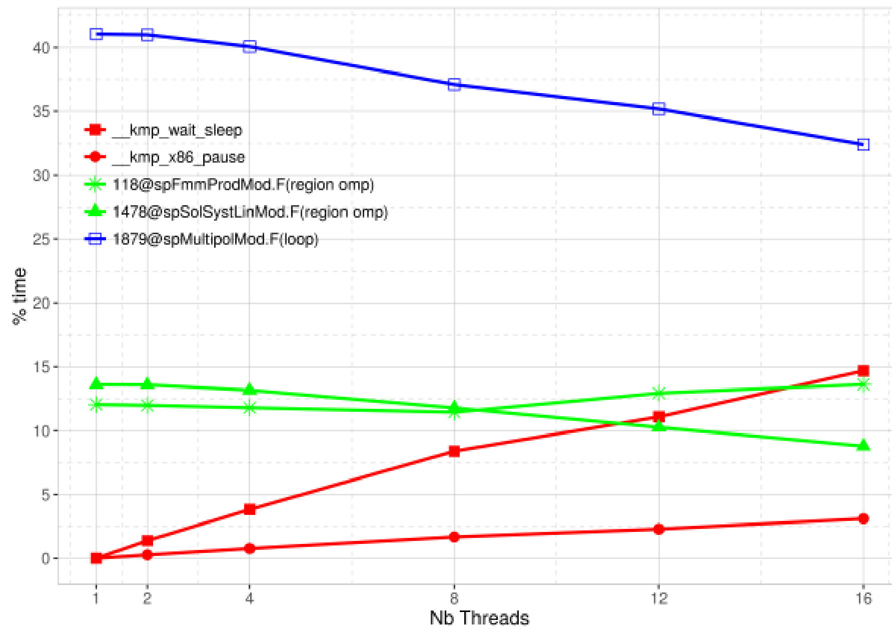


FIGURE 8.2: Hotspot identification with MAQAO

a growing execution time spent in kmp_wait and kmp_pause, represented in red, reaching almost 20% with 16 threads. To examine these results more carefully and provide clear explanation of the idle time, MAQAO provides call chain classification. However the linux kernel available on the Dassault Aviation cluster does not allow the required dependencies. Nonetheless, this results cleary pinpoint a load balancing issue which is generating large idle times.

The blue line corresponds to the main loop in the L2P operator. This loop is parallelized with OpenMP. To update the final unknown vector, it iterates over the nodes, and over the elements of each node. Since multiple elements may impact the same unknown, the unknowns are protected with mutexes. As a result, this operator is the most time consuming. This observation is coherent with figure 8.1, where the L2P kernel is the most time consuming kernel. The two other green lines correspond to sub-parts of the near field and preconditioner. They correspond to previously identified code sections already parallelized with OpenMP regions or loops.

Conclusion To summarize, these experiments show that the node level parallelization scheme has a large room for improvement. Indeed, the MLFMM operators hardly scale over eight threads and spend considerable time being idle. In production mode, SPECTRE is often executed with one MPI process per socket. This configuration mitigates the low shared memory scalability, but increases the communications. To efficiently benefit from modern multicore systems, it is mandatory to optimize the shared memory parallelism. In the next section, we detail the current limitations and present our proposition relying on tasks.

8.3 Towards asynchronous task-based parallelism

8.3.1 Current parallelization scheme

Over-synchronization As described in section 3.2, the FMM algorithm carries data dependencies both during the upward as well as the downward phases. During the upward phase, it is necessary to wait for completion of the computation for the children's contribution. During the downward phase, the result of the parent octree node computation, and incoming neighboring contributions are required. The current version of the algorithm still carries

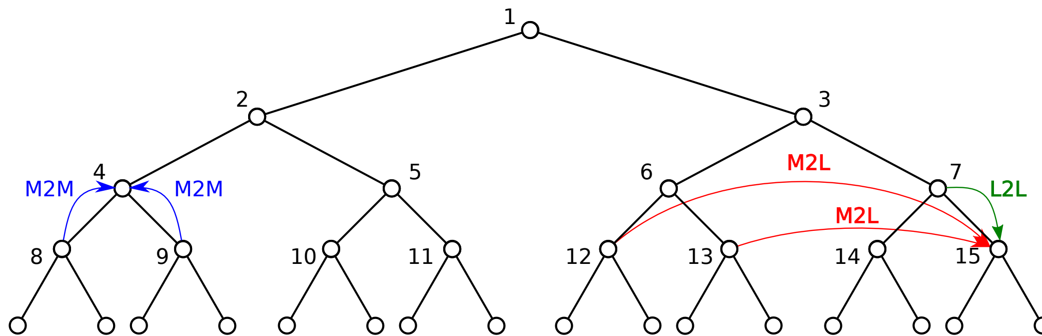


FIGURE 8.3: Asynchronous task-based execution of the FMM algorithm

over-synchronizations, as illustrated in figure 8.3, where FMM operators are executed level by level. Indeed, the computation of the node number 4 can start as soon as nodes 8 and 9 are ready. And likewise, the downward operator L2L on number 15 can start after completion of L2L on the parent node 7 and reception of the contributions of the neighboring nodes 12 and 13.

Loop parallelization The current shared memory parallelism relies on loop parallelization. However, the OpenMP loops are used in an unconventional way and the work is manually scheduled over the threads. This process is illustrated with a Fortran pseudo-code in figure 8.4. The loop size corresponds to the number of threads. The work is cut into chunks and distributed to the threads during the execution. A global counter is maintained to handle the remaining work and protected by a mutex. Each thread executes a loop inside which it gets a chunk of work and executes it. When the code was developed, the different OpenMP schedulers have been tested and this method was the most efficient. Also, since each thread executes only one iteration of the loop, this method allows each thread to allocate a private buffer.
However, with a large number of threads, the use of a mutex may introduce a bottleneck. It is not suitable for current, and future, multicore architectures with large number of cores. Moreover, OpenMP provides simpler methods to declare private variables. We propose an alternative version of the loop parallelization in section 8.4. Furthermore, to move from a bulk synchronous parallel loop based version towards an asynchronous task-based version we propose two strategies detailed hereafter.

```
!$OMP PARALLEL DO PRIVATE(..)

  ! loop on the threads
  do i_thread=1,n_threads
    allocate(tab(size), stat=ok)  ! allocate private buffer
    ipaq = firstpaq               ! initialize node id to the first node to handle

    ! second loop: get a chunk of work and execute it
    do while (ipaq <= lastpaq)

      if(mod(..) =0)  then        ! test if the complete chunk has been handled
        ! get a new chunk of work
        call omp_set_lock(mutex)  ! protect the global counter
        ipaq = ipaqGlobal + 1     ! update pointer with global counter
        ipaqGlobal = ipaqGlobal + chunkSize - 1   ! update global counter
        call omp_unset_lock(mutex)
      else
        ipaq = ipaq + 1           ! execute next node
      endif
```

FIGURE 8.4: SPECTRE's OpenMP Loop Parallelization

8.3.2 OpenMP 4.0 tasks with depend clause

The OpenMP tasking model is presented in section 2.5.1. It is well-suited for dynamic and irregular problems. Moreover, the OpenMP 4.0 release introduces the *depend* clause, which enables fine synchronization between tasks. The *depend* keyword specifies data access modes, which can be *in*, *out* or *inout*. Indeed, the complete task graph can be expressed by specifying data dependencies.
This model is interesting for SPECTRE's MLFMM algorithm. Hereafter, we detail how we plan to use the depend clause to handle the dependencies between the FMM operators:

- *Scoreboards* The FMM algorithm carries dependencies on both upward and downward phases. To simplify the expression of these dependencies we use three scoreboards:
  - *Scoreboard_up* is an array of the size of the number of nodes in the octree. It maintains the list of octree nodes which upward phase computations are completed.
  - *Scoreboard_down* is the equivalent for the downward phase.
  - *Scoreboard_comm* contains for each node, an array the size of which corresponds to the node's interaction list size. It is updated to keep track of the incoming M2L contributions.
- *Upward Phase* During the upward phase, multipoles corresponding to octree nodes are computed. When an FMM operator completes a multipole expansion computation, it updates the corresponding octree node in the scoreboard_up array. Hereafter, we give an example of how to express the P2M and M2M operators dependencies, using the scoreboard_up. The P2M operator has no entry dependency. It starts from the

leaves and computes the first multipoles. Assuming the current node is identified with `nodeID`, we specify the P2M dependencies as follows:

```
!$OMP TASK depend (out: scoreboard_up(nodeID))
```

In contrast, the M2M operator can compute the multipole of a node only if all the children multipoles are ready. Thus, assuming the first and last sons are contiguous and identified with `firstSon` and `lastSon`, and since OpenMP supports the array section notation, we specify the M2M dependencies as follows:

```
!$OMP TASK depend(in:scoreboard_up(firstSon:lastSon) depend(out:scoreboard_up(nodeID))
```

Finally, in order to initialize the downward phase, the last level M2M tasks will update the scoreboard_down array instead of the scoreboard_up array.

- *Downward Phase* Conversely, the downward phase is composed by the M2L, L2L and L2P operators, and the completed nodes are marked in the scoreboard_down array. However, some dependencies are more complex.
  - The M2L operator starts when the M2M operator is completed. It updates the scoreboard_comm.
  - The L2L operator starts when the parent node L2L is completed, and the incoming neighboring contributions carried by the M2L operators are received.
  - Finally, the L2P operator can start when the parent node L2L is completed.

Distributed memory limit The advantage of using the OpenMP 4.0 depend clause is the simplicity to express the dependencies. In the bibliography, we identify different experiences using them for FMM [29] [36] on shared memory architectures. They both demonstrate interesting performance results, however, to our best knowledge; they have not been tested on distributed architectures. Indeed, the OpenMP scheduler was not designed to handle dependencies on data contained in incoming messages. However, one possibility to work around this limitation could be to insert a task in charge of handling the messages receptions. This task would carry the list of the expected contributions as an output dependency, which therefore would be managed by the OpenMP scheduler.

Recursivity Moreover, the model does not support recursivity. Indeed, the OpenMP 4.0 standard specifies that the dependencies are handled only within sibling tasks. Therefore, all the tasks must be created on the same level. Figure 8.5, illustrates how the tasks can be created. We note that the same methodology is used in [33]. We start by creating a parallel region. Inside this parallel region, a single thread traverses the complete octree and spawns all the corresponding tasks. Since no task spawns another task, all the generated tasks are siblings. They are executed by the pool of tasks from the parallel region. However, this may results in a large pool of tasks, which might stress the scheduler.

Scheduling limitations Another limitation lies in the implementation of the OpenMP scheduler, which seems to favor the task insertion order [31], and may result in parallel starvation. In the case of FMM, this is not the optimum behavior. Indeed, the incoming contributions are summed to form the final contribution but there should

```
!$OMP PARALLEL           ! start parallel region
!$OMP SINGLE          ! each thread allocates a private buffer

  ! upward phase
  call P2M_tasks
  do i=nivterm-1,levcom
    call M2M_task(i)

  ! downward phase
  do i=levcom,nivterm-1
    call M2L_task(i)

  do i=levcom,nivterm-1
    call L2L_task(i)

  call L2P_tasks

!$OMP END SINGLE
!$OMP END PARALLEL
```

FIGURE 8.5: Creating the OpenMP depend tasks

not be any constraint on the order. Moreover, as identified in [123], this version may suffer from runtime overhead due to task dependency tracking and managing.
To sum up, this version does not seem well fitted for distributed memory parallelism and may suffer from scheduling overheads and limitations. To mitigate all these constraints we propose an alternative task-based version in which tasks are spawn recursively and dependencies are managed manually through scoreboards.

8.3.3 Recursive version with scoreboards

In this section we present our future work for a recursive parallel task-based version of the FMM algorithm. Scoreboards are used to track the dependencies and tasks are spawned recursively when dependencies are satisfied.

Scoreboards In this version, we also use three similar scoreboards. The difference is that we use explicit counters to keep track of the unsatisfied dependencies.

- Scoreboard_up, used during the upward phase, is initialized with the number of children per node.
- Scoreboard_down, used during the downward phase, is initialized with the number nodes in the interaction list + 1 for the father.
- Scoreboard_comm, used to track the interaction list contributions, is initialized with the number of nodes in the interaction list, for each node.

Recursive execution The algorithm starts at the leaf level and spawns one P2M task per leaf node. When a P2M task is completed it updates the node's parent cell in the scoreboard_up by subtracting one to the number of children. This indicates that one children dependency is satisfied. When the counter reaches zero, all dependencies are satisfied, and the task spawns the father's M2M task. The principle is the same with the M2M tasks. The downward phase follows the same scheme. Each task updates the scoreboard_down to subtract satisfied dependencies, and tasks are spawned progressively when the dependency counters reach zero.

Distributed version Since the dependencies are managed manually, this model should be compatible with distributed memory parallelism. We plan to use a task to check for incoming interaction list contributions and update the scoreboard_comm array. When all expected messages are arrived, it is possible to apply the M2L translation operations, and to update the corresponding cells in the scoreboard_down array.

This method has the advantage of not being impacted by task dependency management overhead. Moreover, dependencies are managed more finely, since there is no implicit task insertion ordering. Tasks are spawned as soon as possible, when the last dependency is satisfied. It can be used on distributed memory, since we also manually handle the incoming contribution dependencies. However, at the time of writing this thesis, implementation work of both versions is still ongoing. In the next section, we present the work that has been done to develop the task-based versions and summarize the remaining work.

8.4 Progress report

Since we work directly on a large industrial code, we progress step by step towards a fully asynchronous task-based version.

OpenMP loop scheduling

Motivation The current shared memory parallelization scheme is presented in section 8.3. Improving this parallelization scheme is necessary to progress towards tasks. Indeed, depending on the operator, the iteration space should be the number of nodes or the number of elements. This will allow for the creation of tasks that will handle the iterations. In order to propose a fair comparison with tasks, we made our best effort to provide an optimized loop-based version, with the most efficient scheduling strategy.

Methodology We proceed as follows:

- We suppress the current OpenMP parallelization scheme and replace it with a standard loop iterating on the nodes or elements.
- We replace the original $OMP PARALLEL DO with $OMP PARALLEL and $OMP DO, which enables us to allocate the thread private arrays inside the parallel region and before entering the parallel loop.
- Finally, we use macros to select an OpenMP scheduling strategy at compile time.

Figure 8.6 illustrates the modified loops, with pseudo Fortran code.

```
!$OMP PARALLEL PRIVATE(tab, ...)  ! start parallel region
  allocate(tab)          ! each thread allocates a private buffer

!$OMP DO PRIVATE(...)        ! start parallel loop
#ifdef AUTO
!$OMP SCHEDULE (auto)
#elif DYNAMIC
!$OMP SCHEDULE (dynamic)
#elif GUIDED
!$OMP SCHEDULE (guided)
#elif STATIC
!$OMP SCHEDULE (static)
#else
  ! no scheduler specified, default behavior
#endif

  ! parallel loop
  do ipaq=first,last
    ! do some computation
  enddo

  !$ OMP END DO NOWAIT       ! end parallel loop
  deallocate(tab)          ! each thread deallocates its private buffer
!$OMP END PARALLEL           ! end parallel region
```

FIGURE 8.6: Replacing SPECTRE's OpenMP manual loop parallelization with standard loop parallelism

Measure We launch an experiment on four Sandy Bridge nodes with the UAV test case. We use one MPI process with 16 threads per node. Figure 8.7 displays the execution time for the M2M, L2P and P2P kernels.
P2M, M2L and L2L graphs are not displayed; they are very similar to the M2M graph. For these operators, the change for standard OpenMP loop scheduling did not improve the parallel performance. The original, auto and guided strategies are equivalent. The L2P kernel is different from the other ones. It does not iterate on octree nodes but on elements, and it contains mutexes to protect writes in the unknown vector. In this case, the auto and guided strategies obtain slightly better results, with an improvement of order of 7 to 10%. Finally, the P2P kernel is the most impacted. In this case, static, auto and guided are equivalent, and improve the original version by 22 to 30%.

Conclusion We conclude that the guided and automatic schedulers give the best results, for all operators. They provide equivalent or better performance than the original version, and with less programming effort. As presented in 2.4.2, OpenMP loops are known to be well-suited for regular problems and less efficient for irregular ones. The results obtained with P2M, M2M, M2L and L2L operators suggest that, for these operators, the work is not so

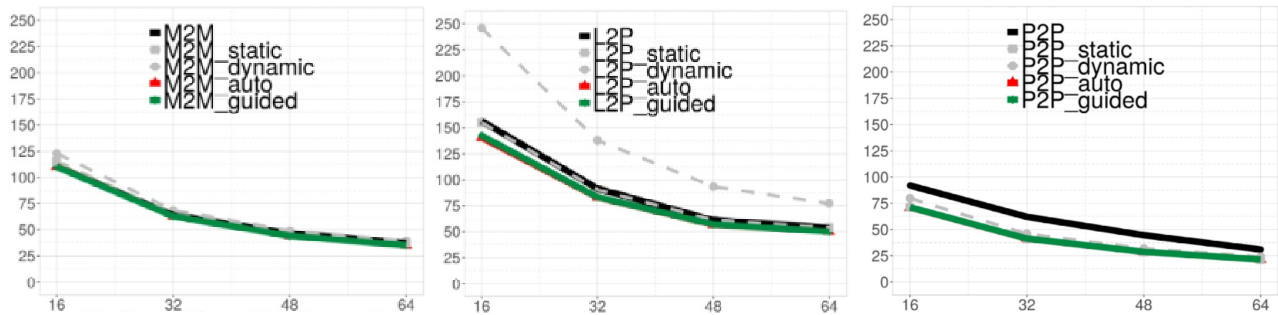


FIGURE 8.7: Testing the OpenMP scheduler strategies on M2M, L2P and P2P

much irregular. Indeed, the parallel loop iterates on nodes. And, for a given kernel, nodes of the same level have similar computation load. On the contrary, the P2P kernel computes the direct SPMV product and iterates on the matrix columns. The matrix is sparse and, depending on the test case, the work per column may not be balanced. This could explain that the P2P kernel benefits better from the guided scheduling strategy which is designed to load balance irregular work.

First OpenMP task implementation

Motivation The current implementation is still executed level by level. As a first step to introduce tasks, we keep this scheme and replace the parallel loops with tasks. The rationale is to limit the risk of introducing errors and to progress step by step. Moreover, from the bibliography, we are aware of two pitfalls. First, the OpenMP tasking runtime may generate overhead, and second, it is important to carefully choose the task granularity. Indeed, a too small granularity may induce scheduling overhead, and a too large granularity reduces parallel slackness. Thus, this first version allows us to verify that we have not introduced overhead when using tasks.

Methodology The parallel loops, as presented in figure 8.6, are well-suited to be turned into tasks. Indeed, it is possible to replace each iteration with a task. We also introduce a *chunksize* parameter per FMM kernel, which defines the number of iterations to gather inside a task. This parameter is adjustable at runtime through an environment variable. Thus, it allows us to easily test a large number of task sizes, and to identify the best granularity, for each FMM kernel. The resulting code is illustrated with pseudo Fortran in figure 8.8. In this example, we start by computing the number of chunks, based on the chunksize parameter. Then, in a parallel loop, a task is spawned for each chunk. Finally, the remaining iterations are handled in a last task.

Measure We run our experiment on one single Sandy Bridge node with the UAV use case. We varied the size of each MLFMM operator task to contain from one to ten nodes, or elements depending on the kernel. We measured and compared the corresponding execution time. For each operator, the execution time decreases along the chunk size until containing 1, or for some cases 2, iterations. With this version, we did not improve the operators efficiency, but did also not generate any additional overhead.

```
startIdx = firstBox              ! first iteration of the original OpenMP loop
lastIdx = lastBox                ! last iteration
nbIter = lastBox - firstBox + 1  ! total number of iterations
nbChunks = nbIter/chunkSize      ! number of chunks
tail = nbIter - (nbChunks*chunkSize)! remaining iterations

!$OMP PARALLEL PRIVATE(tab, ...)  ! start parallel region
  allocate(tab)          ! each thread allocates a private buffer

  !$OMP DO             ! loop over chunks
  do i=1, nbChunks
    !$OMP TASK           ! spawn a task by chunk
    call task_kernel(.., firstBox + (i-1)*chunkSize, firstBox + (i*chunkSize)-1, .. ) !
        address computations
    !$OMP END TASK
  !$OMP END DO


  !$OMP SINGLE           ! handle the remaining iterations
  if(tail > 0) then
    !$OMP TASK
    call task_kernel(.., firstBox + (nbChunks*chunkSize), firstBox + (nbChunks*chunkSize) +
        tail -1, .. )
    !$OMP END TASK
  !$OMP END SINGLE
  !$OMP TASKWAIT

  deallocate(tab)          ! each thread deallocates its private buffer
!$OMP END PARALLEL           ! end parallel region
```

FIGURE 8.8: First task-based version, limited to the scope of an octree level

Conclusion We draw two conclusions from this experiment. First, the MLFMM tasks can handle 1 or 2 iterations without generating overhead. This comes from the fact that compared to standard FMM, the CEM kernel operators are compute-intensive. Therefore, in the case of SPECTRE/MLFMM the tasks do not need to be coarsened. Second, the use of tasks did not generate any scheduling overhead, in comparison to standard loop parallelization. However, these observations stand for standard OpenMP tasks, the effect of scheduling OpenMP tasks with dependency clauses has not been analyzed yet.

Introducing the communications inside the tasks The version presented above has been used to support the task-based version of GASPI one-sided communications. We have added a threshold to parametrize the communication size and studied the impact of the message size. The implementation and the results are presented in the task-based communications section 7.6.

Preparing for a recursive version Some arrays are computed inside the original FMM kernels, and therefore re-computed for each iteration. It is necessary to pre-compute them, because inserting them into the tasks would add unnecessary computations of the same data. Therefore, we have added some fields in the octree structure to pre-compute and hold these arrays.

8.5 Conclusion

In this chapter, we focus on moving from a bulk synchronous parallel loop based model towards an asynchronous and hierarchical task-based model. We profile the current algorithm and identify a bottleneck in the shared memory parallelization which relies on parallel loops. The MLFMM operators do not scale over eight threads.

Therefore, we elaborate two task-based strategies, with different methods to handle the dependencies between tasks. The first one uses the the OpenMP 4.0 depend clause, to handle the dependencies between the tasks. Relying on this new OpenMP feature has the advantage of simplifying the programming. However, we identify some limitations. First, the depend clauses are not compatible with recursive tasks. Therefore it is necessary to spawn all the tasks in a same parallel region, which results in a very large pool of tasks challenging the scalability of the scheduler. Moreover, in a distributed version, some dependencies include incoming neighboring data, which the OpenMP scheduler was not designed to handle directly. Therefore, we add a task to handle the receptions, which will carry an output dependency on the incoming data.
The second version uses recursive tasks and handles the dependency manually through scoreboards. The recursive version benefits from smaller, and more local, pools of tasks. Each task updates the scoreboards and spawns a new task as soon as a dependency is satisfied. We also include a task to handle the incoming communications.

Finally, we summarize our early results in implementing these versions. We have demonstrated that the use of tasks, without dependency clauses, does not generate any overhead. We have also experimented different task sizes for all MLFMM operators, which lead us to the conclusion that the MLFMM operator tasks are compute intensive enough to have size as low as one or two octree nodes or elements.
At the time of writing this thesis, both versions are still under development.

Conclusion and Perspectives

The main goal of this thesis consists in proposing approaches for parallelizing complex industrial HPC applications. We targeted two main simulations in use in the Dassault Aviation aircraft design process: the AETHER CFD simulation code and SPECTRE, the CEM application.
First, we document and evaluate a code modernization methodology based on the use of proto-applications. This methodology is used to optimize AETHER, and compared to directly working on the industrial code, which is the process for SPECTRE.
The following and main contributions relate to the CEM application. We proceeded to an extensive profiling phase, during which we identified three bottlenecks: load-balancing, communications and shared memory parallelism. For each of these aspects, we proposed and documented a methodology to address the related issue. We developed two load balancing strategies and made them publicly available through an open-source library. Then, we implemented an alternative one-sided asynchronous communication scheme based on GASPI to replace standard blocking MPI communications, also available in our FMM-lib library. Finally, we pursued the shift towards a fully asynchronous algorithm by replacing the standard loop parallelization with asynchronous tasks. Load balancing and communication involve the distributed memory level, while tasks target the shared memory level. All in one, combining the different approaches, we propose a global strategy which tackles different parallelism layers.
We summarize and discuss our contributions hereafter.

Code Modernization Methodology

The first contribution of this thesis explored the methodology for modernizing and optimizing complex scientific simulation codes. We experimented two different approaches. For AETHER, we used the previously developed Mini-FEM *proto-application*. For SPECTRE, we integrated our optimization strategies *directly* into the code, without any stripped down version.

Mini-FEM is a proto-application stripped down version from the mesh deformer code developed by Dassault Aviation. It was used to develop the D&C library which targets the assembly step of a CFD method. Our contribution is twofold. First, we successfully integrated the D&C library into AETHER, another much larger CFD code, and confirmed the performance results. Second, we reiterated the process by extending the D&C library with an OpenMP tasks as an alternative to Cilk tasks, and confirmed the performance behavior on AETHER. Using the proto-application allows quick prototyping, developing and debugging of the optimization libraries, with portable results. However, we raised the problem of privatizing shared variables commonly used in Fortran programming and proposed a workaround. The proto-application being based on DEFMESH, differences with AETHER showed

up during the porting of the library.

In comparison, the higher confidentiality level of SPECTRE, did not allow us to build a shareable representative proto-application. Direct access to the final code allowed for complete knowledge and identification of potential pitfalls during the development phase. But manipulating complex industrial codes is costly. Compiling, executing, debugging and even profiling is difficult. As a result, the optimization process is much slower. Moreover, developing directly within the target application limits the reuse of the proposed strategy. To mitigate this, we extracted the load balancing and communication strategies into the FMM-lib library. However, since OpenMP supports tasking in the Fortran language, and because of time constraints, the shared memory optimization developments took place directly within the CEM code. Using a library would necessitate to extract and wrap each task code, which is not only error prone but also creates a large number of function calls. Directly working on a single target allows for a tailored solution, but at the price of limited reuse.

We have validated the effectiveness of using proto-applications to support code modernization efforts. Obtaining reproducible performance results on large, complex and even legacy industrial software validated the process. It has also confirmed the representativeness of the Mini-FEM proto-application. In our view, proto-applications are a great opportunity to develop and validate code optimization. Moreover, providing a proto-application to the community permits to share the problems and to benefit from additional support. However, when the application is too sensitive to be shared, developing the optimization strategies within libraries provides reuse opportunities for other similar applications, and with community engagement, the industrial user can still benefit from outside improvements.

Load Balancing

The second contribution of this thesis targets the load balancing of the MLFMM algorithm as implemented in SPECTRE. We started by analyzing the current load balancing strategy. Even if the existing algorithm provides good balance for current use cases and architectures used at Dassault Aviation, some weak points were identified. Indeed, some empirical choices were made. They relied on the current industrial usage and may have to be revisited for future larger use cases or clusters.

We proposed two alternative load-balancing strategies. To favor reuse, the strategies were included in an external library. Since they handle sets of particles, they are not limited to MLFMM methods. Moreover, for applications in which particles move, efficient and scalable particle exchange functions are provided using MPI and one-sided asynchronous GASPI.
The first strategy, based on histograms, improved standard ORB partitioning with the use of multi-sections, and the second one was a recursive Morton version based on a tree traversal. We demonstrated the effect of modifying the partitions on the communication matrices and on the overall execution time. The histogram version has the advantage of providing very well balanced distributions, but applied to the UAV test case, the cuts generated an important communication volume resulting in an overall execution time slowdown. With the Morton based version we experimented the rotation of the Z-shaped Morton pattern in order to analyze the

effect of the first cut. We obtained a limited 5 to 10% reduction in execution time compared with the original version.

Future work consists in elaborating a strategy which takes into account the communication cost. The rationale is to balance the computational work while minimizing communications. It will be used in conjunction with the asynchronous GASPI communication scheme. The key idea is to translate the octree into a weighted graph, with nodes representing the computations and edges reflecting the remaining non-overlapped communications. A graph partitioning tool with a multi-constraints strategy can then be used to compute balanced partitions while minimizing the communication cost.

Communications

The third contribution focuses on modernizing the communication scheme of SPECTRE's MLFMM. We first conducted an extensive profiling phase to characterize the communication's scalability and identify the bottlenecks. This led us to target the far field exchanges of the MLFMM algorithm. The communications are organized in bulk synchronous manner: the iterative process alternates computation and communication phases.

The first phase aimed at shifting the current communications from two-sided blocking and synchronous towards one-sided and asynchronous. Since the MPI asynchronous communications are known to suffer from slow progression, we conducted a comparative study between the MPI and GASPI libraries. We developed different versions relying on asynchronous MPI and one-sided GASPI. In order to make fair comparisons, we did not modify the communication scheme. Experiments showed a clear advantage for the GASPI implementations, communication time being almost reduced by half.
We introduced asynchronism and communication overlapping following a three-step methodology. First, we replaced the blocking two-sided MPI communications by one-sided asynchronous GASPI messages. Then, we introduced overlapping opportunities. We split the large communication phase, into small sending and receiving operations spread along the MLFMM algorithm. The rationale is to send the data as soon as possible, and to check for receptions as late as possible, only when the data is needed. Finally, we broke down a collective communication into point-to-point one-sided and asynchronous messages. All these modifications allowed for a significant gain in performance: the communication time for the far field exchanges has been divided by 6.

The second phase consisted in setting the foundations for task-based communications. We proceeded in two steps. In the first one, we multithreaded the communications through loop parallelization. Instead of having one thread writing all the available data into the recipient's memory, all the threads participate by taking care of a subpart of the communication. In a similar way, multiple threads can handle the incoming messages and unpack the received data. The second step consisted in moving towards task-handled communications. Standard OpenMP loop parallelization is replaced by OpenMP tasks, and communications are inserted inside these tasks. Since the task granularity directly impacts the size and the number of messages, we have chosen to introduce a size threshold. Threads execute tasks and write data into a buffer, which is sent only when the size threshold is reached. Finally, by varying the threshold, we empirically determined the optimal message size. GASPI-based task-level functions

are ready to be used in the future fully asynchronous task-based version of the MLFMM.

We demonstrated that, without modifying the communication scheme, replacing the standard MPI library by the GASPI library can lead to significant reduction of the communication time. However, using a low-level PGAS API requires more programming effort than MPI. It is necessary to manually declare shared memory segments, to handle multiple offsets for writing and reading in local and remote memories, to manage the communication queue by ensuring that there is enough place to host another message, and to manage the notification mechanism. The GASPI learning curve is stiff, but the performance achievements worth it. This is calling for library development specific to a given class of algorithms. With DC-lib and FMM-lib we are proposing solutions for a large set of similar HPC applications.

Tasks

The last contribution relates to the shared memory parallelization. The current shared memory level parallelization relies on OpenMP loop parallelization. Profiling of the shared memory revealed a bottleneck resulting in a lack of scalability. To take benefit from current and future architectures, which contain more and more cores, we propose a dynamic and hierarchical version based on tasks. We have elaborated two task-based versions.

The first one relies on OpenMP 4.0 tasks and uses the *depend* clause to express fine dependencies. This method has the advantage of being easy to develop. However, it is not compatible with recursivity, and therefore imposes to spawn all tasks at the same level, which results in a very large pool of tasks to be scheduled.
The second version is recursive and handles the dependencies manually through scoreboards. With this method, tasks are spawned as soon as the dependencies are satisfied, and pools of tasks are smaller and more local.

For both versions, in order to be compatible with distributed parallelism, and to handle dependencies on incoming data, we propose to include tasks in charge of managing the received messages. In the first case, it will carry an output dependency which will therefore be integrated by the OpenMP scheduler. And in the second case, it will update a scoreboard and spawn the tasks whose dependencies are satisfied by the received data.

Our early results showed no overhead while using OpenMP tasks, and allowed for determining the ideal task size. Unlike experiences reported in the literature, we can use small tasks sized to one or two loop iterations. This difference comes from the fact that, compared to standard FMM, the MLFMM CEM kernels are highly compute intensive. The two proposed strategies will be integrated with the task-based GASPI communications, already developed. They are currently under development, and should allow obtaining the targeted fully asynchronous version.

Other identified optimization opportunities

During this thesis other optimization opportunities have arisen. First, the near field computations could be used to overlap partially or completely the remaining communication time. Indeed, it is possible to test if some message

has arrived, and if not, some near field computations can be executed during the waiting time. Similarly, in simulations involving multi-materials, each material is independent. Therefore, they could be computed in parallel. The L2P operator does not scale, mainly because it uses a mutex to protect the accesses to the unknown vector. A coloring strategy could be used to avoid concurrent writes. Alternatively, OpenMP 5.0 provides the ability to handle a reduction between tasks. The impact of the data layout was not studied. Reorganizing the data to favor locality may optimize the cache reuse, and improve the vectorization. The automatic vectorization by the compiler has not been studied. A first attempt was performed by using MAQAO to identify the hotspots and analyze the corresponding vectorization reports. Finally, at the communication level, the proportion of the unoptimized time spent to communicate the unknowns has grown mechanically and could benefit from optimization with recent GASPI collective communications.

Overall Conclusion

Modernizing and optimizing large industrial simulation applications is a tedious and complex task. They are often very large, mixing different programming languages and styles, mainly Fortran, and complex to analyze. Even diagnosis and measurement are non-trivial. Moreover, the industrial context and confidentiality restrictions may impose more or less drastic constraints on the code manipulation, including running on in-house clusters.

Moreover, optimization remains execution context dependent. Indeed, the efficiency of the MPI library depends on the implementation, the configuration, the cluster topology, the network and even the choice of an algorithm to achieve a communication. The same is true for the OpenMP library, which is compiler dependent. During this thesis, we have experimented different unexpected behaviors. Sometimes, introducing asynchronous communications in one part of the code generated side-effects on other parts of the code, which would even be different on an other machine. Similarly, updating the compiler impacted the OpenMP tasks behavior. Finally, combining these different runtimes may lead to possible interactions between them. These issues question directly the validity domain of the optimization found in this thesis and in the literature in general, especially when based on a benchmark or synthetic example. The experience also highlights one of the potential downsides of using proto-applications. A careful attention must be given to the conceptual justification of the observed acceleration.

Despite these difficulties, we managed to obtain significant results on the modernization of the MLFMM of Dassault Aviation CEM simulation code. We demonstrated the advantage of using a proto-application for code modernization, and how to mitigate with libraries the impossibility to outsource some applications. We replaced the bulk synchronous MPI communication scheme by asynchronous one-sided GASPI communications, and divided the communication time by 6. We explored the load balancing strategy, elaborating two alternatives to the current algorithm, with a 5 to 10% gain, depending on the test case and the number of domains, on the solver execution time. Finally, we laid the ground for a fully asynchronous FMM implementation, efficiently mixing shared and distributed memory levels to maximize the available concurrency and locality.

With the current trend of heterogeneous computations, industrial simulations still have to evolve. This will bring new problematics, in particular for algorithms like FMM, with many dependencies, communications, and sensible task granularity depending on the underlying physics. Therefore, the proto-application approach seems well-suited to support future explorations. Asynchronism, concurrency and locality are key concepts to take into account to benefit from future architectures.

Annex 1 - Difficulties encountered with MPI Runtimes

In order to study the behavior of our GASPI asynchronous version, on much larger clusters we intended some runs on the INTI Sandy Bridge Runs.

OpenMPI runs

We compared five versions: Ref, and corrected Ref_IncLevcom, which includes the communication level modification from global into local point-to-point exchanges, the MPI non-blocking, and the two GASPI bulk and asynchronous versions. For the sake of simplicity, these last three versions contain the same modification but the IncLevcom suffix is not used anymore.
Complete FMM execution time, see figure 8.9, shows that the asynchronous GASPI and the non-blocking MPI versions decrease until 16 nodes, but then start to grow again.
Figure 8.10 displays the global communication time spent in the far field exchanges. First of all, one can see on the ordinate axis that the communications are much slower than on the Dassault Aviation's equivalent Sandy Bridge cluster. The MPI non-blocking version has not the expected behavior. With growing number of nodes, it becomes significantly slower than the reference one. GASPI Versions also behave differently, even if they scale correctly, the gap between the two versions is not comparable to what we experienced previously.
More surprisingly, the global allreduce communications, see figure 8.11, is significantly impacted. As introduced earlier in figure 4.2 this communication takes place after the FMM algorithm, and particularly, it involves different data: the vector of unknowns. None of the modifications we applied did impact these data.

Intel MPI runs

On the basis of these observations, we made another set of experiments, replacing OpenMPI by IntelMPI. Not only are the results also inconclusive, but they are different. We executed only 40 iterations, which is sufficient to examine the communications. Left side graph in figure 8.12, displays the complete MLFMM execution time, and the asynchronous GASPI version collapses totally. Right side graph of the same figure, shows that this comes from the MPI_Barrier, which takes place between the two calls to the MLFMM subroutine. The Intel MPI Library provides tuning methodologies to enable the developer to change the global communication algorithms [22]. Nine versions of the MPI_Barrier are available and listed in Table 8.1. They are easy to select by setting the environment variable MPI_ADJUST_BARRIER. We tested them all, and did not obtain any valuable improvement.

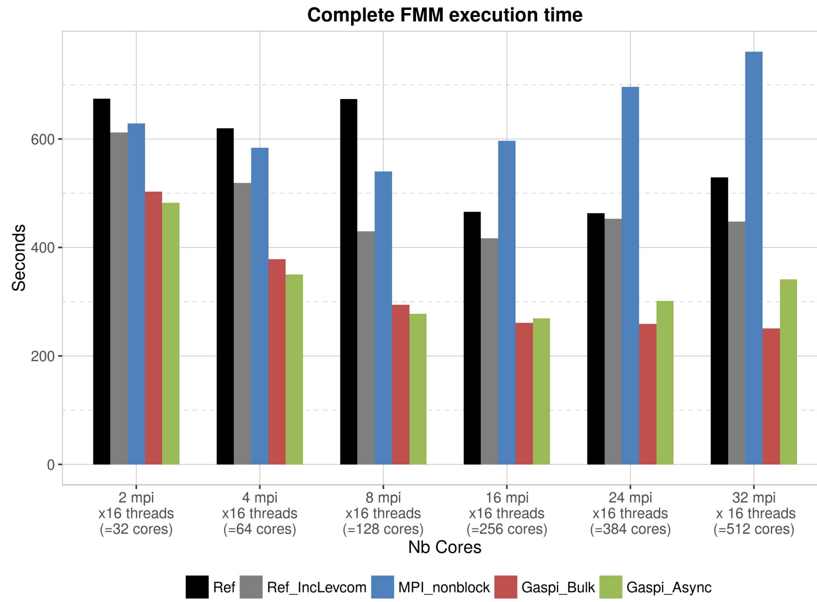


FIGURE 8.9: Complete FMM execution time on UAV test case, INTI Sandy Bridge cluster, OpenMPI.

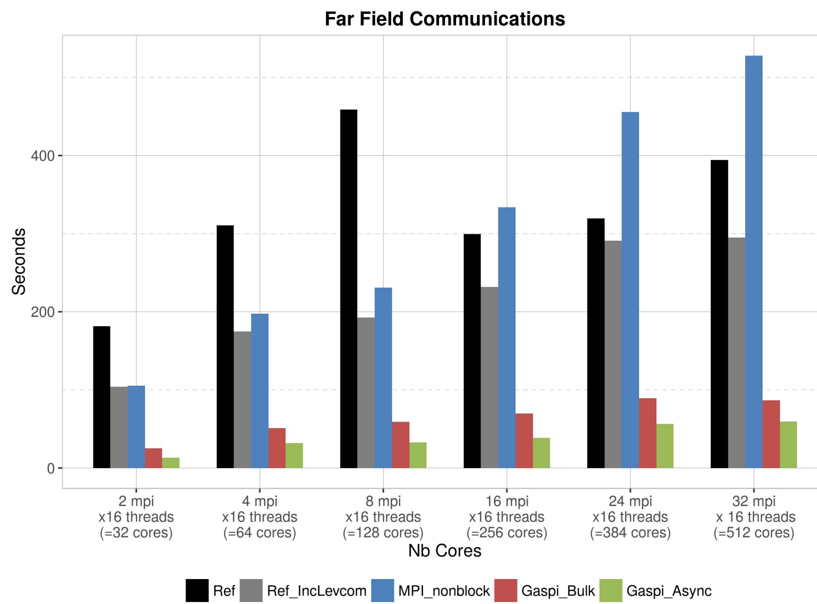


FIGURE 8.10: Far Field Communication time on UAV test case, INTI Sandy Bridge cluster, OpenMPI.

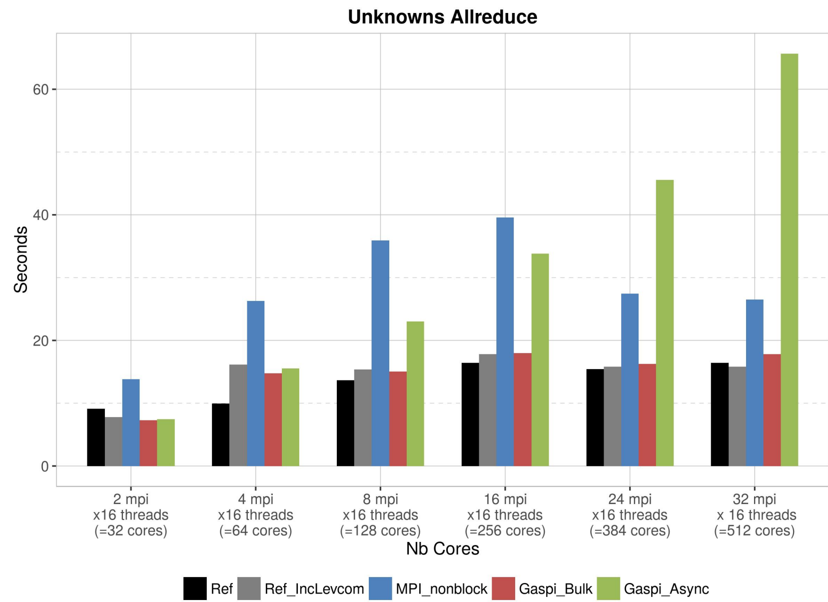


FIGURE 8.11: Side effects on global communications of the unknowns vector, OpenMPI.

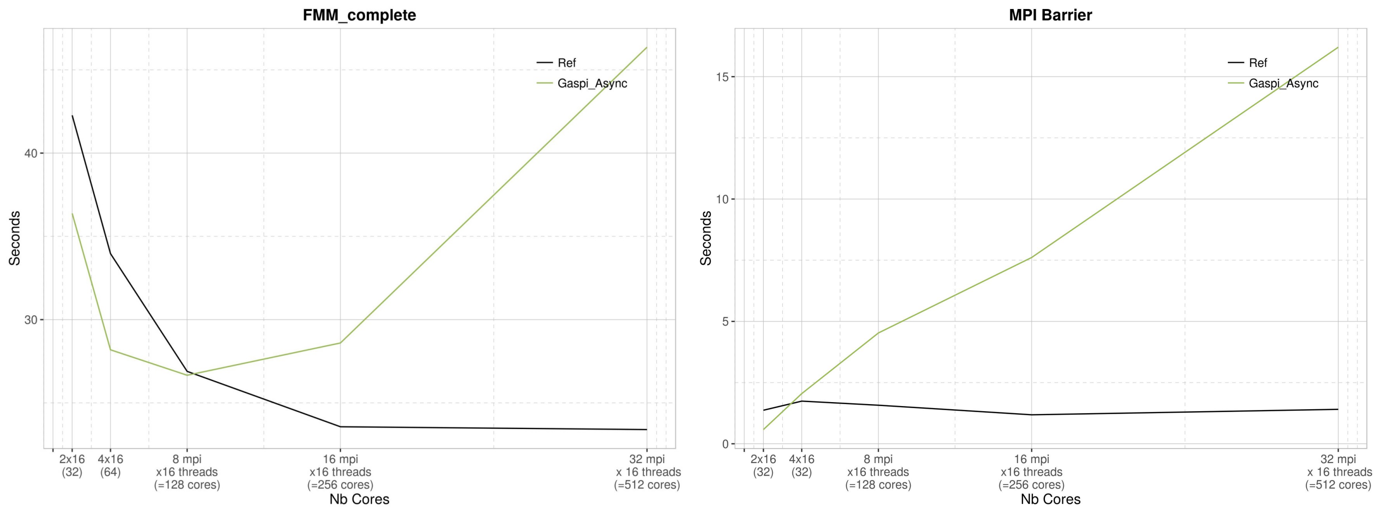


FIGURE 8.12: Left: Complete FMM execution time, Right: MPI_Barrier. Both on UAV test case, INTI Sandy Bridge cluster, Intel MPI.

Environment Variable	Collective Operation	Algorithms
	MPI_Barrier	1. Dissemination 2. Recursive doubling 3. Topology aware dissemination 4. Topology aware recursive doubling 5. Binominal gather + scatter 6. Topology aware binominal gather + scatter 7. Topology aware SHM-based flat 8. Topology aware SHM-based Knomial 9. Topology aware SHM-based Knary

TABLE 8.1: MPI Barrier algorithms list, available in the Intel MPI Library

For the time being, we have no explanation for these observations, and cannot run any reliable performance analysis on these platforms.

Annex 2 - Summary

For many years, the stability of the architecture paradigm has facilitated the performance portability of large HPC codes from one generation of supercomputers to another. The announced breakdown of the Moore's Law, which rules the progress of microprocessor engraving, ends this model and requires new efforts on the software's side.
Code modernization, based on an algorithmic which is well adapted to the future systems, is mandatory. This modernization is based on well-known principles as the computation concurrency, or degree of parallelism, and the data locality. However, the implementation of these principles in large industrial applications, which often are the result of years of development efforts, turns out to be way more difficult than expected. Interesting methods shifting towards an asynchronous paradigm, while preserving load balancing and locality are demonstrated in the academic field. Among them, very few large scale applications in production today are exploiting asynchronous parallel tasks and asynchronous multithreaded communications to take full advantage of the available concurrency.

In this PhD thesis, we explore methods to modernize large industrial simulation codes. To support our study, we work with two main simulation applications in use in the Dassault Aviation aircraft design process: the AETHER CFD (Computational Fluid Dynamics) simulation code and SPECTRE, the CEM (Computational Electro-Magnetics) application. We experiment two different approaches to bring state-of-the-art progress in HPC into legacy applications.

Our contributions are twofold :
On the one hand, we explore a **methodology** of software modernization based on the concept of proto-applications and compare it with the direct approach. We experiment two different approaches to bring state-of-the-art progress in HPC into legacy applications. The first one is the continuation of the work of L. Thébault based on the concept of proto-applications [117]. The critical part to optimize is extracted as a stand alone benchmark that can be shared with the HPC community. In this case, it is representative of the assembly step of a Finite Element Method (FEM) used in CFD codes. This proto-application was then used to support the development of an optimization library, called D&C library. The first contribution consists in porting back and evaluating this optimization in another more complex production code, AETHER, and to document the process. Then the process is reused to extend the D&C library with OpenMP tasks, to propose an alternative to the current implementation based on Cilk tasks. We validated the methodology by demonstrating comparable results on the proto-application and on the target industrial application. The results were published in the ICCS 2015 ALCHEMY workshop [100].

On the other hand, we focus on the identification of the main challenges for the architecture, the programming models and the applications. This second, and main contribution, aims at optimizing the industrial application for CEM simulations. More specifically, we target the MLFMM (Multi-Level Fast Multipole Method) used in SPECTRE, which is developed with similar tools, environment and programming rules than AETHER. For the

sake of comparison, and also because of confidentiality constraints, the optimization work is done directly on the application, without any representative proto-application. Based on measurement and literature, we identify three separate aspects to optimize.

Load balancing the work among the processor nodes directly impacts the global performance. A poor load balancing leads to idle time and bounds the overall performance to the slowest process' execution time. We propose two load balancing methodologies. The first one is based on a Morton space-filling curve and the second on Histograms. We paid great attention on developing a lightweight optimized algorithm so that it can be integrated with minimal overhead. Our load balancing strategies allow for a 5% improvement on communication time.

Nonetheless, load balancing efforts may be harmed by a poor synchronous communication model, therefore we studied and optimized the **communications**. To benefit from today's architectures, it is necessary to evolve towards asynchronous, thread-based and local exchanges. In SPECTRE, the communications are organized in a bulk synchronous manner: the iterative process alternates computation and communication phases. Knowing from literature and past experiences that MPI asynchronous communications lack progression, we tried an alternative PGAS-based library called GASPI, and demonstrated its efficiency. Then, we gradually desynchronize the communication scheme, by breaking down large communications into smaller and more local exchanges. We propose different versions progressing from the two-sided blocking synchronous model towards a fully asynchronous task-handled model. These optimizations allow us to divide the communication time by a factor of 6. Along with load balancing, this work was presented in a short paper at Europar 2017 COLOC workshop [99] and published in ICCS 2019 ALCHEMY workshop [98]. To demonstrate our load balancing and communications improvement for MLFMM, we are releasing *FMM-lib* library [4] under LGPL-3 license.

Finally, the last aspect relates to the shared memory parallelization with the use of **tasks**. In irregular applications, the iterations of a same loop carry very different amounts of computational work. Therefore, loop-based parallelization can lead to load imbalance, resulting in idle threads. To take benefit from current and future architectures, which contain more and more cores, we propose a dynamic and hierarchical version based on tasks. We explore two versions that differ in the way of controlling the task dependencies. The first one relies on OpenMP 4.5 tasks, which provides a specific clause to express data dependencies and then schedules the tasks depending on their readiness. The second is a manually handled recursive version, using scoreboards to track the dependencies and spawn the tasks. The task-level GASPI communications are included inside the tasks. The work on task-based parallelization is ongoing. We present the progress achieved so far and give the guidelines for the future work.

Annexe 3 - Résumé

Durant de longues années, la stabilité dans le paradigme d'architecture a facilité la portabilité de performance des grands codes en Calcul de Haute Performance d'une génération à l'autre de supercalculateurs. L'effondrement programmé de la loi de Moore - qui règle les progrès en gravure des micro-processeurs - bouscule ce modèle et requiert un effort nouveau du côté logiciel.
Une modernisation des codes basée sur une algorithmique adaptée aux futurs systèmes est ainsi nécessaire. Cette modernisation repose sur des principes de base connus tels que la concurrence des calculs et la localité des données. Cependant, la mise en œuvre de ces principes dans le cadre d'applications réelles en milieu industriel, lesquelles applications sont souvent le fruit d'années d'efforts de développement, s'avère bien plus compliquée que ne le laissait prévoir leur simplicité apparente. Des méthodes intéressantes sont étudiées dans le domaine académique: celles-ci privilégient le paradigme asynchrone tout en préservant l'équilibrage de charge et la localité des données. Cependant, parmi les grandes applications utilisées en production, celles exploitant les tâches parallèles, multithreadées et asynchrones afin de profiter pleinement du parallélisme exposé par les architectures se font rares.

Dans cette thèse, nous explorons des méthodologies de modernisation de grands codes industriels de simulation. Nous utilisons les deux principales applications de simulation, utilisée dans le processus de conception des avions de Dassault Aviation: AETHER, application de simulation en mécanique des fluides, et SPECTRE, application de calcul de champs électromagnétiques.

Notre contribution est double :
D'une part, nous explorons une **méthodologie de modernisation de codes** basée sur l'utilisation de proto-applications et la confrontons à une approche directe. Nous expérimentons deux différentes approches pour moderniser les applications HPC.
La première consiste à utiliser le concept de proto-applications, et s'inscrit dans la continuation du travail de L. Thébault [117]. Une mini-application est créee à partir de la la partie critique de l'application principale. Elle peut ainsi être analysée séparément et partagée avec la communauté HPC. La mini-application Mini-FEM est représentative de l'étape d'assemblage d'une méthode des éléments finis utilisée dans les codes de mécanique des fluides. Elle a été utilisée pour supporter le développement d'une librairie d'optimisation, appelée D&C library. La première contricution de cette thèse consiste à porter cette librairie dans une autre application, plus complexe et utilisée en production, AETHER. Nous évaluons l'optimisation obtenue et documentons le processus. Ensuite, nous réitérons le processus afin d'étendre la librairie D&C avec l'usage de tâches OpenMP comme alternative à l'implémentation courante basée sur Cilk. Nous validons cette méthodologie en obtenant des résultats comparables sur la proto-application et sur l'application industrielle ciblée. Les résultats on été publiés au workshop ALCHEMY

dans la conférence ICCS 2015 [100].

D'autre part, nous nous concentrons sur les principaux défis de la modernisation de code, au niveau de l'architecture, des modèles de programmations et des application. Dans cette seconde et principale contribution, notre objectif est d'optimiser une application industrielle dédiée à la simulation de champs électro-magnétiques. Plus précisément, nous travaillons sur la partie MFLFMM (Multi-Level Fast Multipole Method), utilisée dans l'application SPECTRE. Cette application a été dévelopée dans des conditions similaires que l'application AETHER, dans le même environnement, avec les mêmes outils et les mêmes règles de programmation. Pour des raisons de confidentialité, les optimisations sont dévéloppées directement sur la base de l'application industrielle sans l'usage d'une proto-application représentative. Cela nous permet de comparer et de documenter les deux méthodes de travail. En nous référant à la littérature, nous identifions trois principaux aspects à optimiser

L'équilibrage de charge à travers les noeuds de processeurs impacte la performance globale. Un mauvais équilibrage de charge entraîne des temps d'attente, limitant la performance globale au temps d'exécution du processeur le plus lent. Nous proposons deux stratégies d'équilibrage de charge. La première est basée sur une courbe de remplissage de l'espace de type Morton et la seconde utilise des histogrammes. Nous portons une attention particulière au développement d'algorithmes optimisés qui puissent être intégrés avec un surcoût minimal. Nos strategies d'équilibrage de charge permettent de réduire les temps de communication de 5%.
L'équilibrage de charge peut néanmoins voir son efficacité réduite par l'utilisation d'un mauvais schéma de **communication**. Pour exploiter les architectures actuelles il est nécessaire de désynchroniser les communications au moyen de tâches séparées et de privilégier les échanges de données locaux. Dans le code de simulation SPECTRE, les phases de calculs et de communications sont alternées selon un schéma dit « bulk synchronous ». Pour relâcher les synchronisations induites par ce schéma il est nécessaire d'utiliser les primitives de communication asynchrones de la bibliothèques MPI. Cependant, ces primitives ne possède pas d'indicateurs de progression et nous envisageons donc l'utilisation d'une autre bibliothèque nommée GASPI basée sur l'approche PGAS et montrons son intérêt en terme de performance. Nous désynchronisons ensuite plus finement les communications en les découpant en communications plus petites et plus locales. Nous proposons différentes implantations allant d'un modèle synchrone jusqu'à un modèle complètement asynchrone géré par des tâches pour montrer l'intérêt de l'asynchronisme. La version la plus asynchrone nous permet de diviser le temps de communication par un facteur 6. Ce travail a fait l'objet de publications au workshop COLOC [99] de la conférence Euro-Par 2017 et au workshop ALCHEMY de la conférence ICCS 2019 [98]. Les implantations de l'équilibrage de charge et des communications asynchrones pour la MLFMM sont intégrées dans la bibliothèque *FMM-lib* publiée sous licence LGPL-3.

Enfin, les améliorations de la parallélisation en mémoire partagée reposent sur l'utilisation de **tâches**. Dans le contexte des applications irrégulières, les itérations d'une même boucle peuvent représenter des quantités de calcul très variables. Une parallélisation sur les itérations d'une telle boucle peut conduire à un mauvais équilibrage et ainsi à des temps d'attente inutiles pour certaines tâches. Pour occuper au mieux les processeurs actuels qui contiennent de plus en plus de coeurs, nous proposons une approche dynamique et hiérarchique pour la gestion des tâches. Nous considérons deux implantations qui diffèrent dans leur manière de contrôler les dépendances entre tâches. La première repose sur l'utilisation de la bibliothèque OpenMP 4.5 qui permet d'exprimer les dépendances de données et l'ordonnancement des tâches à l'aide de clauses spécifiques. La seconde repose sur une approche récursive

explicite et sur l'utilisation d'une structure de données ad-hoc pour exprimer les dépendances de données et générer les tâches. Les communications GASPI sont directement incluses dans ces tâches. Le travail sur la parallélisation à base de tâches est en cours de développement, nous présentons l'état d'avancement de ces travaux et donnons les consignes à suivre pour le poursuivre.

List of Figures

1 Aerodynamic (top) and stealth - in the electromagnetic (bottom) and infra-red (middle) ranges - analyses in the multi-disciplinary design of the Neuron UCAV demonstrator . . . 1

2.1 Domain decomposition . . . 12
2.2 Basic MPI communication based on blocking send and receive calls . . . 14
2.3 MPI collective communications . . . 14
2.4 Overlapping communication with computation . . . 16
2.5 Blocking and non-blocking two-sided communications . . . 16
2.6 Eager and Rendezvous Protocols . . . 18
2.7 MPI One-sided Communications . . . 19
2.8 Left: MPI_Win_Fence, Right: PSCW . . . 19
2.9 The PGAS model . . . 21
2.10 Communication example with Gaspi . . . 23
2.11 GASPI asynchronous and overlapped communication example . . . 23
2.12 Bulk Synchronous parallelism. . . . 26
2.13 Static vs Dynamic scheduling. . . . 26
2.14 OpenMP loop scheduling. . . . 27
2.15 From parallel sections to tasks . . . 29
2.16 OpenMP Producer/Consumer model . . . 29
2.17 Recursive tree traversal with Intel Cilk Plus tasks . . . 31
2.18 Intel Cilk Puls work-stealing protocol . . . 33

3.1 Near Field and Far Field characterization in 2D. . . . 37
3.2 U and V lists of a node B in a two-dimensional space and a uniform quadtree. U-list contains the near field and V-list contains the far field interaction list. . . . 38
3.3 Adaptative quadtree covering a non uniform distribution. . . . 38
3.4 FMM two-phase tree traversal operators . . . 39
3.5 M2M, M2L and L2L operators . . . 40
3.6 ORB partitioning of a two-dimensional space . . . 46
3.7 Left: Four iterations of Morton. Right: Six iterations of Hilbert . . . 47

4.1 Left: Hierarchically 3D partitioning, Right: 3-dimensional octree. . . . 54
4.2 MLFMM communications . . . 55
4.3 Generic metallic UAV use case . . . 56

4.4 F7X aircraft use case . . . 56
4.5 Flow around a business jet aircraft with complex shock-boundary layer interaction, as captured through a section in the CFD unstructured mesh . . . 57
4.6 Illustrations of the assembly step: building the matrix. . . . 58
4.7 Halo exchange . . . 58
4.8 The EIB fuel tank position optimization use case. . . . 60
4.9 D&C recursive traversal and corresponding pseudo-code. The left and right parts can be treated in parallel. The edges on the cut form a separator treated after these left and right parts. . . . 61
4.10 The elements in left (L), right (R) and separator (S) parts are reordered. . . . 61
4.11 Last level cache (L3) cache misses measurement on a 12 cores two sockets Ivy Bridge node . . . 62

5.1 Optimization scenario: From the development of the proto-application to the integration in the final application . . . 66
5.2 Comparison between CilkView scalability reports. . . . 67
5.3 Optimization scenario: Extension of the D&C library with OpenMP tasks. . . . 69
5.4 Proto-application: Intra-node scalabity on the EIB use-case . . . 70
5.5 Proto-application: Strong scaling on Xeon Phi KNC . . . 70
5.6 AETHER: Strong scaling with F7X mesh on 1024 Sandy Bridge cores. . . . 71

6.1 Original load balancing algorithm . . . 74
6.2 Original load balancing algorithm applied to the UAV test-case on 4 and 64 processors. On the Left side: visualization of the points hosted by each processor. Neighboring points are represented in red, they require communications. On the right side: the communication matrix generated for 64 processors. . . . 75
6.3 FMM-viz, our OpenGL-based visualization tool screenshot . . . 77
6.4 Optimization of the computation of the separators. . . . 78
6.5 OpenGL visualization of histogram load balancing on UAV test-case Top: 4 domains, Bottom: 8 domains . . . 80
6.6 Side-effect of the Histogram load balancing algorithm: Communication degradation. Left: Communication volume in bytes for 2 to 64 domains, Right: Communication matrix with 64 domains. . . . 81
6.7 Morton ordering on a 2D dimensional space represented with a quadtree . . . 81
6.8 Morton recursion on a 3D space with cutting off unnecessary refinements. . . . 82
6.9 Axis order influence on communication matrices with 64 ranks. First column: X-axis first: XYZ and XZY - Second column: Y-axis first: YXZ and YZX, - Third column: Z-axis first : ZXY and ZYX. . . 83
6.10 Execution time spent in the solver, UAV test case, with various Morton space filling curves. Base 100 represents the original execution time without modifying the original load balancing . . . 84

7.1 FMM strong scalability analysis with the UAV test case . . . 89
7.2 MPI communication time ratio on total FMM, with larger F7X test case. . . . 89
7.3 MPI communication volume analysis. Left: quantity of data transferred - Right: corresponding elapsed time . . . 90
7.4 Bulk synchronous parallel model applied to the two phase FMM octree traversal. . . . 91

7.5 Original Mpi comunication scheme 93
7.6 New Mpi comunication scheme 93
7.7 Pseudo code illustrating the five non-blocking MPI versions 95
7.8 Expected overlapping opportunities while pushing or pulling the messages 96
7.9 Far Fields total communication time. Comparison between the original blocking MPI and five non-blocking MPI versions 97
7.10 GASPI Bulk version, from sender and receiver sides. 99
7.11 Introducing asynchronism and overlapping opportunities in the communication scheme 100
7.12 Far Field terms array organization 100
7.13 Far Fields total communication time, UAV test case. Comparison between MPI and GASPI. 101
7.14 Far Fields communication time, larger F7X test case. Top of the tree Allreduce replaced with point-to-point communications, for all versions. 102
7.15 Pseudo code illustrating the two multithreaded GASPI versions 104
7.16 Message size variation impact experiment with GASPI. 106
7.17 New Communication time ratio on total FMM, F7X test case. With breakdown between Far Fields and Unknowns 107

8.1 Shared memory scalability with the UAV test case. 112
8.2 Hotspot identification with MAQAO 113
8.3 Asynchronous task-based execution of the FMM algorithm 114
8.4 SPECTRE's OpenMP Loop Parallelization 115
8.5 Creating the OpenMP depend tasks 117
8.6 Replacing SPECTRE's OpenMP manual loop parallelization with standard loop parallelism 119
8.7 Testing the OpenMP scheduler strategies on M2M, L2P and P2P 120
8.8 First task-based version, limited to the scope of an octree level 121
8.9 Complete FMM execution time on UAV test case, INTI Sandy Bridge cluster, OpenMPI. 130
8.10 Far Field Communication time on UAV test case, INTI Sandy Bridge cluster, OpenMPI. 130
8.11 Side effects on global communications of the unknowns vector, OpenMPI. 131
8.12 Left: Complete FMM execution time, Right: MPI_Barrier. Both on UAV test case, INTI Sandy Bridge cluster, Intel MPI. 131

List of Tables

1.1 Clusters used 10

2.1 MPI Broadcast algorithms list, available in the Intel MPI Library 15

6.1 Load balancing of the UAV test-case among 8 ranks: number of octree nodes per rank and per level 76

8.1 MPI Barrier algorithms list, available in the Intel MPI Library 132

Bibliography

[1] Chapel documentation. https://chapel-lang.org/docs/index.html.

[2] Cube. http://scalasca.org/software/cube-4.x/download.html.

[3] Exafmm. https://github.com/exafmm/exafmm.

[4] Fmm-lib. https://github.com/EXAPARS/FMM-lib.

[5] Gaspi project. http://www.gaspi.de/gaspi/.

[6] Gaspi project. https://cw.infinibandta.org/document/dl/7859.

[7] Intel® mpi library. https://software.intel.com/en-us/mpi-library.

[8] Maqao. https://www.maqao.org.

[9] Massivethreads library. http://code.google.com/p/massivethreads/.

[10] Metis - serial graph partitioning and fill-reducing matrix ordering. http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.

[11] Minifmm. https://github.com/uob-hpc/minifmm.

[12] Mpi 4.0. https://www.mpi-forum.org/mpi-40/.

[13] The mpi forum. https://www.mpi-forum.org/.

[14] nsimd. https://github.com/agenium-scale/nsimd.

[15] Opengl. https://opengl.org/.

[16] Openmp5. https://www.openmp.org/about/members/.

[17] Parmetis. http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.

[18] Pgas languages. http://pgas.org/index.php?option=com_weblinks&view=category&id=38&Itemid=67.

[19] The scalfmm library. http://people.bordeaux.inria.fr/coulaud/Softwares/scalFMM.html.

[20] Score-p. https://www.vi-hps.org/projects/score-p/.

[21] Top 10 algorithm. https://archive.siam.org/pdf/news/637.pdf.

[22] Tuning the intel® mpi library: Basic techniques. https://software.intel.com/en-us/articles/tuning-the-intel-mpi-library-basic-techniques.

[23] Workshop em isae 2016. https://websites.isae-supaero.fr/workshop-em-isae-2016/accueil, 2016.

[24] Workshop em isae 2018. https://websites.isae-supaero.fr/workshop-em-isae-2018/workshop-em-isae-2018, 2018.

[25] M. Abduljabbar, M. Al Farhan, R. Yokota, and D. Keyes. Performance evaluation of computation and communication kernels of the fast multipole method on intel manycore architecture. In F. F. Rivera, T. F. Pena, and J. C. Cabaleiro, editors, *Euro-Par 2017: Parallel Processing*, pages 553–564, Cham, 2017. Springer International Publishing.

[26] M. Abduljabbar, G. S. Markomanolis, H. Ibeid, R. Yokota, and D. Keyes. Communication reducing algorithms for distributed hierarchical n-body problems with boundary distributions. In J. M. Kunkel, R. Yokota, P. Balaji, and D. Keyes, editors, *High Performance Computing*, pages 79–96, Cham, 2017. Springer International Publishing.

[27] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '00, pages 1–12, New York, NY, USA, 2000. ACM.

[28] E. Agullo, O. Aumage, B. Bramas, O. Coulaud, and S. Pitoiset. Bridging the gap between OpenMP 4.0 and native runtime systems for the fast multipole method. Research Report RR-8953, Inria, Mar. 2016.

[29] E. Agullo, O. Aumage, B. Bramas, O. Coulaud, and S. Pitoiset. Bridging the gap between OpenMP and task-based runtime systems for the fast multipole method. *IEEE Transactions on Parallel and Distributed Systems*, page 14, Apr. 2017.

[30] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi. Pipelining the Fast Multipole Method over a Runtime System. In *SIAM Conference on Computational Science and Engineering (SIAM CSE 2013)*, Boston, United States, Feb. 2013.

[31] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi. Task-based FMM for multicore architectures. Technical Report RR-8277, INRIA, Mar. 2013.

[32] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi. Task-based FMM for heterogeneous architectures. Research Report RR-8513, Inria, Apr. 2014.

[33] E. Agullo, B. Bramas, O. Coulaud, M. Khannouz, and L. Stanisic. Task-based fast multipole method for clusters of multicore processors. Research Report RR-8970, Inria Bordeaux Sud-Ouest, Mar. 2017.

[34] A. Amer, N. Maruyama, M. Pericàs, K. Taura, R. Yokota, and S. Matsuoka. Fork-join and data-driven execution models on multi-core architectures: Case study of the fmm. In J. M. Kunkel, T. Ludwig, and H. W. Meuer, editors, *Supercomputing*, pages 255–266, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[35] A. Amer, S. Matsuoka, M. Pericàs, N. Maruyama, K. Taura, R. Yokota, and P. Balaji. Scaling fmm with data-driven openmp tasks on multicore architectures. volume 9903, pages 156–170, 10 2016.

[36] P. Atkinson and S. McIntosh-Smith. On the performance of parallel tasking runtimes for an irregular fast multipole method application. In B. R. de Supinski, S. L. Olivier, C. Terboven, B. M. Chapman, and M. S. Müller, editors, *Scaling OpenMP for Exascale Performance and Portability*, pages 92–106, Cham, 2017. Springer International Publishing.

[37] 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*, 23(2):187–198, 2011.

[38] E. Ayguadé, A. Duran, J. Hoeflinger, F. Massaioli, and X. Teruel. An experimental evaluation of the new openmp tasking model. In V. Adve, M. J. Garzarán, and P. Petersen, editors, *Languages and Compilers for Parallel Computing*, pages 63–77, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[39] E. Ayguadé, N. Copty, I. C. Society, R. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of openmp tasks. *IEEE Trans. Parallel Distrib. Syst*, pages 404–418.

[40] 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. *Advances in Parallel Computing*, Volume 24: Transition of HPC Towards Exascale Computing:141–155.

[41] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions.

[42] R. D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Cambridge, MA, USA, 1995. Not available from University Microfilms Int.

[43] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[44] F. Broquedis, T. Gautier, and V. Danjean. libKOMP, an Efficient OpenMP Runtime System for Both Fork-Join and Data Flow Paradigms. In *IWOMP*, pages 102–115, Rome, Italy, 2012.

[45] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta. Productive cluster programming with ompss. In E. Jeannot, R. Namyst, and J. Roman, editors, *Euro-Par 2011 Parallel Processing*, pages 555–566, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[46] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta. Productive programming of gpu clusters with ompss. pages 557–568, 05 2012.

[47] Q. Carayol. *Development and analysis of a multilevel multipole method for electromagnetics*. PhD thesis, Paris 6, 2002.

[48] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[49] A. Chandramowlishwaran, K. Madduri, and R. Vuduc. Diagnosis, tuning, and redesign for multicore performance: A case study of the fast multipole method. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–12, Washington, DC, USA, 2010. IEEE Computer Society.

[50] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, and R. W. Vuduc. Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*, pages 1–12, 2010.

[51] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 2:1–2:3, New York, NY, USA, 2010. ACM.

[52] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, Oct. 2005.

[53] J. Choi, A. Chandramowlishwaran, K. Madduri, and R. Vuduc. A cpu: Gpu hybrid implementation and model-driven scheduling of the fast multipole method. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, pages 64:64–64:71, New York, NY, USA, 2014. ACM.

[54] 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*, E2SC '13, pages 2:1–2:8, New York, NY, USA, 2013. ACM.

[55] O. Coulaud, P. Fortin, and J. Roman. Hybrid MPI-thread parallelization of the Fast Multipole Method. In *ISPDC 2007, 6th International Symposium on Parallel and Distributed Computing*, Hagenberg, Austria, July 2007.

[56] F. A. Cruz, M. G. Knepley, and L. A. Barba. Petfmm–a dynamically load-balancing parallel fast multipole library. *CoRR*, abs/0905.2637, 2009.

[57] M. De Wael, S. Marr, B. De Fraine, T. Van Cutsem, and W. De Meuter. Partitioned global address space languages. *ACM Comput. Surv.*, 47(4):62:1–62:27, May 2015.

[58] A. Duran, J. Corbalán, and E. Ayguadé. Evaluation of openmp task scheduling strategies. In R. Eigenmann and B. R. de Supinski, editors, *OpenMP in a New Era of Parallelism*, pages 100–110, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[59] T. El-Ghazawi and L. Smith. Upc: Unified parallel c. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[60] V. End. *On Collective Communication and Notified Read in the Global Address Space Programming Interface (GASPI)*. PhD thesis, 03 2017.

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

[62] G. E. Fagg, J. Pjesivac-grbovic, G. Bosilca, J. J. Dongarra, and E. Jeannot. Flexible collective communication tuning architecture applied to open mpi. In *In 2006 Euro PVM/MPI*, 2006.

[63] J. T. Fineman and C. E. Leiserson. *Race Detectors for Cilk and Cilk++ Programs*, pages 1706–1719. Springer US, Boston, MA, 2011.

[64] P. Fortin. *High performance parallel hierarchical algorithmic for N-body problems*. Theses, Université Sciences et Technologies - Bordeaux I, Nov. 2006.

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

[66] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[67] 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*, SPAA '09, pages 79–90, New York, NY, USA, 2009. ACM.

[68] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998.

[69] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open mpi: Goals, concept, and design of a next generation mpi implementation. In D. Kranzlmüller, P. Kacsuk, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

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

[71] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73(2):325–348, Dec. 1987.

[72] W. Gropp. Mpi 3 and beyond: Why mpi is successful and what challenges it faces. In J. L. Träff, S. Benkner, and J. J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, pages 1–9, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[73] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. The MIT Press, 2014.

[74] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.

[75] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, USA, 1999.

[76] D. Grünewald. Bqcd with gpi: A case study. *2012 International Conference on High Performance Computing and Simulation (HPCS)*, pages 388–394, 2012.

[77] D. Grünewald and C. Simmendinger. The gaspi api specification and its implementation gpi 2.0. 10 2013.

[78] 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*, SPAA '10, pages 145–156, New York, NY, USA, 2010. ACM.

[79] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[80] T. Hoefler and A. Lumsdaine. Message Progression in Parallel Computing - To Thread or not to Thread? IEEE Computer Society, Oct. 2008.

[81] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra. Dynamic task discovery in parsec: A data-flow task-based runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '17, pages 6:1–6:8, New York, NY, USA, 2017. ACM.

[82] S. Hunold and A. Carpen-Amarie. Autotuning mpi collectives using performance guidelines. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, HPC Asia 2018, pages 64–74, New York, NY, USA, 2018. ACM.

[83] H. Ibeid, R. Yokota, and D. Keyes. A performance model for the communication in fast multipole methods on HPC platforms. *CoRR*, abs/1405.6362, 2014.

[84] M. A. Jabbar, R. Yokota, and D. Keyes. Asynchronous execution of the fast multipole method using charm++. *CoRR*, abs/1405.7487, 2014.

[85] S. Jubertie, I. Masliah, and J. Falcou. Data layout and simd abstraction layers: Decoupling interfaces from implementations. pages 531–538, 07 2018.

[86] M. Korch and T. Rauber. A comparison of task pools for dynamic load balancing of irregular algorithms: Research articles. *Concurr. Comput. : Pract. Exper.*, 16(1):1–47, Dec. 2003.

[87] N. Kroll, T. Gerhold, S. Melber, R. Heinrich, T. Schwarz, and B. Schöning. *Parallel Large Scale Computations for Aerodynamic Aircraft Design with the German CFD System MEGAFLOW*, pages 227–236. 12 2002.

[88] J. Kurzak and B. M. Pettitt. Communications overlapping in fast multipole particle dynamics methods. *Journal of Computational Physics*, 203(2):731 – 743, 2005.

[89] J. Kurzak and B. M. Pettitt. Massively parallel implementation of a fast multipole method for distributed memory machines. *Journal of Parallel and Distributed Computing*, 65(7):870 – 881, 2005.

[90] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros. A massively parallel adaptive fast-multipole method on heterogeneous architectures. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 58:1–58:12, New York, NY, USA, 2009. ACM.

[91] C. E. Leiserson. The cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, Mar 2010.

[92] H. Ltaief and R. Yokota. Data-driven execution of fast multipole methods. *CoRR*, abs/1203.0889, 2012.

[93] R. Machado, S. Abreu, and D. Diaz. Parallel local search: Experiments with a pgas-based programming model. *CoRR*, abs/1301.7699, 2013.

[94] M.A.Heroux. Mantevo home page. https://mantevo.org/, 2008.

[95] J. Makino. A Fast Parallel Treecode with GRAPE. *Publications of the Astronomical Society of Japan*, 56(3):521–531, 06 2004.

[96] J. Milthorpe, A. P. Rendell, and T. Huber. Pgas-fmm: Implementing a distributed fast multipole method using the x10 programming language. *CCPE*, 26(3):712–727, 2014.

[97] P. Miquel, A. Abdelhalim, F. Keisuke, M. Naoya, Y. Rio, and M. Satoshi. Towards a dataflow fmm using the ompss programming model. Technical Report 12, Global Scientific Information and Computing Center, Tokyo Institute of Technology, Department of Mathematical and Compute Sciences, Tokyo Institute of Technology, Department of Mathematical and Compute Sciences, Tokyo Institute of Technology, Advanced Institute for Computational Science, Riken, King Abdullah University of Science and Technology, Global Scientific Information and Computing Center, Tokyo Institute of Technology, sep 2012.

[98] N. Möller, E. Petit, Q. Carayol, Q. Dinh, and W. Jalby. Scalable fast multipole method for electromagnetic simulations. In J. M. F. Rodrigues, P. J. S. Cardoso, J. Monteiro, R. Lam, V. V. Krzhizhanovskaya, M. H. Lees, J. J. Dongarra, and P. M. Sloot, editors, *Computational Science – ICCS 2019*, pages 663–676, Cham, 2019. Springer International Publishing.

[99] N. Möller, E. Petit, Q. Carayol, Q. Dinh, and W. Jalby. Asynchronous One-Sided Communications for Scalable Fast Multipole Method in Electromagnetic Simulations, Aug. 2017. Short Paper presented at COLOC workshop, Euro-Par 2017, Santiago de Compostela, August 29, 2017.

[100] N. Möller, E. Petit, L. Thébault, and Q. Dinh. A case study on using a proto-application as a proxy for code modernization. *Procedia Comput. Sci.*, 51(C):1433–1442, Sept. 2015.

[101] Y. Nakamura and H. Stüben. Bqcd - berlin quantum chromodynamics program. 10 2010.

[102] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, Aug. 1998.

[103] L. Oden. Gpi2 for gpus: A pgas framework for efficient communication in hybrid clusters. 01 2013.

[104] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. Uts: An unbalanced tree search benchmark. In G. Almási, C. Caşcaval, and P. Wu, editors, *Languages and Compilers for Parallel Computing*, pages 235–250, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[105] S. L. Olivier and J. F. Prins. Evaluating openmp 3.0 run time systems on unbalanced task graphs. In M. S. Müller, B. R. de Supinski, and B. M. Chapman, editors, *Evolving OpenMP in an Age of Extreme Parallelism*, pages 63–78, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[106] 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*, 38(5-6):341–360, 2010.

[107] B. R. de Supinski, T. R. W. Scogland, A. Duran, M. Klemm, S. Mateo Bellido, S. L. Olivier, C. Terboven, and T. Mattson. The ongoing evolution of openmp. *Proceedings of the IEEE*, PP:1–16, 08 2018.

[108] W. T. Rankin, III. *Efficient Parallel Implementations of Multipole Based N-body Algorithms*. PhD thesis, Durham, NC, USA, 1999. AAI9928860.

[109] J. Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[110] F. Shahzad, M. Wittmann, M. Kreutzer, T. Zeiser, G. Hager, and G. Wellein. PGAS implementation of SpMVM and LBM with GPI. In *Proceedings of the 7th International Conference on PGAS Programming Models*, pages 172–184, Edinburgh, 2013. The University of Edinburgh. UnivIS-Import:2015-04-16:Pub.2013.tech.IMMD.phleir.pgasim.

[111] C. Simmendinger, J. Jägersküpper, R. Machado, and C. Lojewski. A pgas-based implementation for the unstructured cfd solver tau. *PGAS11, USA*, 2011.

[112] C. Simmendinger, M. Rahn, and D. Grünewald. *The GASPI API: A Failure Tolerant PGAS API for Asynchronous Dataflow on Heterogeneous Architectures*, pages 17–32. 11 2015.

[113] Staff. Using mpi-portable parallel programming with the message-passing interface, by william gropp. *Sci. Program.*, 5(3):275–276, Aug. 1996.

[114] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda. Rdma read based rendezvous protocol for mpi over infiniband: Design alternatives and benefits. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 32–39, New York, NY, USA, 2006. ACM.

[115] S.-H. Teng. Provably good partitioning and load balancing algorithms for parallel adaptive n-body simulation. *SIAM J. Sci. Comput.*, 19(2):635–656, Mar. 1998.

[116] R. Thakur, W. Gropp, and B. Toonen. Optimizing the synchronization operations in message passing interface one-sided communication. *The International Journal of High Performance Computing Applications*, 19(2):119–128, 2005.

[117] L. Thebault. *Scalable and Efficient Algorithms for Unstructured Mesh Computations*. Theses, Université Paris-Saclay, Oct. 2016.

[118] 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.

[119] L. Thébault and E. Petit. The mini-fem proto-application. https://github.com/EXAPARS/Mini-FEM, 2016.

[120] L. Thébault and E. Petit. Asynchronous and multithreaded communications on irregular applications using vectorized divide and conquer approach. *Journal of Parallel and Distributed Computing*, 114:16 – 27, 2018.

[121] S. Treichler, M. Bauer, and A. Aiken. Realm: An event-based low-level runtime for distributed memory architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 263–276, New York, NY, USA, 2014. ACM.

[122] S. Vadhiyar, G. Fagg, and J. Dongarra. Performance modeling for self adapting collective communications for mpi. Santa Fe, NM, 10-2001 2001.

[123] P. Virouleau, P. BRUNET, F. Broquedis, N. Furmento, S. Thibault, O. Aumage, and T. Gautier. Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite. In *10th International Workshop on OpenMP, IWOMP2014*, 10th International Workshop on OpenMP, IWOMP2014, pages 16 – 29, Salvador, Brazil, France, Sept. 2014. Springer.

[124] W. W Carlson, J. M Draper, D. Culler, K. Yelick, E. Brooks, K. Warren, and L. Livermore. Introduction to upc and language specification. 04 1999.

[125] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, pages 12–21, New York, NY, USA, 1993. ACM.

[126] M. Wittmann, G. Hager, T. Zeiser, and G. Wellein. Asynchronous MPI for the masses. *CoRR*, abs/1302.4280, 2013.

[127] A. YarKhan, J. Kurzak, and J. Dongarra. Quark users' guide: Queueing and runtime for kernels. Technical Report ICL-UT-11-02, 00-2011 2011.

[128] L. Ying, G. Biros, and D. Zorin. A kernel-independent adaptive fast multipole algorithm in two and three dimensions. 07 2003.

[129] L. Ying, G. Biros, D. Zorin, and M. H. Langston. A new parallel kernel-independent fast multipole method. pages 14– 14, 11 2003.

[130] R. Yokota and L. Barba. Treecode and fast multipole method for n-body simulation with cuda. *GPU Computing Gems Emerald Edition*, 10 2010.

[131] R. Yokota and L. A. Barba. Fast n-body simulations on gpus. *CoRR*, abs/1108.5815, 2011.

[132] R. Yokota and L. A. Barba. A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems. *CoRR*, abs/1106.2176, 2011.

[133] R. Yokota, T. Hamada, J. P. Bardhan, M. G. Knepley, and L. A. Barba. Biomolecular electrostatics simulation by an fmm-based bem on 512 gpus. *ArXiv*, abs/1007.4591, 2010.

[134] R. Yokota, G. Turkiyyah, and D. Keyes. Communication complexity of the fast multipole method and its algebraic variants. *CoRR*, abs/1406.1974, 2014.

[135] B. Zhang. Asynchronous task scheduling of the fast multipole method using various runtime systems. *Proceedings - 2014 4th Workshop on Data-Flow Execution Models for Extreme Scale Computing, DFM 2014*, pages 9–16, 04 2015.

[136] J. A. Zounmevo, X. Zhao, P. Balaji, W. Gropp, and A. Afsahi. Nonblocking epochs in MPI one-sided communication. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, pages 475–486, 2014.

[137] R. B. Ávila, C. D. P, A. E. Ci, E. D. Computac, R. Bohrer, P. Dr, and A. Navaux. A comparative study on dpc++ and other concurrent object-oriented languages.

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

universite PARIS-SACLAY

Titre: Adaptation de codes industriels de simulation en Calcul Haute Performance aux architectures modernes de supercalculateurs

Mots clés: Parallélisme, Mémoire Partagée, Mémoire Distribuée, Méthode Multipôle, CFD, CEM, Communications, Programmation par tâches

Résumé: Durant de longues années, la stabilité dans le paradigme d'architecture a facilité la portabilité de performance des grands codes en Calcul de Haute Performance.L'effondrement programmé de la loi de Moore - qui règle les progrès en gravure des microprocesseurs - bouscule ce modèle et requiert un effort nouveau du côté logiciel. Une modernisation des codes basée sur une algorithmique adaptée aux futurs systèmes est ainsi nécessaire. Cette modernisation repose sur des principes de base connus tels que la concurrence des calculs et la localité des données. Cependant, la mise en œuvre de ces principes dans le cadre d'applications réelles en milieu industriel – lesquelles applications sont souvent le fruit d'années d'efforts de développement - s'avère bien plus compliquée que ne le laissait prévoir leur simplicité apparente. Les contributions de cette thèse sont les suivantes : D'une part, nous explorons une méthodologie de modernisation de codes basée sur l'utilisation de proto-applications et la confrontons à une approche directe, en optimisant deux codes de simulation dévéloppés dans un contexte similaire. D'autre part, nous nous concentrons sur l'identification des principaux défis concernant l'adéquation entre applications, modèles de programmation et architectures. Les deux domaines d'application choisis sont la dynamique des fluides et l'électromagnétisme.

Title: Industrial Code Modernization of High Performance Computing simulations on modern supercomputer architectures

Keywords: Parallelism, Distributed Memory, Shared Memory, Multipole Method, CFD, CEM, Communications, Task-based programming

Abstract: For many years, the stability of the architecture paradigm has facilitated the performance portability of large HPC codes.The announced breakdown of the Moore's Law, which rules the progress of microprocessor engraving, ends this model and requires new efforts on the software's side. Code modernization, based on an algorithmic which is well adapted to the future systems, is mandatory. This modernization is based on well-known principles as the computation concurrency, or degree of parallelism, and the data locality. However, the implementation of these principles in large industrial applications, which often are the result of years of development efforts, turns out to be way more difficult than expected. This thesis contributions are twofold : On the one hand, we explore a methodology of software modernization based on the concept of proto-applications and compare it with the direct approach, while optimizing two simulation codes developed in a similar context. On the other hand, we focus on the identification of the main challenges for the architecture, the programming models and the applications. The two chosen application fields are the Computational Fluid Dynamics and Computational Electro Magnetics.