



HAL
open science

Numerical methods for the accelerated resolution of large scale linear systems on massively parallel hybrid architecture

Abal-Kassim Cheik Ahamed

► **To cite this version:**

Abal-Kassim Cheik Ahamed. Numerical methods for the accelerated resolution of large scale linear systems on massively parallel hybrid architecture. Other. Ecole Centrale Paris, 2015. English. NNT : 2015ECAP0035 . tel-02918210

HAL Id: tel-02918210

<https://theses.hal.science/tel-02918210v1>

Submitted on 20 Aug 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.

École doctorale n° 287 : Sciences pour l'ingénieur


Doctorat École Centrale Paris

THÈSE

pour obtenir le grade de docteur délivré par

École Centrale des Arts et Manufactures
“CentraleSupélec”

Spécialité «**Mathématiques Appliquées et Informatique**»

Laboratoire d'accueil: Mathématiques Appliquées aux Systèmes 

présentée et soutenue publiquement par

Abal-Kassim CHEIK AHAMED

le 07 juillet 2015

Méthodes numériques pour la résolution accélérée des systèmes linéaires de grandes tailles sur architectures hybrides massivement parallèles

Directeur de thèse: **Frédéric MAGOULÈS**

JURY

Mme. Corinne ANCOURT LE QUELLENEC	MINES ParisTech	Rapporteur
M. Raphaël COUTURIER	Univ. de Franche Comté	Examinateur
M. Che-Lun HUNG	Providence Univ.	Rapporteur
M. Michaël KRAJECKI	Univ. de Reims	Président
Frédéric MAGOULÈS	École Centrale Paris	Directeur de thèse
M. Jean-François MÉHAUT	Univ. Joseph Fourier	Examinateur

— 2015ECAP0035 —

École Centrale des Arts et Manufactures (CentraleSupélec)
Mathématiques Appliquées aux Systèmes
Grande Voie des Vignes, 92295 Châtenay-Malabry Cedex, France

I RECEIVED my french Baccalauréat S. (Scientific) in Mayotte, France, in 2006. In 2008, I graduated with three B.Sc. Degree at the Université de Pau et des Pays de l'Adour and Université de Cergy-Pontoise, France. I graduated with Engineer degree in the Department of HPC and Scientific Computing at the Ecole Internationale des Sciences du traitement de l'information (EISTI), France, in 2011. I also graduated with a M.Sc. degree in the Department of Science, Technology and Health at the Université de Cergy-Pontoise, France, in 2011. I candidate toward the PhD degree of Ecole Centrale Paris (CentraleSupélec), France.

Abal-Kassim CHEIK AHAMED

Fr.: *Méthodes numériques pour la résolution accélérée des systèmes linéaires de grandes tailles sur architectures hybrides massivement parallèles*


En.: *Numerical methods for the accelerated resolution of large scale linear systems on massively parallel hybrid architecture*

Date de production: 07 octobre 2015

Date de soutenance: 07 juillet 2015

Doctorat École Centrale Paris (CentraleSupélec)

Spécialité “Mathématiques Appliquées et Informatique”

Laboratoire d'accueil: Mathématiques Appliquées aux Systèmes 

Directeur de thèse: Frédéric MAGOULÈS

JURY :	Mme. Corinne ANCOURT LE QUELLENEC	MINES ParisTech	Rapporteur
	M. Raphaël COUTURIER	Univ. de Franche Comté	Examineur
	M. Che-Lun HUNG	Providence Univ.	Rapporteur
	M. Michaël KRAJECKI	Univ. de Reims	Président
	Frédéric MAGOULÈS	École Centrale Paris	Directeur de thèse
	M. Jean-François MÉHAUT	Univ. Joseph Fourier	Examineur

AUTRES : M. Denis GERRER Nvidia France Invité

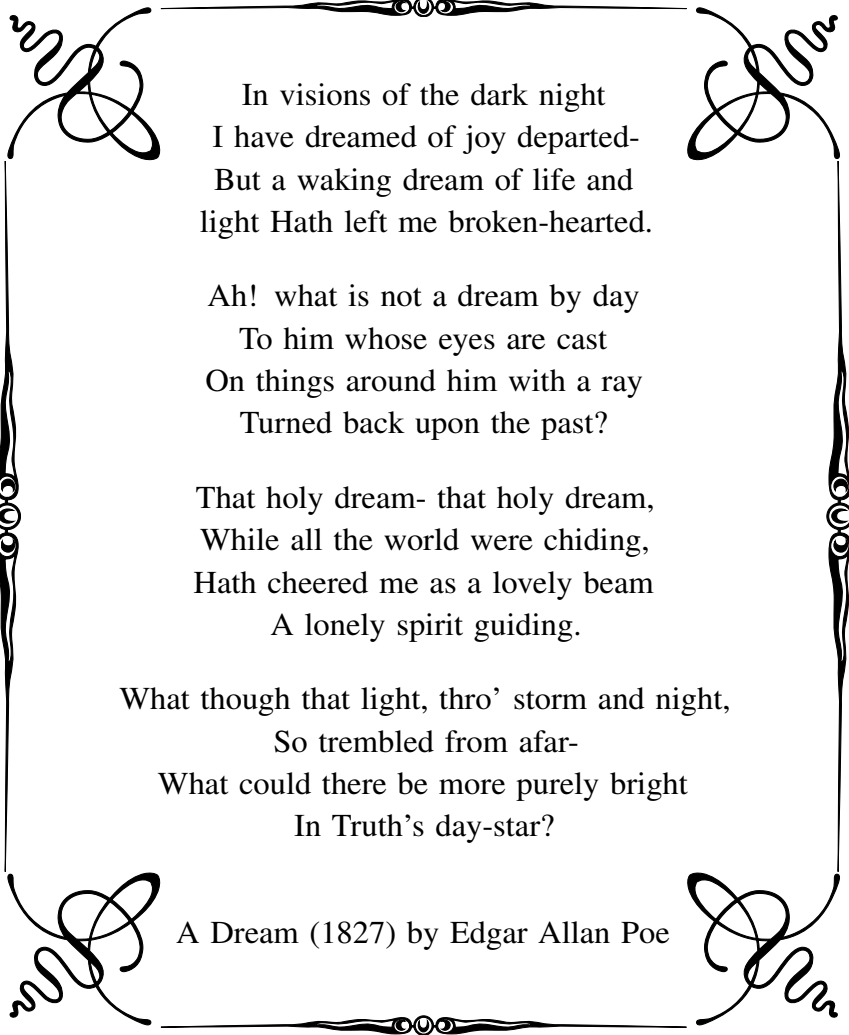
École doctorale n° 287 : Sciences pour l'ingénieur — 2015ECAP0035 —

École Centrale des Arts et Manufactures (École Centrale Paris)  «CentraleSupélec» 

Mathématiques Appliquées aux Systèmes

Grande Voie des Vignes, 92295 Châtenay-Malabry Cedex, France

Copyright © Abal-Kassim CHEIK AHAMED



In visions of the dark night
I have dreamed of joy departed-
But a waking dream of life and
light Hath left me broken-hearted.

Ah! what is not a dream by day
To him whose eyes are cast
On things around him with a ray
Turned back upon the past?

That holy dream- that holy dream,
While all the world were chiding,
Hath cheered me as a lovely beam
A lonely spirit guiding.

What though that light, thro' storm and night,
So trembled from afar-
What could there be more purely bright
In Truth's day-star?

A Dream (1827) by Edgar Allan Poe

ABSTRACT OF THE THESIS

Numerical methods for the accelerated resolution of large scale linear systems on massively parallel hybrid architecture

Abstract — Advances in computational power have led to many developments in science and its applications. Solving linear systems occurs frequently in scientific computing, as in the finite element discretization of partial differential equations. The running time of the overall resolution is a direct result of the performance of the involved algebraic operations.

In this dissertation, different ways of efficiently solving large and sparse linear systems are put forward. We present the best way to effectively compute linear algebra operations in an heterogeneous multi-core-GPU environment in order to make solvers such as iterative methods more robust and therefore reduce the computing time of these systems. We propose new techniques to speed algorithms up the auto-tuning of the threading design, according to the problem characteristics and the equipment level in the hardware and available resources. Numerical experiments performed on a set of large-size sparse matrices arising from diverse engineering and scientific problems, have clearly shown the benefit of the use of GPU technology to solve large sparse systems of linear equations, and its robustness and accuracy compared to existing libraries such as Cusp.

The main priority of the GPU program is computational time to obtain the solution in a parallel environment, *i.e.*, “How much time is needed to solve the problem?”. In this thesis, we also address another question regarding energy issues, *i.e.*, “How much energy is consumed by the application?”. To answer this question, an experimental protocol is established to measure the energy consumption of a GPU for fundamental linear algebra operations accurately. This methodology fosters a “new vision of high-performance computing” and answers some of the questions outlined in green computing when using GPUs.

The remainder of this thesis is devoted to synchronous and asynchronous iterative algorithms for solving linear systems in the context of a multi-core-GPU system. We have implemented and analyzed these algorithms using iterative methods based on sub-structuring techniques. Mathematical models and convergence results of synchronous and asynchronous algorithms are presented here, as are the convergence results of the asynchronous sub-structuring methods. We then analyze these methods in the context of a hybrid multi-core-GPU, which should pave the way for exascale hybrid methods.

Lastly, we modify the non-overlapping Schwarz method to accelerate it, using GPUs. The implementation is based on the acceleration of the local solutions of the linear sub-systems associated with each sub-domain using GPUs. To ensure good performance, optimized conditions obtained by a stochastic technique based on the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) are used. Numerical results illustrate the good performance, robustness and accuracy of synchronous and asynchronous algorithms to solve large sparse linear systems in the context of an heterogeneous multi-core-GPU system.

Keywords — Parallel algorithm, GPU, OpenCL, CUDA, Auto-tuning, Green computing, Energy consumption, Iterative methods, Synchronous, Asynchronous, Substructuring methods, Domain Decomposition Method, Schwarz, Hybrid methods, Exascale

Méthodes numériques pour la résolution accélérée des systèmes linéaires de grandes tailles sur architectures hybrides massivement parallèles

Résumé — Les progrès en termes de puissance de calcul ont entraîné de nombreuses évolutions dans le domaine de la science et de ses applications. La résolution de systèmes linéaires survient fréquemment dans le calcul scientifique, comme par exemple lors de la résolution d'équations aux dérivées partielles par la méthode des éléments finis. Le temps de résolution découle alors directement des performances des opérations algébriques mises en jeu.

Cette thèse a pour but de développer des algorithmes parallèles innovants pour la résolution de systèmes linéaires creux de grandes tailles. Nous étudions et proposons comment calculer efficacement les opérations d'algèbre linéaire sur plateformes de calcul multi-cœur hétérogènes-GPU afin d'optimiser et de rendre robuste la résolution de ces systèmes. Nous proposons de nouvelles techniques d'accélération basées sur la distribution automatique (*auto-tuning*) des threads sur la grille GPU suivant les caractéristiques du problème et le niveau d'équipement de la carte graphique, ainsi que les ressources disponibles. Les expérimentations numériques effectuées sur un large spectre de matrices issues de divers problèmes scientifiques, ont clairement montré l'intérêt de l'utilisation de la technologie GPU, et sa robustesse comparée aux bibliothèques existantes comme Cusp.

L'objectif principal de l'utilisation du GPU est d'accélérer la résolution d'un problème dans un environnement parallèle multi-cœur, c'est-à-dire "Combien de temps faut-il pour résoudre le problème?". Dans cette thèse, nous nous sommes également intéressés à une autre question concernant la consommation énergétique, c'est-à-dire "Quelle quantité d'énergie est consommée par l'application?". Pour répondre à cette seconde question, un protocole expérimental est établi pour mesurer la consommation d'énergie d'un GPU avec précision pour les opérations fondamentales d'algèbre linéaire. Cette méthodologie favorise une "nouvelle vision du calcul haute performance" et apporte des réponses à certaines questions rencontrées dans l'informatique verte ("*green computing*") lorsque l'on s'intéresse à l'utilisation de processeurs graphiques.

Le reste de cette thèse est consacré aux algorithmes itératifs synchrones et asynchrones pour résoudre ces problèmes dans un contexte de calcul hétérogène multi-cœur-GPU. Nous avons mis en application et analysé ces algorithmes à l'aide des méthodes itératives basées sur les techniques de sous-structurations. Dans notre étude, nous présentons les modèles mathématiques et les résultats de convergence des algorithmes synchrones et asynchrones. La démonstration de la convergence asynchrone des méthodes de sous-structurations est présentée. Ensuite, nous analysons ces méthodes dans un contexte hybride multi-cœur-GPU, qui devrait ouvrir la voie vers les méthodes hybrides exaflopiques.

Enfin, nous modifions la méthode de Schwarz sans recouvrement pour l'accélérer à l'aide des processeurs graphiques. La mise en œuvre repose sur l'accélération par les GPUs de la résolution locale des sous-systèmes linéaires associés à chaque sous-domaine. Pour améliorer les performances de la méthode de Schwarz, nous avons utilisé des conditions d'interfaces optimisées obtenues par une technique stochastique basée sur la stratégie CMA-ES (Covariance Matrix Adaptation Evolution Strategy). Les résultats numériques attestent des bonnes performances, de la robustesse et de la précision des algorithmes synchrones et asynchrones pour résoudre de grands systèmes linéaires creux dans un environnement de calcul hétérogène multi-cœur-GPU.

Mot clés — Parallel algorithm, GPU, OpenCL, CUDA, Auto-tuning, Green computing, Energy consumption, Iterative methods, Synchronous, Asynchronous, Substructuring methods, Domain Decomposition Method, Schwarz, Hybrid methods, Exascale

Acknowledgements

With hindsight, I am convinced that research is far from lonely work. Indeed, I could never have done this work without the support of so many people whose good humor, generosity and interest with regard to my research have allowed me to move on through the sensitive stages of being a trainee researcher.

First, I warmly thank Prof. Dr. Frédéric Magoulès, my thesis tutor, for lending his support throughout my thesis and for the help and advice he has provided during the various stages. He has always given me valuable ideas and suggestions.

Many thanks to my thesis reviewers: Mrs. Corinne Ancourt Le Quellenec and Mr. Che-Lun Hung, and the defense jury committee: Mr. Raphaël Couturier (president), Mr. Michaël Krajecki, Mr. Jean-François Méhaut. Many thanks also to Mr. Denis Gerrer for participating in the jurying process.

I also express my thanks to my colleagues and friends, Baptiste Zhang, Guillaume Gbikpi-Benissan, Mugabe Alain Rucyahana, Rémi Cerise, Ricardo Bispo Vieira and many other students at Ecole Centrale Paris. They have been helpful and very pleasant to be with me and to have enriched my experience throughout the time we spent together. I also express my thanks to all staff of MAS laboratory and doctoral school.

I acknowledge partial financial support from the OpenGPU project (Pôle de Compétitivité Systematic, France), and from the CRESTA (Collaborative Research into Exascale Systemware, Tools and Applications) project (European Commission). I also commend the CUDA Research Center at Ecole Centrale Paris “CentraleSupélec” (France) for its support and for providing the computing facilities.

I dedicate this PhD thesis to all my family and especially to my parents **Kassab** and **Cheik Ahamed**, who taught me the value of education, and my wife **Nourou**, my strong supporter. All my life, my parents have encouraged my curiosity and encouraged me to explore, for which I am eternally grateful.

I also thank my siblings: Marachi (Ma Omar), Momed (Ba Cheik), Said (Ba Hayrati), Zaoid (Ma Bilale), Ousséni (Ba Fazal), Anfane (Ba Belhadji), Oumrati (Ma Saidali), Anoir (Ba Rahime), Toybina (Ma Ankidine), Anli (Doulou Star, Ba Meïmouna), Bibi-Echati (Ma Djamel), Ambdoulhakime (Ba Raymane), Mouhamadi El-Had (Ba Afzal), Ambdourrahmane (Chehou, Ba Sâraah), Soibahaddine (Ba Yaqîne), Ambdourazak (Ba Layane) and Ahmad (Ba Sajîa) for their contribution. I do not forget their wives and their children.

I would also like to give credit to my stepmother (Gnadza Matou) with her “bata-bata na gnama” and my stepfather (Baninga) for their daily support.

I also thank Kamaldine (DKama) and Ousséni (DD Mtou) for their warm welcome when I arrived in Paris and M. Soumaïla Salime (Raïs) for their continuous support.

Finally, I would like to thank all the teachers I have met, who have each given me some of their knowledge throughout my schooling.

Abal-Kass. Thank you so much.

Je dédie cette thèse de doctorat à tous mes proches, et particulièrement à mes parents **Kassab** et **Cheik Ahamed**, qui m'ont appris les valeurs de l'éducation, et à ma femme **Nourou**, mon fervent soutien. Toute ma vie, mes parents ont nourri ma curiosité et m'ont encouragé à l'explorer, ce pourquoi je suis et serais éternellement reconnaissant.

Je tiens également à remercier chaleureusement ma fratrie: Marachi (Ma Omar), Momed (Ba Cheik), Said (Ba Hayrati), Zaoid (Ma Bilale), Ousséni (Ba Fazal), Anfane (Ba Belhadji), Oumrati (Ma Saidali), Anoir (Ba Rahime), Toybina (Ma Ankidine), Anli (Doulou Star, Ba Meïmouna), Bibi-Echati (Ma Djamel), Ambdoulhakime (Ba Raymane), Mouhamadi El-Had (Ba Afzal), Ambdourrahmane (Chehou, Ba Sâraah), Soibahaddine (Ba Yaqîne), Ambdourazak (Ba Layane) et Ahmad (Ba Sajîa) pour leur contribution. Sans oublier leurs épouses et leurs enfants. Mes dédicaces vont également vers ma belle mère (Gnadza Matou) et mon beau père (Baninga) pour leur soutien quotidien.

Je souhaite également remercier Kamaldine (DKama) et Ousséni (DD Mtou) pour l'accueil chaleureux qu'ils m'ont réservé à mon arrivée à Paris, et M. Soumaïla Salime (Raïs) pour son soutien indéfectible.

Et bien évidemment tous les professeurs que j'ai rencontrés et qui m'ont tous apporté un peu de leur savoir tout au long de mon parcours scolaire.

Abal-Kass. Mille remerciements.

Contents

Abstract	i
Résumé	ii
Acknowledgements	iii
Dedication	v
List of Listings	xxiii
1 General Introduction	1
1.1 Motivation and Background	1
1.1.1 Linear system solvers	1
1.1.2 Parallel architectures	2
1.1.3 GPU Computing	3
1.1.4 Green Computing with GPU	4
1.1.5 Parallel algorithms	4
1.1.6 Project support	5
1.2 Organization of this thesis and contributions	6
2 GPU Computing	9
2.1 Introduction	9
2.2 History and evolution	10
2.3 General-Purpose GPU Computing	11
2.3.1 GPU Programming languages	12
2.3.2 Hardware architecture	12
2.4 GPU Programming Model	14
2.4.1 Memory structure	14
2.4.2 Kernel and gridification	15
2.4.3 Auto-tuning of the grid	17
2.4.4 Memory access and data transfers	20
2.4.5 Implementing complex number arithmetics on GPU	22
2.5 Energy consumption	24
2.5.1 Introduction	24
2.5.2 Experimental protocol	26
2.5.3 Signal processing	28
2.5.4 Acquisition process	30
2.5.5 Detection of the phases	32

2.5.6	Prediction	33
2.5.7	Evaluation of the execution time and energy consumption	35
2.6	Benchmark environments	35
2.6.1	Measurement of execution time	35
2.6.2	Environment for benchmarks	36
2.7	Conclusion	37
3	Linear Algebra on Parallel and GPU Computing	39
3.1	Introduction	39
3.2	Sparse linear system solvers	41
3.2.1	Matrix storage format	41
3.2.2	Direct methods	44
3.2.3	Iterative methods	46
3.3	Overview of existing scientific libraries	52
3.3.1	Basic Linear Algebra Subprograms	52
3.3.2	Toward accelerated BLAS	53
3.4	BLAS level-1 operations	54
3.4.1	Addition of vectors (axpy)	54
3.4.2	Element-Wise product	55
3.4.3	Scaling vectors	56
3.4.4	Dot product and norm	57
3.4.5	Numerical results	58
3.5	Sparse matrix-vector product	63
3.5.1	Basic algorithms	63
3.5.2	GPU Implementation	65
3.5.3	Numerical results	67
3.6	Implementation of Sparse linear system solvers	80
3.6.1	Iterative Krylov Algorithms	81
3.6.2	Implementation on a GPU	86
3.6.3	Numerical results	87
3.7	Alinea: an hybrid CPU/GPU library	94
3.7.1	Introduction	94
3.7.2	BLAS level-1	95
3.7.3	Matrix storage format	97
3.7.4	BLAS level-2 and level-3	99
3.7.5	Solvers	100
3.7.6	Conclusion	103
3.8	Conclusion	104
4	Introduction to parallel linear system solvers	105
4.1	Introduction	105
4.2	Classification of parallel iterative methods	106
4.2.1	Iterative methods SISC	106
4.2.2	Iterative methods SIAC	107
4.2.3	Iterative methods AIAC	108

4.2.4	Towards asynchronous algorithms	110
4.3	Graph and Matrix partitioning	112
4.3.1	Band-row splitting	113
4.3.2	Band-column splitting	116
4.3.3	Block-diagonal splitting	116
4.3.4	Mesh and Graph Coloring with METIS	117
4.3.5	Sub-structuring splitting	120
4.3.6	Evaluation of the partitioning	127
4.4	Mathematical model	127
4.4.1	Synchronous iterations	128
4.4.2	Totally asynchronous iterations (T-AI)	130
4.4.3	Jacobi method, a stationary iterative method based on splittings	136
4.5	Conclusion	139
5	Implementation of parallel linear system solvers	141
5.1	Introduction	141
5.2	Sub-structuring methods	142
5.2.1	Overview of the principle of sub-structuring methods	142
5.2.2	Matrix-based splitting	146
5.2.3	Synchronous sub-structuring algorithm	149
5.2.4	Asynchronous sub-structuring algorithm	153
5.2.5	Convergence detection	158
5.2.6	Implementation on a cluster of multi-core-GPUs	159
5.2.7	Fault-tolerance and iteration penalty behavior	159
5.3	Theoretical speed-up of the fully parallelizable iterative method	161
5.3.1	Background and motivations	161
5.3.2	Speed-up of the fully parallelizable iterative method in parallel computing	165
5.3.3	Estimation of the costs of data exchange	169
5.3.4	Synchronous vs Sequential	170
5.3.5	Asynchronous vs Sequential	171
5.3.6	Asynchronous vs Synchronous	172
5.3.7	Application to Jacobi method	172
5.4	Numerical results	174
5.4.1	Test cases	174
5.4.2	Parallel test cases	180
5.4.3	Jacobi's numerical results using GPU	183
5.4.4	Numerical analysis of the theoretical speed-up	186
5.4.5	Parallel synchronous and asynchronous numerical results	190
5.4.6	Fault-tolerance and iteration penalty behavior	195
5.5	Conclusion	199
6	Implementation of the Optimized Schwarz method without overlap for the gravitational potential equation on a cluster of GPUs	201
6.1	Introduction	201
6.2	Optimized Schwarz method	202

6.3	Stochastic-based optimization	204
6.4	Numerical experiments	207
6.5	Conclusion	208
7	Conclusions and Perspectives	211
7.1	Summary	211
7.2	Perspectives and future works	212
	Bibliography	215

List of Figures

2.1	Evolution of the peak performance of GPUs	11
2.2	CPU architecture vs GPU architecture	12
2.3	CUDA and OpenCL memory models	14
2.4	The different hierarchy of execution process and gridification	16
2.5	GPGPU API architecture and GPU computing processing flow	16
2.6	Grid computing strategies: one-dimensional grid technique (a) and two-dimensional grid technique (b)	20
2.7	Data transfers between CPU side and GPU side	21
2.8	Cosines raw efficiency (a) and computing time in microseconds (b)	21
2.9	Memory access mode: $i + k * step$ (a) and $i * step + k$ (b)	22
2.10	Number of memory access per microseconds	22
2.11	Riser uncut and riser cut connected to a clamp	27
2.12	Schematic representation of the protocol	27
2.13	Experiment wiring: clamp on GPU computer	27
2.14	Original signal measured by the oscilloscope and its associated frequency spectrum	28
2.15	Low-pass filters with different cut-off frequencies	29
2.16	Simple averaging filters	30
2.17	Gaussian filters	30
2.18	Summary of filters	31
2.19	Procedure of the acquisition and screen of the oscilloscope	32
2.20	Energy consumption of the different phases of the execution and the approximated function of the signal	32
2.21	GENCI-CEA hybrid computing clusters	37
3.1	Example of non-zero pattern of a sparse matrix A	42
3.2	<i>CO</i> ordinate (<i>COO</i>) storage format	42
3.3	Compressed-Sparse Row (<i>CSR</i>) storage format	43
3.4	Compressed-Sparse Column (<i>CSC</i>) storage format	43
3.5	<i>ELL</i> PACK (<i>ELL</i>) storage format	44
3.6	<i>HYB</i> rid (<i>HYB</i>) storage format	44
3.7	<i>LU</i> decomposition into a product of a lower triangular matrix L and an upper triangular matrix U	45
3.8	Design of the splitting algorithm	49
3.9	Summation of two vectors	54
3.10	Summation of two vectors in parallel	55
3.11	Multiplying two vectors	56
3.12	Multiplication of two vectors in parallel	56

3.13	Dot product scheme for tree cutting passes sums	57
3.14	Execution times of vector construction and copy vector in μs on <i>NVIDIA</i> card	59
3.15	Running times of the addition of vectors and the dot product in μs on <i>NVIDIA</i> card	59
3.16	Execution times of the addition of vectors and the dot product on <i>NVIDIA</i> and <i>ATI</i>	59
3.17	DDOT: Time in seconds (s) and Energy in Joules (J)	62
3.18	Band and multiple diagonal matrices	70
3.19	Double precision execution time of band and multiple diagonal in milliseconds upon k for CSR format: matrix of size $h = 10,000$. And Alinea ELL SpMV.	70
3.20	Double precision execution time of band and multiple diagonal in milliseconds upon k for CSR format: matrix of size $h = 20,000$	71
3.21	Double precision execution time of band and multiple diagonal in milliseconds upon k for CSR format: matrix of size $h = 100,000$. And Alinea ELL SpMV.	71
3.22	Double precision execution time of band and multiple diagonal in milliseconds upon gridification for CSR format: matrix of size $h = 10,000$	72
3.23	Double precision execution time of band and multiple diagonal in milliseconds upon gridification for CSR format: matrix of size $h = 20,000$	72
3.24	Double precision execution time of band and multiple diagonal in milliseconds upon gridification for CSR format: matrix of size $h = 100,000$	72
3.25	Chicxulub crater in the Yucatan Peninsula	73
3.26	FE mesh associated with the <i>gravitational potential equation</i>	73
3.27	SpMV CSR of the gravitational potential matrices: time in milliseconds (ms) and energy in Joules (J)	76
3.28	Audi 3D, $h = (0.133425, 0.066604, 0.033289, 0.016643)$	77
3.29	Twingo 3D, $h = (0.077866, 0.038791, 0.019379)$	77
3.30	CAD and mesh of the <i>O</i> -ring	79
3.31	Running times of the sparse matrix-vector product (SpMV) in CSR format (μs)	80
3.32	Conjugate Gradient (CG) CSR with gravitational potential matrices: Time in seconds (s) and Energy in Joules (J)	91
3.33	Illustration of the obtained solution for the Audi 3D, with $h = 0.033289$ (see Figure 3.28, Page 77)	93
3.34	CG execution times in seconds with <i>O</i> -ring matrices and zoom at small sizes	94
3.35	Diagram of the BLAS level-1	95
3.36	Diagram of the available matrix formats	97
3.37	Diagram of the BLAS level-2	99
3.38	Diagram of the BLAS level-3	99
3.39	Diagram of the solvers	101
3.40	Diagram of the iterative Krylov Solvers (1)	101
3.41	Diagram of iterative Krylov Solvers (2)	102
3.42	Diagram of the stationary iterative Solvers (3)	102
3.43	Diagram of the stationary iterative Solvers (4)	102
4.1	Example of the execution scheme of the parallel iterative “ <i>Synchronous Iterations and Synchronous Communications (SISC)</i> ” with 2 processors	107
4.2	Example of the execution scheme of another variant of the parallel iterative “ <i>SISC</i> ” with 2 processors	107

4.3	Example of the execution scheme of the parallel iterative “ <i>Synchronous Iterations and Asynchronous Communications (SIAC)</i> ” with 2 processors	108
4.4	Example of the execution scheme of the parallel iterative “ <i>Synchronous Iterations and Asynchronous Communications (SIAC)</i> ” with flexible send, with 2 processors	108
4.5	Example of the execution scheme of the parallel iterative “ <i>Synchronous Iterations and Asynchronous Communications (AIAC)</i> ” with 2 processors	109
4.6	Example of the execution scheme of the parallel iterative “ <i>Synchronous Iterations and Asynchronous Communications (AIAC)</i> ” with flexible send and receive, with 2 processors	109
4.7	Example of band-row splitting of a matrix	113
4.8	Send/Recv ordering of the processor p	115
4.9	Example of the band-column and block-diagonal splitting of a matrix	116
4.10	Example of non-zero pattern of a sparse matrix A	117
4.11	Notation of a directed graph and an undirected graph	118
4.12	Example of directed graph and its corresponding adjacency matrix	118
4.13	Example of undirected graph and its corresponding adjacency matrix	119
4.14	An example of the CSR format for storing sparse graph given in Figure 4.13(a)	120
4.15	Direct partitioning by METIS (API C/C++) using CSR structure of the graph of the matrix given in Figure 4.13(a)	121
4.16	Notation of the line-graph with $e_{(1)} \in \mathcal{E}$ and $e_{(1,2)} \in \mathcal{E}$. Note that the directed edge $\hat{e}_{e_{(1,2)}, e_{(1,2)}}$ is for the illustration.	122
4.17	Line-graph $\mathcal{L}(\mathcal{G})$ of the undirected graph given in Figure 4.13(a)	122
4.18	Adjacency matrix of the line-graph described in Figure 4.17 and the corresponding non-zero pattern	123
4.19	The CSR format for storing the sparse line-graph given in Figure 4.17	123
4.20	METIS (API C/C++) 2-partitioning and 3-partitioning of the line-graph described in Figure 4.17 using CSR structure given in Figure 4.19	123
4.21	Sub-domains obtained from the METIS (API C/C++) 2-coloring and 3-coloring of the line-graph given in Figure 4.20. n_l (n_g) stands for n_l : local node number, n_g : global node number	124
4.22	Sub-domain matrices of the 2-partitioning described in Figure 4.21(a)	126
4.23	Sub-domain matrices of the 3-partitioning described in Figure 4.21(b)	126
4.24	Design of the Jacobi algorithm	137
5.1	Splitting into two sub-domains	143
5.2	Sub-structure interface description	144
5.3	Speed-up upon the number of processes	163
5.4	Speed-up upon the parallel portion of an algorithm with $2^1, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7, 2^8$ processes	163
5.5	Speed-up upon the parallel portion of an algorithm with $2^5, 2^6, 2^7, 2^8, 2^9, 2^{10}, 2^{11}, 2^{12}$ processes	164
5.6	Global efficiency with 16 (a) and 128 (b) processes upon parallel proportion μ and parallel local efficiency λ	164
5.7	Global efficiency with 256 (a) and 512 (b) processes upon parallel proportion μ and parallel local efficiency λ	164

5.8	Example of the mesh with 4 layers	175
5.9	Finite element mesh examples of the cube	177
5.10	Computer-Aided Design (CAD) model of the Royaumont church (exterior view of the architecture)	179
5.11	CAD model of the Royaumont church (interior view of the architecture)	179
5.12	Finite element mesh examples of the church	180
5.13	Number of nodes (# of nodes) in each sub-domain for 32 and 64-partitioning of the matrix “church-484507”	181
5.14	Number of interface nodes (# of interface nodes) in each sub-domain for 32 and 64-partitioning of the matrix “church-484507”	182
5.15	Number of neighboring (# of neighboring) in each sub-domain for 32 and 64-partitioning of the matrix “church-484507”	182
5.16	Number of nodes (# of nodes) in each sub-domain for the 32 and 64-partitioning of the matrix “lupcuf_cube-274625”	182
5.17	Number of interface nodes (# of interface nodes) in each sub-domain for the 32 and 64-partitioning of the matrix “lupcuf_cube-274625”	183
5.18	Number of neighboring (# of neighboring) in each sub-domain for the 32 and 64-partitioning of the matrix “lupcuf_cube-274625”	183
5.19	Projection of the 32- and 64-partitioning of the matrix “church-484507” into the associated mesh given in Figure 5.12	183
5.20	Projection of the 32- and 64-partitioning of the matrix “lupcuf_cube-274624” into the associated mesh given in Figure 5.9(b)	184
5.21	Proportion of <i>communication latencies</i> : $t_l(m, p) = t_s + m \times t_t + \lfloor \frac{m}{p} \rfloor \times t_p$	186
5.22	Theoretical speed-up $\mathcal{S}^{para seq}$ and $\mathcal{S}^{async sync}$ of sub-structuring Jacobi with <i>luf_cube-35937</i>	187
5.23	Theoretical speed-up $\mathcal{S}^{para seq}$ and $\mathcal{S}^{async sync}$ of band-row Jacobi with <i>luf_cube-35937</i>	188
5.24	Theoretical speed-up $\mathcal{S}^{async seq}$ and $\mathcal{S}^{async sync}$ upon δ of sub-structuring with <i>luf_cube-35937</i>	188
5.25	Theoretical speed-up $\mathcal{S}^{para seq}$ and $\mathcal{S}^{async sync}$ of the sub-structuring Jacobi with <i>luf_cube-274625</i>	189
5.26	Theoretical speed-up $\mathcal{S}^{async seq}$ and $\mathcal{S}^{async sync}$ upon δ of sub-structuring with <i>luf_cube-274625</i>	190
5.27	Sub-structuring theoretical speed-up $\mathcal{S}^{sync seq}$ and $\mathcal{S}^{async seq}$ upon degree of freedoms	190
5.28	Simulation of chaotic iterations with the sub-structuring Jacobi method for <i>luf_cube-274625</i> with 48 processors	196
5.29	Number of iterations of the sub-structuring Jacobi method of the <i>luf_cube-274625</i> test case with 48 processors where the iterations of processor #7 are penalized from the p_a^{th} iteration during p_d iterations	196
5.30	Simulation of chaotic iterations with the sub-structuring Jacobi method for <i>gravi_hexas100x100x6_1</i> with 48 processors	197

5.31	Number of iterations of the sub-structuring Jacobi method of the <i>gravi_hexas100x100x6_1</i> test case with 48 processors where the iterations of processor #7 are penalized from the p_a^{th} iteration during p_d iterations	197
5.32	Number of iterations of the sub-structuring Jacobi method of the <i>luf_cube-274625</i> and <i>gravi_hexas100x100x6_1</i> tests cases with 48 processors with $s = 1$	198
5.33	Number of iterations of the sub-structuring Jacobi method with <i>luf_cube-274625</i> regarding the number of processors (8, 16, 24, 32, 48, 56, 64) and breakdown points ($k = n \times \frac{N[s]}{10}$, $n = \{1, \dots, 10\}$)	199
5.34	Number of iterations of the sub-structuring Jacobi method with <i>gravi_hexas100x100x6_1</i> regarding the number of processors and breakdown points	199
6.1	Non-overlapping domain decomposition, $\Omega = \Omega_1 \cup \Omega_2$	203
6.2	Convergence rate isolines of the Schwarz algorithm with zeroth order optimized transmission conditions	205
6.3	Convergence rate isolines of the Schwarz algorithm with second order optimized transmission conditions	206
6.4	Fourier convergence rate of the Schwarz algorithm with zeroth order optimized transmission conditions	206
6.5	Fourier convergence rate of the Schwarz algorithm with second order optimized transmission conditions	206
6.6	Perspective view of the main structural features and gravity anomaly map of the Chicxulub impact crater	207

List of Tables

2.1	Nvidia/AMD: GPGPU nomenclature	17
2.2	Double precision data transfers from the CPU to the GPU	33
2.3	Double precision addition of vectors (DAXPY) and element wise product (DXMY)	34
2.4	Linear prediction of DAXPY	34
2.5	Linear prediction of element wise product	34
2.6	MRG/LISA hybrid computing clusters	37
3.1	Double precision kernel execution time in milliseconds	60
3.2	Alinea double precision Scale (DSCAL) operation	60
3.3	Alinea double precision addition of vectors (DAXPY)	60
3.4	Alinea double precision element wise product (DXMY)	61
3.5	Alinea double precision dot product (DDOT)	61
3.6	Alinea complex double precision scale (ZSCAL) operation	61
3.7	Alinea complex double precision addition of vectors (ZAXPY)	62
3.8	Alinea complex double precision element wise product (ZXMY)	62
3.9	Alinea complex double precision dot product (ZDOT)	62
3.10	Linear prediction of the dot product	63
3.11	Sketches of engineering matrices from University of Florida collection	68
3.12	Double precision running time of sparse matrix-vector product in milliseconds (ms)	68
3.13	GPU execution time of the sparse matrix-vector product for CSR format in milliseconds (ms) with auto-tuning	69
3.14	Sketches of matrices from the 3D FE discretization of the gravitational potential equation	75
3.15	Example pattern of matrices from the 3D FE discretization of the gravitational equation	75
3.16	Double precision SpMV CSR of the gravitational potential matrices	75
3.17	Linear prediction of the SpMV CSR of the gravitational potential matrices	76
3.18	Sketches of the Audi 3D FE acoustic matrices	78
3.19	Sketches of the Twingo 3D FE acoustic matrices	79
3.20	Alinea complex double precision CSR SpMV execution time in milliseconds (ms)	79
3.21	Execution time of the P-CG (seconds)	88
3.22	Running time of the P-CG (seconds) with auto-tuning	88
3.23	Number of iterations and execution times (in seconds) of P-GCR, P-Bi-CGCR and P-TFQMR Alinea algorithms for CSR format	89
3.24	Execution time of the P-Bi-CGSTAB (seconds)	89
3.25	Execution time of the P-Bi-CGSTAB(l) (seconds) for CSR format	90

3.26	Double precision CSR preconditioned Krylov methods: P-Bi-CGSTAB, P-Bi-CGCR, P-TFQMR	90
3.27	Double precision CG CSR with gravitational potential matrices	91
3.28	Predictions of computational time and energy consumption of CG solver with gravitational potential matrices	91
3.29	Audi3D - Speed-up of the CSR acoustic solver (complex number arithmetics with double precision): P-Bi-CGSTAB, P-Bi-CGSTAB(8) and P-TFQMR	92
3.30	Twingo3D - Speed-up of the CSR acoustic solver (complex number arithmetics with double precision): P-Bi-CGSTAB, P-Bi-CGSTAB(8) and P-TFQMR	92
3.31	Alinea (<i>Advanced LINEar Algebra</i>) main levels	95
3.32	Alinea specifications and diagram of the different levels	95
4.1	List of receiving and sending dependencies of the splitting described in Figure 4.7(b).	115
4.2	Execution time used for partitioning (s)	127
5.1	Hydraulic conductivity ($m.s^{-1}$), μ , depending on the environment of the medium (material)	175
5.2	Statistics of the original and projected meshes with 4 layers	176
5.3	Statistics of the level #0 and #1 of the meshes with 4 layers: <i>muluf_hexas100x100x6_0</i> (level #1) and <i>muluf_hexas100x100x6_1</i> (level #2)	176
5.4	Statistics of the academic cube meshes: <i>luf_cube-35937</i> and <i>luf_cube-274624</i>	176
5.5	Statistics of the meshes used in the discretization of $\Delta\psi + \eta\psi = f$: <i>lupcuf_cube-35937</i> , <i>lupcuf_cube-274624</i> and <i>church-484507</i>	179
5.6	Sketches of matrices obtained by the finite element discretization of the presented test cases	181
5.7	Double precision CSR Jacobi method for $\varepsilon = 10^{-6}$, $\varepsilon = 10^{-8}$ and $\varepsilon = 10^{-10}$. . .	185
5.8	Double precision CSR CG method for $\varepsilon = 10^{-6}$, $\varepsilon = 10^{-8}$ and $\varepsilon = 10^{-10}$. . .	185
5.9	Double precision CSR preconditioned CG (P-CG) method for $\varepsilon = 10^{-6}$, $\varepsilon = 10^{-8}$ and $\varepsilon = 10^{-10}$	185
5.10	Degree of freedoms (dof) and non-zero values of the matrix of each sub-domain	191
5.11	Execution time for parallel sub-structuring P-CG (seconds) for CSR format	192
5.12	Speed-up of parallel sub-structuring P-CG (seconds) for CSR format	192
5.13	<i>luf_cube-35937</i> : Numerical results for the Synchronous (BANDROW, BANDROW-OP and SUB-STRUCTURING) and Asynchronous (SUB-STRUCTURING) Jacobi method	193
5.14	<i>luf_cube-274625</i> : Numerical results for the Synchronous (BANDROW, BANDROW-OP and SUB-STRUCTURING) and Asynchronous (SUB-STRUCTURING) Jacobi method	194
5.15	<i>gravi_hexas100x100x6_0</i> : Numerical results for the Synchronous (BANDROW, BANDROW-OP and SUB-STRUCTURING) and Asynchronous (SUB-STRUCTURING) Jacobi method	194
5.16	<i>gravi_hexas100x100x6_1</i> : Numerical results for the Synchronous (BANDROW, BANDROW-OP and SUB-STRUCTURING) and Asynchronous (SUB-STRUCTURING) Jacobi method	195

6.1 Optimized coefficients obtained from the *CMA-ES* algorithm 207

6.2 Comparison of stochastic-based optimized Schwarz domain decomposition
method on CPU and GPU 208

List of Algorithms

2.1	Addition of vectors on CPU	17
2.2	Addition of vectors on GPU (thread function)	17
3.1	Algorithm of the stationary iterative methods	49
3.2	Matrix-vector multiplication with the COO storage format	64
3.3	Matrix-vector multiplication with the ELL storage format	64
3.4	Matrix-vector multiplication with the CSR storage format	64
3.5	Matrix-vector multiplication with the CSC storage format	64
3.6	Matrix-vector multiplication with the HYB storage format	65
3.7	Preconditioned Conjugate Gradient (P-CG) method	83
3.8	Preconditioned Conjugate Residual (P-CR) method	83
3.9	Precond. Generalized Conjugate Residual (P-GCR) method	84
3.10	Preconditioned Stabilized BiConjugate Gradient (Bi-CGSTAB)	84
4.1	Band-row partitioning for the $p - th$ band, with CSR matrix	113
4.2	Band-column partitioning for the $p - th$ band, with CSR matrix	114
4.3	Algorithm of sequential iterative methods	130
4.4	Jacobi method: element updates version	138
4.5	Jacobi method: vectorial version	138
5.1	Construct inner buffer and send to neighboring	145
5.2	Receiving interface results and updating interface equations	145
5.3	Algorithm of the iterative parallel synchronous sub-structuring methods	151
5.4	Algorithm of the parallel iterative asynchronous sub-structuring methods	157

List of Listings

2.1	Kernel parameters and call	18
2.2	Daxpy (Double-precision real Alpha X Plus Y) on one-dimensional grid . . .	18
2.3	Daxpy (Double-precision real Alpha X Plus Y) version in OpenCL	18
2.4	Basic kernel call	18
2.5	Kernel call with basic self-threading computation with a number of threads per block equals to 100	19
2.6	Daxpy on two-dimensional grid	19
2.7	Grid features computation for one-dimensional grid fully used	19
2.8	Grid features computation for two-dimensional grid	20
2.9	Complex number in memory	23
2.10	Complex class	24
2.11	Complex free functions	24
3.1	Summing two vectors, using on two-dimensional grid	54
3.2	Multiplying two vectors, using on two-dimensional grid	55
3.3	Scaling vector, using on two-dimensional grid	56
3.4	Vector usage	96
3.5	GpuDevice usage	96
3.6	CPU real vector storage	96
3.7	GPU real vector storage	96
3.8	CPU complex vector storage	96
3.9	GPU complex vector storage	96
3.10	GPU usage, BLAS1 operations	97
3.11	GPU usage, BLAS1 (Operator)	97
3.12	Matrix usage	97
3.13	Matrix constructors (CPU)	98
3.14	Matrix constructors (GPU)	98
3.15	Matrix direct allocation (CPU)	98
3.16	Matrix direct allocation (GPU)	98
3.17	Matrix explicit allocation (CPU)	99
3.18	Matrix explicit allocation (GPU)	99
3.19	Matrix copy (CPU↔GPU)	99
3.20	Matrix conversion (only CPU)	99
3.21	BLAS 2: computation on CPU	100
3.22	BLAS 2: computation on GPU	100
3.23	BLAS 3: simple usage on CPU	100

3.24	BLAS 3: simple usage on GPU	100
3.25	Solver usage	101
3.27	Direct solver simple usage (CPU)	101
3.28	CPU iterative solvers usage	103
3.29	CPU iterative solvers usage	103

General Introduction

” *The beautiful thing about learning is nobody can take it away from you.*

— **B. B. King**
(Musician)

This chapter introduces my thesis. In this thesis, we focus on different ways to efficiently solve large and sparse linear systems, arising from the discretization of numerical methods. Among these methods, we choose the finite element method. We have designed parallel iterative algorithms for solving these systems in a context of heterogeneous multi-core-GPU systems. In the first section, the motivations and backgrounds of this work are presented. We introduce the problem, which interests us, *i.e.*, the resolution of large-scale sparse linear systems. Then, we present briefly the different class of parallel architectures. The interest of GPU Computing is highlighted and the necessity of green computing is justified. The use of domain decomposition methods, such as the sub-structuring method and the Schwarz method, is then vindicated for solving these systems in parallel. The first section concludes with the presentation of different projects I have partaken in. Finally, the main contributions of this thesis are highlighted and the plan of the manuscript is presented.

1.1 Motivation and Background

Undoubtedly, parallelism is the future of computing. Advances in computational power have led to many developments in science and its large application areas. High Performance Computing (HPC) uses very large-scale computers to solve some of the world's biggest computational problems.

1.1.1 Linear system solvers

For applied sciences and engineering problems, a system of partial differential equations (PDEs) is used as a fundamental physical and mathematical model, which is solved numerically. Nowadays, several physical parameters can be taken into account and different physical models can be coupled and then the PDE system becomes more and more complex. To obtain numerical solutions of PDEs, discretization methods (finite difference/volume/element methods) are applied and linear systems with very large sparse matrices are obtained after some linearization process, if necessary. In scientific computing, solving large linear systems is often the most expensive part, both in terms of memory space and computation time. There are several methodologies to solve this type of large linear systems, *e.g.*, direct methods, iterative methods and a combination of those. However, the choice of method to solve the linear system is often driven by the properties of the matrix related to the system. The choice also depends on the speed of the method and the desired accuracy for the resolution. The size of the system can also be a determining factor in the choice of the method. Indeed, the size of the system may further slow the resolution process. Due to the hypothesis of a large linear system, these methods

must be simple to parallelize in order to better harness the computing power and to better manage the memory capacity of parallel computing platforms. For large-size linear systems, iterative methods are more appropriate. While fast solvers are frequently associated with iterative solvers, for special problems, a direct solver can be competitive. Iterative algorithms constitute a most suitable solution for the resolution of many problems. These methods have the characteristic to perform several iterations in order to refine the required solution. Krylov subspace methods are the most representative of iterative methods, where an approximate solution is updated repeatedly using products of sparse matrix and vectors. Matrix computing arises in the solution of almost all linear systems of equations. In high-resolution simulations, a method can reach its applicability limits quickly and hence there is a constant demand for fast matrix solvers. Because of their size, these matrices cannot be stored as a simple set of vectors. Instead, taking advantage of their great amount of zero values, we focus on storing matrices in compressed formats, *i.e.*, only non-zero elements are allocated. In this thesis, we pay close attention to storing these sparse matrices in memory more efficiently by taking into account their pattern of non-zero values. Parallel and GPU computing are the key to successful fast solvers. It is the intention of this thesis to present and investigate efficient solutions to solve linear systems rapidly.

1.1.2 Parallel architectures

Parallel computing is a set of hardware and software techniques allowing the simultaneous execution of sequences of independent instructions on different processes and/or cores. The set of hardware and software techniques consists of various architectures of parallel computers and various models of parallel programming.

Parallel computing has two objectives: accelerate the execution of a program by distributing work and executing a big problem with more material resources (memory, storage capacity, processor, etc.). A parallel computer can be: a multi-core processor having at least two physical computing units on the same chip or a supercomputer, which gathers the components of several computers (processors and memories) in only one machine, or a distributed platform made up of several independent machines, homogeneous or heterogeneous, connected with one another by a communication network. In the jargon, the supercomputer is more often called a cluster.

Classifications of parallel architectures In literature, there are several classifications of parallel architectures based on various criteria. In this thesis, we present the most largely used classification in parallel computing: the *taxonomy of Flynn* [1]. This classifies parallel architectures in several categories, according to whether great volumes of data are processed in parallel (or not) or whether a large number of instructions are being carried out at the same time (or not). In the taxonomy of Flynn, one distinguishes four classes: *SISD* (*Single Instruction, Single Data*), *SIMD* (*Single Instruction, Multiple Data*), *MISD* (*Multiple Instruction, Single Data*) and *MIMD* (*Multiple Instruction, Multiple Data*). The *SISD* class represents most conventional computing equipment which has only one computing unit. The sequential computer (traditional uniprocessor machine) can process only one operation at a time. The *SIMD* class corresponds to computers which have a large number of computing units, *e.g.*, array processors or GPUs. The computer can exploit multiple data against a single instruction to perform operations which may be naturally parallelized. At each clock cycle, all the processors

of a SIMD computer carry out the same instruction on different data simultaneously. The *MISD* class corresponds to the parallel machines which perform several instructions, simultaneously, on the same data. These architectures are uncommon and are generally used for fault tolerance. *MIMD* represents the most used class in the taxonomy of Flynn. In this architecture, multiple autonomous processors simultaneously execute different instructions on different data, *e.g.*, distributed systems. They exploit a single shared memory space or a distributed memory space. Computers using *MIMD* have a number of processors that function asynchronously and independently.

Memory model In parallel computing, we have two main models of memory management: shared memory and distributed memory. These two models help define the mode of data access to the data of the other cooperating processors in a parallel computation for a given process. In a shared memory model, the processors are all connected to a "globally available" memory. All the processors have a direct access to the same physical memory space via powerful links of communication (*e.g.*, cores of a single machine). The second memory model is generally used in parallel computers with distributed resources, *e.g.*, a grid or cluster. In this model, each processor has its own individual memory location. Each processor has no direct knowledge of other processors' memory. However, access to the data of the distributed memories is ensured by message passing between cooperating processors through a communication network. In this thesis, we focus on algorithms in parallel distributed-memory computing.

1.1.3 GPU Computing

For years, increases in HPC performance have been delivered through increased clock speeds, larger, faster memories and higher bandwidth and lower latency interconnects. Nowadays, multi-core and many-core platforms lead the computer industry, forcing developers to adopt new programming paradigms in order to fully harness their computing capabilities. Graphics Processing Units (GPUs) are one representative of many-core architectures, and certainly the most widespread. Comparing the evolution of Central Processing Units (CPUs) to that of GPUs, one can talk about a revolution in the case of GPUs, looking at it from the perspective of the reached performance and advances in multi-core designs. Many consumer personal computers are equipped with powerful Graphics Processing Units (GPUs), which can be harnessed for general-purpose computations. Currently, the computational efficiency of GPUs reaches the TeraFlops (Tflops) for single precision computations, and roughly half of that for double precision computations, all this for one single graphics card. Recent developments resulting in easier programmable access to GPU devices, have made it possible to use them for general scientific numerical simulations. The performance reached by add-on graphics cards has attracted the attention of many scientists, with the underlying idea of utilizing GPU devices for tasks other than graphics related computations. Gaining the most from the GPU programming paradigm requires understanding of GPU architectures.

In the early 2000s, this gave birth to the technology known as GPGPU (General-Purpose computation on GPU). As graphics hardware has become more sophisticated, more control of this parallel infrastructure has been given to the programmer. The real boost to this technology was the introduction by Nvidia of their programming paradigm and tools referred as CUDA (*Compute Unified Device Architecture*) [2]. CUDA and OpenCL [3] (*Open Computing Language*) provide tools based on extensions to the most popular programming languages

(most notably C/C++/Fortran), which makes computing resources of GPU devices easily accessible for general tasks. The CUDA programming language has made general-purpose programming on the GPU more accessible and enabled researchers to achieve good performance [4] [5] [6] [7]. GPU applications are used to solve critical, grand challenge issues, including: biomolecular systems, fusion energy, the virtual physiological human, numerical weather prediction and engineering.

Most of the top supercomputers include GPUs, as we can observe in the list of the world's 500 fastest supercomputers [8]. Besides this list, there is another top500 list of supercomputers which are power efficient, the Green 500 [9] [10] [11]. In the latter, all top-10 supercomputers consist of CPU and GPU hybrid architecture.

1.1.4 Green Computing with GPU

Nowadays, we need to address another question, energy related, *i.e.*, “*How much energy is consumed by the application?*”. For high-performance computing (HPC) hardware, the appropriate answer gives a compromise between computational time and energy consumption. Numerical algorithms have already faced a similar question, *i.e.*, “*How much memory is necessary to solve the linear system?*”. This question is also very important because memory limitation is one of the physical constraints of numerical simulation. If the memory falls short, we have to give up the simulation or satisfy ourselves with very slow out-of-core execution. Therefore, algorithms have been optimized in both directions to minimize computational time and memory usage. Unfortunately, there is no way to answer the new question on energy consumption only from a software point of view. To the best of our knowledge, there is no established way to measure real energy consumption for each application. Therefore, we aim to design an experimental protocol to measure GPU energy consumption accurately for fundamental and basic arithmetic operations and some heavy operations involved in numerical linear algebra with sparse matrix. This allows us to build an accurate energy consumption prediction model of the GPU. This methodology could suggest a “new vision of high-performance computing” and answer some of the questions outlined in references [12] [13] [14] [11], which highlight the importance of energy consumption when using GPUs. The computing power and power/energetic consumption ratio of the GPU being superior to standard CPU, the use of GPU represents an opportunity to have more power for a lower energetic cost.

1.1.5 Parallel algorithms

Parallel computations are fundamental and ubiquitous in numerical analysis and large application areas, when we deal with the problem of large-size. Distributed computing constantly gains in importance and has become an important tool in common scientific research work. Recall that a sequential computation consists in performing a program by only one process, sequentially, *i.e.*, only one instruction is carried out at the same time. When computation is done in parallel, the simplest solution consists in synchronizing the processors during iterations. In that particular field of parallel programming, the commonly used model is the synchronous message passing. In the synchronous case, communications are strongly penalizing the overall performance of the application. Indeed, they often involve large idle periods, especially when the processors are heterogeneous. Another solution consists in using asynchronous iterations in order to avoid the waiting points. However, some properties must be verified to guarantee

convergence. These algorithms are called asynchronous iterative algorithms and are applicable to a large class of scientific applications. They allow a significant reduction of the computing time compared to the synchronous methods, due to the suppression of synchronization between the iterations. The performance of algorithms strongly depends on the management of interprocessor communication.

The parallel iterative methods for solving these linear systems are categorized into three main classes, depending on the nature of the scheme (synchronous or asynchronous) of iterations and communications. Parallel algorithms require that a large number of parameters, such as the number and the typology of the processors, be taken into account. Therefore, in parallel processing, the step of data distribution is crucial and strongly impacts the performance of algorithms. In this thesis, we propose a study of the optimization process of parallel algorithms run on modern heterogeneous multi-core-GPU architectures. Different optimization schemes are proposed for overlapping computations with communications.

Domain decomposition methods The domain decomposition methods allow us to define powerful algorithms adapted to the parallel machines. These methods consist in splitting the domain of resolution of an initial problem into sub-domains. The solution to a mathematical problem of “large” size can thus be obtained by the resolution of the associated “small” sub-problems. This technique helps deal with problems which no computer has sufficient memory to solve alone. There are two main classes of decomposition methods: overlapping (*e.g.*, Schwarz methods [15]) and non-overlapping (*e.g.*, sub-structuring methods).

In this thesis, we are interested in non-overlapping domain decomposition methods, applicable to the resolution of linear systems arising from a discretization by finite elements.

Sub-structuring methods In this thesis, we pay special attention to sub-structuring methods. The sub-structuring method is the precursor of non-overlapping domain decomposition methods [16] [17] [18] [19]. The sub-structuring method is based on decomposition of the original structure into several sub-structures. The term *sub-structuring* is a way to describe the general method allowing to decompose splitting among sub-domains sharing a common interface. This method is most often used as a way to reduce the number of unknowns in the linear system by eliminating the interior unknowns.

1.1.6 Project support

My thesis has contributed to various projects and to the label *CUDA Research Center*.

Project OpenGPU The OpenGPU [20] project started in 2010. It has brought together a set of French companies and French academical research centers around the vision of a lack of tool sets, framework and expertise for teaming up the GPGPU. My thesis brings expertise to clear the vision of GPGPU benefits in terms of efficiency and power consumption. We have put forward a new library that helps take advantage of the power of GPUs and gives a new vision of high-performance computing for solving large-size linear systems. Indeed, the library proposes a way to accelerate the computation of linear algebra operations by taking into account their features and the hardware configuration.

Project CRESTA CRESTA [21] (Collaborative Research into Exascale Systemware, Tools & Applications) has brought together four of Europe’s leading supercomputing centers with

one of the world's major equipment vendors, two of Europe's leading programming tools providers and six application and problem owners to explore how the Exascale challenge can be met. The CRESTA project was completed at the end of 2014 (*source: CRESTA website [21]*). The CRESTA project had two integrated strands: one focused on enabling a key set of co-design applications for exascale, the other focused on building and exploring appropriate systemware for exascale platforms. In this project, we propose and analyze parallel algorithms in a context of heterogeneous multi-core environments that should pave the way to exascale hybrid methods.

Label CUDA Research Center In 2013, our team led by Pr. Frédéric Magoulès, received the prize CUDA Research Center, in recognition of the work undertaken in the field of the applied mathematics and scientific computing, using graphics processors.

1.2 Organization of this thesis and contributions

The organization of the thesis and the main contributions are described in the following.

Chapter 2

In order to understand the popularity of GPUs, we first need to travel back in time, in Section 2.2 to explore the history and evolution of the Graphics Processing Unit. Next, Section 2.3 introduces General-Purpose GPU Computing, inter alia, the programming tools that offer access to the power of graphics cards. The GPU programming model and hardware configuration issues are discussed in Section 2.4. Section 2.5 addresses the question of how much energy is consumed for a real numerical simulation running on a Graphics Processing Unit (GPU). In the high-performance computing field, an experimental protocol has been proposed. In Section 2.6, we present all benchmark workstations and clusters used in this dissertation. Finally, Section 2.7 concludes the chapter.

My first contribution to this chapter focuses on a gridification technique of the CUDA grid to help the algorithms take greater advantage of the power of the graphics card.

My second contribution is to address the question of how much energy is consumed for a real numerical simulation running on a GPU. In the high-performance computing field, an experimental protocol has been proposed.

We also propose a way to easily handle complex number arithmetics on GPU for all precisions, using advanced C++ template structures.

Chapter 3

First, an introduction to linear system solvers is given in Section 3.2, which includes matrix storage formats and a description of two basic classes of methods for solving linear systems, namely direct and iterative methods, and especially the iterative Krylov methods. For convenience and thoroughness, Section 3.3 gives an overview of existing scientific libraries targeted for GPU. Section 3.4 and Section 3.5 respectively discuss the best way to efficiently perform BLAS level-1 linear algebra operations and sparse matrix-vector multiplication (SpMV) on GPU. Hints for heterogeneous multi-core+GPU are given. Section 3.6 describes how solvers, particularly iterative Krylov methods are implemented on GPU and multi-core systems, and

how the performance of the basic linear algebra operations influence the overall performance of these methods. These sections also describe, evaluate, analyze and present the optimizations we have developed. Numerical experiments performed on a set of large size sparse matrices arising from diverse engineering and scientific problems, have clearly shown the value of using GPU technology to solve large sparse systems of linear equations, and its robustness and accuracy compared to existing libraries such as Cusp. Section 3.7 presents an overview of the library we have implemented, Alinea, and its competitive usage. Section 3.8 concludes the chapter.

My contribution in this chapter has been to develop, implement and test innovative algorithms for computing linear algebra operations on GPU and their use together within solvers.

My second contribution is to build an accurate energy consumption prediction model of the GPU. This helps predict energy needs of a program, and then allowing a better task scheduling with a given computational time.

Another contribution is also the proposal of “An Advanced Linear Algebra Library for Massively Parallel Computations on Graphics Processing Units”, aimed at providing a new vision of GPU computing that helps harness the power of GPUs easily, without the programming complexity of classical APIs such as CUDA and OpenCL.

Chapter 4

For convenience and thoroughness, Section 4.2 gives an overview of the different classes of parallel iterative methods for readers not familiar with parallel and distributed computing. Given that the first step is also necessary for optimization, in parallel processing consists in distributing the data on the cluster processors. In Section 4.3 we shortly describe how data are distributed among processors for different splitting strategies, in particular using a smart process to partition a sparse matrix for parallel sub-structuring methods. Section 4.4 is dedicated to a state-of-the-art theory of parallel synchronous and asynchronous iterations. We notably introduce the classical model and models with total communication.

My contribution in this chapter was to develop and implement code for partitioning data among processors in order to control and adapt the splitting to the studied parallel algorithms. This choice allowed freedom in the implementation of parallel algorithms as we like.

I have proposed a way to correctly partition a sparse matrix for sub-structuring methods.

Chapter 5

The first section, Section 5.2, describes and discusses sub-structuring methods. They are first presented in the synchronous case. Then a particular attention is paid to the asynchronous case. In this section, we give a proof of the convergence of asynchronous sub-structuring methods. In Section 5.2, we also evaluate the behavior of parallel algorithms in terms of fault-tolerance and iteration penalization. Section 5.3 studies the theoretical speed-up of fully parallelizable iterative methods with synchronous and asynchronous iterations. This section aims to analyze the behavior of asynchronous algorithms. *In Section 5.3, my contribution resides in the theorems for the theoretical speed-up of synchronous and asynchronous parallelizable*

iterative methods. Section 5.4 reports numerical results and shows the advantage of parallel sub-structuring methods, particularly when they are associated with asynchronous iterations with enough processes. The numerical results highlight the effectiveness and robustness of these methods on a platform of multi-core+GPUs. Section 5.5 concludes the chapter.

My contribution to this chapter has been to develop, implement and test innovative algorithms of parallel synchronous and asynchronous sub-structuring methods.

My contribution in also to integrate GPU Computing in parallel algorithms with synchronous and asynchronous iterations.

We give proof of the convergence of asynchronous sub-structuring methods.

We also give some theorems concerning fault-tolerance in parallel processing and we study the behavior of the algorithm when some iterations are penalized.

Chapter 6

Section 6.2 describes the optimized Schwarz method, followed in Section 6.3 by an overview of the stochastic-based technique to determine the optimized transmission conditions. Section 6.4 reports numerical results performed on a realistic test case of our Schwarz methods with optimized stochastic conditions. Finally, Section 6.5 concludes the chapter.

My contribution to this chapter is to modify high-order finite element solvers to be run on a GPU. We use optimized conditions obtained by a stochastic technique based on a CMA-ES algorithm.

I started with an existing code of domain decomposition with optimized Schwarz, developed by Frederic Magoulès *et al.* [22] I modify the sub-domain solvers in order to be run on GPUs. The implementation is based on the acceleration of the local solutions of the linear sub-systems associated with each sub-domain using GPUs.

Chapter 7

We conclude and suggest some important open problems and future research.

GPU Computing

” You can't do better design with a computer, but you can speed up your work enormously.

— Wim Crowel

(Graphic designer and typographer)

2.1 Introduction

As per the constant need to solve larger and larger numerical problems, it is not possible to neglect the opportunity that comes from the close adaptation of computational algorithms and their implementations for particular features of computing devices, *i.e.*, the characteristics and performance of available workstations and servers. In the last decade, the advances in hardware manufacturing, the decreasing cost and the spread of GPUs have attracted the attention of researchers for numerical simulations, given that for some problems, GPU-based simulations can significantly outperform the ones based on CPUs. Comparing the evolution of CPUs to that of GPUs, one can talk about a revolution in the case of GPUs, looking at it from the perspective of performance and advances in multi-core designs. Currently, the computational efficiency of GPU reaches the TeraFlops (Tflops) for single precision computations and roughly half of that for double precision computations, all this for one single graphics card. Recent developments resulting in easier programmable access to GPU devices, have made it possible to use them for general scientific numerical simulations.

Although the field of GPU-based computing matures, there are still some issues regarding communication, memory access patterns and load balancing, that are challenging and require further research. Until the beginning of this century, parallel numerical simulations were most often executed on clusters. The emerging possibility of general-purpose computations on GPU has opened the door for several new simulation approaches. This is due to the fact that GPU architecture differs significantly compared to the classical CPU. Because of their primary use for managing graphics-related calculations, GPU has fewer control units and many more ALUs (Arithmetical and Logical Units). GPU design results in inherently parallel architecture well suited to many parallel simulations. A landmark in GPU-based computing was the appearance of the *Compute Unified Device Architecture* (CUDA) prepared by *NVIDIA*. With it, the developers are given a standard tool, in a form of *C/C++* compiler extension to build parallel programs to be executed on GPUs.

In the market of graphics cards, two major constructors stand out: *NVIDIA* and *AMD*. Although this market is driven by the gaming industry, manufacturers are also struggling in the high-performance computing market since the beginning of the last decade. A simple API development is the key point of this battle. Putting GPU-based computations in a wider context can be helpful in understanding this technology. This is why in subsequent subsections we briefly discuss its history and evolution. Moreover, in order to master GPU programming one

needs to be aware of hardware architecture issues, memory layout, language programming and managing threads of execution. The GPU unit emerged as a specialized and highly parallel microprocessor designed to accelerate 2D or 3D rendering, and to make it independent from the load of the central processing unit. The appearance of add-on GPUs can be traced back to the middle 90s, but it is in the last 10 years that GPUs have become more and more programmable.

In recent years, we have witnessed that GPGPUs (General-Purpose Graphics Processing Units) were adopted as an essential computational platform in all scientific fields. For High Performance Computing (HPC) environments, the use of GPGPU is almost mandatory. The newest graphics card can achieve performance measured in TeraFlops, and the use of programming tools with APIs, based on high level language such as *C*, is probably one of the reasons for that. While offering considerable computational power, the use of GPU architectures requires redesigning most of the algorithms. Implementations and optimizations need to be done carefully, otherwise they can degrade the performance significantly. The community of programmers and graphics card manufacturers are now working on making the power of GPGPU more accessible, with special programming tools and common standards.

My first contribution to this chapter focuses on a gridification technique of the CUDA grid to help the algorithms take greater advantage of the power of the graphics card.

My second contribution is to address the question of how much energy is consumed for a real numerical simulation running on a GPU. In the high-performance computing field, an experimental protocol has been proposed.

We also propose a way to easily handle complex number arithmetics on GPU for all precisions, using advanced C++ template structures.

The chapter is structured as follows. In order to understand the popularity of GPUs, we first need to travel back in time, in Section 2.2 to explore the history and evolution of the Graphics Processing Unit. Next, Section 2.3 introduces General-Purpose GPU Computing, inter alia, the programming tools that offer access to the power of graphics cards. The GPU programming model and hardware configuration issues are discussed in Section 2.4. Section 2.5 addresses the question of how much energy is consumed for a real numerical simulation running on a Graphics Processing Unit (GPU). In the high-performance computing field, an experimental protocol has been proposed. In Section 2.6, we present all benchmark workstations and clusters used in this dissertation. Finally, Section 2.7 concludes the chapter.

Keywords — GPU, Graphics card, OpenCL, CUDA, History, Auto-tuning grid, Green computing, Energy consumption

2.2 History and evolution

GPUs were originally used for graphics computing, *e.g.*, calculations of coordinates of vertices, edges and surfaces, lighting, colors and textures with single floating point arithmetic. As mentioned above, the first 3D add-in graphics card appeared in 1995. The main driving force for its evolution was the demand from the gaming industry. The race to provide more processing power and better video quality to the end users made the graphics card reach such

a level of performance that its capabilities could no longer be ignored by the computational science community. Special purpose graphics programming languages and platforms such as *RapidMind*, *Sh*, *Cg*, *Brook* were another factor that helped spread GPU computing [23] over diverse scientific fields [24] [25]. In the early 2000s, GPU units were able to outperform some

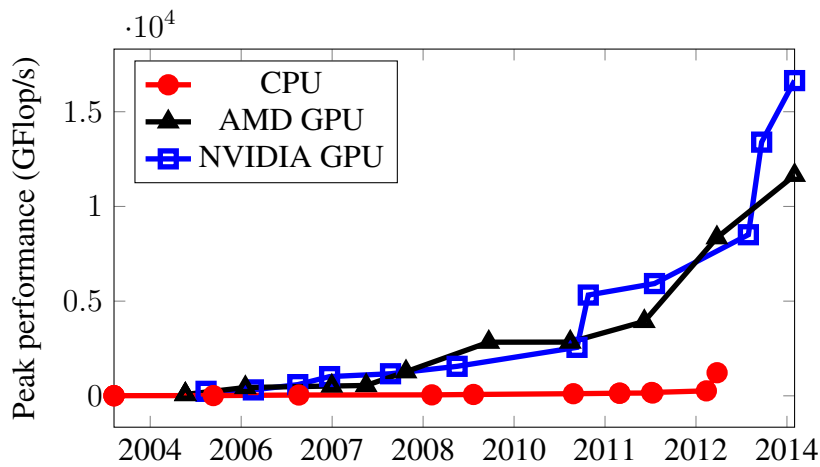


FIGURE 2.1: Evolution of the peak performance of GPUs

of the supercomputers from the previous decade. This is the time when the idea of GPGPU gained wide acceptance. After establishing the usage of GPGPU (General-Purpose computing on GPU) the hardware was installed to manage double floating point arithmetic and widely used for numerical simulation applications, see references [4] [5] [6] [7]. The introduction by *NVIDIA* of its parallel computing platform and programming model, *CUDA* (*Compute Unified Device Architecture*) [2] has set the new direction for making GPU programming more accessible and user friendly to programmers of such high-level languages as *C* or *C++*. That said, GPU programming still is a serious endeavour, especially from the perspective of portability between devices and proving calculations' correctness.

Now the GPU has become an indispensable device for HPC. The important and most attractive character of GPU comes from its developing speed of hardware components. For example, the GPU, *NVIDIA GTX275*, used during my thesis, has 200 cores running with $0.633GHz$ which achieve $126.6G Flop/s$ in double floating point, and is equipped with $127GB/sec$ – bandwidth memory. It was produced in 2009. The most recent GPU, *NVIDIA K20* has 2,496 cores running with $0.706GHz$, which achieve $1.17T Flop/s$ in double floating point, and is equipped with $208GB/sec$ – bandwidth memory. We can see that the computational speed of GPU has become about 10 times faster and memory access speed two times faster during the last four years, whereas both GPU consumes about 220W electricity. This tendency has been ongoing for nearly a decade. Figure 2.1 also shows the same tendency of computational speeds in single floating point with the other leading GPU by *AMD* in comparison to the typical CPU from *Intel*.

2.3 General-Purpose GPU Computing

The performance reached by add-on graphics cards has attracted the attention of many scientists, with the underlying idea of utilizing GPU devices for tasks other than graphics-related computations. In the early 21st century, this gave birth to the technology known as GPGPU (General-Purpose computation on GPU) as introduced in the history. As illustrated by

Figure 2.1, one can note a big difference between the peak performance of CPUs and GPUs, partly due to the inherently different architectures of these processors. This is the reason why an increasing number of researcher focuses their attention on GPU-based numerical simulations. Since the beginning [4] [5] [6] [24], the use of GPUs in high-performance computing, especially in the field of scientific computing, has demonstrated its effectiveness and robustness in terms of computation time. Much like in GPU programming, in this thesis the CPU is classified as a *host* and the GPU as a *device*. Programming on GPU requires a minimum understanding of the underlying mechanisms of the API and hardware features. Thus, it seems reasonable to give a description of GPU hardware and software. In order to better understand the attention paid to GPU Computing, we first introduce the native difference between CPU and GPU architectures. The main difference between both architectures is located at hardware level. And then we describe the issue of programming models in the next section, which includes memory access and the management of computing threads referred to as *gridification*.

2.3.1 GPU Programming languages

GPGPU-based programming has undergone substantial evolution over the past few years. The basic programming model of traditional GPGPU is *stream processing*, which is closely related to *SIMD* (Single Instruction Multiple Data). CUDA and OpenCL [3] (*Open Computing Language*) provide tools based on extensions to the most popular programming languages (most notably *C/C++/Fortran*), that make computing resources of GPU devices easily accessible for general tasks. An effective use of these tools requires an understanding of the implications of GPU hardware architecture, therefore we provide in the following a brief overview of GPU architecture.

2.3.2 Hardware architecture

The GPU device architecture stems from its role in speeding up graphic related data processing independently from the load of the central processing unit. The key point of effective processing of such data lies in specially designed memory hierarchy that allows each processor to access requested data optimally. The simplified CPU architecture contains *Arithmetical and Logical Unit* (the basic unit of computations), several memory units with multiple levels of associated cache memory and a complex control unit. Figure 2.2 shows a simplified comparison between CPU and GPU architecture. The key idea of designing GPU architecture is based on an ALU (Arithmetical and Logical Unit) structure simpler than CPU with low clock frequency, *e.g.*, 1/4 of CPU but with a very large number of ALUs, (see Figure 2.2). Tasks such as texture processing or ray tracing imply doing similar calculations

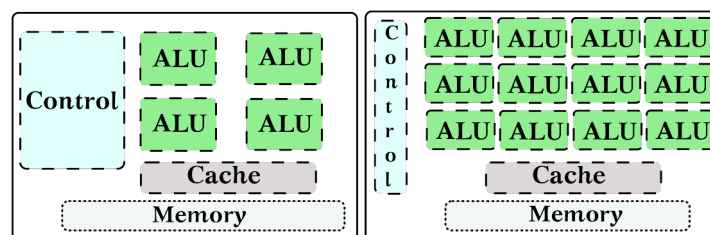


FIGURE 2.2: CPU architecture vs GPU architecture

on a large amount of independent data (Single Instruction Multiple Data). Thus the main idea behind the GPU design is to have several simple floating point processors working on a large amount of partitioned data in parallel. The way this is achieved is, in general, through a memory hierarchy allowing each processor to access needed data in a somehow optimal manner. Then it is easy to develop more massively parallel architecture, whereas most recent CPU houses only up to 12 cores. GPU has in fact become a massively parallel computing device with hundred or thousand of cores (ALUs) that operate together to massively parallel process the computing data. The other important factor accounting for the popularity of GPU computation is the software environment in which to develop computational codes.

Compute Unified Device Architecture (CUDA)

CUDA (*Compute Unified Device Architecture*) [26] is a framework for GPGPU computing developed by NVIDIA. The first version of CUDA appeared in 2007 and it is updated almost every year. Despite being proprietary, CUDA technology sets the *de facto* standard in GPGPU-driven computing, being feature rich and delivering top performance. CUDA defines both hardware and software environment with C-based programming language to manage *SIMD*-type massively parallel execution by GPU. Initially based on C language, nowadays it is also available for C++ and *Fortran*. For other languages, most notably scripting languages commonly used in programming numerical simulations, like *Matlab*, *Python* or *Haskell*, it is possible to find third party wrappers. In this thesis we will show examples based on C/C++ programming language. The CUDA software environment provides an abstract thread computing model which could be common over different generations of GPU hardware. Specifically, CUDA parallel execution has more flexibility than *SIMD* parallel execution, and is called as *SIMT* (Single Instruction Multiple Threads). CUDA lowers the initial barrier for developers to benefit from *NVIDIA* GPGPUs technology for solving intensive computations such as linear algebra operations including matrix-vector multiplication. Thus it enables programs to use the computation power of GPU, facilitated by the CUDA-SDK. Despite enormous improvement, there are still some CUDA [27] issues that need to be resolved. GPUs are originally designed for single precision (32 bits) computations [28]. In fact, single precision is generally sufficient for graphics rendering. Unfortunately, the numerical simulation generally requires more precision. All implementations prior to *CUDA 1.3* are performed in single precision, since double precision was introduced for *CUDA 1.3*. Nevertheless, double precision computation time is still usually 4 to 8 times higher than single precision [29]. Regarding the arithmetic, CUDA implementation of double precision floating point operations differ at some points from the IEEE 754 standard [30]. More research is needed to ensure the correctness, stability and portability of such operations. In this thesis, we will obviously investigate the behavior of performance upon the precision. CUDA has proven a large interest in scientific computing [31] [32] [33] and continues to evolve. CUDA has also demonstrated its efficiency in distributed parallel computing environment [34] [35].

OpenCL

On the other hand, OpenCL (Open Computing Language), a free and open language offered in 2008 by the Khronos group, is intended for use on all the compatible graphics cards of all manufacturers on the graphics card market. It provides a more general framework for a hybrid parallel computation with CPU and GPU. It can be used with both *NVIDIA* and *ATI/AMD* cards.

The first multi-platform GPU programming language was born. OpenCL has also proven its efficiency [36] [37] [38] [39] [40] in numerical simulations. *NVIDIA* includes OpenCL API in the CUDA toolkit since 2008. Today, many engineering and scientific applications are transformed in order to utilize the advantages of GPU-based computing.

2.4 GPU Programming Model

In this section, we will discuss important issues of the CUDA programming paradigm. Understanding the issues of memory access, management of execution threads and the necessity to account for the features of a particular GPU device is important, in order to provide effective implementation of the basic linear algebra operations that affect the performance of more complex algorithms.

2.4.1 Memory structure

The GPU performs the same calculations on a large number of data instances (SIMD model). This alone, however, is not enough to get more performance, because the data must be well organized on memory to optimize access, the key point on which the algorithm performance depends. The memory layout and access patterns are the features that strongly differentiate CPU and GPU devices. This difference compels us to use a specific code optimization model, but such optimization brings better performance in GPU. CPU memory exhibits very low latency for the price of reduced throughput. On the contrary, GPU devices allow to push large portions of data but the access to their memory is rather slow. As seen in Figure 2.2 (left), CPU has a connection to the main memory through cache memory. When CPU accesses data, one of two things happens, *i.e.*, to access to the main memory, or to access to the cache memory. The cache memory can store the last accessed data. If data once read from the main memory is reused several times in certain cycles, CPU needs only access such data residing in cache. However, we need to notice that the usage of the cache memory is completely automatic and there is no way to control the behavior of the cache memory from the user software. The GPU system has a more complicated hierarchy of memory. GPU can access *register memory*, *shared memory*, *constant memory* and *global memory*, which are shown in Figure 2.3(a) for CUDA. The OpenCL version is described in Figure 2.3(b). Without loss of generality and for

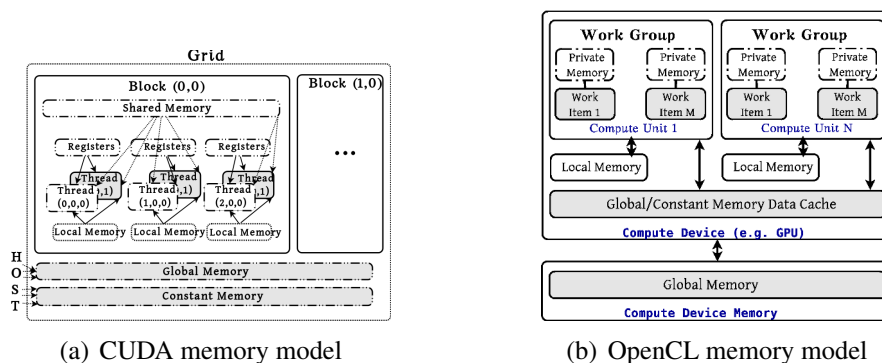


FIGURE 2.3: CUDA and OpenCL memory models

the sake of clarity, we must adopt the terminologies of CUDA. In contrast with CUDA, we will accurate the terminologies whether specific to OpenCL. The *register memory* is seen as

local memory of an ALU in GPU. This memory is accessed in the fastest speed in the GPU board with low latency. In the execution of the code, this latency will be completely hidden by executing an appropriate number of *threads*. The *shared memory* is the next fastest memory, which can be accessed from all ALUs in a group called a *block*. Then the *shared memory* is used as a communication buffer between ALUs in a *block*. In CUDA programming, a user can declare the usage of *shared memory* explicitly, which leads to a big difference from the code for CPU. The *constant memory* also provides faster access but it is only readable. It is necessary to use an host CPU to rewrite the data in the *constant memory*. The *global memory* is the main memory of the CUDA GPU, which can be accessed from all ALUs of GPU. The results of computation by ALUs are written in the *global memory* and sent to the host, which, in reality, is done the opposite way, *i.e.*, the host reads data through the PIC Express bus. Both *constant memory* and *global memory* need to be allocated by the host computer. A comparison on access speeds to memories amounts to:

$$\begin{aligned} \text{register memory} &< \text{shared memory} \\ \text{shared memory} &< \text{constant memory} \\ \text{constant memory} &\ll \text{global memory} . \end{aligned}$$

Since *NVIDIA* GPU uses GDDR5 memory with a higher clock and wider memory bus, the memory access speed of *global memory* is about four times faster than the main memory of CPU which consists of DDR3 memory. However, we need to be aware that *global memory* is shared by all ALUs of the GPU board, and that the speed of the memory to an ALU is slow. Efficient memory access to the *global memory* without ALUs idling is paramount to CUDA GPU computation, for execution time and also for power consumption, because the GPU board consumes some amount of energy even though all ALUs are idling. Computing units are divided into blocks of threads. Each block has its own *shared memory* that is available to blocks' threads. The key point in achieving good performance in GPU computing is the wise use of this memory hierarchy. It might require a redesign of the traditional implementation of some basic algorithms. What is also important, the algorithm design must not only account for the architecture but also be tunable to GPU device characteristics.

2.4.2 Kernel and gridification

Handling scientific calculations on GPU with CUDA technology extends beyond the traditional GPU stream computing by extending the functions of the GPU to address a wide range of mathematical intensive problems. In GPU terminology, in OpenCL and in CUDA, the word *kernel* is used to denote a subroutine executed in parallel by GPU ALUs (see Figure 2.4(a)), with a CUDA grid architecture. A *thread* is the smallest unit of processing that can be scheduled by an operating system. CUDA *threads* are controlled by the dedicated hardware inside GPU. They are grouped into 32 *threads* assigned to one actual ALU hardware component. The unit of 32 *threads* is called a *warp*. This is a fundamental size of *thread* execution and is understood as a size of vector length of vector super computers. Several *warps* are grouped into a *block* whose size is given by the user as a multiplier of 32. Inside a *block*, all *threads* can access a *shared memory* and can be synchronized by the hardware with almost negligible latency. This usage of *shared memory* and synchronization inside a *block* play a key role in

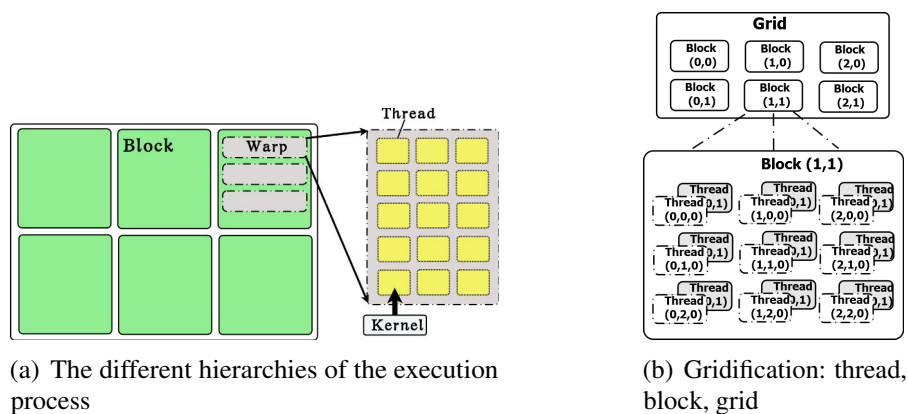


FIGURE 2.4: The different hierarchy of execution process and gridification

computation of the inner product of two vectors for scientific computation. A set of *blocks* is called a *grid*, as shown in Figure 2.4(b).

Following the development history of GPGPU as a graphics hardware, GPU with dedicated memory is packaged in a PCI expansion board and connected to the host computer through the PCI Express bus [41] [42]. Here, the GPU itself has no operating system and the general parallel executing flow is driven by CPU on the host computer. The flow of parallel computation using CUDA GPU is the following. Data is first copied from the main memory to the GPU memory, *global memory* (1). Then the host (CPU) instructs the device (GPU) to carry out calculations (2). The *kernel* is then executed by all threads in parallel on the device, (3). Finally, the results are copied back (from GPU memory) to the host (main memory), (4). Figure 2.5(b) shows this flow schematically. Figure 2.4(a) shows the different structures of

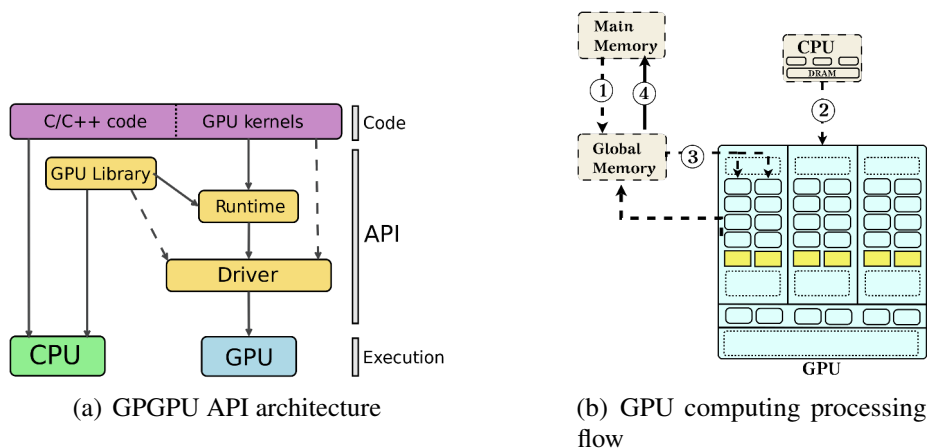


FIGURE 2.5: GPGPU API architecture and GPU computing processing flow

the execution process and Figure 2.5(a) describes the general architecture of GPGPU APIs. The OpenCL syntax is closed to that of CUDA. The main differences in terminology of the language architectures are reported in Table 2.1.

The global index of the current *thread* is obtained from a pair of indices, the index for the *block* and the local *thread* index inside the *block*,

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

where `blockDim.x` is set as a number of *threads* per *block* by user. CUDA has the capability to assign a global index of current threads in a 2D or 3D-shaped grid, which is originally sup-

	nVidia CUDA	AMD (OpenCL)
CPU	<i>host</i>	<i>host</i>
GPU	<i>device</i>	<i>device</i>
GPU function	<i>kernel</i>	<i>kernel</i>
main memory	<i>global</i>	<i>global</i>
block memory bloc	<i>shared</i>	<i>local</i>
texture memory	<i>texture</i>	<i>texture</i>
constant memory	<i>constant</i>	<i>constant</i>
thread memory	<i>local</i>	<i>private</i>
grid	<i>grid</i>	<i>NDRange</i>
block	<i>block</i>	<i>work group</i>
thread	<i>thread</i>	<i>work item</i>
warp	<i>warp (32)</i>	<i>wavefront (64)</i>

TABLE 2.1: Nvidia/AMD: GPGPU nomenclature

posed to manage tasks for the bitmap display. Generalization of these thread index assignments is understood as gridification.

The literature [43] [44] [45] has proved that the gridification, *i.e.*, the grid settings and organization, has a strong impact on the performances of the kernels.

To illustrate the inherent difference between the CPU and GPU code, we present as a basic example the addition of two vectors. CPU code is expressed as a simple loop as described in Algorithm 2.1. On a GPU, the basic idea consists in assigning each index of the vectors to an

Algorithm 2.1: Addition of vectors on CPU

```

1 for  $i = 0$  to  $n - 1$  do
2   |  $c[i] \leftarrow a[i] + b[i]$ 
3 end

```

execution unit and then performing the addition operation on the whole vector. This principle is presented in Algorithm 2.2.

Algorithm 2.2: Addition of vectors on GPU (thread function)

```

1  $i =$  thread number
2  $c[i] \leftarrow a[i] + b[i]$ 

```

2.4.3 Auto-tuning of the grid

In order to build efficient implementations, developers must pay attention to hardware configuration issues. The main problem lies in setting up an optimal threads hierarchy and grid configuration that will match the hardware configuration and the specific problem parameters, for instance the size of processed matrices.

The kernel function requires at least two additional arguments corresponding to the gridification: the number of threads per block, `nThreadsPerBlock`, and the dimension of the block, `nBlocks`. An example of a kernel function declaration for a one-dimensional grid is presented below:

```
KernelEx<<<nBlocks, nThreadsPerBlock>>>(arguments);
```

These values depend both on the total number of necessary threads related to the size of a problem to be solved, and the number of blocks and number of threads per block, *i.e.*, the specificity of the GPU architecture. Choosing a number of threads higher than the amount supported natively will result in non-optimal performance. Accordingly, auto-tuning of the

gridification is a way that helps the CUDA program achieve near-optimal performance on a GPU architecture. The idea behind the tuning of the grid is to recognize device features and then adjust the gridification according to the problem size. We focus on finding the gridification, which gives the best performance.

To illustrate the concept of gridification, let us consider the example described in the Listing 2.1.

```
// Declaration of the kernel
__global__ void KernelEx(double* parameter);
// Use of the kernel
KernelEx <<< Dg, Db, Ns, S >>> (parameter);
```

LISTING 2.1: Kernel parameters and call

where Dg, Db, Ns, S denote respectively: (i) Dg, the number of blocks of type dim3 is used to define the size and dimension of the grid (the product of these three components provides the number of units launched); (ii) Db, the number of threads per block of type dim3 is used to specify the size and dimension of each block (the product of these three components offers the number of threads per block); (iii) Ns, the number of bytes in shared memory of type size_t represents the number of bytes allocated dynamically in shared memory per block in addition to statically allocated memory (by default Ns=0); (iv) S corresponds to the CUDA stream of type cudaStream_t, and gives the associated stream (by default S=0). Each kernel has *read-only* implicit variables of type dim3: blockDim, the number of threads per block, *i.e.*, value of nThreadsPerBlock of the kernel's setting, blockIdx, the index of the block in the grid and threadIdx, the index of the thread in the block.

For the sake of clarity and without loss of generality, the gridification is illustrated for the Daxpy (Double-precision real Alpha X Plus Y) operation, presented in Algorithm 2.2. The corresponding CUDA kernel is given in Listing 2.2.

```
1 __global__ void Daxpy(double alpha, const double* d_x, double* d_y, int size) {
2   unsigned int idx=blockIdx.x*blockDim.x+threadIdx.x;
3   if (idx<size) {
4     d_y[idx] = alpha * d_x[idx] + d_y[idx];
5   }
6 }
```

LISTING 2.2: Daxpy (Double-precision real Alpha X Plus Y) on one-dimensional grid

As a comparison, the OpenCL version is described in Listing 2.3.

```
1 __kernel void Daxpy(double alpha, __global double* d_x, __global double* d_y, int size) {
2   unsigned int idx = get_global_id (0);
3   if (idx<size) {
4     d_y[idx] = alpha * d_x[idx] + d_y[idx];
5   }
6 }
```

LISTING 2.3: Daxpy (Double-precision real Alpha X Plus Y) version in OpenCL

As an example of a one-dimensional grid, consider one block of the *size* number of threads, as shown in Listing 2.4.

```
1 int main(int argc, char** argv) {
2   // -- variables declaration and initialization
3   ...
```

```

4 // -- kernel call where d_x and d_y are device pointers
5 Daxpy<<<1, size>>>(alpha,d_x,d_y, size );
6 // a vector of size element added once
7 }

```

LISTING 2.4: Basic kernel call

In the following we present the auto-tuning technique for CUDA. A basic self-configuration of the number of blocks with a given number of threads can be expressed in Listing 2.5.

```

1 // call function
2 int main(int argc, char** argv) {
3     const int nThreadPerBlock = 100;
4     // -- self-configuration parameters kernel call
5     const int nBlocks = 1 + (size - 1) / nThreadPerBlock;
6     // -- kernel call where d_x and d_y are device pointers
7     Daxpy<<<nBlocks,nThreadPerBlock>>>(alpha,d_x,d_y,size);
8 }

```

LISTING 2.5: Kernel call with basic self-threading computation with a number of threads per block equals to 100

where `nBlocks` and `nThreadPerBlock` of type `dim3` are required for 2-d and 3-d. The code presented in Listing 2.6 shows the *Daxpy* kernel with a two-dimensional grid.

```

1 __global__ void Daxpy(double alpha, const double* d_x,double* d_y, int size) {
2     unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
3     unsigned int y = threadIdx.y + blockIdx.y * blockDim.y;
4     int pitch = blockDim.x * gridDim.x;
5     int idx = x + y * pitch;
6     if (idx < size) {
7         d_y[idx] = alpha * d_x[idx] + d_y[idx];
8     }
9 }

```

LISTING 2.6: Daxpy on two-dimensional grid

The implementation given in Listing 2.6 ensures that when the vector content exceeds the one-dimensional grid, the extra coefficients are stored on the second dimension. To optimize the threading organization, we propose to set the kernel configuration by taking into account the hardware characteristics. Thus, the grid block characteristics are computed as described in Listing 2.7 for a one-dimensional grid and in Listing 2.8 for a two-dimensional grid.

```

1 dim3 ComputeGridBlock(int size, int thread_per_block) {
2     dim3 nblocks;
3     // -- compute the necessary blocks
4     int necessary_blocks=1+(size-1)/thread_per_block;
5     int max_grid_size_x=dev_properties.maxGridSize[0];
6     // -- required two-dimensional grid
7     if (necessary_blocks > max_grid_size_x) {
8         nblocks.x=max_grid_size_x;
9         nblocks.y=(necessary_blocks-1)/max_grid_size_x+1;
10        nblocks.z=1;
11    } else { // dimensional grid sufficient
12        nblocks.x = necessary_blocks;
13        nblocks.y = 1;
14        nblocks.z = 1;

```

```

15 }
16 return nblocks;
17 }

```

LISTING 2.7: Grid features computation for one-dimensional grid fully used

```

1 dim3 ComputeGridBlock(int size, int thread_per_block) {
2   dim3 nblocks;
3   // compute the necessary blocks
4   int necessary_blocks = 1 + (size - 1) / thread_per_block;
5   int max_grid_size_x = dev_properties.maxGridSize[0];
6   if (necessary_blocks <= max_grid_size_x) {
7     // -- dimensional grid sufficient
8     nblocks = dim3(necessary_blocks);
9   } else {
10    // -- required two-dimensional grid
11    int each_block_dim = ceil(sqrt(necessary_blocks));
12    nblocks = dim3(each_block_dim, each_block_dim);
13  }
14  return nblocks;
15 }

```

LISTING 2.8: Grid features computation for two-dimensional grid

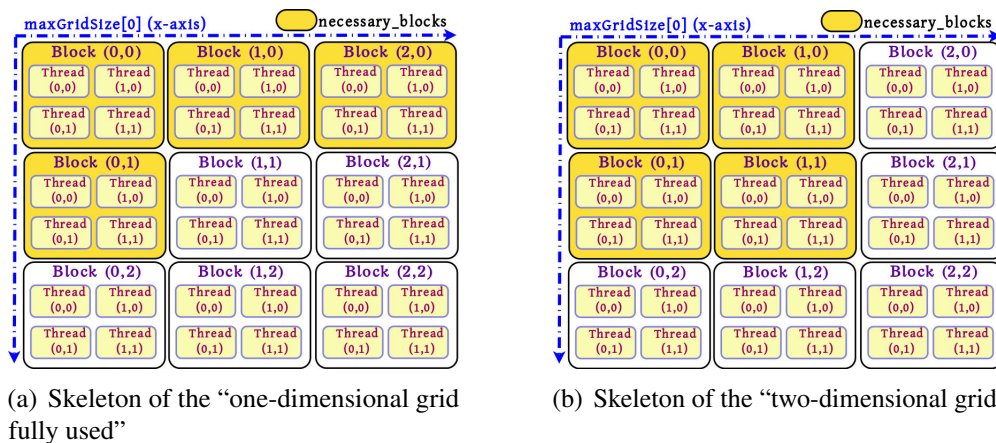


FIGURE 2.6: Grid computing strategies: one-dimensional grid technique (a) and two-dimensional grid technique (b)

Figure 2.6(a) and Figure 2.6(b) illustrate respectively the first technique presented in Listing 2.7 and the second one exhibited in Listing 2.8. Line 5 of both codes calculates the maximum size of the first dimension. The technique presented in (Listing 2.7, Figure 2.6(a)) consists in using all threads of the first dimension of the grid and then completing the remaining ones with the second dimension. The technique implemented and described in (Listing 2.7, Figure 2.6(a), Figure 2.6(b)) utilizes a two-dimensional square grid. This technique involves a greater proximity between the blocks of the first dimension and those of the second leading to better efficiency as we will see in the next sections.

2.4.4 Memory access and data transfers

We focus on the CPU and GPU interaction and communication in order to minimize CPU and GPU idling, and maximize the total throughput.

GPUs are interconnected to a motherboard using a standardized type of slot, as drawn in Figure 2.7. As said above, most source codes using CUDA are divided in multiple steps as

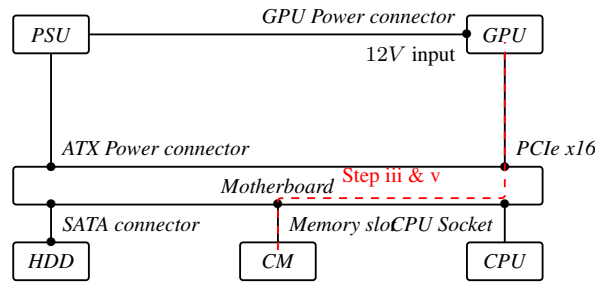


FIGURE 2.7: Data transfers between CPU side and GPU side

shown in Figure 2.5(b). and follow the following steps: (i) allocation of memory on the CPU side, (ii) allocation of memory on the GPU side, (iii) copying allocated values from CPU to GPU, (iv) kernel launch, (v) copy results from GPU to CPU, (vi) deallocation of allocated memory. The copy operations use the motherboard bus to transfer data between the *Central Memory (CM)* of the computer and the *Global Memory* inside the GPU.

When we deal with GPU implementation, the evaluation of raw computing efficiency and memory bottleneck gives a first overview of the performance of the graphics card. Figure 2.8(a) shows the effectiveness of a cosine function upon the number of threads, where the efficiency calculation is based on the number of transactions per second. We compare CPU and GPU with CUDA, and GPU with OpenCL. The analyze has been performed on a workstation equipped with an Intel Core i7 930 running with 2.67GHz, which has 4 cores and 12 GB main memory and two *NVIDIA GTX570* GPU with 1279MB memory. The GPU card has double precision arithmetic capability, and is driven by CUDA 4.0 software. We evaluate the number of transactions carried out per second with 512 threads per block (512 work-items per work group) and each thread performing 1,000 computations. Figure 2.8(a) clearly illustrates that

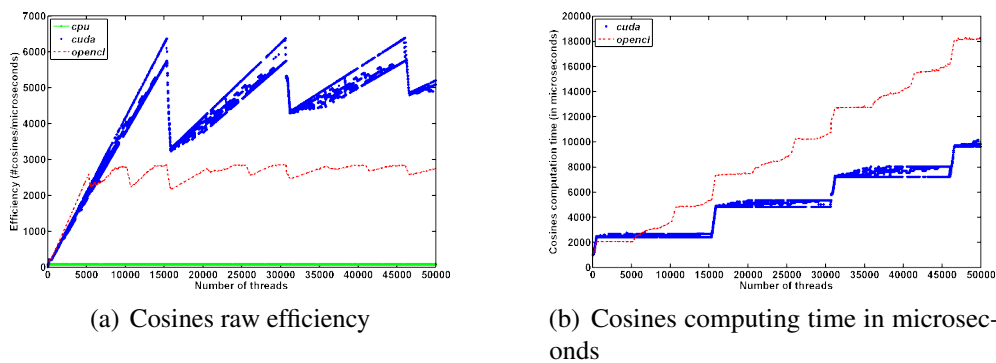


FIGURE 2.8: Cosines raw efficiency (a) and computing time in microseconds (b)

the GPU is more effective when the required number of operations increases. The higher the number of operations, the smaller the influence of a particular gridification. According to the particularity of GPU architecture as described in Figure 2.2, the efficiency limit is achieved asymptotically. The shape of the curve in Figure 2.8(b) that represents the cosine computation time in microseconds, confirms this phenomenon and the importance of the gridification for small data sizes. For CUDA, a gap in the curve is observed when the load remains constant while the block is not fully utilized, which involves an overloading when an extra warp is

required. On the OpenCL curve, levels are less clear-cut because the cutting of blocks was done in a different way, which allows supernumerary threads to release the GPU quickly instead of executing itself in a vacuum. When threads of a warp access the memory, the queries are gathered into words of 32, 64 or 128 bits. More requests are required if each thread accesses a distant memory location, contrary to the case when threads access localized memory locations. The performance bottleneck may occur when memory access is poorly managed.

To illustrate this phenomenon, we carry out two tests where each thread accesses a memory location to indices $i + k * step$ first, as shown in Figure 2.9(a) and second to indices $i * step + k$, as described in Figure 2.9(b), for $k = 0, \dots, 1000$ where i denotes the index of the thread. Figure 2.10 shows that when we vary the $step$ for the indices $i * step + k$, the threads are

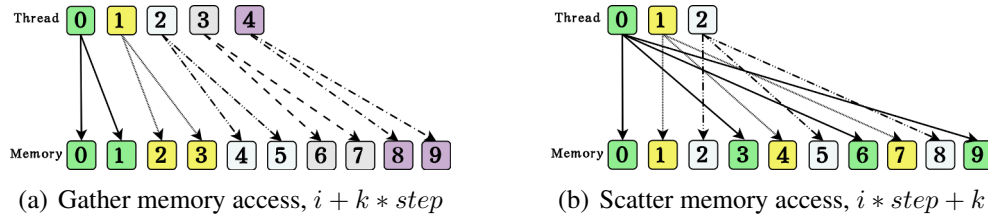


FIGURE 2.9: Memory access mode: $i + k * step$ (a) and $i * step + k$ (b)

accessing memory locations further from each other, and this results in degraded performance because a given memory request can satisfy only one thread. On the other hand, it also illustrates that for the indices $i + k * step$, the threads are always accessing contiguous memory locations, and thus it results in good performance because of localised memory access. Let remark in Figure 2.8(a) the behavior of the CPU access to the RAM, which is constant in time, unlike the access to the cache.

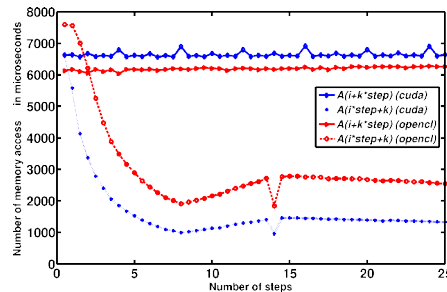


FIGURE 2.10: Number of memory access per microseconds

2.4.5 Implementing complex number arithmetics on GPU

Some numerical methods, such as the finite element discretization of the Helmholtz equations for acoustic problems, leads to complex number arithmetics matrices.

GPUs have initially been developed for integer arithmetics, and using floating point operations with real numbers significantly reduces their performance. This is even more problematic when using double precision floating point operations which imply a drop in performance [46] [47] [48]. Since such problems related to numerical simulations involve complex number arithmetics with double precision floating point operations, the expected performance is extremely low.

As mentioned earlier, CUDA was initially dedicated to real numbers arithmetics. To better take advantage of GPU architectures, we focus on the optimization of complex numbers in CUDA and OpenCL. As indicated in most of state-of-the-art real-number arithmetics, GPU implementation clearly outperforms CPU algorithms, but behavior for complex number arithmetics with double precision is still a challenge.

However, given that a complex number is a set of two real numbers (real part, imaginary part), implementation is possible by defining a structure containing two real numbers. A natural approach to defining a complex number in memory is based on the following structure:

```
struct complex {
    double x; // real part
    double y; // imaginary part
};
```

LISTING 2.9: Complex number in memory

We make sure to avoid padding, and that both real numbers x and y are stored on contiguous memory, *i.e.*, the offset of both is null. The structure of CUDA and the OpenCL complex has the same concept as before, except that data are stored on a GPU device. The CUDA library *libcudart.so*, proposes *cuFloatComplex*, a *float2* structure (*typedef float2 cuFloatComplex*) and *cuDoubleComplex*, a *double2* structure (*typedef double2 cuDoubleComplex*), which correspond to the same design given in Listing 2.9. We have implemented a complex template class *complex<T,U>* (T is the data type name and U the index type name), which redesigns all the operations available in standard *std::complex*. The routines of the *std* complex fall within the “host” category, *i.e.*, they are only available from the CPU. In our complex class, these routines are implemented with *__host__ __device__* (CUDA), which guarantees the usage on both CPU and GPU code. To differentiate from the standard complex, we put our new complex class into “*stdmrg*” namespace.

```

1 namespace stdmrg {
2 	//! @struct complex
3 	//! @brief complex for cuda and
4 	template <class T>
5 	class complex {
6 	public:
7 		typedef T value_type;
8 	private:
9 		T m_x;
10 		T m_y;
11 	public:
12 		__host__ __device__ complex (void);
13 		__host__ __device__ complex (T r, T i=0);
14 		__host__ __device__ complex (const complex<T>& copy_c);
15 	public:
16 		__host__ __device__ T real (void) const;
17 		__host__ __device__ T imag (void) const;
18 	private:
19 		__host__ __device__ T abs (void) const;
20 		__host__ __device__ T norm (void) const;
21 		__host__ __device__ complex conj (void) const;
22 		// ...
23 	public:
24 		__host__ __device__ complex& operator=(const T& value);
25 		__host__ __device__ complex& operator+=(const T& value);
26 		__host__ __device__ complex& operator-=(const T& value);
27 		__host__ __device__ complex& operator/=(const T& value);
28 		__host__ __device__ complex operator-(void);
29 		// ...
30 } // namespace stdmrg {

```

LISTING 2.10: Complex class

```

1 	// -- free functions: example of usage:
2 	// -- stdmrg::complex<T,U> c;
3 	// -- T r = stdmrg::real (c);
4 	//! @brief real part
5 	template <class T>
6 	__host__ __device__ T real (const complex<T> c) {
7 	return c.real ( );
8 }
9 	//! @brief imaginary part
10 template <class T>
11 __host__ __device__ T imag (const complex<T> c) {
12 return c.imag ( );
13 }
14 	//! @brief absolute value of complex
15 template <class T>
16 __host__ __device__ T abs (const complex<T> c) {
17 return c.abs ( );
18 }
19 	//! @brief norm of complex number
20 template <class T>
21 __host__ __device__ T norm (const complex<T> c) {
22 return c.norm ( );
23 }
24 	//! @brief complex conjugate
25 template <class T>
26 __host__ __device__ T conj (const complex<T> c) {
27 return c.conj ( );
28 }
29 	// ...
30 } // namespace stdmrg {

```

LISTING 2.11: Complex free functions

Listing 2.10 shows a piece of the `stdmrg::complex<T,U>` class, and Listing 2.11 illustrates some free functions.

2.5 Energy consumption

To answer the question “*How much energy is consumed for a numerical simulation running on a Graphic Processing Unit?*”, an experimental protocol is herewith established. The current provided to a Graphic Processing Unit (GPU) during computation is directly measured using amperometric clamps. Signal processing of the intensity of the power supplied to a GPU, with a noise reduction technique, gives precise timing of GPU states, which helps establish an energy consumption model of the GPU.

2.5.1 Introduction

During the last decade, GPGPU has proved to be extremely powerful and efficient to accelerate graphics (video games, etc.) and numerical simulations (seismic imaging, etc.). GPGPU has emerged as an alternative to classical approaches in parallel computing, and showed a great promise for supercomputers toward exascale and petascale computing. The power of the GPU can be a real advantage when considering the treatment of an application with a huge data size.

Recently, mobile devices such as personal computers, cell phones, tablets, etc. are equipped with a powerful GPU which can be harnessed for general-purpose computations. Nvidia has newly expanded its CUDA parallel-processing architecture to mobile devices. Usually, embedded systems (aircraft, vehicle, drone, gravimeter, etc.) run with limited computer hardware

resources, such as memory and performance. The recent idea to support the performance of embedded processors consists in coupling the classical microprocessors with GPU devices in order to accelerate the computations. Having a GPU available on embedded systems is interesting for a number of reasons, such as the acceleration of real-time applications when solving large-size problems. However, these mobile devices are battery-dependent, *i.e.*, the battery capacity is dependent upon the equipment level in the device, therefore it is crucial to take into account the portion of energy consumed by the GPU accelerator, which is energy-greedy. Beyond our interest in accelerating computations, we are keen to reduce the energy consumption of embedded systems or supercomputers.

The main contribution of this work is to analyze and understand the behavior of algorithms when using GPU Computing in terms of energy consumption. This analysis can be helpful in designing future GPU-accelerated energy-efficient embedded systems. It can be used, for instance, when managing automatic switching between ordinary microprocessors, GPU and hybrid microprocessors according to the needs of an algorithm, equipment level and available resources, *e.g.*, the battery usage. The data used in this analysis are coming from gravity surveys using a gravimeter. Recent gravimeters (ZLS Dynamic Gravity Meter, Absolute Quantum Gravimeter, Micro-g LaCoste Absolute Gravimeters, etc.) include embedded processors connected to a host computer. In the *ZLS Dynamic Gravity Meter*, the host computer stores all data on a hard disk and can simultaneously display data to the video monitoring. Real-time output is one of the most important constraints. GPU Computing is an excellent candidate for achieving real-time rendering.

With the construction of supercomputers toward exascale computing, a new problem has arisen, *i.e.*, how to reduce energy consumption of the system. In order to make a computer 100 times faster than today's supercomputers, it would require 100 times the energy if the same CPU architecture is used, which is unaffordable. There is a famous list of the 500 fastest supercomputers in the world [8] which is updated twice a year. Besides this list, there is another top-500 list of supercomputers which is power efficient, the Green 500 [9] [10] [11]. In the latter list, all top-10 supercomputers consist of hybrid architecture of CPU and GPU.

GPU is the fastest computational unit, which is approximately ten times faster than a multi-core CPU with similar energy consumption. The usage of a hybrid system with GPU is a promising way to construct energy-efficient embedded systems and clusters. Efforts to reduce energy consumption are based on hardware developments, *e.g.*, using special coolant for efficient heat exchange. The most energy efficient supercomputer TSUBAME-KFC at TITech, Japan[49] uses a special coolant for efficient heat exchange. Usage of GPU is almost the standard in super computing and is *de facto* the future of embedded systems and mobile devices, but what about energy consumption for real applications?

For applied sciences and engineering problems, a system of partial differential equations (PDEs), ordinary differential equations (ODEs) and algebraic differential equations (DAEs) is often used as a physical and mathematical model, which is then solved numerically. Today, several physical parameters can be taken into account, coupling different physical models leading to more and more complex systems.

The main priority of the library is the computational time required to obtain the solution in a parallel environment, *i.e.*, "How much time is necessary to solve the problem?". The second priority of the library is GFlop/s, because this indicator is useful to estimate performance on a

new hardware by evaluating of the ratio between actual G Flop/s by the library and the peak G Flop/s by the hardware.

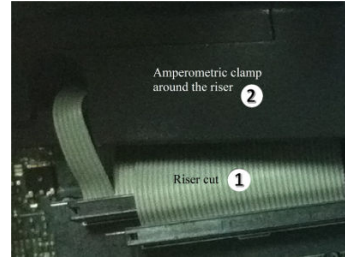
Nowadays, we need to address another question, energy related, *i.e.*, “*How much energy is consumed by the application?*”. For high-performance computing (HPC) hardware, the appropriate answer gives a compromise between computational time and energy consumption. Numerical algorithms [50] [51] [52] [53] [54] [45] have already faced a similar question, *i.e.*, “*How much memory is necessary to solve the linear system?*”. This question is also very important because memory limitation is one of the physical constraints of numerical simulation. If the memory falls short, we have to give up the simulation or satisfy ourselves with very slow out-of-core execution. Therefore, algorithms have been optimized in both directions to minimize computational time and memory usage. Unfortunately, there is no way to answer the new question on energy consumption only from a software point of view. To the best of our knowledge, there is no established way to measure real energy consumption for each application. Therefore, we aim to design an experimental protocol to measure GPU energy consumption accurately for fundamental and basic arithmetic operations and some heavy operations involved in numerical linear algebra with sparse matrix. This allows us to build an accurate energy consumption prediction model of the GPU. This methodology could suggest a “new vision of high-performance computing” and answer some of the questions outlined in references [12] [13] [14] [11], which highlight the importance of energy consumption when using GPUs. The computing power and power/energetic consumption ratio of the GPU being superior to standard CPU, the use of GPU represents an opportunity to have more power for a lower energetic cost.

2.5.2 Experimental protocol

In this section, we describe the physical setting in which to measure the CPU energy consumption through direct measurements of the current sent to the GPU from the host computer, and a technique to understand the state of the GPU from raw experimental data on the intensity of the current to the GPU with noise reduction. In order to get accurate measurements of the energy consumption of a GPU parallelized code, we need to measure the energy consumption by the GPU alone without consumption by other components of the computer. Unfortunately, there is no dedicated GPU sensor which can measure energy with the GPU ALU status, which is why we must plug in a device, an *amperometric clamp*, on the power supply cables of the GPU. An amperometric clamp measures the intensity of the magnetic field created by the current circulation and its output data are values of the current. This current is converted to voltage by an I-V conversion circuit for input of a digital oscilloscope to monitor time dependent electric power. We note that we define positive or negative voltage according to the direction of the original current during the I-V conversion. There is no space to clamp the amperometric device between the GPU card and the mother board, because the GPU card is directly connected to the PCI express expansion slot. Then we use an extension cable called a “riser uncut” in Figure 2.11(a), which allow to make space for the clamp. The power for the GPU is supplied through the PCI Express interface with two voltages, +3.3V and +12.0V, and through supplemental cables with +12.0V, directly from the power supply unit of the workstation. By specification of the PCI Express, each voltage is guaranteed as the constant value. The currencies on the PCI Express slot for +3.3V and +12.0V vary up to 3.0A and 5.5A, respectively, which results in 75W electric power at



(a) Riser uncut



(b) Riser cut and connected to a clamp

FIGURE 2.11: Riser uncut and riser cut connected to a clamp

the maximum. The current of supplemental cables varies up to $18.75A$ with $225W$ electric power. We note that one GPU card can consume power up to $300W$. Figure 2.12 presents the experimental setting schematically. To measure the currencies for $+3.2V$ and $+12.0V$, we

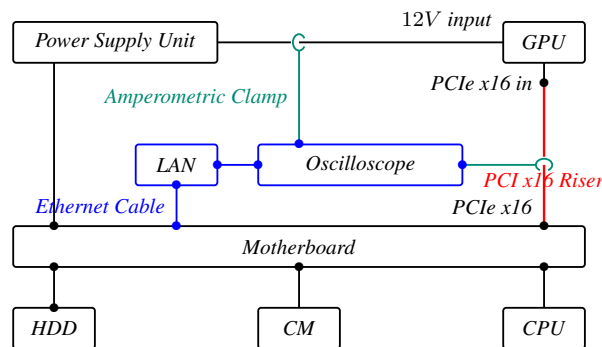


FIGURE 2.12: Schematic representation of the protocol

need to separate the $+3.3V$ cable from the $+12.0V$ cable of riser uncut. A way of separation is shown in Figure 2.11(b), where (1) the riser cut and (2) the amperometric clamp around the riser. Figure 2.13 shows the global wiring of the experiment, where (1) represents the GPU computer (workstation), (4) the first amperometric clamp, (2) the second amperometric clamp, (5) the USB plug for recording, (6) the numerical oscilloscope. In total, we used three

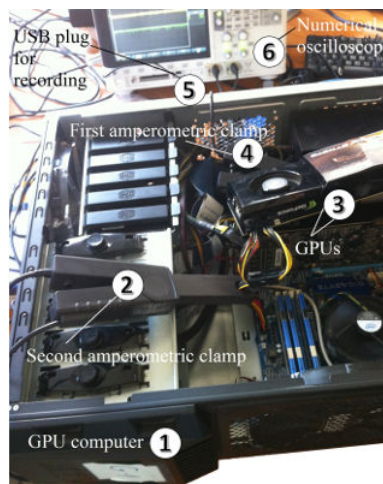


FIGURE 2.13: Experiment wiring: clamp on GPU computer

clamps to measure each current of the power-supplying cable. By measuring the currents $I_{3.3V}$, $I_{12V\ riser}$ and $I_{12V\ ext}$, we can get the energy consumption of the GPU card as

$$P = 3.3 \times I_{3.3V} + 12.0 \times (I_{12V\ riser} + I_{12V\ ext}) .$$

The foreground clamp is plugged into the power supply unit, whereas the background one is plugged into the riser. To retrieve the measure we use a numerical oscilloscope, which is connected to the local area network using an ethernet cable. The computer has two NVIDIA GTX 275 GPUs. The experiments have been performed on the first GPU, the second one being mostly used for the video output. We have designed our own experimental protocol that measures precisely the GPU energy consumption, from the raw data obtained by the amperometric clamps. The raw data are first processed by a digital oscilloscope, and then as numerical data.

2.5.3 Signal processing

The intensity is categorized into two levels,

- a level when the GPU is idle,
- another when the GPU is busy.

Unfortunately, as shown by Figure 2.14(a), the measured data are very noisy due to disturbances caused by the other components of the computer.

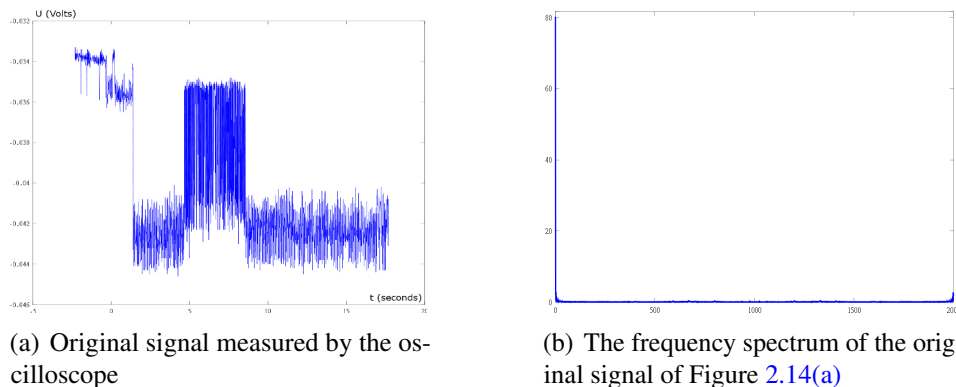


FIGURE 2.14: Original signal measured by the oscilloscope and its associated frequency spectrum

To retrieve these values, it is definitely necessary to remove noise. Then we need to use a filtering technique. There are several filtering techniques to process digital signals. First we have examined three famous filters and then chosen the best one for our purpose.

Ideal low-pass filter

The basic idea of the low-pass filter is based on the assumption that the noise oscillates much faster than the signal itself, which means the noise resides in higher frequencies. We have apply the *Fast Fourier Transform (FFT)* to the signal to get the frequency spectrum. Figure 2.14(a) and Figure 2.14(b) show the original signal and frequency spectrum respectively. Most amount of the noise is located in the high spectrum region, then we cut these frequencies. By applying *Inverse FFT*, we get a smoother signal by reducing the noise, but it is a little disrupted as a result of filtering. There is a tuning parameter to define which frequency will

be cut. We show the results of the processed signal with different cut-off frequencies in Figure 2.15.

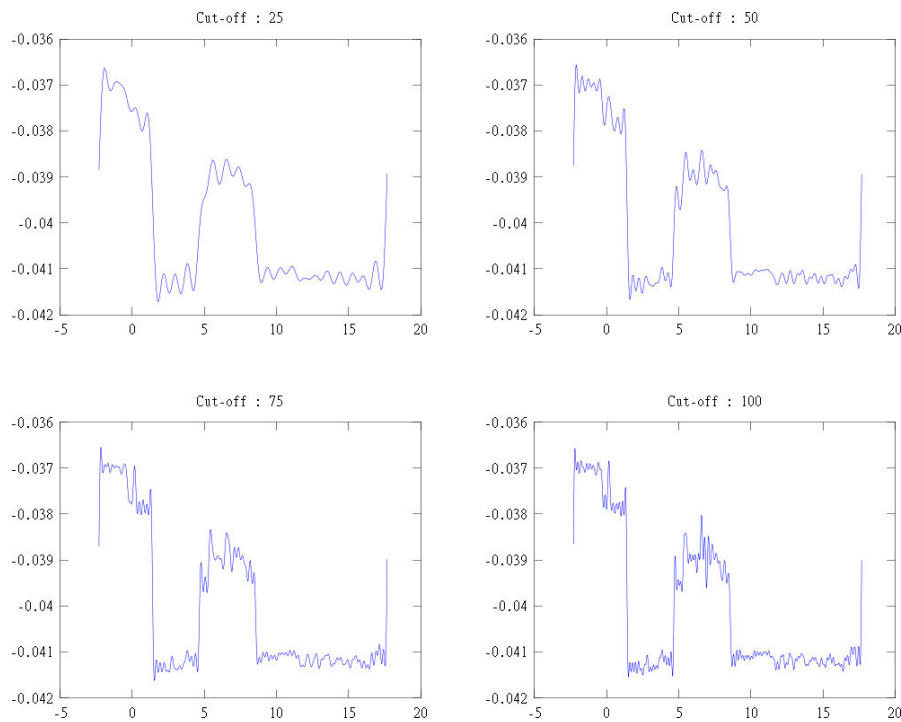


FIGURE 2.15: Low-pass filters with different cut-off frequencies

Simple averaging filter

The other fundamental filtering technique is the *averaging filter*, which uses the fact that the noise causes fluctuations which will be removed by replacing the value on each point by the average of several neighboring points. *Simple averaging filter* uses a uniform weight on the points. The computation is realized by a convolution product. There is a parameter to define the number of averaging points. We show the results of the processed signal with a different number of points in Figure 2.16.

Weighted averaging filter

This filter is an extension of the simple averaging filter by introducing weights for averaging. The weight with a *Gaussian* distribution results in a combination of a simple averaging filter in the temporal domain and a low-pass filter in the frequency domain. The standard deviation σ defines the distribution of a *Gaussian* function. We show the results of the processed signal with a different σ in Figure 2.17. Figure 2.18 summarizes the results of three different filters. We see that a *simple averaging filter* can not remove the noise completely without losing information. The low-pass filter and the Gaussian filter produce almost the same results but the Gaussian is better in terms of accuracy and preserving intensity of the signal. Therefore, we have chosen the Gaussian filter to eliminate noise from the signal measured by the digital oscilloscope in our experimental protocol. Before carrying out the benchmark, we have ensured that the results provided by the protocol are relevant. We evaluate our protocol in the following subsection.

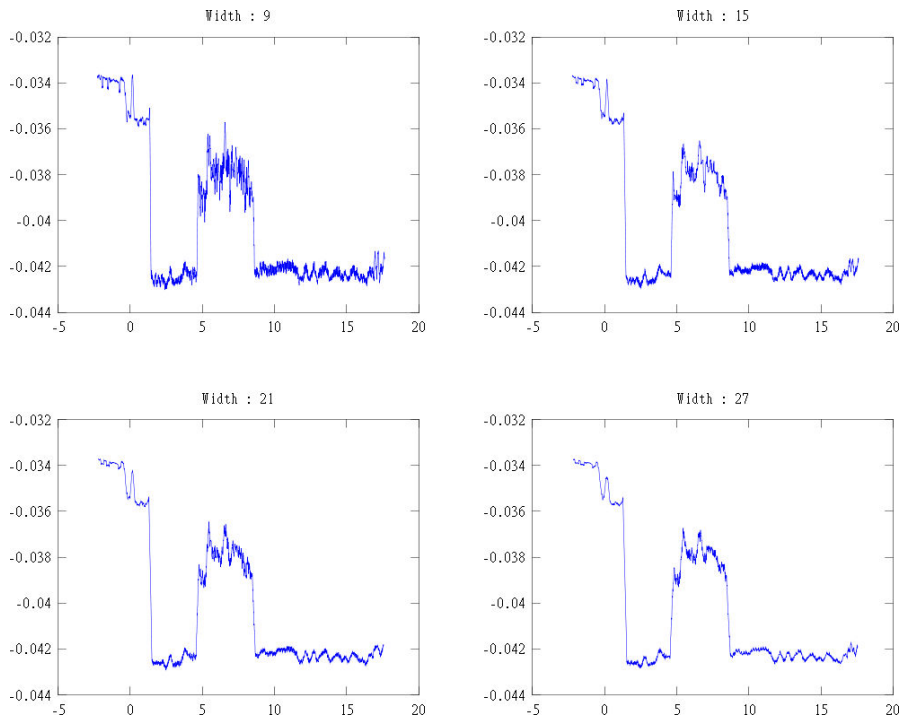


FIGURE 2.16: Simple averaging filters

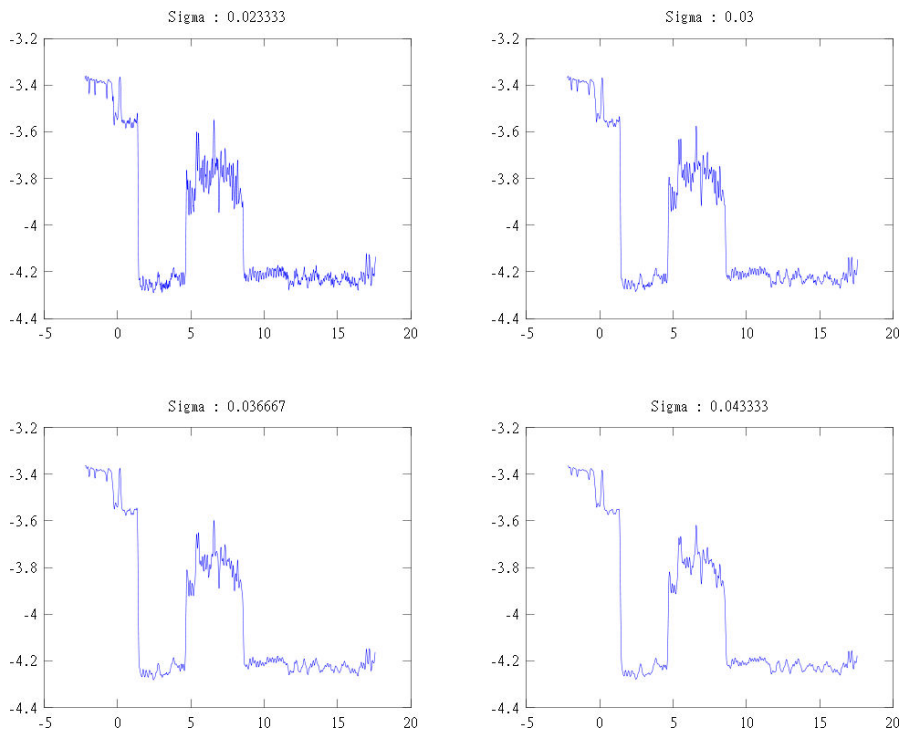


FIGURE 2.17: Gaussian filters

2.5.4 Acquisition process

To evaluate our results and validate our experimental protocol, we have qualitatively studied the response of the GPU for elementary operations such as memory allocation, addition (called DAXPY) and product of two vectors (element wise product), and communication (memory copy) between the GPU and the CPU. Since our target application is a numerical solution of

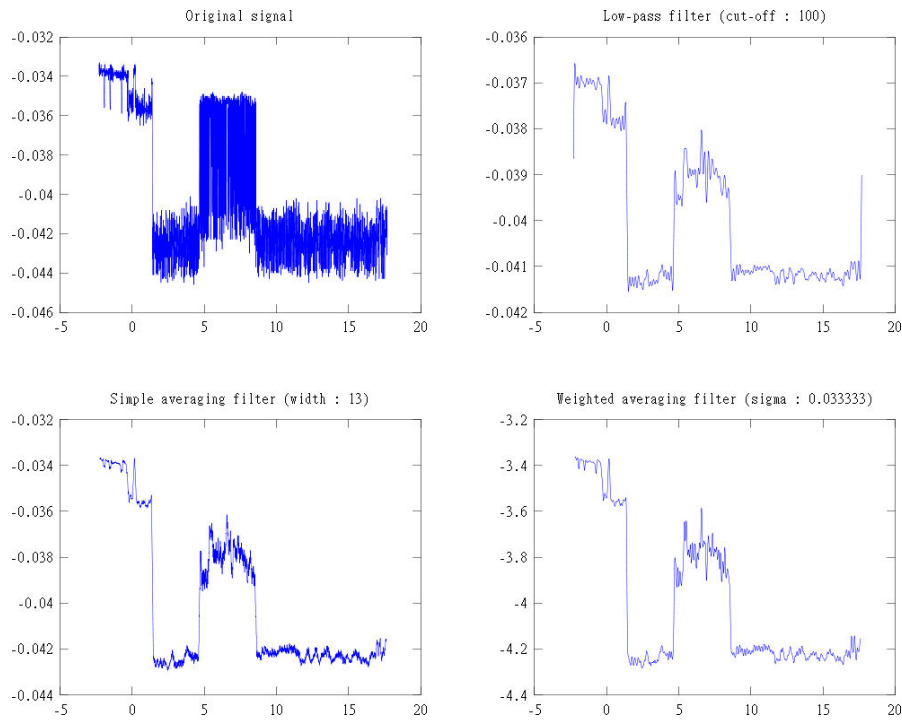


FIGURE 2.18: Summary of filters

real engineering and scientific problems, we deal with double precision arithmetic, which has been the standard in numerical simulations for three decades.

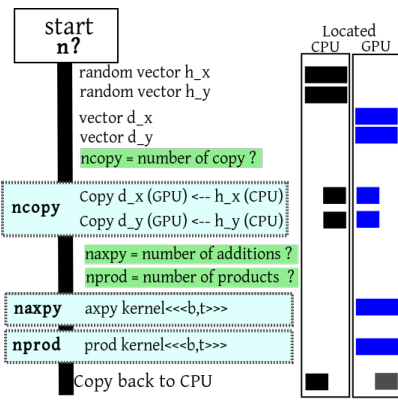
Procedure

A graph obtained by the digital oscilloscope allows us to follow the execution steps of the code, the CPU preparation and launch of the GPU kernel corresponding to each type of basic operation. Below are the steps, which are also illustrated in Figure 2.19(a):

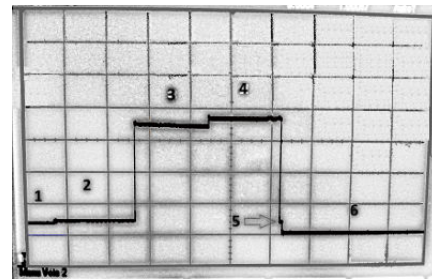
- launch the program by given n , which defines the size of the vectors
- generate random vectors in the main memory of the CPU
- allocate working vectors in the global memory of the GPU
- enter a parameter: n_copy , the number of copy operations will be performed
- copy data from CPU to GPU
- enter parameters: n_axpy , number of DAXPY and n_prod , number of products to be performed, and number of threads per block
- execute the kernel of DAXPY n_axpy times and save the total elapsed time
- execute the kernel product n_prod times and save the total elapsed time
- return the results of all computations from GPU to CPU

The result on the screen of the oscilloscope shows clearly six different phases corresponding to the following states:

1. GPU is allocating the memory without touching the values.
2. GPU is copying data from the CPU to the memory.
3. GPU is performing DAXPY.
4. GPU is performing the product.



(a) Procedure of the acquisition



(b) Screen of the oscilloscope

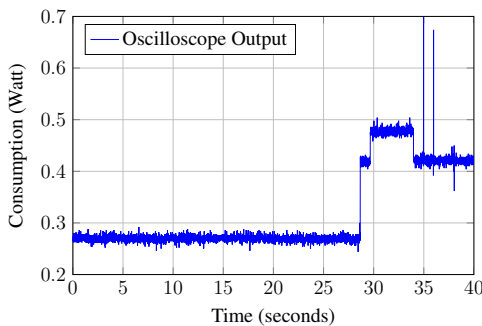
FIGURE 2.19: Procedure of the acquisition and screen of the oscilloscope

5. GPU is copying back data to the CPU and deallocating the memory.
6. GPU is back to stand-by.

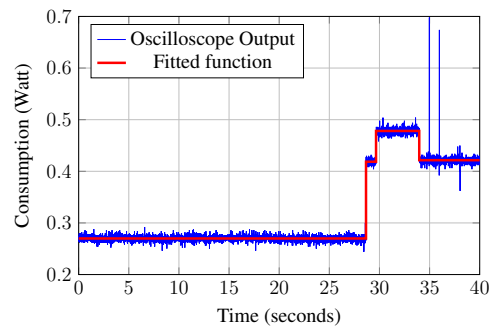
This is verified by Figure 2.19(b) by the human eye, but to get a precise timing of the phase changes, we need to proceed with further steps.

2.5.5 Detection of the phases

Figure 2.20(a) shows an example of measurement of the energy consumption of the execution obtained from the oscilloscope with the electrical power in Watts or Joules per second.



(a) Energy consumption of the different phases of the execution



(b) Approximated function of the signal

FIGURE 2.20: Energy consumption of the different phases of the execution and the approximated function of the signal

We can see four phases with some amount of noise in the figure. Each phase corresponds to each state: *sleep*, *sum*, *product*, and *copy*. In order to calculate the energy consumption and the elapsed time of each state, we needed to construct an algorithm which can detect each step precisely. We can assume that the energy consumption of each phase can be represented by a constant value, and then an algorithm needs to identify the time when the relevant jump occurs as well as the values before and after the jump. We have constructed such an algorithm as follows. The algorithm identifies the relevant jump by computing the mean value of the current phase and verifies the tenth forwarding value. If the forward value is too far from the current mean value, the algorithm concludes there is a relevant jump. This algorithm is improved

by considering the variation of the standard deviation of the values. Figure 2.20(b) shows an example of an approximate function with three relevant jumps computed by the algorithm. We also get information on the timing of the execution from this curve.

2.5.6 Prediction

In this section, we evaluate and analyze the effectiveness and robustness of our experimental protocol. We used a workstation equipped with an Intel Core i7 920 running with 2.67GHz, which has 4 cores and 6 GB main memory and one dedicated NVIDIA GTX275 GPU with 895MB memory. The GPU card has a capability of double precision arithmetic and is driven by CUDA 4.0 software.

Our aim is to establish a model that can predict the program's execution time and energy consumption from the number of elementary operations in the source code. For the development of our methodology, we first discuss the dependency of the execution time and the energy consumption of elementary operations on the size of the data set (vectors) for elementary operations which are executed on a single GPU. For a multiple GPU environment, since GPUs are independently controlled by the host hardware, our data on a single GPU are easily extrapolated to the case of multiple GPUs.

To evaluate the elementary operations, we use the program detailed in Figure 2.19(a) for different sizes of the vectors. With our current protocol, the acquisition and analysis of the measurements consumes considerable time, therefore we prepared a script that automates the process.

Data transfers

The cost of transferring data between CPU and GPU is not negligible. We have measured the time in seconds and the power in Watts for transferring double precision data. Table 2.2 shows the time and power with several sizes of the data.

size	time (s)	power (W)
1,000	0.00012219	0.000786827
2,000	0.000149068	0.000964973
5,000	0.000172517	0.001386241
10,000	0.000321641	0.002139488
20,000	0.000418555	0.003222079
2,000,000	0.030316534	0.20801877
5,000,000	0.061184495	0.536819129

TABLE 2.2: Double precision data transfers from the CPU to the GPU

Since the complexity of data transfer is linear, we can apply linear prediction for both time and power of GPU execution. From the table, we can determine the time prediction in seconds for the size of the vector x , by

$$7.0 \times 10^{-5} + 10^{-8}x \quad (2.1)$$

with a 0.9922 coefficient of determination. The prediction of the electrical power in Watts is given by

$$2.0 \times 10^{-5} + 10^{-7}x \quad (2.2)$$

with a 0.9998 coefficient of determination.

Elementary operations: DAXPY and DXMY

The programs of the elementary operations used in this analysis will be detailed in the next chapter, Chapter 3.

An addition operation of two vectors of double precision with a scalar weight $y_i = y_i + \alpha x_i$ ($1 \leq i \leq n$) is called DAXPY, which belongs to BLAS (Linear Algebra Subprograms) level-1. This operation is completely parallelized without any overhead of parallelization. We note that the CUDA hardware processes the Fused Multiply and Addition (FMA) operation which allows to compute $y = y + \alpha x$ by a dedicated hardware as one operation. An element wise product computes $c_i = x_i y_i$ ($1 \leq i \leq n$), which we call DXMY in this thesis. This operation is also completely parallelized as DAXPY. However, it needs to access three arrays for the operation, which results in a more memory-intensive operation than DAXPY.

(a) DAXPY			(b) DXMY		
size	time (s)	power (W)	size	time (s)	power (W)
1,000	0.000005008	0.000305963	1,000	0.00000512	0.000318909
2,000	0.000006545	0.000390259	2,000	0.000006673	0.000462524
5,000	0.000006763	0.000408337	5,000	0.000007001	0.000563243
10,000	0.000006535	0.000420392	10,000	0.000006603	0.000541402
20,000	0.000009760	0.000571913	20,000	0.000009795	0.000982404
2,000,000	0.000456454	0.027676176	2,000,000	0.000325738	0.046621519
5,000,000	0.001142272	0.069889335	5,000,000	0.001115328	0.160956136

TABLE 2.3: Double precision addition of vectors (DAXPY) and element wise product (DXMY)

Table 2.2(b) shows the GPU time in seconds and the electrical power in Watts and Table 2.2(a) shows the GPU time in seconds and the electrical power in Watts for DAXPY. Since the complexity of the addition operation is linear, the linear predictions of GPU execution time and electrical power are appropriate. These predictions are listed in Table 2.4.

	equation	coef. of det
Time (s)	$4.905 \times 10^{-6} + 2.272 \times 10^{-10}x$	0.999
Power (W)	$2.682 \times 10^{-4} + 1.389 \times 10^{-8}x$	0.999

where x is the size of the vectors.

TABLE 2.4: Linear prediction of DAXPY

We can see that the time for each size is exactly same as with DAXPY but the consumed energy is larger than with DAXPY due to the greater memory access. Since the complexity of the DXMY operation is linear, the linear predictions of the GPU execution time and electrical power listed in Table 2.5 are appropriate.

	equation	coef. of det
Time (s)	$-6.668 \times 10^{-6} + 2.164 \times 10^{-10}x$	0.9857
Power (W)	$-1.417 \times 10^{-4} + 3.131 \times 10^{-8}x$	0.9856

where x is the size of the vectors.

TABLE 2.5: Linear prediction of element wise product

2.5.7 Evaluation of the execution time and energy consumption

Following the experimental results of data transfer, vector addition (DAXPY), and element wise product (DXMY), we propose a model of energy consumption.

Let us define the computation time $T_{compute}$ of arithmetic operations (DAXPY or DXMY) by the unitary time $TU_{compute}$ multiplied by the size of the vectors n . Let $T_{CPU \leftrightarrow GPU}$ be the communication time. The execution time $T_{execute}$ becomes the sum of those three values.

$$TU_{compute} = TU_{DAXPY} \text{ or } TU_{DXMY} \quad (2.3)$$

$$T_{compute} = n \times TU_{compute} \quad (2.4)$$

$$T_{CPU \leftrightarrow GPU} = n * TU_{CPU \leftrightarrow GPU} \quad (2.5)$$

$$T_{execute} = T_{CPU} + T_{CPU \leftrightarrow GPU} + T_{compute} \quad (2.6)$$

To define the whole energy consumption in addition to the execution time, we need to consider the energy consumption corresponding to other phases,

- P_{idle} , power at rest: 47.4 W
- P_{active} , power when the card is activated (memory allocated but not yet used): 58.43 W
- P_{pause} , power during a pause in the calculation: 60.77 W
- P_{end} , power at the end of the program, before the process is completed: 59.14 W

These values correspond to the states in Section 2.5.5, Page 32, Figure 2.20(b). The energy consumption is calculated by

$$\begin{aligned} Energy &= P_{idle} \times t_{idle} + P_{active} \times t_{active} + P_{pause} \times t_{pause} + P_{end} \times t_{end} \\ &+ P_{copy}(n) \times T_{copy}(n) + P_{compute}(n) \times T_{compute}(n) \end{aligned} \quad (2.7)$$

with

- t_{idle} , the time during which the program runs without using the GPU card
- t_{active} , the time during which the memory will be allocated without being used
- t_{pause} , the time during which the GPU will not be used in the middle of the calculations, *e.g.*, computation is done on the CPU
- t_{end} , the time during which the main program continues to run and the GPU is not used

To validate your protocol, we will give an evaluation by a real problem in Section 3.5 and Section 3.6, which includes analysis of linear algebra operations and their use together within iterative Krylov methods.

2.6 Benchmark environments

2.6.1 Measurement of execution time

The resolution and accuracy of the clock depends on the hardware environment: host/CPU or device/GPU. The original clock of a graphic card has an accuracy of a few nanoseconds

while the host has an accuracy of a few milliseconds. This difference might negatively influence the objective comparison of the measured times. To overcome this problem, the proposed solution is to perform the same operation several times, at least 10 times and until the total time measured is greater than 100 times the accuracy of the clock. In this thesis we adopted this model for all benchmarks. In particular, we will specify the benchmarking steps explicitly.

2.6.2 Environment for benchmarks

In this dissertation, we have to experiment different analyses. Given the different nature of the experiments, we will have to use several workstations. Obviously, the performance of our algorithms depends on the characteristics of the test machines. The calculations depend on the precision (single or double) and the hardware support. Thus, in this section, we describe all the workstations used in this thesis. Each workstation will be identified by a unique name throughout this thesis.

Platform-1

The first workstation is equipped with an Intel Core i7 920 2.67GHz processor, which has 8 cores composed of four physical cores and four logical cores, 5.8 GB RAM memory and two *NVIDIA GTX275* GPUs fitted with 895MB memory. The workstation hardware and software configuration allows to use CUDA 4.0 features.

Platform-2

The second platform consists of a workstation equipped with an Intel Core i7930 running with 2.67GHz, which has four physical cores and four logical cores, and 12 GB main memory and two *NVIDIA GTX570* GPU with 1279MB memory. The GPU card has the capability of double precision arithmetic and is driven by CUDA 4.0 software.

Platform-3

Platform-3 is composed of an Intel Xeon E3-1225 with four physical cores and four threads, running at 3.1 GHz and 8 GB RAM memory. The GPU consists of an ATI FirePro V4800 equipped with 1 GB memory with 400 stream processors. This graphics card is compatible with OpenCL 1.0.

Platform-4

The fourth platform consists of a workstation equipped with an Intel Core i7930 running with 2.67GHz, which has four physical cores and four logical cores, and 12 GB main memory and two *NVIDIA* graphics cards: a Tesla K20c (dev#0) with 4799GB memory and GeForce GTX 570 with 1279MB memory (dev#1).

Platform-5

The *Platform-4* corresponds to *Titane* supercomputer of CEA-GENCI. *Titane* is an hybrid CPU/GPU cluster, of which a schematic diagram is given in Figure 2.21. The cluster consists of 1,068 compute nodes and 24 nodes dedicated to IO and administration. Each node has 2 Intel Xeon 5570 Nehalem quad-cores (2.93 GHz) and 24 GB of memory (3Go per core). *Titane* includes 48 Tesla S1070 servers, each with 4 processors and 4 GB of memory. Each server is attached to two compute nodes via the PCI-Express bus. The compute nodes are interconnected by a *Voltaire* network, based on the InfiniBand DDR technology. In 2010,

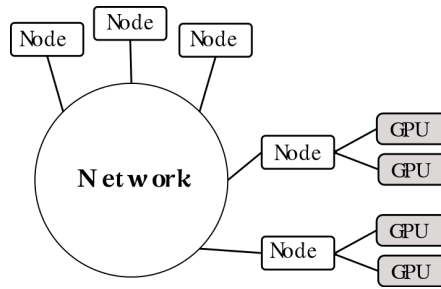


FIGURE 2.21: GENCI-CEA hybrid computing clusters

Titane got an extension of 432 nodes with also 2 Intel Nehalem quad-cores and 24 GB of memory per node. The compiler Intel suite is considered. This system provides a peak performance of 100 Tflops for the Intel part and 200 Tflops for the GPU part. The Tesla S1070 includes 4 processors of 240 cores each, *i.e.*, 960 processing units available for each server.

Platform-6 (MRG/LISA)

MRG/LISA is a in-house cluster of our team. I assembled this cluster during my thesis. *MRG/LISA* is an hybrid CPU/GPU cluster, whose characteristics are given in Table 2.6. The cluster consists of 6 CPU nodes (C_1, C_2, \dots, C_6) and 8 CPU/GPU nodes (G_1, G_2, \dots, G_8). The version of the MPI library used is OpenMPI (*OpenRTE*) 1.6.5.

node	OS	CPU	GPU
C_1, C_2, C_3	Linux 64 bits Ubuntu 14.04 LTS system \prime : 55GB	Intel(R) Xeon(R) E5410 2,33GHz, $2 \times 4 = 8$ cores RAM: 8 GB	none
C_4, C_5, C_6	swap: 30GB /home: 180GB	MPI (OpenRTE) 1.6.5	
G_1, G_2, G_3, G_4	Linux 64 bits Ubuntu 14.04 LTS system \prime : 55GB swap: 30GB /home: 180GB	Intel(R) Core(TM) i7 2,80GHz, $2 \times 4 = 8$ cores RAM: 8 GB MPI (OpenRTE) 1.6.5	Tesla K20c 4799MB GTX 570 1279MB CUDA v6.5 Double precision
G_5, G_6, G_7, G_8	Linux 64 bits Ubuntu 14.04 LTS system \prime : 55GB swap: 30GB /home: 180GB	Intel(R) Xeon(R) E5-2609 2,10GHz, $4 \times 6 = 24$ cores RAM: 16 GB MPI (OpenRTE) 1.6.5	G_5, G_6, G_7 Quadro K4000 3071MB G_8 Quadro K600 1023MB G_5, G_6, G_7, G_8 CUDA v6.5 Double precision
The interconnected network is a switched, star shaped 10Mb/s Ethernet network.			

TABLE 2.6: MRG/LISA hybrid computing clusters

2.7 Conclusion

In this section, we mentioned in the introduction the challenge of GPU computing in the field of computational science. We have give an overview of the GPU programming model and looked closely at hardware specifications in order to understand the specific aspects of graphic card computations and their challenges. We have cared to demonstrate how to capitalize on the use of GPU Computing and how we proceed to optimize CPU and GPU operations, data transfers and memory management. In order to ensure even better efficiency, we have proposed gridification strategies to auto-tune the grid of the GPU architecture, which strongly impacts

the performance of algorithms. In the next chapter, we will give an analysis to evaluate how fast are the algorithms.

In this section, we have also established an experimental protocol for measuring the power consumption of GPU during computation.

Due to the lack of a GPU monitoring sensor for state and energy consumption, we used amperometric clamps to directly measure the current provided to GPU. Precise phase analysis of the time-dependent current of GPU combined with an efficient noise reduction technique provided accurate timing of GPU states and the intensity of each state's current.

By applying our experimental protocol to fundamental and basic GPU operations, DAXPY and the element wise product, which are completely parallelized within the GPU, and to memory copy operations between CPU and GPU, we have determined the energy consumption of the GPU for an idling state, an activated state, a pausing state, and a finalizing state. It was interesting to discover that GPU consumes a certain amount of energy, even during its idling state, by maintaining the state of the GPU global memory.

The next chapter presents linear algebra operations on multi-cores/multi-GPUs environment. We describe how to implement the efficient iterative Krylov methods on GPU by using gathered experience. Different Krylov methods are then developed and compared for different data from real problems.

Linear Algebra on Parallel and GPU Computing

” Create your own visual style... let it be unique for yourself and yet identifiable for others.

— Orson Welles
(Actor)

3.1 Introduction

FOR many scientific and engineering applications, the overall computational performance is the direct derivative of the performance of the linear algebraic equations solver, and this in turn can be related to the performance of basic linear algebra operations, like dot products, scaling or matrix-vector products. Engineering problems involve the solution of large sparse linear systems, which could be very expensive in terms of computing time for classical CPU. They require therefore fast and high-performance algorithms for algebra operations. For sparse vectors and matrices, it is the matrix-vector product that appears to be the most time-consuming operation. The basic indicator for the computational complexity of sparse linear algebra operations is the number of non-zero values (nnz) of the matrix. For the Krylov algorithm, the operations that are performed for each iteration step consist of several scaled vector additions, dot products and one or more sparse matrix-vector multiplications. In this chapter of the dissertation, we aim focus on the best way to perform these operations effectively in a multi-core/multi-GPUs environment, in order to make solvers such as the iterative Krylov methods, more robust and therefore reduce the computing time. In this thesis, we have also paid attention to alerting readers about the energy consumption of the hardwares used, especially the graphics card. The performance of our algorithms is evaluated on several matrices arising from engineering problems and compared to different existing libraries.

Matrix and vector computations constitute the core of many scientific numerical programs. Many linear algebra libraries such as BLAS [55] [56], LINPACK [57] propose various algorithms of linear algebra operations on CPU. Others, such as ITSOL [58] for sparse matrices or MAGMA [59] for dense matrices, provide functions to solve linear systems. Since the apparition of GPU Computing, researchers have paid more attention to graphics cards in order to accelerate their algorithms. As explained in this chapter, over the past decade, GPU has clearly demonstrated better performance compared to classical CPU, and many libraries appear in the scientific community, such as: cuBLAS [60], a BLAS level-one GPU version, cuSPARSE [61], a BLAS level-2 GPU version, Cusp [62], a library of sequential and GPU iterative solvers.

The performance of algorithms varies depending on the library used. For example, a library may lead to a faster matrix-vector product than another, but may not be faster for other operations. The idea that was chosen in my thesis consists in implementing our own library,

which provides the necessary algorithms of linear algebra and various linear solvers. We also paid attention to their optimizations. Thus, we can address the implementation of advanced parallel solvers such as parallel sub-structuring methods, domain decomposition methods, etc., with optimized, robust and effective linear algebra operations on both CPU and GPU environments. We illustrate in this chapter how these calculations can be accelerated using innovative algorithms. We explain how numerical linear algebra solvers can be designed to harness the power of current heterogeneous multi-core-GPU environments.

My contribution in this chapter has been to develop, implement and test innovative algorithms for computing linear algebra operations on GPU and their use together within solvers.

My second contribution is to build an accurate energy consumption prediction model of the GPU. This helps predict the energy needs of a program, and allow a better task scheduling with a given computational time.

My contribution is also the proposal of “An Advanced Linear Algebra Library for Massively Parallel Computations on Graphics Processing Units”, to provide a new vision of GPU computing that helps harness the power of GPUs easily, without the programming complexity of classical APIs such as CUDA and OpenCL.

This chapter is organized as follows. First, an introduction to linear system solvers is given in Section 3.2, which includes matrix storage formats and a description of two basic classes of methods for solving linear systems, the direct and iterative methods, especially the iterative Krylov methods. For convenience and thoroughness, Section 3.3 gives an overview of the existing scientific libraries targeted for GPU. In Section 3.4 and Section 3.5 respectively, we discuss the best way to efficiently perform BLAS level-1 linear algebra operations and sparse matrix-vector multiplication (SpMV) on GPU. Hints for heterogeneous multi-core-GPU are given. Section 3.6 describes how solvers, particularly iterative Krylov methods are implemented on GPU and multi-core systems, and how the performance of the basic linear algebra operations influence the overall performance of these methods. These sections also describe, evaluate, analyze and present the optimizations we have developed. Numerical experiments performed on a set of real problems show the robustness, competitiveness and efficiency of the proposed implementation on GPU for computing linear algebra operations and solving linear equations. Section 3.7 presents an overview of the library we have implemented, Alinea, and how its usage is competitive. Section 3.8 concludes the chapter.

Keywords — BLAS-1, BLAS-2, BLAS-3, Saxpy, Element-wise product, Dot product, Storage formats, Sparse matrix, CSR, CSC, COO, ELL, HYB, Matrix-vector product, SpMV, Linear system, Solver, Direct methods, Iterative methods, Krylov, CG, GCR, Bi-CGSTAB, Bi-CGSTAB(1), QMR, TFQMR, Bi-CGCR, Preconditioning techniques, Diagonal preconditioner, Gravitational potential equation, Acoustic problem, Helmholtz equation, Alinea, cuBLAS, cuSPARSE, Cusp

3.2 Sparse linear system solvers

For applied sciences and engineering problems, a system of partial differential equations (PDEs) is used as a fundamental physical and mathematical model which is solved numerically. Nowadays, several physical parameters can be taken into account and different physical modelings can be coupled and then the PDE system becomes more and more complex. To obtain a numerical solution of PDE, discretization methods (finite difference/volume/element methods) are applied, and linear systems with very large sparse matrices are obtained after some linearization process, if necessary. In scientific computing, solving large linear systems is often the most expensive part, both in terms of memory space and computation time. A system of linear equations can be written in the matrix notation as

$$Ax = b, \quad (3.1)$$

where A is a $n \times n$ matrix, b the right-hand side in \mathbb{C}^n and x in \mathbb{C}^n represents the solution vector we are looking for. The system (3.1) has a unique solution if and only if the A is non-singular, which means that the determinant of the matrix A is non-zero.

The rest of this section will discuss suitable methods for solving sparse linear systems. We will concentrate only on systems of equations with unique solutions. There are several methodologies to solve this type of large linear system, *e.g.*, sparse direct solvers, iterative solvers, and a combination of those. However, the choice of method to solve the linear system is often driven by the properties of the matrix related to the system. The choice also depends on the speed of the method and the desired accuracy for the resolution. The size of the system can also be a determining factor in the choice of the method. Indeed, the size of the system may further slow the resolution process. Due to the hypothesis of a large linear system, these methods must be facile to parallelize in order to better harness the computing power and to better manage the memory capacity of parallel computing platforms. The methods for solving the linear system (3.1) are classified in two categories: direct methods and iterative methods. A direct method or an iterative method may be faster or more accurate according to the properties of the matrix A and the speed with which solution x_i converges to $x = A^{-1}b$. Several numerical methods, for instance the finite element method, result in algebraic problems for large-size sparse matrices. Because of their size, these matrices cannot be stored as a simple set of vectors. Instead, taking advantage of their great amount of zero values, we focus on storing matrices in compressed formats, *i.e.*, only non-zero elements are allocated. First, we present diverse data structures that can be considered to store matrices more efficiently in memory. Then, we discuss the direct methods, followed by the iterative methods that are grouped into two basic classes: stationary (*e.g.*, Jacobi), and non-stationary (*e.g.*, Krylov methods). For the sake of generality, we work over the field of complex numbers \mathbb{C} .

3.2.1 Matrix storage format

Origin and Motivation of sparse matrices

The treatment of large sparse matrices in parallel requires a good choice of storage format that helps the computations of involved operations. The basic idea behind sparse matrix storage is to store only the non-zero matrix elements. The distribution of non-zero coefficients

depends on the features of the original problem. Sparse matrix is called *structured* when the non-zero values form a regular pattern along diagonals, otherwise it is called *unstructured*. The performance of the algorithms strongly depends on the data structure of the sparse matrices as demonstrated in [63] [31] [64] [65] [66] [53] [67] [68], and many others.

Such data structures offer several advantages in terms of memory usage and algorithm execution strategy, compared to naive matrix data structure implementation. Unfortunately, each special data format also exhibits disadvantages and trade-offs when compared to other special purpose formats. The purpose of this part is to present the data structures, as well as their advantages and disadvantages. The purpose of each of these formats is to gain efficiency both in terms of the number of arithmetic operations and memory usage. We describe in detail the following storage schemes: COOrdinate format (*COO*), Compressed-Sparse Row format (*CSR*), Compressed-Sparse Column format (*CSC*), ELLPACK format (*ELL*) and HYBRid format (*HYB*). The sparse matrix A described in Figure 3.1(a) will be used to illustrate each case. In the following examples, we consider matrices indexed from 1. Figure 3.1(b) draws the pattern of non-zero values of the matrix A .

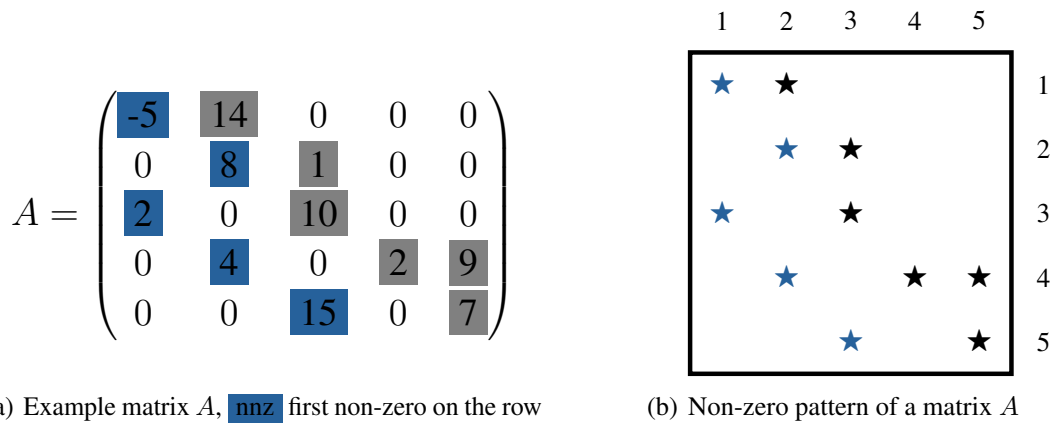


FIGURE 3.1: Example of non-zero pattern of a sparse matrix A

COO format

The structure of the COOrdinate (*COO*) Figure 3.2 format is trivial. According to [69], COO format consists of three one-dimensional arrays of size nnz , *i.e.*, the number of non-zero matrix coefficients. An array AA holds the non-zero coefficients in such a way that $AA(i) = A(IA(i), JA(i))$, $1 \leq i \leq n$, where IA and JA store respectively the indices of rows and columns of the i^{th} value of A . The COO format is efficient for unstructured matrices.

$$\begin{aligned}
 IA &= 1 / 1 / 2 / 2 / 3 / 3 / 4 / 4 / 4 / 5 / 5, \\
 JA &= 1 / 2 / 2 / 3 / 1 / 3 / 2 / 4 / 5 / 3 / 5, \\
 AA &= -5 / 14 / 8 / 1 / 2 / 10 / 4 / 2 / 9 / 15 / 7
 \end{aligned}$$

FIGURE 3.2: COOrdinate (*COO*) storage format

Indeed, when the sparse matrix is structured, the COO format may contains successive redundant informations. When the matrix is large in size, the COO format will lose its interest, especially in terms of memory space. An alternative of this format consists in compressing

the index of rows or the index of columns. The following presents two formats based on this compression.

CSR/CSC formats

Compressed Sparse Row (*CSR*), described in Figure 3.3, is widely used because of minimal memory usage and the simplicity of the implementation. The sparse matrix $A \in \mathbb{C}^{n \times n}$ is stored in three one-dimensional arrays. Two arrays of size nnz , AA and JA , store respectively the non-zero values, through major row storage (row by row) and the column indices, thus $JA(k)_{1 \leq k \leq nnz}$ is the column index in A matrix of $AA(k)_{1 \leq k \leq nnz}$. Finally, IA , an array of size $n + 1$ that stores the list of indices at which each row starts. $IA(i)_{1 \leq i \leq n}$ and $IA(i + 1) - 1$ correspond respectively to the beginning and the end of the i^{th} row in arrays AA and JA , i.e., $IA(n + 1) = nnz + 1$. It is interesting to mention the Compressed-Sparse Column

$$\begin{aligned} AA &= -5 / 14 / 8 / 1 / 2 / 10 / 4 / 2 / 9 / 15 / 7, \\ JA &= 1 / 2 / 2 / 3 / 1 / 3 / 2 / 4 / 5 / 3 / 5, \\ IA &= 1 / 3 / 5 / 7 / 10 / 12 \end{aligned}$$

FIGURE 3.3: Compressed-Sparse Row (*CSR*) storage format

storage format (*CSC*) Figure 3.4, which is the transpose of the *CSR* format. This format is the same as above, but replacing the role of the rows by those of columns. As the *CSR* format, the sparse matrix $A(n \times m)$ is stored in three one-dimensional arrays. Two arrays of size nnz , AA and IA , store respectively the non-zero values, through major column storage (column by column) and the row indices, thus $IA(k)_{1 \leq k \leq nnz}$ is the row index in A matrix of $AA(k)_{1 \leq k \leq nnz}$. Finally, JA , an array of size $m + 1$ that stores the list of indices at which each column starts. $JA(j)_{1 \leq j \leq m}$ and $JA(j + 1) - 1$ correspond respectively to the beginning and the end of the i^{th} column in arrays AA and IA , i.e., $JA(m + 1) = nnz + 1$.

Figure 3.4 describes the corresponding *CSC* storage of the matrix A .

$$\begin{aligned} AA &= -5 / 2 / 14 / 8 / 4 / 1 / 10 / 15 / 2 / 9 / 7, \\ IA &= 1 / 3 / 1 / 2 / 4 / 2 / 3 / 5 / 4 / 4 / 5, \\ JA &= 1 / 3 / 6 / 9 / 10 / 12 \end{aligned}$$

FIGURE 3.4: Compressed-Sparse Column (*CSC*) storage format

ELLPACK format

The *ELLPACK* (*ELL*) format [70], illustrated in Figure 3.5, for a $n \times n$ sparse matrix consists in storing a $n \times k$ dense matrix (where k is the maximum number of non-zero values by row), $COEF$, and a full dense matrix, $JCOEF$, which stores the column indices of non-zero elements. In terms of implementation, the star symbols \star in $COEF$ (see Figure 3.5(a)) and $JCOEF$ (see Figure 3.5(b)) can be replaced by any arbitrary value and not be ambiguous. We have chosen zero in the algorithms used in this thesis. In fact, zero corresponds to a fictitious value. For example we could use -1 . For robustness reasons, we can put a coefficient that refers to a zero coefficient of $COEF$ to increase the alignment and regularity of memory

access. This type of storage is very efficient for each row when the number of non-zero values approaches the average number of non-zero values per line, *i.e.*, when the standard deviation of the number of non-zero values by row is low. Indeed, if a sparse matrix has a full line, the storage of the matrix in ELL format takes more space than the matrix itself, therefore this format loses all interest. The COO format allows rows of different sizes without wasting

$$COEF = \begin{pmatrix} -5 & 14 & \star \\ 8 & 1 & \star \\ 2 & 10 & \star \\ 4 & 2 & 9 \\ 15 & 7 & \star \end{pmatrix} \qquad JCOEF = \begin{pmatrix} 1 & 2 & \star \\ 2 & 3 & \star \\ 1 & 3 & \star \\ 2 & 4 & 5 \\ 3 & 5 & \star \end{pmatrix}$$

(a) ELL dense *COEF* matrix (b) ELL dense *JCOEF* matrix

FIGURE 3.5: ELLPACK (ELL) storage format

memory space in contrary to the ELL storage.

HYBrid format

The *HYBrid* (*HYB*) format introduced by Bell and Garland [31] is based on the ELL and COO formats. The *HYB* structure consists of two parts: the ELL part and COO part, as illustrated in Figure 3.6 for a 5×5 matrix. The principle of this format in storing a $n \times k$ dense matrix for the ELL part where k is a typical number, threshold, that depends on both the context and the structure of the matrix. To determine the ELL portion, *i.e.*, the number of columns of the ELL matrix, as presented in [32], which use a heuristic for computing the largest number k such that there are at least $K = \max(4096, \frac{nnz}{3})$ rows with k or non-zero values, in a matrix with nnz non-zero values. All non-zero values of the matrix that would result in decreasing the benefits of ELL format are stored in COO format (see Figure 3.6(b)). In the example given in Figure 3.6, the typical number $k = 2$. To conclude, the *HYB* format

$$COEF = \begin{pmatrix} -5 & 14 \\ 8 & 1 \\ 2 & 10 \\ 4 & 2 \\ 15 & 7 \end{pmatrix}; JCOEF = \begin{pmatrix} 1 & 2 \\ 2 & 3 \\ 1 & 3 \\ 2 & 4 \\ 3 & 5 \end{pmatrix} \qquad AA = [9]; JA = [5]; IA = [4]$$

(a) ELL part of the *HYBrid* format (b) COO part of the *HYBrid* format

FIGURE 3.6: *HYBrid* (*HYB*) storage format

benefits both from the efficiency of the ELL format due to its regular memory access, and from the flexibility of the COO format that is insensitive to the distribution of non-zero values of the matrix.

3.2.2 Direct methods

The principle of the classical direct methods, the oldest methods for solving linear systems, is discussed in this section. They are called direct methods, because they theoretically give an exact solution in a predictable finite number of operations. A direct algorithm ensures the absence of errors related to the method, the only errors that occur are roundoff errors. In the absence of roundoff errors, the algorithm would give the exact solution of $Ax = b$. The

basic idea behind the direct methods consists in decomposing the matrix into a product of particular and suitable matrices, which are easier to solve with a direct algorithm. This process consists of the factorization of the matrix. The matrices obtained after the factorization are simple, based on diagonal matrices and triangular matrices. The solution is then obtained by successively solving problems associated with each of the factors of the decomposition. This second step of the resolution is called successive substitution algorithms using forward or/and backward techniques.

For the direct methods, we will focus on the LU factorization method, which is one of the most used and robust methods. The LU decomposition is a method that consists in decomposing a matrix into a product of a lower triangular matrix L (see Figure 3.7(a)) and an upper triangular matrix U (see Figure 3.7(b)). As said above, once the factorization is

$$L = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ l_{2,1} & 1 & 0 & \cdots & 0 \\ l_{3,1} & l_{3,2} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \cdots & l_{n,n-1} & 1 \end{pmatrix} \quad U = \begin{pmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,n-1} & u_{1,n} \\ 0 & u_{2,2} & \cdots & u_{2,n-1} & u_{2,n} \\ 0 & 0 & \cdots & u_{3,n-1} & u_{3,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & u_{n,n} \end{pmatrix}$$

(a) Lower triangular matrix L

(b) Upper triangular matrix U

FIGURE 3.7: LU decomposition into a product of a lower triangular matrix L and an upper triangular matrix U

completed, the linear system is solved with two successive substitutions: $Ly = b$ by forward algorithm and then $Ux = y$ by backward algorithm. Unfortunately, direct methods have to perform a significant number of arithmetic operations and are greedy in terms of memory space. The usage of sparse matrices techniques in terms of storage and computation is a solution to improve the robustness and effectiveness of the LU decomposition method. However, the implementation of sparse direct methods is more complicated and requires an additional effort. The main difficulty lies in the management of the sparse data structure. Indeed, some zero values of the initial matrix could become non-zero during the factorization, which involves rearranging the data, i.e zero value in A does not necessarily imply a zero value in L or U . LU factorization strongly depends on the sparse structure of the initial matrix and also on the associated lower and upper triangular matrices. References [71] [72] show that a sparse direct method is usually decomposed in four steps:

1. **Pretreatments and symbolic factorization:** reorders the rows and columns of the matrix in order to reduce the filling and re-balances the terms in order to limit rounding errors, i.e., improve the accuracy of calculations. This phase is crucial for computational efficiency. This steps also consists in constructing the sparse structure of factorized matrices. It estimates the tasks' tree of dependency and the total memory consumption planned.
2. **Step of renumbering:** renumbers the order of the matrix's unknowns using the permutation matrix in order to reduce the filling that implies the factorization.
3. **Phase of numerical factorization:** this is the most expensive phase, which will explicitly construct the sparse factorizations LU .

4. **Solving the sparse linear system:** performs the successive forward and backward substitutions, from factors L and U respectively.

The steps 1 and 4 are independent while the steps 1, 2 and 3 are related. According to the algorithmic approach, these steps are managed differently. It is interesting to mention the existence of optimized libraries for solving sparse linear systems with direct methods such as *MUMPS* [73] where steps 1 and 2 are merged, *SuperLu* [74] where steps 1 and 3 are merged and *UMFPACK* [75] where 1, 2 and 3 are related. The implementation of the LU solver on GPU [53] is even more complicated. The complexity of the direct solvers is $\mathcal{O}(n^3)$, where n is the size of the matrix. For large-size linear systems, iterative methods are more appropriate. The next two subsections discuss iterative methods.

3.2.3 Iterative methods

Iterative methods are used either for solving large linear systems, or when we would like to improve a given estimate of a solution. An iterative method consists in constructing a sequence $x^{(0)}, x^{(1)}, x^{(2)}, x^{(3)}, \dots$ of approximate solutions of $Ax = b$, which stops iterating when the current solution $x^{(k)}$ is accurate enough. At the stopping step, the convergence suite $(x^{(k)})_{k \in \mathbb{N}}$ satisfies the relation (3.2).

$$\lim_{k \rightarrow +\infty} x^{(k)} = x^*, \quad (3.2)$$

where x^* is the solution of the system (3.1), which could be considered as the exact solution ($x^* = A^{-1}b \in \mathbb{C}^n$) under some conditions. The suite $(x^{(k)})_{k \in \mathbb{N}}$ is obtained from a given starting point $x^{(0)}$ by iteratively performing a relation that computes $x^{(k+1)}$ from an already known $x^{(k)}$. In contrast to direct methods, the number of steps required by iterative methods for reaching convergence is not predictable in advance. There are different classes of iterative methods for solving linear systems. In this section, we discuss two categories. The first category is called *stationary iterative methods* (e.g., Jacobi) and the second is called *non-stationary iterative methods* (e.g., Conjugate Gradient). The suite $(x^{(k)})_{k \in \mathbb{N}}$ constructed from an arbitrary initial guess $x^{(0)}$ can be expressed by the recurrence relation (3.3).

$$x^{(k+1)} = B^{(k)}x^{(k)} + C^{(k)}b, \quad k \in \mathbb{N}, \quad (3.3)$$

where $B^{(k)} \in \mathbb{C}^{n \times n}$ and $C^{(k)} \in \mathbb{C}^{n \times n}$ are the *iteration matrices* of the method. The nature of the iterative methods differs with the choice of $B^{(k)}$ and $C^{(k)}$. Different choices of $B^{(k)}$ and $C^{(k)}$ follow. Before presenting the principle of both classes of methods, we will give some general results concerning iterative methods.

Definition 3.1 An iterative method is *convergent* if (3.2) from any starting point $x^{(0)} \in \mathbb{C}^n$.

Definition 3.2 The iterative method (3.3) is *consistent* with the system (3.1) if the iteration matrices $B^{(k)}$ and $C^{(k)}$ satisfy (3.4).

$$x^* = B^{(K)}x^* + C^{(K)}b, \quad k \in \mathbb{N}, \quad (3.4)$$

where x^* is the solution of the system (3.1), converged at the K^{th} iteration, and $B^{(K)}$ and $C^{(K)}$ are the corresponding K^{th} matrices. Considering the exact solution $x^* = A^{-1}b$ and the relation (3.4), the iteration matrix, $C^{(K)}$, is expressed as follows

$$C^{(K)} = (I_n - B^{(K)})A^{-1}. \quad (3.5)$$

Definition 3.3 Let us define $e^{(k)}$ the *error* vector at the iteration k of the iterative method such as

$$e^{(k)} = x^{(k)} - x^*, \quad k \in \mathbb{N}, \quad (3.6)$$

where $x^* = A^{-1}b$ is the solution of the system (3.1). We call the vector *residual*:

$$r^{(k)} = b - Ax^{(k)}, \quad k \in \mathbb{N}. \quad (3.7)$$

Lemma 3.4 A consistent iterative method of the form (3.4) converges to the solution of the system (3.1), from any chosen $x^{(0)}$ iff

$$\lim_{k \rightarrow +\infty} e^{(k)} = 0, \quad k \in \mathbb{N}, \quad (3.8)$$

That also ensures $\lim_{k \rightarrow +\infty} r^{(k)} = \lim_{k \rightarrow +\infty} Ae^{(k)} = 0, \quad k \in \mathbb{N}$.

Theorem 3.5 An iteration sequence $x^{(k)}, k \in \mathbb{N}$ of the form (3.4) converges to the solution of the system (3.1), from any chosen $x^{(0)}$ iff $\lim_{k \rightarrow +\infty} e^{(k)} = 0, \quad k \in \mathbb{N}$, which means that

$$\lim_{k \rightarrow +\infty} B^{(k)}B^{(k-1)} \dots B^{(0)} = 0, \quad k \in \mathbb{N}, \quad (3.9)$$

An iterative algorithm computes an approximate solution of the system (3.1). In practice, the iteration process should be stopped at the first iteration for which the error is “sufficiently small”. We define three stop criteria: *absolute error* (3.10), *relative error* (3.11) and *residual solution* (3.12). Let us define K the maximum number of iterations, ε the tolerance threshold required for calculation accuracy and $\|\cdot\|$ a given vectorial norm.

Definition 3.6 The *absolute error* criterion is defined as follows

$$\|x^{(k+1)} - x^{(k)}\| < \varepsilon \quad (3.10)$$

where $x^{(k)}$ is the approximate solution at iteration $k \geq K$.

Definition 3.7 The *relative error* criterion is defined as follows

$$\frac{\|x^{(k+1)} - x^{(k)}\|}{\|x^{(k)}\|} < \varepsilon \quad (3.11)$$

where $x^{(k)}$ is the approximate solution at iteration $k \geq K$.

Definition 3.8 The *residual solution* criterion is defined as follows

$$\frac{\|r^{(k)}\|}{\|b\|} = \frac{\|b - Ax^{(k)}\|}{\|b\|} < \varepsilon \quad (3.12)$$

where $x^{(k)}$ is the approximate solution at iteration $k \geq K$.

The algorithm associated with the iterative method converges depending on the properties of the linear system. The rate of convergence also depends on the properties of the linear system.

Stationary iterative methods

Stationary iterative methods consist of iterative methods that satisfy (3.3) with constant iteration matrices, $B^{(k)} = B$ and $C^{(k)} = C$, $k \in \mathbb{N}$. The recurrence relation becomes (3.3).

$$x^{(k+1)} = Bx^{(k)} + Cb, \quad k \in \mathbb{N}. \quad (3.13)$$

Theorem 3.9 *An iteration sequence $x^{(k)}$, $k \in \mathbb{N}$ of the form (3.13) converges to the solution of the system (3.1), from any chosen $x^{(0)}$ iff*

$$\lim_{k \rightarrow +\infty} \|B^k\| = 0. \quad (3.14)$$

A necessary and sufficient condition that ensures $\lim_{k \rightarrow \infty} \|B^k\| = 0$ is that the spectral radius $\rho(B) < 1$.

Proof. Let us assume that the method is of the form (3.4), i.e., consistent to the system (3.1). The error at the iteration $k + 1$, $k \in \mathbb{N}$, satisfies the following relation of recurrence

$$\begin{aligned} e^{(k+1)} &= x^{(k+1)} - x^* \\ &= Bx^{(k)} + Cb - (Bx^* + Cb) \\ &= B(x^{(k)} - x^*) \\ &= Be^{(k)} \\ &= B^k e^{(0)} \end{aligned}$$

The stationary iterative algorithm converges if $\lim_{k \rightarrow \infty} \|\zeta^{(k)}\| = 0$, which means that $\lim_{k \rightarrow \infty} \|B^k\| = 0$, i.e. the matrix B^k be close to the zero matrix. A necessary and sufficient condition that ensures $\lim_{k \rightarrow \infty} \|B^k\| = 0$ is that the spectral radius $\rho(B) < 1$. \square

For a dense matrix, the complexity of iterative methods is $\mathcal{O}(n^2)$ operations at each iteration. As seen above, the complexity of the direct solvers is $\mathcal{O}(n^3)$, where n is the number of equations. Therefore, an iterative method will be competitive compared to direct methods if it converges into a number of iterations that does not increase linearly with the size of the matrix. The main stationary iterative methods are based on a “*splitting*” of the matrix A of the form

$$A = M - N, \quad (3.15)$$

where $M \in \mathbb{C}^{n \times n}$ is a non-singular matrix and $N \in \mathbb{C}^{n \times n}$. The stationary iterative methods are motivated by the following observation. Let A be a matrix with non-zero diagonal values, $M \in \mathbb{C}^{n \times n}$ be a non-singular matrix and $N \in \mathbb{C}^{n \times n}$ a matrix such as $A = M - N$. The system (3.1) can be rewritten as $(M - N)x = b \Leftrightarrow Mx = Nx + b$, which gives

$$Mx^{(k+1)} = Nx^{(k)} + b, \quad k \in \mathbb{N}. \quad (3.16)$$

The equation (3.16) can be rewritten as

$$x^{(k+1)} = M^{-1}Nx^{(k)} + M^{-1}b, \quad k \in \mathbb{N}. \quad (3.17)$$

The iteration matrices are: $B = M^{-1}N$ and $C = M^{-1}$. Different choices of $M \in \mathbb{C}^{n \times n}$ and $N \in \mathbb{C}^{n \times n}$ define different stationary iterative methods. The *Jacobi* and *Gauss-Seidel* methods that will be discussed in this thesis are a particular instance of stationary iterative methods with:

- **Jacobi**: $M = D$ and $N = -(L + U)$,
- **Gauss-Seidel**: $M = D + L$ and $N = -U$,
- **Richardson (constant step descent)**: $M = \frac{1}{\alpha}I$ and $N = \frac{1}{\alpha}I - A$,
- **Successive overrelaxation method (SOR)**: $M = \frac{\omega L + D}{\omega}$ and $N = \frac{(1 - \omega)D - \omega U}{\omega}$.

It is interesting to mention that the *Richardson* method is stationary with $B = I_n - \omega A$ and $C = \omega$, $\omega \in \mathbb{R}^{+*}$. Figure 3.8 shows the definition of matrices D , L and U . Algorithm 3.1

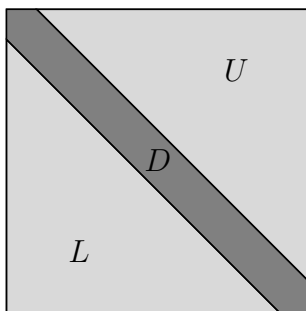


FIGURE 3.8: Design of the splitting algorithm

gives the main steps of the algorithm of stationary iterative methods.

Algorithm 3.1: Algorithm of the stationary iterative methods

input : $A \in \mathbb{C}^{n \times n}$: initial matrix, $B \in \mathbb{C}^{n \times n}$ and $C \in \mathbb{C}^{n \times n}$: iteration matrices,
 b : right-hand side vector, $x^{(0)}$: initial guess,
 ε : tolerance threshold, K : maximum number of iterations

output : x : solution vector

variable: k , convergence

```

1 // initialization
2 convergence ← false
3 k ← 0
4 // loop until convergence
5 while .not. convergence .OR. k < K do
6   compute  $x^{(k+1)} \leftarrow Bx^{(k)} + Cb$ 
7   convergence ←  $\left( \frac{\|b - Ax^{(k)}\|}{\|b\|} \leq \varepsilon \right)$ 
8   k ← k + 1
9 end
```

As said above, the update of the solution performed at Line 6 of Algorithm 3.1 becomes:

- $x^{(k+1)} = D^{-1}b - D^{-1}(L + U)x^{(k)}$ for **Jacobi**, where D is the diagonal part of A and non-singular, L and U are the strictly lower and upper triangular of A . So, the method

consists in computing with a given $x^{(0)}$ the sequence of vectors $x^{(k)}$, which are defined as follows

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^n a_{ij} x_j^{(k)} \right), \quad i = \{1, \dots, n\}, \quad k \in \mathbb{N}. \quad (3.18)$$

- $x^{(k+1)} = (D + L)^{-1}b - (D + L)^{-1}Ux^{(k)}$ for **Gauss-Seidel**, which can be rewritten as

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^n a_{ij} x_j^{(k)} \right), \quad i = \{1, \dots, n\}, \quad k \in \mathbb{N}. \quad (3.19)$$

The convergence test is controlled using the relative error on the residual vector for a given precision, ε , as described at Line 7 of Algorithm 3.1. Unlike the Gauss-Seidel algorithm, the Jacobi algorithm converges more slowly, but is fully parallelizable. More details will emerge on these methods, when we have to analyze them. The choice of method is driven by the properties of the linear system as said previously, but it is often motivated by the speed of convergence of the algorithm. Stationary iterative methods are suitable for sparse matrices. However, the speed of convergence remains insufficient. Recently, non-stationary methods such as the iterative Krylov methods have demonstrated [76] [77] [78] [79] their efficiency and robustness for solving linear systems, especially sparse linear systems. They have also proved their efficiency on GPU and multi-core/multi-GPUs [80] [81] [82] [83] [84]. The value of these methods is discussed below.

Non-stationary iterative methods

The non-stationary iterative methods consist of iterative methods with non-constant iteration matrices $B^{(k)}$ and $C^{(k)}$ in the recurrence relation $x^{(k+1)} = B^{(k)}x^{(k)} + C^{(k)}b$, $k \in \mathbb{N}$. These methods involve data that changes at every iteration. The non-stationary methods are the more recent iterative methods. The non-stationary methods are harder to understand, but are more effective compared to stationary methods, which have a slow convergence but simpler to understand. The methods, in general, are based on the idea of orthogonal vectors and subspace projections such as the *Krylov iterative methods*, which are based on the *Krylov subspace*. The main idea of the iterative Krylov methods introduced by Youcef Saad [69] consists in generating a basis of the Krylov subspace and looking for an approximate solution $x^{(k)}$ of the system (3.1) from this subspace, \mathcal{K}_k .

Krylov subspace methods The Krylov subspace methods are the most representative iterative methods, where an approximate solution is updated repeatedly using products of a sparse matrix and some vectors.

In a Krylov subspace method,

$$x^{(k)} - x^{(0)} \in \mathcal{K}_k(A, r^{(0)}) = \text{span} \{r^{(0)}, Ar^{(0)}, A^2r^{(0)}, \dots, A^{k-1}r^{(0)}\}, \quad (3.20)$$

where $A \in \mathbb{C}^{n \times n}$ is a square matrix and $r^{(0)} = b - Ax^{(0)}$ the residual of a given non-zero initial vector of solution.

We call $\mathcal{K}_k(A, r^{(0)})$ a k -th Krylov subspace. Equivalently,

$$x^{(k)} \in x^{(0)} + \text{span} \{r^{(0)}, Ar^{(0)}, A^2r^{(0)}, \dots, A^{k-1}r^{(0)}\}. \quad (3.21)$$

Recall that the minimal polynomial of a vector b is the non-zero monic polynomial P of the lowest degree such that $P(A)b = 0$. Clearly, the Krylov subspace \mathcal{K}_k is the subspace of all vectors in \mathbb{C}^n , which can be written as $x = P(A)b$, where P is a polynomial of degree not exceeding $k - 1$. In other terms, the idea of the Krylov subspace methods is to construct a basis of the Krylov subspace $\mathcal{K}_k = \text{span} \{r^{(0)}, Ar^{(0)}, A^2r^{(0)}, \dots, A^{k-1}r^{(0)}\}$, and seek an approximate solution $x^{(k)}$ from this subspace, \mathcal{K}_k . Given an initial guess $x^{(0)}$ to the linear system (3.1), a general *projection method* looks for an approximate solution $x^{(k)}$ from an affine subspace $x^{(0)} + \mathcal{K}_k$ of order k such that the *Petrov-Galerkin* condition is satisfied:

$$b - Ax^{(k)} \perp \mathcal{L}_k \quad (3.22)$$

where \mathcal{L}_k is another Krylov subspace of order k . Krylov methods differ in the choice of \mathcal{L}_k and \mathcal{K}_k . The following give an overview of the iterative Krylov methods with different choices of \mathcal{L}_k and \mathcal{K}_k . The *Conjugate Gradient (CG)* method is a particular instance of the Krylov subspace method with $\mathcal{L}_k = \mathcal{K}_k = \mathcal{K}_k(A, r_0)$ when the matrix is hermitian positive-definite. The *Conjugate Residual (GCR)* method, an analogue of the conjugate gradient method, is a Krylov subspace method with $\mathcal{L}_k = A\mathcal{K}_k$ where $\mathcal{K}_k = \mathcal{K}_k(A, r_0)$. This choice of \mathcal{L}_k gives an approximate solution $x^{(k)}$ which minimizes the residual norm $\|b - Ax\|_2$ over all vectors in $x_0 + \mathcal{K}_k$. Presented in 1986 by *Y. Saad and M. H. Schultz* in [85], the *Generalized Minimal RESidual (GMRES)* algorithm, is an extension of the GCR method to non-hermitian problems. The GMRES algorithm is a projection method based on the *Arnoldi algorithm* [86]. GMRES uses a *Gram-Schmidt modified* [87] [88] algorithm to compute an orthonormal basis $\{v_1, v_2, \dots, v_k\}$ of the Krylov subspace $\mathcal{K}_k(A, v_0) = \text{span} \{v_0, Av_0, A^2v_0, \dots, A^{k-1}v_0\}$. There exist many other Krylov methods such as Bi-Conjugate Gradient Conjugate Residual (BiCGCR), transpose-free Quasi Minimal Residual (TFQMR), Stabilized BiConjugate Gradient (BiCGStab), Stabilized BiConjugate Gradient (L) (BiCGStabl), etc. In general, Krylov methods have a rapid convergence compared to the stationary methods. The conditioning of the matrix has a fundamental role in the convergence speed of the Krylov method. Nevertheless, certain sparse linear systems require a high number of iterations to converge. In practice, preconditioning techniques are used in order to help the Krylov methods accelerate their convergences [89] [90] [91] [92]. Preconditioning a linear system consists in replacing it with an equivalent system, where the conditioning of the matrix is smaller, which allows a faster convergence. The basic principle consists in replacing the sparse linear system (3.1) with one of the following systems:

$$MAx = Mb, \quad (3.23)$$

$$AM\hat{x} = b, \quad \text{with } x = M\hat{x}, \quad (3.24)$$

where $M \in \mathbb{C}^{n \times n}$ is an approximation of A^{-1} . The system (3.23) is called *left preconditioning* and system (3.24) consists in *right preconditioning*. In general, there are two approaches for constructing preconditioners: *matrix-based* and *operator-based*. Within the first class, the

most basic one is the diagonal (Jacobi) preconditioning. We also find the incomplete LU (ILU) factorizations. Examples of *operator-based* preconditioners are: separation of variables [93] and analytic ILU (AILU) [94].

In practice, the iterative Krylov methods performance depends on the matrix conditioning, and efficient preconditioning techniques must be used to ensure fast convergence such as ILU factorization [95] [96] and domain decomposition methods [18] [97] [98], which involve various interface conditions [99], either based on a continuous optimization [100] [101] [102] using an approximation of the Steklov-Poincaré operators or an algebraic optimization [103] [104] using an approximation of the Schur complement matrix [105], etc. Gander *et al.* analyzes and evaluates ILU-type preconditioners used with QMR algorithms to solve the Helmholtz equation. Several library software solutions [106] [107] [58] [62] [108] are developed to provide fundamental components of the Krylov subspace methods. In particular when solving extremely large sparse linear systems, new programming paradigms of the Krylov methods should be defined and evaluated with respect to the state-of-the-art scientific methods. Iterative Krylov methods involve linear algebra operations such as dot product, norm, addition of vectors and sparse matrix-vector multiplication, which are computationally expensive for large-size matrices. The optimization of this type of computation is crucial, and the large-size of the data structure involves the use of parallel and appropriate methods. Parallel iterative methods are well suited for their resolutions. Algorithms need to be adapted to these new constraints. One solution is to use a GPU or multi-core system to accelerate them. In this thesis, we will pay a special attention to iterative methods. In the next section, we give an overview of existing linear algebra libraries.

3.3 Overview of existing scientific libraries

In my thesis, we have decided to implement our own research library, Alinea, that will be described in the following in order to better optimize the algorithms according to the hardware characteristics and the properties of the problem to be solved. This section gives the motivations of our choice.

3.3.1 Basic Linear Algebra Subprograms

The *BLAS* (Basic Linear Algebra Subprograms) is a set of basic routines for performing basic operations of numerical linear algebra, *i.e.*, basic vector and matrix operations. The initial version of the subprogram specifications arose in [109], where Hanson, Krogh, and Lawson demonstrated the advantages of adopting a set of basic routines for linear algebra problems. In [110] [111], Lawson, Hanson, Kincaid, and Krogh presented the original basic linear algebra subprograms commonly referred to as the BLAS. This BLAS [111] of Lawson *et al.* is labeled as “BLAS level-1”, which concerns elementary vector operations. The library has been used in a wide range of software in numerical science such as LinPack [57] and many fields of engineering [112] [113] [114] [115], and others. In numerical science, BLAS has become a *de facto* standard for the elementary vector operations.

From 1984 to 1986, motivated by the development of vector-processing machines with extended precision arithmetic registers, for example machines performing IEEE arithmetic [116], an extension to the set of Basic Linear Algebra Subprograms was proposed by Dongarra *et al.*, [117]. The extensions are targeted at matrix-vector operations which should provide for

efficient and portable implementations of algorithms for high-performance computers. Subsequently, a set of Level 3 BLAS for matrix-matrix operations, motivated by high-performance computers, especially those with hierarchical memory and parallel processing capability, was proposed by Dongarra *et al.* [118]. The Numerical Algorithms Group Ltd (NAG) has proposed a C equivalents library [119] of the Fortran BLAS. The linear algebraic operations are frequently used in numerical simulations, which leads us to pay more attention to their implementation. In the massively parallel simulation, their use is multiplied requiring to look for the best way to implement them. Given their massively parallel character, it is especially important to adjust their algorithms upon the environment of computations and the hardware specifications. The BLAS exploits real and complex objects in single and double precision. Each BLAS routine comes in several versions, one for each precision. The first letter of the routine name specifies the precision used: *S*: Real single precision, *D*: Real double precision, *Z*: Complex single precision and *C*: Complex double precision. In the remainder of this thesis, when we talk about general routine without specific type, we will remove the first letter (*S*, *D*, *Z*, *C*) or replace it with “?”.

3.3.2 Toward accelerated BLAS

In our comparative study of scientific libraries targeted for GPU, we concentrate exclusively on libraries implemented with CUDA-provided languages. The primary target of the study is our Alinea library. Alinea is designed to provide efficient linear algebra operations on hybrid multi-CPU and GPU clusters. The framework offers direct and iterative solvers for solving large, both dense and sparse, linear systems. Our primary aim is to ease the development of engineering and scientific applications on both CPU and GPU by hiding most of the intricacies encountered when dealing with these architectures. In Alinea we pay particular attention to the auto-tuning implementation of GPU as said in Section 2.4, Page 14, with accordance to the present and future hardware on which the library is likely to be used. For maximum flexibility, the library provides a whole set of matrix data storage formats.

In the following, we will have to compare our algorithms with the following libraries: *cuBLAS*, *cuSPARSE*, *Cusp*. The *cuBLAS* library [60], which is a CUDA version of the Basic Linear Algebra Subprograms (BLAS) library, gives the basic set of tools to access the computational resources of Nvidia GPUs. As written in the *cuBLAS Library Guide* [60], *the basic model by which applications use the cuBLAS library, is to create matrix and vector objects in GPU memory space, fill them with data, call a sequence of cuBLAS functions, and, finally, upload the results from GPU memory space back to the host. To accomplish this, cuBLAS provides help functions for creating and destroying objects in GPU space, and for writing data to, and retrieving data from these objects.* The *cuSPARSE* [61] library is an implementation of basic linear algebra used for sparse matrix operations, on top of an Nvidia CUDA library. The *Cusp* [62] library provides flexible, high-level interface for manipulating sparse matrices and solving sparse linear systems on GPU. *Cusp* appears as the popular library that offers GPU iterative solvers. *Cusp* library includes various sparse matrix formats such as *COO*, *CSR*, *ELL* and *HYB*.

It is interesting also to mention the *CUDA_ITSOL* [58] library, which is developed under CUDA language by Ruipeng Li [PhD Student supervised by Yousef Saad, Univ. of Minnesota]. It supports several sparse matrix operations and, more importantly, provides a variety of GPU linear system iterative solvers.

3.4 BLAS level-1 operations

In this section, we present “BLAS level-1”, which concerns vector operations.

3.4.1 Addition of vectors (axpy)

Alpha X Plus Y (AXPY), *i.e.*, $y = y + \alpha x$, is an addition of two vectors with a scalar weight, which belongs to the BLAS level-1 package. This BLAS-1 operation is completely parallelized without any overhead of parallelization.

GPU Implementation

We note that CUDA hardware processes the Fused Multiply and Addition (FMA) operation which allows to compute $y = y + \alpha x$ by a dedicated hardware as one operation. The first

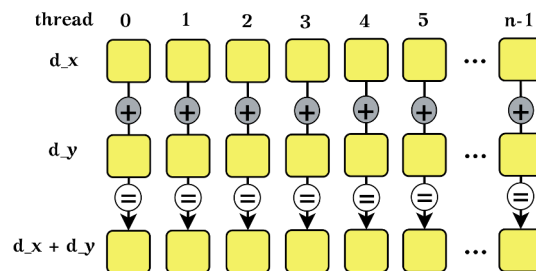


FIGURE 3.9: Summation of two vectors

implementation of BLAS on GPU has been reported in cuBLAS. Apxy is inherently parallel, which makes them excellent candidates for implementation on a GPU. The update, which is done in place, *i.e.*, $y_i = y_i + \alpha x_i$ ($0 \leq i \leq n - 1$), where n is the size of vectors, is done with simple GPU kernel of the form given in Listing 3.1, using a two-dimensional grid. In Listing 3.1, `d_x` and `d_y` are the vector on global memory, `size` is the length of vectors, *i.e.*, the number of elements in `d_x` and `d_y`. The scheme of the vector summation is illustrated in Figure 3.9.

```
1 template <class T, class U>
2 __global__ void Apxy ( T alpha, double* d_x, T* d_y, const U size ) {
3     unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
4     unsigned int y = threadIdx.y + blockIdx.y * blockDim.y;
5     unsigned int pitch = blockDim.x * gridDim.x;
6     unsigned int idx = x + y * pitch;
7     if (idx < size) {
8         d_y[idx] = alpha * d_x[idx] + d_y[idx];
9     }
10 }
```

LISTING 3.1: Summing two vectors, using on two-dimensional grid

Let us note that all of our kernels are template functions on indices (U) and on values (T). The kernel is unique for real and complex arithmetic numbers in single and double precision. The handling of complex arithmetic number templates in CUDA will be presented in the following. The threads of (Figure 3.9, Listing 3.1) are distributed using auto-tuning techniques that take into account the characteristics of the graphics card and the size of vectors.

MPI Implementation

For the sake of simplicity, at this level of the dissertation we do not care about the contribution and the cost of communications between processes.

For the purposes of our advanced algorithms (sub-structuring, domain decomposition, ...) that will be presented in the following, we have adopted the following convention: pre-treatment has been performed in order to distribute our global data locally to each process. The algorithms presented for the parallel MPI version assume that the input data is local to each process. Let us consider a system capable of executing $N \in \mathcal{P}$ processes of execution.

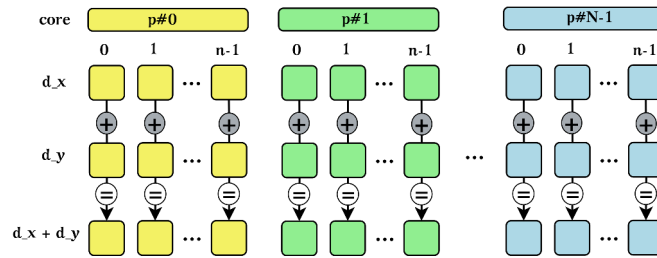


FIGURE 3.10: Summation of two vectors in parallel

Let $\mathcal{P} = \{P_1, \dots, P_N\}$ be the set of the processors.

As said above and described in Figure 3.10 the input data are local to each processor. Let us note that the local size can differ with each processor. The parallel scheme described in Figure 3.10 consists in each processor computing local addition of two vectors, and then a gather of the results of all processors. The global vector is obtained, if necessary, by a gather operation of local results. In case of multi-core/multi-GPUs, all local computations are performed on the GPU associated with the process, using the algorithm presented in Figure 3.9 and Listing 3.1.

3.4.2 Element-Wise product

As the addition of two vectors, the element-wise multiplication or scalar-vector multiplication between two vectors computes $c_i = x_i y_i$ ($1 \leq i \leq n$), which we call *X Multiply Y (XMY)* in this dissertation. This operation is also completely parallelized as *AXPY*. However, it needs to access three arrays for the operation, which results in a more memory-intensive operation than *AXPY*.

GPU Implementation

The scheme of this operation is close to that of *AXPY* as described in Figure 3.11, except that it performs multiplication, element by element, and the results are stored in a new third array. Listing 3.2 presents the CUDA kernel of *XMY* operation.

```

1 template <class T, class U>
2 __global__ void Axy ( double* d_x, T* d_y, T* d_c, const U size ) {
3   unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
4   unsigned int y = threadIdx.y + blockIdx.y * blockDim.y;
5   unsigned int pitch = blockDim.x * gridDim.x;
6   unsigned int idx = x + y * pitch;
7   if (idx < size) {
8     d_c[idx] = d_x[idx] * d_y[idx];

```

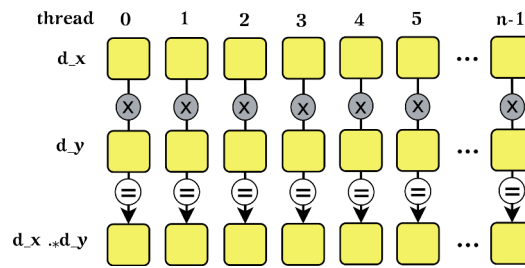


FIGURE 3.11: Multiplying two vectors

```
9 }
10 }
```

LISTING 3.2: Multiplying two vectors, using on two-dimensional grid

As in Listing 3.1, we check whether `idx` is less than the size of vectors in Listing 3.2. In fact the `idx` should always be less than the size, since we have specifically launched our kernel so that this assumption holds.

MPI Implementation

The parallel version is similar to AXPY as described in Figure 3.12. Each process computes its local element-wise product. In heterogeneous multi-core-GPU systems, all local

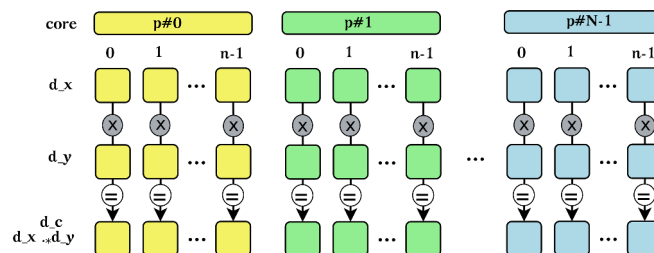


FIGURE 3.12: Multiplication of two vectors in parallel

computations are carried out on the GPU associated with the process, using the algorithm presented in Figure 3.11 and Listing 3.2.

3.4.3 Scaling vectors

The “Scale” operation (SCAL) is equivalent to the level-one (vector) operation in the Basic Linear Algebra Subprograms (BLAS) package, which scales all elements of a vector by a scalar *alpha*. This operation is done “in-place”, *i.e.*, the result will overwrite elements of the input vector. The Listing 3.3 presents the CUDA kernel of the SCAL operation.

```
1 template <class T, class U>
2 __global__ void SCAL ( T alpha, double* d_x, const U size ) {
3     unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
4     unsigned int y = threadIdx.y + blockIdx.y * blockDim.y;
5     unsigned int pitch = blockDim.x * gridDim.x;
6     unsigned int idx = x + y * pitch;
7     if (idx < size) {
8         d_x[idx] = alpha * d_x[idx];
9     }
10 }
```

LISTING 3.3: Scaling vector, using on two-dimensional grid

This operation in heterogeneous multi-core-GPU is similar to AXPY and XMY operations.

3.4.4 Dot product and norm

The computation of dot products and euclidean norms are prevailing linear algebra operations, which could be very expensive both in terms of computing time and memory storage for large-size vectors. *DOT*, computes a dot product of two vectors and belongs to BLAS level-1, same as AXPY. However, the implementation of *DOT* requires some effort to achieve good performance in a parallel environment. The operation $\sum_i x_i y_i$ is distributed over several processors by decomposing indices into a union of blocks, and finally it is necessary to sum up scalar values from all processors. This operation implies a kind of synchronization between processors. Unfortunately, there is no dedicated hardware for synchronization between CUDA ALUs. The host CPU needs to participate to get the final value of the inner product.

GPU Implementation

The dot product operation is characterized by a simple loop with simultaneous summation, which is computationally expensive in a sequential implementation on CPU. The optimized dot product operation given here is computed in two steps. The first task consists in performing an element-wise operation, and the second task consists in calculating the sum of each of these products to get the final result. This second step is achieved in a hierarchical way by summing product components in neighboring threads as indicated by the gridification and illustrated in Figure 3.13. At the final level of the reduction, the partial sum of all elements

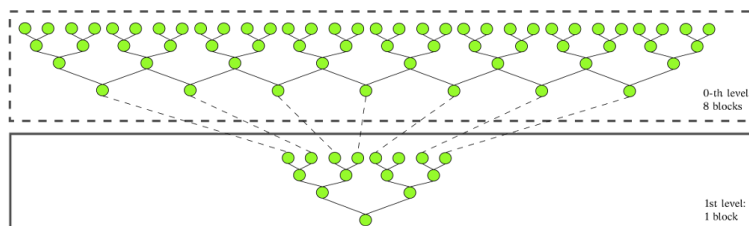


FIGURE 3.13: Dot product scheme for tree cutting passes sums

of a block' threads is stored in the first thread (thread [0]) of the current block. Finally, the dot product results in a sum of all partial sums of all blocks. The calculation of the euclidean vector norm is done in a similar way. The number of threads harnessed by the shared memory while performing dot product operations depends on the fixed number of threads per block. Obviously, we need to use a given number of threads that will provide good performance when treated in shared memory, and which also ensure optimal performance for the reduction of the global sum between all blocks; this depends on the number of threads per block. The threads in a warp spread multiple rows and each thread handles multiple non-zero values of a row.

MPI Implementation

The computation of dot products in parallel MPI is a relatively simple operation. Each processor then computes the dot product on the piece that it has, *i.e.*, the local dot product. Finally, the pieces of the dot product must be summed together to obtain the full dot product on the required process, using a reduction operation. This reduction is performed with the `MPI_REDUCE` operation, using predefined constant `MPI_SUM`. This constant means that the desired reduction operation is to sum the values and return the sum to the process collecting the

results. When the dot product is required by all processes, an `MPI_ALLREDUCE` operation is used instead of `MPI_REDUCE`. This operation ensure that all processes will collect the result of the dot product. For the GPU version, local operations have been performed on graphics cards, and MPI handles the sum reduction.

3.4.5 Numerical results

In this section, we report the collected numerical results of the experiments we have performed to evaluate the speed of our linear algebra operations both on CPU and GPU.

In the experiments the default gridification ($nBlocks, nThreadsPerBlock$) of “Scale”, “Addition of Vectors”, “Element wise product” is set to the following values

$$nBlocks = \frac{size + numb_thread_block - 1}{numb_th_block}$$

$$numb_th_block = 256$$

where $size$ and $numb_thread_block$ represent respectively the size of the vector and the thread block size. For dot products, we consider the following default gridification

$$nBlocks = \frac{size + numb_thread_block - 1}{numb_th_block}$$

$$numb_th_block = 128$$

For a particular configuration, we will take care to specify the gridification explicitly. The cost of transferring data from CPU/GPU to GPU/CPU is not negligible. In all the algorithms presented here, the step before launching the kernel consists in sending data from CPU to GPU and then copy back the result from GPU to CPU. In all our programs, we pay attention to minimizing data transfers, *i.e.*, only necessary transfers are performed. The first set of numerical results consists in analyzing OpenCL and CUDA when both algorithms are carried out on *Platform-2*, which consists of *NVIDIA* cards. We used in this first set a one-dimensional grid. In the following benchmarks, the execution time corresponds to the exclusive time of the kernel, *i.e.*, our measurements do not include time spent transferring data between the host and the device memory. This benchmark setting will be considered as *default running*. As a first step, we take a look at the impact of data construction and copy on host and device. Figure 3.14(a) and Figure 3.14(b) provide comparative operations of elementary transfer on CPU, GPU with CUDA and GPU with OpenCL. As we can see in these figures, allocation and transfer times are quasi-linear operations in size with different penalization factors except for CUDA double, where fluctuations appear. Figure 3.15(a) presents the execution time in microseconds of the addition operation performed on CPU, GPU/CUDA and GPU/OpenCL. These figures clearly show the effectiveness of GPU compared to CPU for this inherent parallel operation for large-size data. On the other hand, single precision is faster than double precision. OpenCL is more efficient than CUDA for these operations. Figure 3.15(b) compares the scalar product execution time in microseconds of CPU, CUDA and OpenCL. It appears from Figure 3.15(b) that the dot product on CPU is exactly linear in the size of the vector, which is consistent with the traditional method of calculation ($\sum_k a_k * b_k$). On the GPU, we take advantage of its parallel architecture.

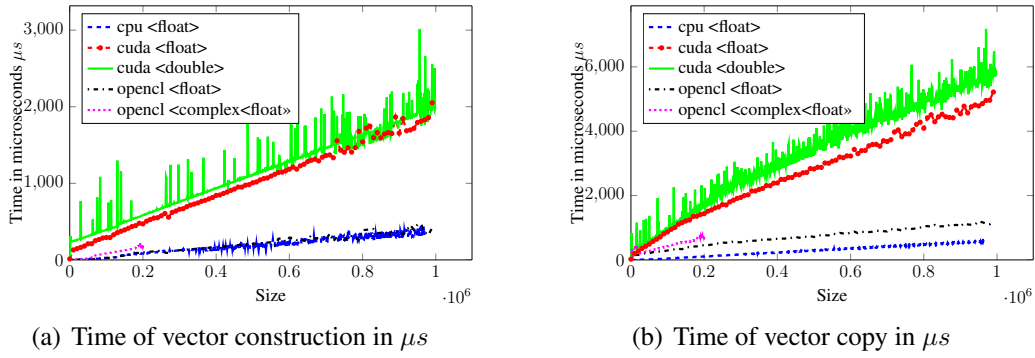


FIGURE 3.14: Execution times of vector construction and copy vector in μs on *NVIDIA* card

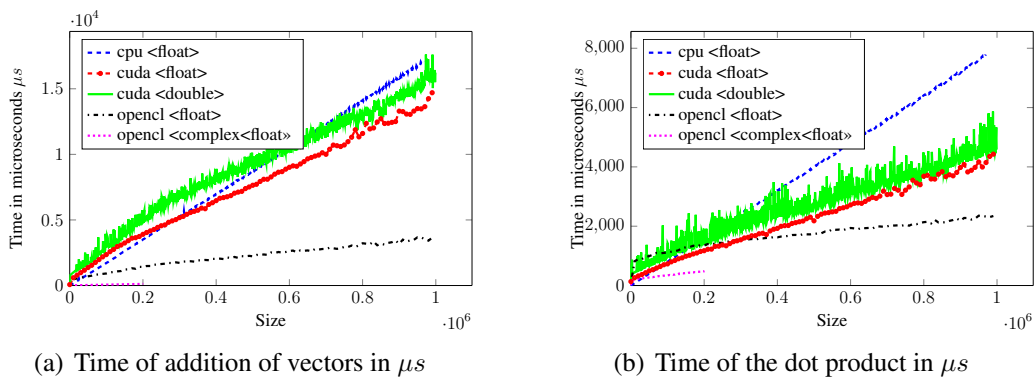


FIGURE 3.15: Running times of the addition of vectors and the dot product in μs on *NVIDIA* card

The second set of numerical experiments consists in evaluating OpenCL and CUDA when algorithms are performed on different types of graphics cards from *Platform-2* and *Platform-3*. We used in this second set a one-dimensional grid. The second set of experiments confirms the

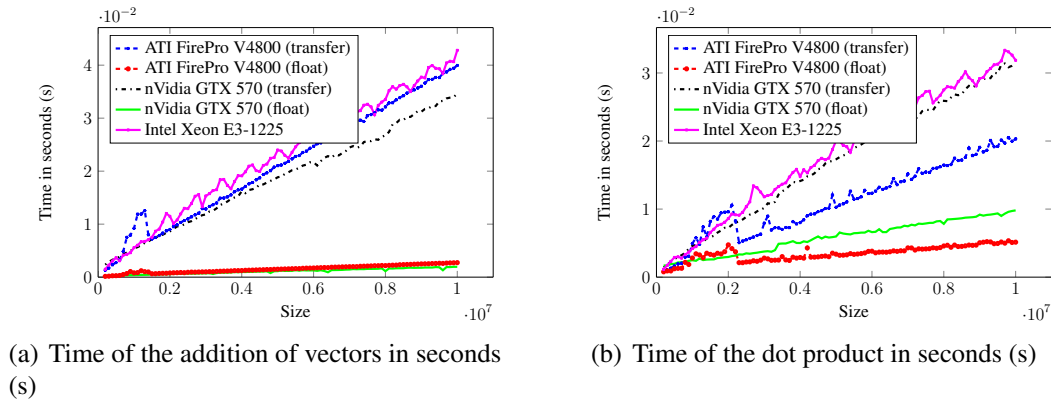


FIGURE 3.16: Execution times of the addition of vectors and the dot product on *NVIDIA* and *ATI*

effectiveness of GPU to compute linear algebra operations according to the execution time for the addition of vectors (OpenCL) and dot products (OpenCL). The results are shown in Figure 3.16(a) and Figure 3.16(b). OpenCL gave better results for the *AMD/ATI* card compared to *NVIDIA*; this can be explained by the fact that *NVIDIA* is more optimized for CUDA. In this thesis, we have chosen to concentrate our study on CUDA with double precision. So, after a brief comparison between CUDA and OpenCL, we now focus on our CUDA algorithms. The

experiments reported in Table 3.1 have been performed on vectors of different sizes changing from 10^3 to 10^7 , on *Platform-I*.

Size	DAXPY			Dot product			Norm		
	Alinea	cuBLAS	Cusp	Alinea	cuBLAS	Cusp	Alinea	cuBLAS	Cusp
10^3	0.045	0.030	0.025	0.045	0.060	0.202	0.053	0.151	0.200
10^5	0.070	0.033	0.051	0.082	0.077	0.234	0.114	0.059	0.317
10^7	2.490	0.037	2.607	1.693	0.082	1.958	5.301	0.077	1.146

TABLE 3.1: Double precision kernel execution time in milliseconds

Table 3.1 shows the computation time in milliseconds for the DAXPY, dot product and norm operations, by considering 256 threads per block for the Alinea DAXPY experiment and, 128 threads per block for the Alinea dot products and norms. Note that the times given in Table 3.1 are strictly the times of the kernel execution, which uses a one-dimensional grid. The Alinea dot product outperforms the Cusp dot product as we can see in Table 3.1. Indeed cuBLAS gives better results than Cusp and Alinea for basic linear operations (BLAS 1). Now, we consider a two-dimensional grid for the rest of experiments of this subsection. The speed-ups presented in these results are computed as follows: $\text{ratio} = \text{CPU_time} / \text{GPU_time}$.

h	CPU time (ms)	GPU time (ms)	CPU Gflops	GPU Gflops	CPU/GPU time ratio
101,168	0.414	0.059	0.243	1.7	6
296,208	1.176	0.083	0.251	3.5	14
544,563	2.222	0.108	0.245	5.0	20
650,848	2.777	0.121	0.234	5.3	22
848,256	3.571	0.149	0.237	5.6	23
1,213,488	5.555	0.194	0.218	6.2	28
1,325,848	5.555	0.208	0.238	6.3	26

TABLE 3.2: Alinea double precision Scale (DSCAL) operation

Table 3.2 collects the double precision execution time of our implementation for the *Scale* operation. The first column h gives the size of the vector. The second and third columns contain the execution time in milliseconds for CPU and GPU algorithms, respectively. The fourth and fifth columns give the floating point operations per second for CPU and GPU algorithms. The last column shows the ratio of the CPU to the GPU time. Table 3.2 clearly shows the better results of GPU compared to CPU. As this operation is inherently parallel, we have observed higher speed-up for larger vectors. The numerical results for our implementation of the *DAXPY* operation in double precision are reported in Table 3.3.

h	CPU time (ms)	GPU time (ms)	CPU Gflops	GPU Gflops	CPU/GPU time ratio
101,168	0.833	0.067	0.364	4.5	12
296,208	2.439	0.102	0.364	8.6	23
544,563	4.761	0.155	0.343	10.5	30
650,848	5.555	0.175	0.351	11.1	31
848,256	7.692	0.188	0.330	13.4	40
1,213,488	10.000	0.283	0.364	12.8	35
1,325,848	11.111	0.291	0.357	13.6	38

TABLE 3.3: Alinea double precision addition of vectors (DAXPY)

Table 3.3 clearly confirms the efficiency of the GPU implementation compared to the CPU. As we can see by comparing Table 3.2 and Table 3.3, the speed-up for DAXPY is better than

for DSCAL. Table 3.4 presents the double precision execution time of our implementation for the *element-wise product* operation.

h	CPU time (ms)	GPU time (ms)	CPU Gflops	GPU Gflops	CPU/GPU time ratio
101,168	0.781	0.072	0.129	1.4	10
296,208	2.439	0.116	0.121	2.5	20
544,563	4.166	0.146	0.130	3.7	28
650,848	5.000	0.166	0.130	3.8	29
848,256	6.666	0.209	0.127	4.0	31
1,213,488	9.090	0.294	0.133	4.1	30
1,325,848	10.000	0.289	0.132	4.5	34

TABLE 3.4: Alinea double precision element wise product (DXMY)

The GPU remains clearly better than CPU for element-wise product operations. Table 3.5 gives the comparison of our implementation on both CPU and GPU for the *dot product* double precision execution time.

h	CPU time (ms)	GPU time (ms)	CPU Gflops	GPU Gflops	CPU/GPU time ratio
101,168	0.564	0.103	0.358	1.9	5
296,208	1.612	0.131	0.367	4.5	12
544,563	3.125	0.192	0.348	5.6	16
650,848	3.571	0.196	0.364	6.6	18
848,256	5.263	0.204	0.322	8.3	25
1,213,488	7.142	0.312	0.339	7.7	22
1,325,848	7.692	0.336	0.344	7.8	22

TABLE 3.5: Alinea double precision dot product (DDOT)

As illustrated in Table 3.5, GPU implementation outperforms the CPU with a speed-up 27 times higher, for the largest vector case.

As a conclusion, the simulations performed demonstrate that with 128 threads per block we have a good rate of performance and a compromise between the use of shared memory and the reduction of the global sums. As proved above, GPU clearly gives effective results compared to CPU for real number arithmetics. But performance for complex number arithmetics with double precision remains a defiance, and dynamic auto-tuning of the GPU grid should be considered. The analysis of performance for complex number arithmetics with double precision has been performed on *Platform-4*, which consists of an Intel Core i7 920 2.67Ghz with four physical cores and four logical cores, 12GB RAM, and two NVIDIA graphics cards: a Tesla K20c with 4799GB memory and GeForce GTX 570 with 1279MB memory. In the following Table 3.6, Table 3.7, Table 3.8 and Table 3.9, the cards Tesla K20c and GTX 570 will be denoted respectively *dev#0* and *dev#1*. In Table 3.6, we collect the execution times of the *scale operation* for complex number arithmetics with double precision.

h	cpu time (ms)	cpu Gflops	dev#0 time (ms)	dev#0 Gflops	dev#1 time (ms)	dev#1 Gflops	ratio#0 cpu/#0	ratio#1 cpu/#1
648,849	5.56	0.70	0.20	19.47	0.21	18.34	27.78	26.17
2,000,000	15.71	0.76	0.46	26.04	0.53	22.56	34.10	29.54
9,000,000	80.00	0.68	1.92	28.08	2.33	23.22	41.60	34.40
14,000,000	120.00	0.70	2.94	28.56	3.57	23.52	40.80	33.60

TABLE 3.6: Alinea complex double precision scale (ZSCAL) operation

In Table 3.7, we present the complex number arithmetics with double precision execution times in milliseconds (ms) of the *ZAXPY* operation.

h	cpu <i>time (ms)</i>	cpu <i>Gflops</i>	dev#0 <i>time (ms)</i>	dev#0 <i>Gflops</i>	dev#1 <i>time (ms)</i>	dev#1 <i>Gflops</i>	ratio#0 <i>cpu/#0</i>	ratio#1 <i>cpu/#1</i>
648,849	5.56	0.93	0.26	20.04	0.27	19.52	21.44	20.89
2,000,000	16.67	0.96	0.69	23.20	0.81	19.68	24.17	20.50
9,000,000	75.00	0.96	3.03	23.76	3.33	21.60	24.75	22.50
14,000,000	120.00	0.93	4.76	23.52	5.26	21.28	25.20	22.80

TABLE 3.7: Alinea complex double precision addition of vectors (*ZAXPY*)

Table 3.8 exhibits the double precision execution times of the *element by element product* operation.

h	cpu <i>time (ms)</i>	cpu <i>Gflops</i>	dev#0 <i>time (ms)</i>	dev#0 <i>Gflops</i>	dev#1 <i>time (ms)</i>	dev#1 <i>Gflops</i>	ratio#0 <i>cpu/#0</i>	ratio#1 <i>cpu/#1</i>
648,849	8.33	0.47	0.28	13.66	0.29	13.55	29.25	29.00
2,000,000	25.00	0.48	0.72	16.56	0.85	14.16	34.50	29.50
9,000,000	120.00	0.45	3.03	17.82	3.33	16.20	39.60	36.00
14,000,000	180.00	0.47	4.76	17.64	5.00	16.80	37.80	36.00

TABLE 3.8: Alinea complex double precision element wise product (*ZXMY*)

The *dot product* execution times for double precision with complex number arithmetics on both CPU and GPU are exposed in Table 3.9.

h	cpu <i>time (ms)</i>	cpu <i>Gflops</i>	dev#0 <i>time (ms)</i>	dev#0 <i>Gflops</i>	dev#1 <i>time (ms)</i>	dev#1 <i>Gflops</i>	ratio#0 <i>cpu/#0</i>	ratio#1 <i>cpu/#1</i>
648,849	5.56	0.93	0.33	15.83	0.33	15.94	16.94	17.06
2,000,000	16.67	0.96	0.83	19.20	0.76	20.96	20.00	21.83
9,000,000	80.00	0.90	3.23	22.32	3.23	22.32	24.80	24.80
14,000,000	130.00	0.86	4.76	23.52	4.55	24.64	27.30	28.60

TABLE 3.9: Alinea complex double precision dot product (*ZDOT*)

For the dot product, 128 threads per block give good rate of performance and a compromise between the use of shared memory and the reduction of the global sums. Table 3.6, Table 3.7, Table 3.8 and Table 3.9 have demonstrated the efficiency of GPU in terms of computing time for double precision with complex number arithmetics.

We now need to address question, energy related, *i.e.*, how much energy is consumed by the dot product? We compare the Alinea and Cusp libraries in terms of energy computation. We used *Platform-1*. Figure 3.17(a) and Figure 3.17(b) respectively show a graphic presentation of

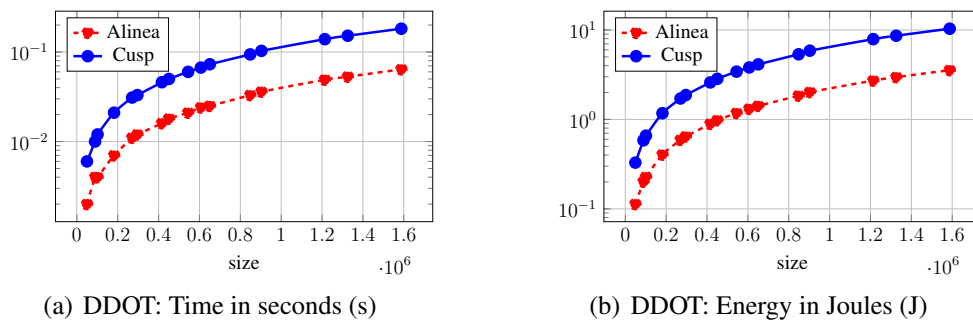


FIGURE 3.17: DDOT: Time in seconds (s) and Energy in Joules (J)

the execution time in seconds and energy consumption in Joules. Here the kernel is performed by both Alinea and Cusp. The energy consumption in Joules is calculated by the multiplication of the operations' energy and time plus the fundamental energy consumption by the GPU. We can see that Alinea is faster than Cusp and consumes less energy than Cusp. Since the complexity of DDOT is linear, the linear prediction of the GPU execution time and energy consumption is appropriate and is given in Table 3.10. We can see Alinea's better energy consumption performance than Cusp, by verifying the coefficient of linear dependency. The prediction formula for Alinea is 1/3 of Cusp.

	equation	coef. of determination
Alinea Time (s)	$-2.216 \times 10^{-4} + 4.002 \times 10^{-8}x$	0.9996
Cusp Time (s)	$-6.325 \times 10^{-4} + 1.142 \times 10^{-7}x$	0.9996
Alinea Energy (J)	$-1.897 \times 10^{-2} + 2.241 \times 10^{-6}x$	0.9996
Cusp Energy (J)	$-5.477 \times 10^{-2} + 6.514 \times 10^{-6}x$	0.9996

where x is the size of the vectors.

TABLE 3.10: Linear prediction of the dot product

3.5 Sparse matrix-vector product

The sparse matrix-vector product (SpMV) is the basic operation not only for Krylov algorithms. It can be found at the core of many scientific numerical programs. In [45] [120] [121], we proved that the sparse matrix-vector multiplication is the crucial operation to reach a good performance. Many numerical methods, among them finite element methods, involve extremely large-size sparse matrices. Crucial to the effective implementation of algebraic operations for such matrices is the matrix storage format. Sparse data structures such as Compressed-Sparse Row (CSR) [69] are also essential with respect to storing the matrices on GPU memory effectively. We have showed in [122] that the computation time of GPU kernels strongly depends on both gridification, *i.e.*, threads distribution upon the grid, and the performance of the machine (CPU and GPU). This result is also corroborated in references [44] [43] [123]. Let us consider the following matrix-vector product

$$y = Ax, \quad (3.25)$$

where $A \in \mathbb{C}^{n \times n}$, left operand of the SpMV, is a sparse matrix, $x \in \mathbb{C}^n$, right operand of the SpMV, is a vector, and $y \in \mathbb{C}^n$ is the vector that stores the result of the SpMV.

3.5.1 Basic algorithms

Algorithm 3.2 describes the matrix-vector multiplication algorithm for the COO storage format.

Algorithm 3.2: Matrix-vector multiplication with the COO storage format

input : n (size of the matrix), nnz (number of non-zero values),
 r_numb (row numbers),
 c_numb (column numbers),
 $coef$ (non-zero values),
 x (vector)
output : y (vector)
variable: i, j, tid

```
1 // - initialization of the vector result
2 for  $i = 0$  to  $n - 1$  do  $y[i] \leftarrow 0$ 
3 // - compute the COO SpMV
4 for  $tid = 0$  to  $nnz - 1$  do
5    $i \leftarrow r\_numb[tid]$ 
6    $j \leftarrow c\_numb[tid]$ 
7    $y[i] \leftarrow y[i] + coef[tid] \times x[j]$ 
8 end
```

Algorithm 3.3: Matrix-vector multiplication with the ELL storage format

input : n (size of the matrix), m (maximum number of non-zero values per row),
 c_numb (column numbers),
 $coef$ (non-zero values),
 x (vector)
output : y (vector)
variable: i, j, k, tid

```
1 // - initialization of the vector result
2 for  $i = 0$  to  $n - 1$  do  $y[i] \leftarrow 0$ 
3 // - compute the ELL SpMV
4 for  $i = 0$  to  $n - 1$  do
5   for  $k = 0$  to  $m - 1$  do
6      $tid \leftarrow k + m \times i$ 
7      $y[i] \leftarrow y[i] + coef[tid] \times x[j]$ 
8   end
9 end
```

Algorithm 3.4 presents the matrix-vector multiplication algorithm for the CSR storage format. As we can see in Algorithm 3.4, each atomic update ($y[i] \leftarrow y[i] + coef[tid] \times x[j]$) of the CSR SpMV requires to get the column index and the corresponding value of vector x via an indirect memory access.

Algorithm 3.4: Matrix-vector multiplication with the CSR storage format

input : n (size of the matrix), nnz (number of non-zero values),
 $rows_indices$ (position of the first entry of each row),
 c_numb (matrix column numbers),
 $coef$ (matrix non-zero values),
 x (vector)
output : y (vector)
variable: i, j, tid

```
1 // - initialization of the vector result
2 for  $i = 0$  to  $n - 1$  do  $y[i] \leftarrow 0$ 
3 // - compute the CSRSpMV
4 for  $i = 0$  to  $n - 1$  do
5   for  $tid = rows\_indices[i]$  to
6      $rows\_indices[i + 1] - 1$  do
7      $j \leftarrow c\_numb[tid]$ 
8      $y[i] \leftarrow y[i] + coef[tid] \times x[j]$ 
9   end
```

Algorithm 3.5: Matrix-vector multiplication with the CSC storage format

input : n (size of the matrix), nnz (number of non-zero values),
 r_numb (matrix row numbers),
 $columns_indices$ (position of first entry of each column),
 $coef$ (matrix non-zero values),
 x (vector)
output : y (vector)
variable: i, j, tid

```
1 // - initialization of the vector result
2 for  $j = 0$  to  $n - 1$  do  $y[j] \leftarrow 0$ 
3 // - compute the CSC SpMV
4 for  $j = 0$  to  $n - 1$  do
5   for  $tid = columns\_indices[j]$  to
6      $columns\_indices[j + 1] - 1$  do
7      $i \leftarrow r\_numb[tid]$ 
8      $y[i] \leftarrow y[i] + coef[tid] \times x[j]$ 
9   end
```

The transpose of the CSR matrix corresponds to a format called CSC. The algorithm of the CSC matrix-vector product is given in Algorithm 3.5. As in CSR SpMV, each atomic operation ($y[i] \leftarrow y[i] + coef[tid] \times x[j]$) of the CSC SpMV (Algorithm 3.5) involves getting the row index and then updating the vector result y at the corresponding place via an indirect memory access.

Algorithm 3.3 gives the matrix-vector multiplication algorithm for the ELL storage format.

The matrix-vector multiplication algorithm for the HYB storage format is described in Algorithm 3.6.

Algorithm 3.6: Matrix-vector multiplication with the HYB storage format

```

input  :  $ell\_n$  (size of the ELL matrix),
           $ell\_m$  (maximum number of non-zero values per row of ELL matrix),
           $m\_ell\_c\_numb$  (ELL matrix of column numbers),
           $m\_ell\_coef$  (ELL matrix of non-zero values)
input  :  $coo\_r\_numb$  (COO matrix row numbers),
           $coo\_c\_numb$  (COO matrix column numbers),
           $coo\_coef$  (COO matrix non-zero values),
           $coo\_nnz$  (COO number of non-zero values)
input  :  $x$  (vector)
output :  $y$  (vector)
variable:  $i, j, k, tid$ 

1 // - initialization of the vector that stores the result
2 for  $i = 0$  to  $n - 1$  do  $y[i] \leftarrow 0$ 
3 // - compute the ELL matrix-vector multiplication
4 for  $i = 0$  to  $ell\_n - 1$  do
5   | for  $k = 0$  to  $ell\_m - 1$  do
6   |   |  $tid \leftarrow k + n \times i$   $j \leftarrow m\_ell\_c\_numb[tid]$ 
7   |   |  $y[i] \leftarrow y[i] + m\_ell\_coef[tid] \times x[j]$ 
8   |   end
9 end
10 // - compute the COO matrix-vector multiplication
11 for  $tid = 0$  to  $coo\_nnz - 1$  do
12 |    $i \leftarrow coo\_r\_numb[tid]$ 
13 |    $j \leftarrow coo\_c\_numb[tid]$ 
14 |    $y[i] \leftarrow y[i] + coo\_coef[tid] \times x[j]$ 
15 end

```

3.5.2 GPU Implementation

COO format

The idea behind the SpMV COO algorithm consists in performing n -element-wise products between each row of the sparse matrix and the right operand x of the SpMV, followed by a summation. This scheme corresponds to n -dot products. In practice, each thread computes the product of a non-zero value of the matrix and the corresponding value in vector x of (3.25). Then, the final result y_i is obtained from a dot product of a row of the matrix by the vector x . The scheme of the dot product performed in the SpMV is similar to that presented in Section 3.4.4, Page 57, which uses a reduction step to accumulate the partial products. As explained in Section 3.4.4, Page 57 for the dot product, the performance of SpMV on a matrix in COO format depends on the efficiency of the segmented sum.

CSR format

The parallel algorithm we designed for computing the sparse matrix-vector multiplication for CSR format is inspired by the algorithm described in references [31] and [32]. The very basic, intuitive approach called the CSR scalar version, consists in simply assigning a thread of execution to each matrix row. This results in a straightforward CUDA implementation.

However the consequence of such design is that the access to storage arrays of the matrix is not contiguous [31], which is not efficient. In fact, the manner in which threads within a warp access the coefficients has serious drawbacks.

The algorithm we implemented in this thesis is close to the CSR vector version. Threads are grouped into warps of size 32 (or 16, *i.e.*, a half-warp, or 8, *i.e.*, half a half-warp). We assume a warp uses 32 threads, so on each cycle, every thread in the warp executes the same instruction (*SIMD*). A row of the matrix is thus worked on by a warp performing 32 (number of threads per warp) computations per iteration. The main step of the computation is to sum, the values computed by the threads of the warp in an array of shared memory that is faster than global memory. This allows us to avoid the main constraint of the CSR scalar approach, due to the recovered contiguous memory access into storage arrays and column indices. When a thread requires access to the shared memory outside the range of cached data, global memory access is needed for these data. Despite having optimized the access to matrix elements, the sparse matrix-vector computation remains handicapped, because of the non-contiguous access to the values of the dense vector. The access pattern to the elements of this vector is closely related to the distribution of the non-zero values.

ELLPACK format

The idea behind the SpMV ELL algorithm we implemented consists in assigning each thread the treatment of a row of the matrix. The matrix is stored in an array arranged in an increasing order to ensure continuous and linear access to memory. If the number of non-zero values per row of a matrix alters a lot, *i.e.*, if the variance of the density of non-zero values per row is high, this data storage format turns out to be very suitable. Our investigation has shown that this leads to fewer problems when the matrices are well structured, but in cases where the distribution is non balanced, the algorithm loses its effectiveness.

HYBrid format

The format is hybrid, the kernel is also implemented in hybrid mode The HYB SpMV kernel consists of a ELL SpMV followed by a COO SpMV as described in Algorithm 3.6. Obviously, the COO SpMV kernel is performed only if the matrix has a COO part (see Figure 3.6, Page 44). The performance of the HYB kernel strongly depends on performance of the ELL and COO SpMV. Therefore, we have paid attention to optimize both kernels, using auto-tuning technique of the grid. The hybrid format depends on both the portion of ELL and COO formats. The HYB SpMV takes benefits both of the efficiency of ELL SpMV and the regularity of the COO SpMV.

SpMV gridification

The required number of blocks in SpMV for CSR, ELL and HYB kernels of execution is calculated as follows:

$$\frac{(numb_rows \times numb_thread_warp) + numb_thread_block - 1}{numb_thread_block}$$

where *numb_rows*, *numb_thread_warp* and *numb_thread_block* represent respectively the number of rows of the matrix, the number of threads inside the warp and the thread block size. The number of threads per warp and the number of threads per block are given upon a dynamic tuning of the gridification according to the properties of the graphics card and size of

the problem. The SpMV implementation is based on tuning the performance of warps in order to speed the computation. This dynamic control strategy consists in calculating the required number of blocks used in a grid for computing the SpMV, with a given number of threads per block, a number of threads to be used in a warp and the size of the matrix. A row of the matrix is associated with a warp. The proposed optimization for accelerating the calculation is based on the full utilization of the *performance of warps*. Therefore, the number of warps used depends on the number of threads in a block. Moreover, depending on the structure and size of the matrix, a row can be handled by several warps, which can be located in different blocks.

The idea of our approach also consists in looking for the optimal number of threads to use together within a warp according to the pattern of the matrix, *i.e.*, the distribution and the density of non-zero values per row. According to the non-zero values of the $row \times vector$ multiplication, we aim to find the optimal warp size to use. Note that a $row \times vector$ product can be handled by several warps. We exploit the power of warp and manage a kind of balance on the density of the distribution of the number of non-zero elements per row. Thus, we focus on looking for a grid layout that provides contiguous memory access, because the performance is strongly impacted by the memory accesses patterns as discussed in Section 2.4, Page 14.

3.5.3 Numerical results

In this section, we summarize the results of experiments performed to evaluate our SpMV algorithms on both CPU and GPU. The set of data presented in this part will be used in the remainder of this dissertation, especially in the next analysis of the iterative Krylov methods.

SpMV comparisons: sparse formats

The first set of numerical results analyzes and evaluates our SpMV algorithms on both CPU and GPU. Our algorithms are also compared to *cuSPARSE* and *Cusp*. The tests have been performed on large sparse matrices coming from the University of Florida collection [124] with different size and properties of the structure. Table 3.11 collects the Florida set of matrices and gives the main properties of each matrix: h the size of the matrix, nnz the number of non-zero values, $density$ the density that corresponds to the number of non-zero values divided by the total number of matrix coefficients, nnz/h the mean row density, max_row the maximal row density, $\sigma(nnz/n)$ or $nnz/h\ stddev$ the standard deviation of the mean row density (nnz/h) and $bandwidth$ the upper bandwidth, which is equal to the lower bandwidth in the case of a symmetric matrix. The first and second pictures represent the pattern of non-zero values and an histogram of the distribution of non-zero values per row respectively for each matrix. The matrices are arranged in increasing order of non-zero values (nnz), from top to bottom, left to right.

The first set of experiments has been performed on *Platform-1*, which consists of on workstation equipped with an Intel Core i7 920 2.67GHz processor, which has 8 cores composed of 4 physical cores and 4 logical cores, 5.8 GB RAM memory and two *NVIDIA GTX275* GPUs fitted with 895MB memory. This configuration is adequate for carrying out the SpMV operations for the selection of matrices used in the tests. Unless otherwise stated, we define the *default gridification* with 256 threads per block and 8 used threads per warp. The sparse matrix-vector multiplication total running time (in milliseconds) for different implementations, that is Alinea (our implementation), *cuSPARSE* and the *Cusp* library for CSR storage, is detailed in Table 3.12 in columns two to five.

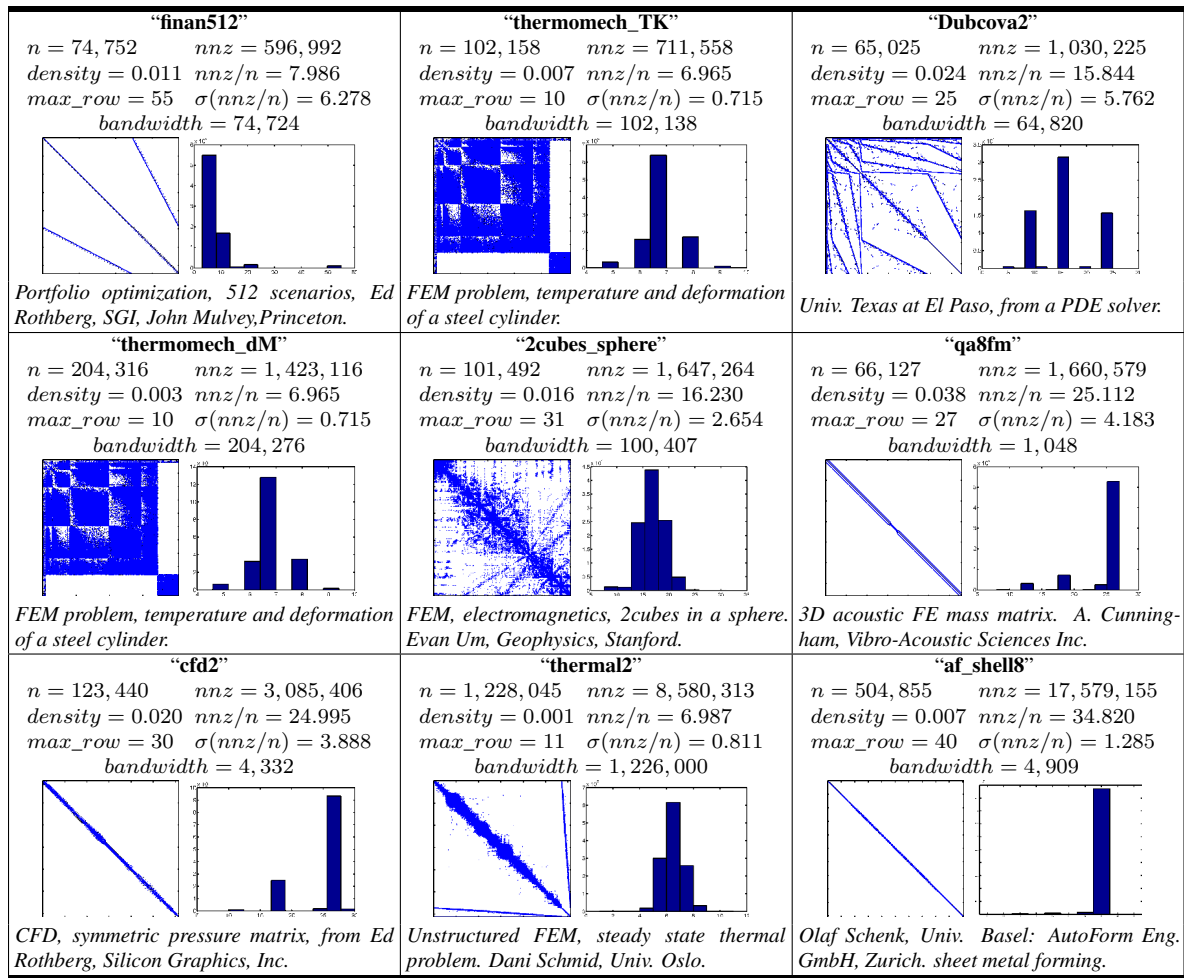


TABLE 3.11: Sketches of engineering matrices from University of Florida collection

Matrix	Alinea CPU	Alinea GPU	cuSPARSE GPU	Cusp GPU	Alinea GPU	Alinea GPU
	CSR	CSR	CSR	CSR	ELL	HYB
2cubes_sphere	14.717	0.943	0.988	0.939	0.904	1.044
cfd2	24.303	1.187	2.433	1.465	0.958	1.189
thermomech_dM	15.906	0.757	1.009	0.813	0.685	0.866
thermomech_TK	7.827	0.465	0.509	0.484	0.445	0.668
qa8fm	12.852	0.633	1.319	0.788	0.529	0.395
Dubcova2	8.326	0.488	0.670	0.498	0.477	0.663
af_shell8	131.043	5.798	9.302	7.103	4.512	3.838
finan512	5.555	0.360	0.452	0.362	0.485	0.517

TABLE 3.12: Double precision running time of sparse matrix-vector product in milliseconds (ms)

Results for the CSR format in Table 3.12 clearly show the good performance of the GPU compared to the CPU. We can also see in this table that our Alinea implementation outperforms cuSPARSE and Cusp libraries for double precision computations. Now we focus on the question of how to improve our library by taking into account the features of the GPU and the specification of the problem. We propose to vary the parameters used in the gridification, *i.e.*, the number of threads inside the warp and the thread block size, for the SpMV with the CSR format. The results with the best gridification are reported in Table 3.13.

A comparison between the computational times of the Alinea library with the default gridification and the Alinea library with auto-tuning of the gridification are given in Table 3.13.

Matrix	Alinea	Grid Alinea	Alinea T.	Grid Alinea T.
		$\langle ntb, tw \rangle$		$\langle ntb, tw \rangle$
2cubes_sphere	0.943	$\langle 256, 8 \rangle$	0.928	$\langle 64, 8 \rangle$
cfid2	1.187	$\langle 256, 8 \rangle$	1.141	$\langle 64, 8 \rangle$
thermomech_dM	0.757	$\langle 256, 8 \rangle$	0.751	$\langle 128, 8 \rangle$
thermomech_TK	0.465	$\langle 256, 8 \rangle$	0.460	$\langle 64, 8 \rangle$
qa8fm	0.633	$\langle 256, 8 \rangle$	0.609	$\langle 64, 8 \rangle$
thermal2	4.158	$\langle 256, 8 \rangle$	4.158	$\langle 256, 8 \rangle$

TABLE 3.13: GPU execution time of the sparse matrix-vector product for CSR format in milliseconds (ms) with auto-tuning

The second column presents the SpMV GPU time in milliseconds by considering the *default gridification* indicated in the third column. The fourth column reports the GPU execution time in milliseconds of Alinea with the tuned grid ($\langle ntb, tw \rangle$) given in the fifth column, where ntb and tw describe respectively the number of threads per block and the number of threads used per warp.

Our experiments have produced results that are encouraging to continue our investigation of the best gridification approach. The best results are obtained when we consider a gridification with half a half-warp, *i.e.*, 8 threads in a warp. In some rare cases, when our implementation is not the best (see Table 3.12), by tuning the gridification, we obtain better results. For example, cuSPARSE is better than Alinea for the `2cubes_sphere` matrix by considering 256 threads per block and *half a half-warp* used, but Alinea becomes better than cuSPARSE for 64 threads per block and 8 threads used per warp. The gridification tuning, by optimizing the number of threads per block and the number of threads per warp depending on the matrix size, definitely increases the performance. Knowing that the gridification strongly impacts the performance of our algorithms, as demonstrated with CSR matrices, we now focus on the question of which storage format results in the best performances for the Alinea library. The results are pointed out in Table 3.12 in the two last columns. As we can see in column six and seven of Table 3.12, the ELL format is very effective when the matrix has a nearly uniform distribution of non-zero values in the rows, see nz/h in Table 3.11. This is the case when the standard deviation of the number of non-zero values per row is low. However, if a sparse matrix has a full line, this ELL format may take more space than the matrix itself. In this case the ELL format loses all its advantages. When we deal with a matrix with a high standard deviation of the non-zero values per row, *i.e.*, the matrix is poorly structured, COO format is better almost in every case. If the matrix is moderately structured, ELL format treats the “*healthy*” part efficiently and COO supports the rest. When the number of zeros per row is almost constant then the matrix is well structured, and the ELL format shows its full advantages. Furthermore, it is worthwhile to choose the ELL format rather than the CSR format even if the standard deviation of the number of values per row is not uniform, because it is memory-efficient when the average row density is close to the greatest row density. According to the obtained results, the format to use for computing a sparse matrix-vector product should be carefully selected to fit the matrix sparse pattern characteristics, especially the row density of non-zero entries. The results clearly show that the sparse matrix-vector product on the GPU device is faster than on the CPU host.

SpMV: influence of the gridification

We investigate the sparse matrix-vector multiplication when the matrix stored in the CSR format comes from a finite element discretization [125] [126]. A finite element matrix is

associated with a finite element mesh and the coefficients of the matrix are correlated with the interaction of the cells of the mesh. As a consequence, except for high-order finite elements which require a different analysis outside the scope of this paper [127] [128], the finite element matrices are sparse matrices with less than 60 coefficients per row. We vary the parameters used in the gridification in order to evaluate their influence on the SpMV algorithm for finite element analysis. We compare cuSPARSE and Cusp to our in-house implementation (Alinea). The experiments have been performed on two typical types of matrices coming from a discretization of academic finite element problems. The workstation used for the experiments consists of *Platform-1*.

We define a “band diagonal matrix” as a $n \times n$ matrix A satisfying the following conditions: $A_{i,l} \neq 0$ for $l = i - k, \dots, i + k$ where $i \geq 1, i \leq n$, and k is a given integer ($k \geq 1$ and $k \leq n$). We define a “multiple diagonal matrix” as a $n \times n$ matrix A satisfying: $A_{i,i-s_l-l} \neq 0$, where $i \geq 1, i \leq n$, and k is a given integer ($k \geq 1$ and $k \leq n$), for $s_l = l \frac{n}{k+1}$ ($l \geq 1$ and $l \leq k$). These two kinds of matrices correspond to two typical finite element matrices when the nodes of the associated mesh follow a frontal renumbering [129] [130]. Figure 3.18(a)

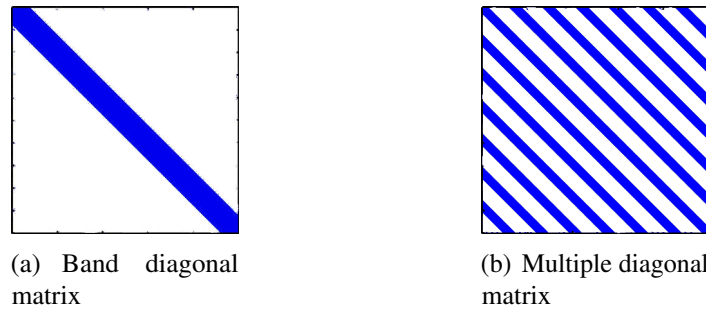


FIGURE 3.18: Band and multiple diagonal matrices

and Figure 3.18(b) illustrate respectively a “band diagonal matrix” and a “multiple diagonal matrix”. In this experiment, we vary the parameters k from $k = 0$ (a diagonal matrix) to $k = 40$ of band and multiple real matrices described in Figure 3.18. The default gridification considered consists of 256 threads per block and 8 threads per warp. Figure 3.19 represents the

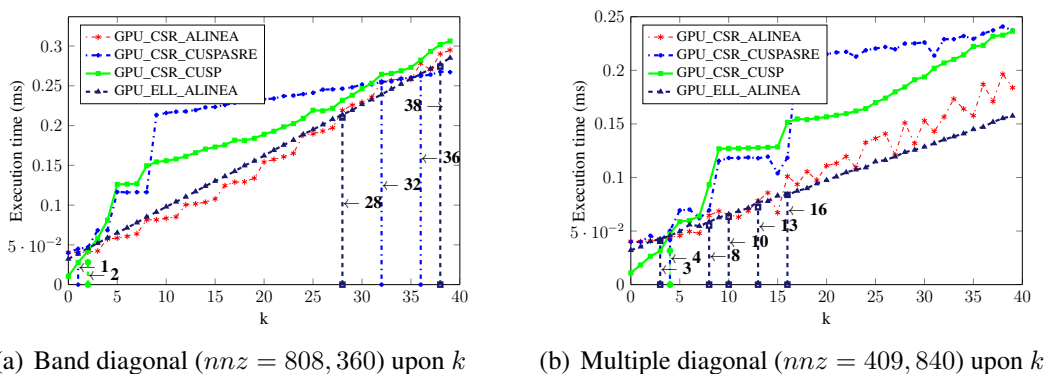


FIGURE 3.19: Double precision execution time of band and multiple diagonal in milliseconds upon k for CSR format: matrix of size $h = 10,000$. And Alinea ELL SpMV.

double precision CSR SpMV running times in milliseconds for Alinea, Cusp and cuSPARSE for a band diagonal matrix (Figure 3.19(a)) and a multiple diagonal matrix (Figure 3.19(b)) of size 10,000. The double precision CSR SpMV execution time in milliseconds for Alinea,

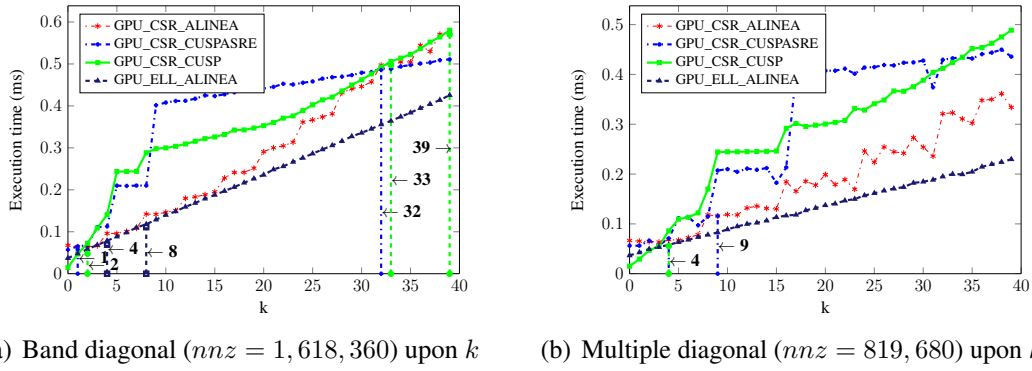


FIGURE 3.20: Double precision execution time of band and multiple diagonal in milliseconds upon k for CSR format: matrix of size $h = 20,000$

Cusp and cuSPARSE for a band diagonal matrix of size 20,000 and a multiple diagonal matrix of size 20,000 are presented in Figure 3.20(a) and Figure 3.20(b) respectively. Figure 3.21(a)

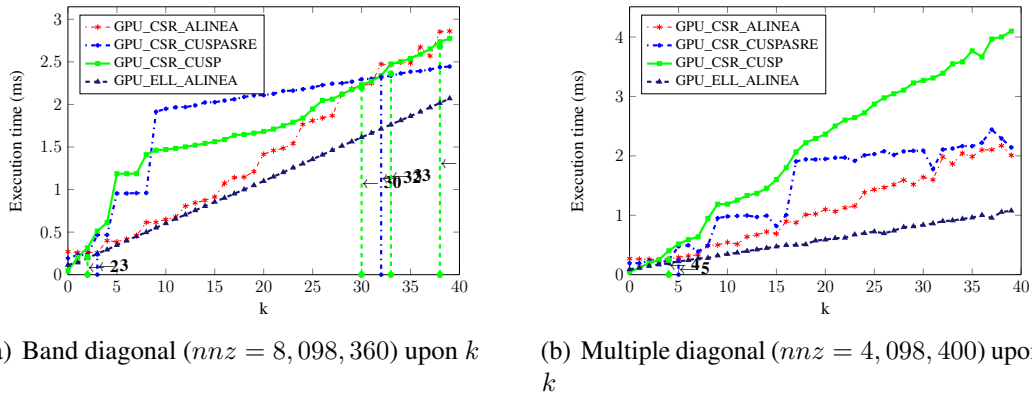


FIGURE 3.21: Double precision execution time of band and multiple diagonal in milliseconds upon k for CSR format: matrix of size $h = 100,000$. And Alinea ELL SpMV.

and Figure 3.21(b) report for Alinea, Cusp and cuSPARSE, the double precision times of SpMV (CSR format) in milliseconds for a band diagonal and multiple diagonal matrices of size 100,000.

Figure 3.19, Figure 3.20 and Figure 3.21 also collect the in-house double precision times of SpMV (ELL format) in milliseconds. The additional information allows us to compare in-house SpMV for ELL and CSR storage formats. As we can see in these figures, for a parameter k greater than 32, *i.e.*, matrices with around 64 non-zero value per line, Alinea loses leadership to cuSPARSE. On the opposite, Alinea is better than Cusp except for certain values of parameter k greater than 32. In summary, Alinea outshines cuSPARSE and the Cusp library for the CSR format when the number of non-zero values per row is lower than 60, which corresponds to typical finite element matrices. In addition, the results confirm the effectiveness of ELL SpMV compared to CSR. The ratio is more important when size increases. As described in Figure 3.19(a) ($n = 10000$), the SpMV CSR is efficient compared to ELL for a band diagonal except for some values of k higher than 28. On the contrary, for a multiple diagonal (Figure 3.19(b)), ELL is more effective than CSR, except for some punctual values of k . In order to analyze and evaluate the impact of the threading distribution on the grid of in-house implementations, we propose a benchmark that consists in varying the number of threads per block and the number of threads per warp. Note that the number of blocks

on the grid also changes. Should the graphics card configuration fail for the combination “warp+threads per block”, the corresponding bar will not be drawn. Figure 3.22(a) and Figure 3.22(b) give respectively the Alinea CSR SpMV running times in milliseconds for a band diagonal and multiple diagonal matrices of size 10,000. Figure 3.23(a) and Figure 3.23(b)

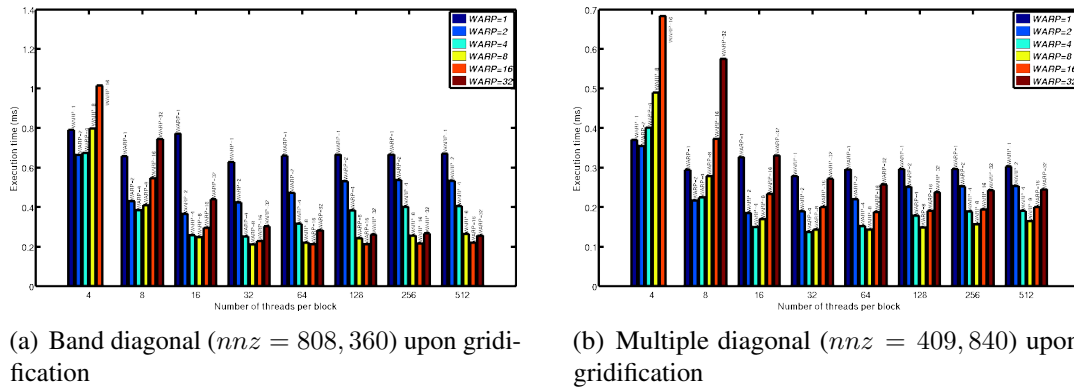


FIGURE 3.22: Double precision execution time of band and multiple diagonal in milliseconds upon gridification for CSR format: matrix of size $h = 10,000$

report those of size 20,000. Figure 3.24(a) and Figure 3.24(b) collect those of size 100,000.

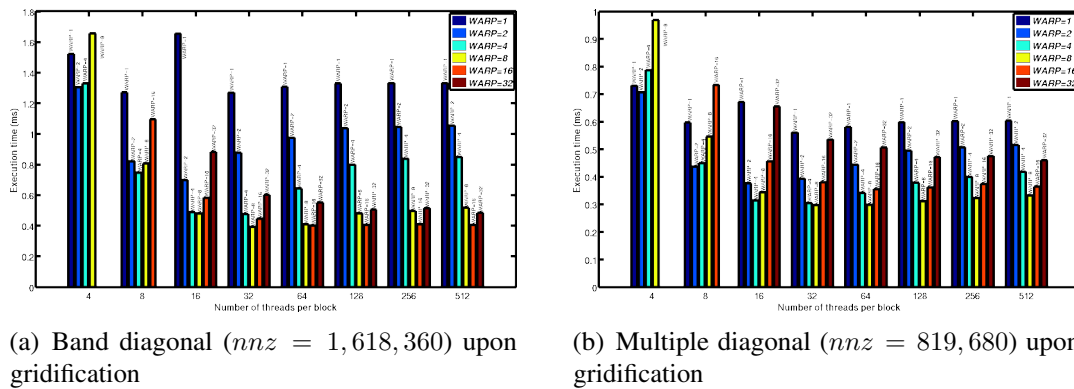


FIGURE 3.23: Double precision execution time of band and multiple diagonal in milliseconds upon gridification for CSR format: matrix of size $h = 20,000$

These figures point out that a half-warp, *i.e.*, 16 threads in a warp, gives the best results for

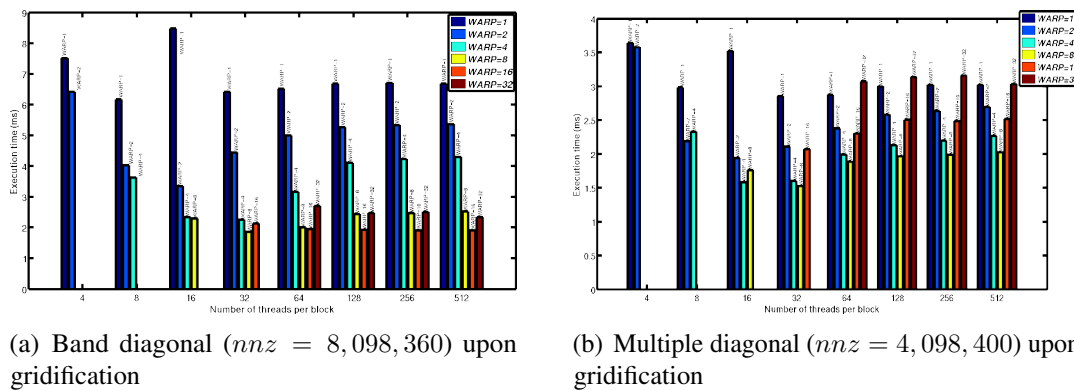


FIGURE 3.24: Double precision execution time of band and multiple diagonal in milliseconds upon gridification for CSR format: matrix of size $h = 100,000$

the “band diagonal matrix” and half a half-warp, *i.e.*, 8 threads in a warp, is better suited to

a “multiple diagonal matrix”. We can also conclude that an optimization of the number of threads per block, which depends on the matrix size, surely promotes better performance.

SpMV: dataset matrices from gravity equation, a realistic physical problem

Previously, we have performed experiments of a SpMV operation on the GPU using the matrix market data which contain various data in a wide range of applications, but with somewhat moderate sizes. Now, we aim to understand the efficiency of GPU computation for realistic engineering and scientific problems which are described by partial differential equations. In this set of experiments, we focus on SpMV in the CSR format. The SpMV CSR is evaluated in terms of computing time and in terms of energy consumption. Our algorithm is also compared to Cusp SpMV CSR. We tested a large 3D finite element matrix for the simulation of a realistic scientific problem from geophysics. The third set of experiments has also been performed on *Platform-1*. The sparse matrices are obtained from the *gravitational potential*

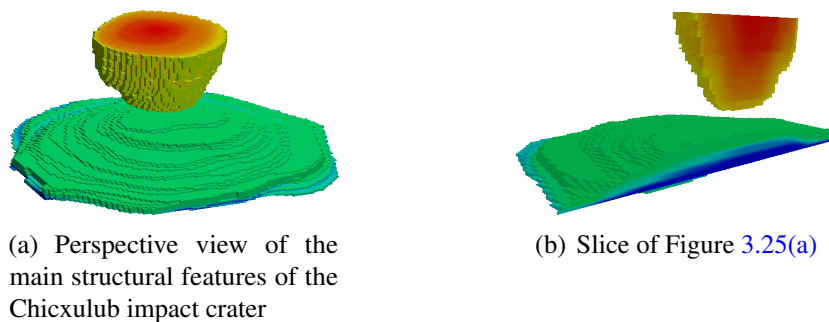


FIGURE 3.25: Chicxulub crater in the Yucatan Peninsula

equation applied on the Chicxulub crater in the Yucatan Peninsula in Mexico, Figure 3.25. An example of mesh used in the finite element method is given in Figure 3.26. First, we present

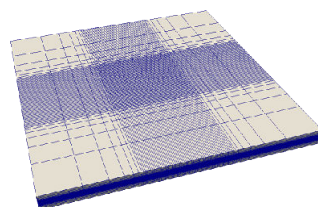


FIGURE 3.26: FE mesh associated with the *gravitational potential equation*

the context of the realistic problem, and then characterize the set of matrices used for the third set of numerical experiments. Then, we present the results for the sparse matrix-vector multiplication.

Gravitational potential equation Earth’s geological processes destroy most of the records of impact craters on Earth’s surface, however there is evidence that the Earth has also experienced collisions with other bodies in the Solar System. One such evidence is the ancient Chicxulub impact crater, located underneath the town of Chicxulub in Yucatán, southwest of Mexico on the Yucatán Peninsula. Yucatán is bordered by the states of Campeche to the southwest and Quintana Roo to the east, and the Gulf of Mexico to the north. Geological and other investigations suggest that this impact crater dates back more than 65 million years approximately and was caused by a collision with a meteorite. This event coincides with the mass extinction in the

late Cretaceous period, when the last of the dinosaurs vanished. The Chicxulub impact crater is now widely accepted as the main footprint of the global mass extinction event that marked the Cretaceous/Paleogene (K/Pg) boundary. Because of its relevance, the area of the impact crater was subjected to several geophysical studies including land, marine and aerial surveys. One of the techniques used to study the geological formations of the crater area was the gravity method. This method allows to quantify differences in the Earth's gravitational field at specific locations. Detected anomalies of the gravitational field allow to draw conclusions about the morphology and formation processes of geological structures. The gravity method allows to determine the depth, density and geometry of the gravitational anomaly source. In the case of the Chicxulub impact crater, the collected survey is relevant to a physical domain covering an area of $250km \times 250km$ and reaching $15km$ in depth.

The other way for geological exploration is the analysis of seismic waves, however it requires solving an “inverse problem”. By its very nature the problem is ill-posed and its solution might not sufficiently describe the subsurface, especially in case of strong contrasts in the velocity of seismic waves appearing near the surface. Below a certain depth, it is important to define the structures of the basement that the seismic method can not resolve. Hence, other techniques such as the potential method, best describe the subsurface. It consists in determining mass density distribution correlated with seismic velocities. The study of the gravitational potential equation gives additional information as to the mass density distribution. The gravity force is expressed as the sum of the gravitational force and the centrifugal force. The potential of gravity of a spherical density distribution is $\Phi(r) = Gm/r$, where m and r respectively represent the mass of the object and the distance to the object, and G is the universal gravity constant equal to $G = 6.672 \times 10^{-11} m^3 kg^{-1} s^{-2}$. For the spatial position x , the gravitational potential is expressed as $\Phi(x) = G \int (\rho(x')/||x - x'||) dx'$ where ρ is an arbitrary density distribution and x' is the point position within the considered density. The effect due to the rotation of the Earth, *i.e.*, the component of centrifugal force, is neglected. Hence, only the regional scale of the gravity potential equation is considered. The technique that has been used for the direct calculation of gravity anomalies is based on finite element (FE) method approximation of the Poisson equation [131]. The gravitational potential of a density anomaly distribution is thus given as the solution to the Poisson equation. The mathematical modeling is based on the following Poisson equation,

$$\begin{cases} -\Delta\Phi = 4\pi G\delta\rho = f & \text{on } \Omega, \\ \mathcal{C}(\Phi) = b & \text{on } \partial\Omega, \end{cases} \quad (3.26)$$

where $\delta\rho$ is the density anomaly and G the gravitational constant and \mathcal{C} represents Dirichlet boundary conditions. This technique does not require defining discontinuities in the discrete formulation of the problem and enables us to treat a larger number of discrete points than semi analytic methods. The discretization of this problem leads to a linear system which can be very large when we consider high order finite elements.

Datasets matrices The computational domain Ω consists of a box $250km \times 250km \times 15km$, Figure 3.26. A sparse matrix is obtained by a finite element method with Q1 discretization. One should note that the sizes of matrices correspond to those used in testing basic linear algebra operations, *i.e.*, SCAL, AXPY, XMY and DOT. The properties of matrices

with different mesh sizes are summarized in Table 3.14 and examples of pattern are given in Table 3.15. The quantities h , nnz , $density$, $bandwidth$, max_row , nnz/h , $nnz/h\ stddev$ have the same meaning as in properties given in Table 3.11.

h	nnz	$density$	$bandwidth$	max_row	nnz/h	$nnz/h\ stddev$
101,168	2,310,912	0.022	6,209	27	22.752	8.666
296,208	7,101,472	0.008	12,869	27	23.975	7.544
544,563	13,348,267	0.005	19,439	27	24.512	6.932
650,848	16,044,032	0.004	21,929	27	24.651	6.758
848,256	21,073,920	0.003	26,225	27	24.844	6.504
1,213,488	30,434,592	0.002	33,389	27	25.080	6.171
1,325,848	33,321,792	0.002	33,389	27	25.132	6.094

TABLE 3.14: Sketches of matrices from the 3D FE discretization of the gravitational potential equation

In Table 3.15 the second column represents the sparse matrix pattern and the third column represents the distribution of the non-zero elements. All those matrices have the same non-zero pattern structure.

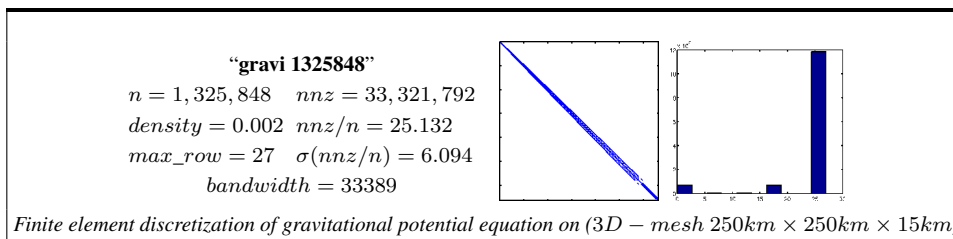


TABLE 3.15: Example pattern of matrices from the 3D FE discretization of the gravitational potential equation

The sizes of generated matrices vary from 544,563 to 1,325,848 rows, and the numbers of non-zeros values vary from 13,348,267 to 33,321,792.

Numerical experiments Table 3.16 shows the times in seconds (s), the electrical power in Watt (W), and the energy consumption in Joules (J). Both results by Alinea (in-house implementation) and Cusp for CSR SpMV with gravitational potential matrices in double precision are listed. The electrical power in Watts is computed using the experimental protocol and the execution process described in Section 2.5, Page 24. The energy consumption in Joules is calculated by the multiplication of the energy and time of the operation plus the fundamental energy consumption by the GPU. Table 3.16 also compares the double precision execution times in seconds of our in-house CSR SpMV implementation for CPU and GPU. They are respectively collected in the second and third columns.

n	Alinea				Cusp GPU		
	CPU time (s)	GPU time (s)	P (W)	E (J)	time (s)	P (W)	E (J)
101,168	0.017	0.001	144.229	0.136	0.001	151.423	0.183
296,208	0.047	0.003	148.591	0.401	0.003	154.756	0.541
544,563	0.085	0.005	151.245	0.754	0.006	156.787	1.007
650,848	0.100	0.006	149.42	0.891	0.008	156.229	1.198
848,256	0.130	0.008	149.289	1.176	0.01	155.357	1.552
1,213,488	0.200	0.011	149.354	1.676	0.014	155.28	2.219
1,325,848	0.220	0.012	145.663	1.807	0.016	155.419	2.427

TABLE 3.16: Double precision SpMV CSR of the gravitational potential matrices

As we can see in Table 3.16, the speed-ups become more than thirty three in favour of GPU. Numerical experiments clearly illustrate the gains from GPU computations for large-size problems and especially for linear algebra operations. Figure 3.27(a) and Figure 3.27(b)

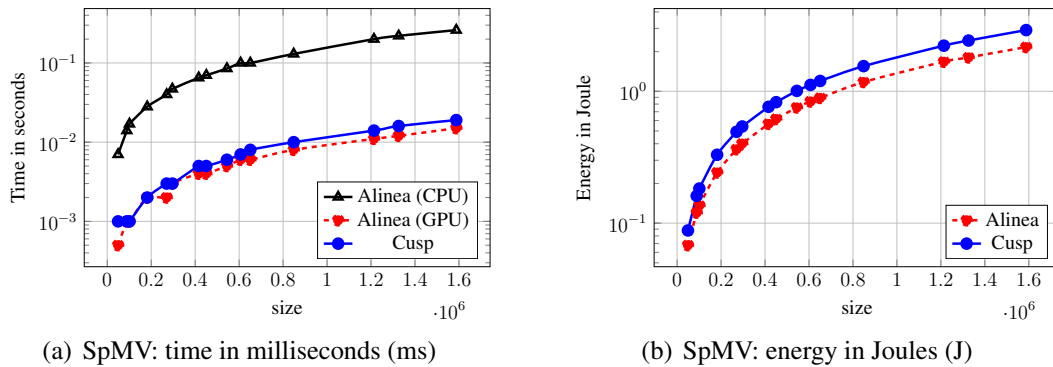


FIGURE 3.27: SpMV CSR of the gravitational potential matrices: time in milliseconds (ms) and energy in Joules (J)

respectively show a graphic data presentation of execution times in seconds and energy consumption in Joules, as given in Table 3.16. We can see Alinea achieves better results than Cusp in terms of execution time and energy consumption. Alinea greatest advantage is energy consumption, 25% less energy for the largest size. The complexity of SpMV for the matrix generated by equation (3.26) is linear because non-zero in each row is almost constant for all sizes of the matrices. The linear prediction of the GPU execution time and energy consumption for double precision SpMV are given in Table 3.17.

	equation	coef. of determination
Alinea Time (s)	$-3.552 \times 10^{-5} + 9.286 \times 10^{-9}x$	0.9996
Cusp Time (s)	$1.165 \times 10^{-5} + 1.177 \times 10^{-8}x$	1.000
Alinea Energy (J)	$-1.223 \times 10^{-3} + 1.371 \times 10^{-6}x$	0.9999
Cusp Energy (J)	$-6.667 \times 10^{-4} + 1.834 \times 10^{-6}x$	1.000

where x is the size of the problem.

TABLE 3.17: Linear prediction of the SpMV CSR of the gravitational potential matrices

SpMV: complex number arithmetics

This set of experiments gives an analysis and an evaluation of the sparse matrix-vector product on the GPU with complex number arithmetics with double precision. Numerical experiments carried out on a set of acoustic matrices arising from the modeling of acoustic phenomena within a car compartment are exposed, exhibiting the performance, robustness and efficiency of our algorithms, with a ratio up to 10x in complex double precision arithmetics.

Simulation of acoustic problems Amongst other industries, acoustic performance is a major concern in automotive companies. To tackle this, several models are used. In this study, we limit ourselves to acoustic models applying to closed cavities where the acoustic problem is independent from the surrounding structure. We are able to assume that the pressure field does not have any interactions with the enclosed structure. The acoustic problem, in its simplest linear form, is governed in the frequency domain by the Helmholtz equation with suitable boundary conditions. When the high frequency regime is considered, the matrix of the linear system becomes very large. The matrices we aim to evaluate arise from the discretization of

the Helmholtz equation (3.27) in a bounded domain Ω , with a boundary condition considered on the outside boundary $\Gamma = \partial\Omega$. The Helmholtz equation is expressed as follows

$$\begin{cases} -\nabla^2 u - k^2 u = g & \text{on } \Omega, \\ \mathcal{C}(u) = b & \text{on } \Gamma = \partial\Omega, \end{cases} \quad (3.27)$$

where $k = \frac{2\pi F}{c}$ is the wavenumber associated with the frequency $F \in \mathbb{R}$ and $c \in \mathbb{R}$ denotes the velocity of the medium that is different in space. In this experiment, Dirichlet boundary conditions are considered along a part of Γ . The frequency domain in which the solution is sought is usually limited, so as to analyze the acoustic response at specific places of the cavity (for instance around the driver's ears). To carry this out, a suitable numerical model has to be used. For complex geometries, two models can be chosen, depending on the boundary conditions. If there are conditions on all boundaries of the domain, then boundary element (BE) methods can be used. Else, finite element (FE) methods, which are methods based on the domain, are used to solve a weak formulation of the problem. When using FE methods, mesh requirements (around 10 nodes per wavelength are necessary) make the mesh sizes gigantic when dealing with high frequencies. This experiment focuses on effectively handling large-size acoustic problems using FE method.

Dataset matrices: automotive acoustic Now, we present the finite element meshes used for solving the acoustic problems arising from the automotive industry [132].

We focus on numerical examples that enable us to evaluate the performances of our procedures. We consider the study in a car compartment with Audi (Audi3D) and Twingo (Twingo3D). Let us look at the example of a *car compartment*. The goal is to construct the

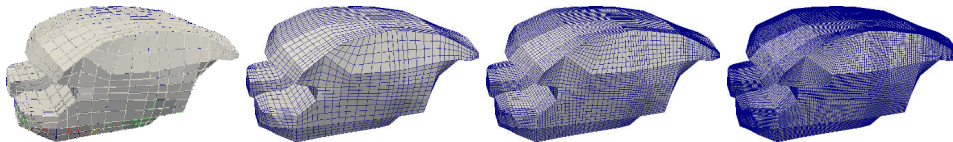


FIGURE 3.28: Audi 3D, $h = (0.133425, 0.066604, 0.033289, 0.016643)$

frequency response function at the driver's ear from the velocity boundary conditions along the firewall. Understanding this problem can help solve similar problems where the evaluation of the acoustic response to vibrating panels inside a cavity is at stake. Several sources can explain such mechanical vibrations. The vibrations can indeed be air-borne or structur-borne. And the prediction of these vibrations can be a difficult task. In the case of automotive applications, the higher the frequency is, the worst the quality of numerical predictions for mechanical vibrations. Acoustic predictions depend on the treatment of these mechanical vibrations,

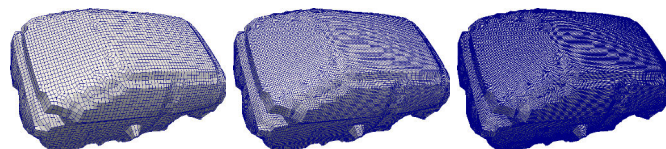


FIGURE 3.29: Twingo 3D, $h = (0.077866, 0.038791, 0.019379)$

precise acoustic predictions are possible only if correct vibration profiles along the car body are provided. According to advanced FE methodologies used on car bodies, computing accurate results becomes difficult when the frequencies are higher than 2500 Hz. Such difficulty to

produce correct results at high frequencies can be explained by the complex mechanical structure of a car body.

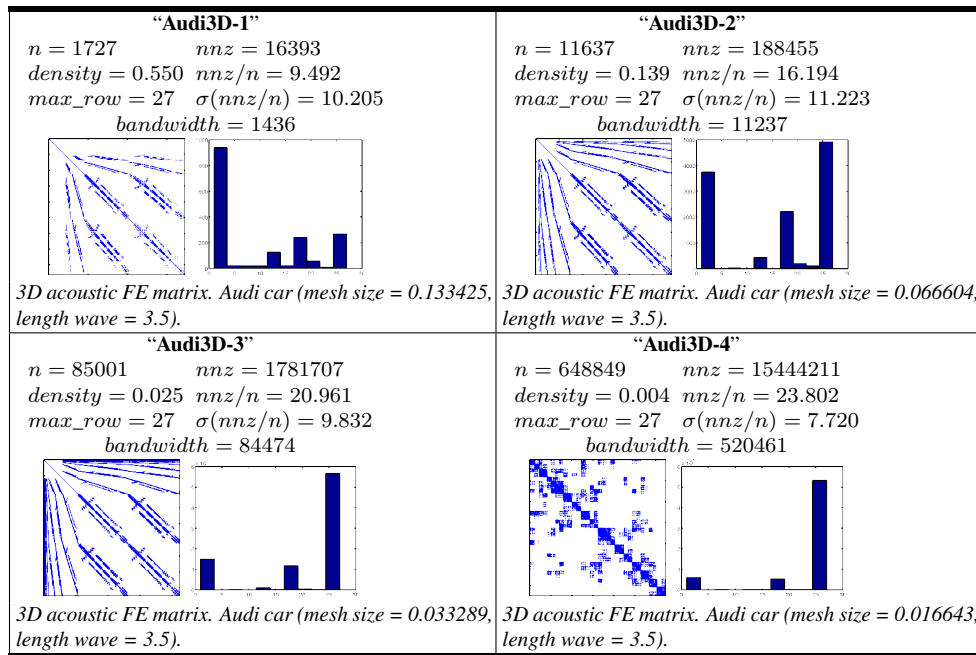


TABLE 3.18: Sketches of the Audi 3D FE acoustic matrices

Usual models do not consider parameters which are essential to understand the behavior of a car body at high frequencies, such as the characteristics of the connections, the damping properties, etc. We can note that modifying the models to make them take such parameters into account is not easy. At these high frequencies, some variability effects become important and complicate the predictions. For instance, two car bodies produced in the same way that could be considered identical, may present drastically different vibro-acoustic behaviors at such frequencies. The meshes of Audi 3D with different refinements (size h) are presented in Figure 3.28. Figure 3.29 describes the meshes of Twingo 3D with different refinements (size h).

Numerical experiments The experiments have been performed on the fourth platform, which consists of a workstation equipped with an Intel Core i7930 running with 2.67GHz, which has 4 cores and 12 GB main memory and two *NVIDIA* graphics cards: a Tesla K20c (dev#0) with 4799GB memory and GeForce GTX 570 with 1279MB memory (dev#1). The matrices are coming from the FE discretization of the Helmholtz equation for car compartment acoustic problems. The finite element discretization of the Helmholtz equations for acoustic problems leads to complex number arithmetics matrices. Table 3.18 and Table 3.19 collect a set of matrices respectively arising from Audi3D and Twingo3D. The quantities h , nnz , $density$, $bandwidth$, max_row , nnz/h , $nnz/h\ stddev$ have the same meaning as in properties given in Table 3.11.

The matrices of the acoustic problem are of large-size and are sparse. In this experiment, the SpMV is evaluated in CSR format. The default gridification and advanced auto-tuned techniques to organize threads on the CUDA grid are considered. We report in Table 3.20 the SpMV execution time and the number of floating operations per second when using the CSR format for in-house implementation of complex number arithmetics with double precision.

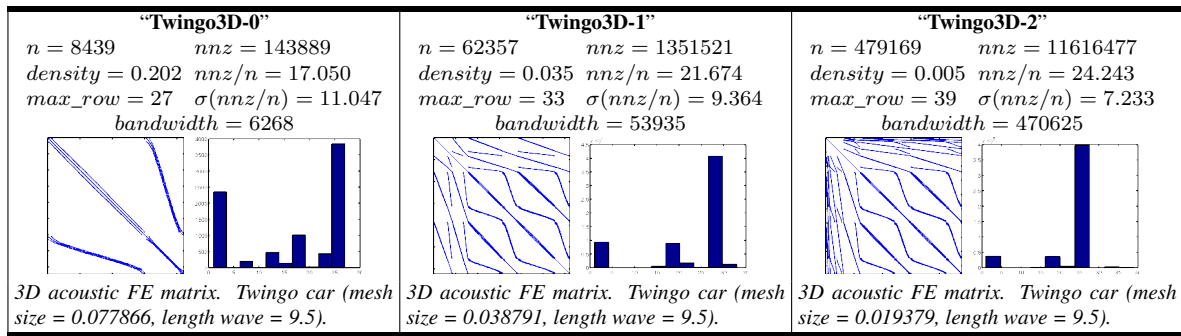


TABLE 3.19: Sketches of the Twingo 3D FE acoustic matrices

problem	cpu time (ms)	cpu Gflops	dev#0 time (ms)	dev#0 Gflops	dev#1 time (ms)	dev#1 Gflops	ratio#0 cpu/#0	ratio#1 cpu/#1
Audi3D-0	0.01	0.61	0.07	0.12	0.06	0.13	0.19	0.21
Audi3D-1	0.20	0.67	0.11	1.23	0.12	1.07	1.84	1.60
Audi3D-2	2.22	0.68	0.37	4.03	0.42	3.56	5.93	5.24
Audi3D-3	20.00	0.71	2.22	6.41	3.03	4.70	9.00	6.60
Audi3D-4	180.00	0.69	18.33	6.74	24.00	5.15	9.82	7.50
Twingo3D-0	1.67	0.69	0.28	4.06	0.33	3.45	5.88	5.00
Twingo3D-1	15.71	0.69	1.79	6.05	2.44	4.43	8.80	6.44
Twingo3D-2	140.00	0.66	14.29	6.51	16.67	5.58	9.80	8.40

TABLE 3.20: Alinea complex double precision CSR SpMV execution time in milliseconds (ms)

Numerical experiments clearly show that GPU operations are more efficient than CPU operations when calculating the sparse matrix-vector product in CSR format for complex number arithmetics with double precision.

SpMV: what about OpenCL?

The last set of numerical experiments consists in an evaluation of SpMV algorithms with OpenCL. The matrices used in the experiments arise from the finite element discretization of the equations of the deformation and the stress distribution in an *O*-ring under pressure. This is done by determining the expression of local constraints and deformations at each point of the joint. We vary the size of the finite element mesh in Figure 3.30(b) of the Computer-Aided

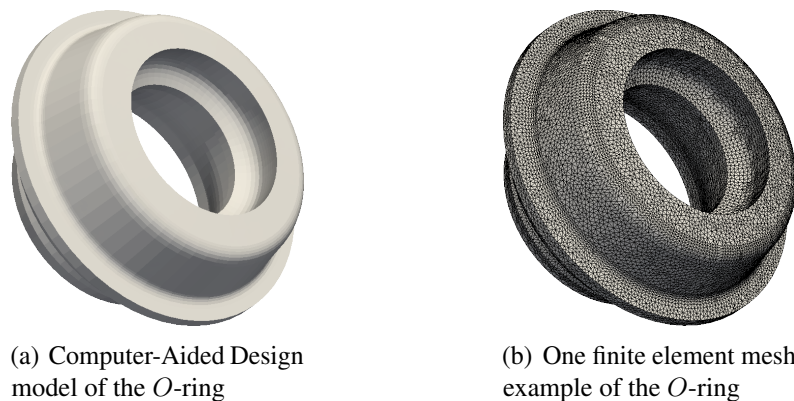


FIGURE 3.30: CAD and mesh of the *O*-ring

Design model shown in Figure 3.30(a). The experiments have been performed on *Platform-2*.

Figure 3.31(a) compares the SpMV execution time in microseconds of CPU, CUDA and OpenCL.

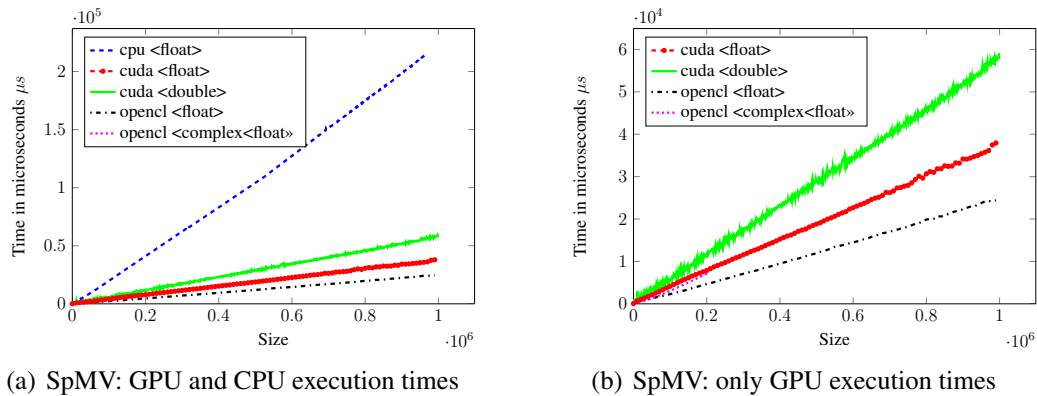


FIGURE 3.31: Running times of the sparse matrix-vector product (SpMV) in CSR format (μs)

As demonstrated above, the results clearly show that GPU is more effective than CPU for large-size. The gain is even more important when the problem size is increasing. The gridification is here optimized to ensure high-performance for CUDA. As we can see in the Figure 3.31(a), OpenCL outperforms CUDA for the considered gridification.

The parallel multi-core-GPU SpMV will be detailed within parallel algorithms, *e.g.*, sub-structuring, etc.

3.6 Implementation of Sparse linear system solvers

Solving linear systems represents an indispensable step in numerical methods such as the finite difference method, the finite element method, etc. Generally, these methods require the solution of large-size sparse linear systems. The choice of a concrete algebraic solver is governed by several factors, but the two main ones are the size of the system and the parallelization issues. Direct solvers are tuned almost to perfection but above a certain size, in the linear system, they are inferior to iterative solvers due to the available memory constraints. Also iterative solvers are more amenable for parallel implementation. Among many available iterative methods, preconditioned Krylov methods [79] [84] are the ones used most often, especially for very large systems. Effective implementation of Krylov solvers depends mostly on the effective sparse matrix-vector multiplication. Having investigated the different strategies to optimize linear algebra operations, in this section we explain the implementation of iterative algorithms, especially the iterative Krylov algorithms. We propose to explore and compare different iterative Krylov methods, which require a computational effort of linear algebra operations.

We focus on various Krylov methods: *Conjugate Gradient (CG)* for symmetric positive-definite matrices, *Generalized Conjugate Residual (GCR)*, *Stabilized BiConjugate Gradient (Bi-CGSTAB)*, *Stabilized BiConjugate Gradient (l) (Bi-CGSTAB(l))*, *Quasi Minimal Residual (QMR)*, *Transpose-Free Quasi Minimal Residual (TFQMR)* and *Bi-Conjugate Gradient Conjugate Residual (Bi-CGCR)* for the solution of sparse linear systems with non-symmetric matrices. We propose numerical experiments for matrices resulting from a large scale of

different applications of the finite element method, such as the gravitational potential equation presented in Section 3.5.3, Page 67. First, we give an overview of the mathematical aspects of the investigated iterative Krylov methods. We present how we implement these methods efficiently on GPU, then we analyze and evaluate their performances referring to the collected numerical results.

3.6.1 Iterative Krylov Algorithms

Before illustrating the implementation of the algorithms of the different solvers, we will first present the mathematical aspects of these methods in order to better understand the key points of each algorithm. Let's consider the system (3.1), $Ax = b$, where $A \in \mathbb{C}^{n \times n}$ is a sparse matrix, $b \in \mathbb{C}^n$ the right hand side and $x \in \mathbb{C}^n$ represents the solution vector we looking for. We consider large-size sparse matrices, *i.e.*, $n \in \mathbb{N} \gg 0$, *i.e.*, n is large enough. Let's consider the system (3.23), $M^{-1}Ax = M^{-1}b$, where $M \in \mathbb{C}^{n \times n}$ such as M^{-1} , the *left preconditioning* is an approximation of A^{-1} . In this thesis, we consider *left preconditioning*. Let us define (\cdot, \cdot) the scalar product of two vectors in \mathbb{C} considering the bilinear form $(x, y) = y^T x$ as the governing "inner product".

Conjugate Gradient method (CG)

The *Conjugate Gradient (CG)* method is an iterative method for solving linear systems of the system form (3.1) where matrices are symmetric positive-definite. The CG method has been propounded initially in 1952 by Hestenes *et al.* [133]. The CG method is based on an orthogonal projection technique onto the Krylov subspace of order k , $\mathcal{K}_k(A, r^{(0)})$, where $r^{(0)}$ is the initial residual. The idea of the CG method much like Krylov subspace methods, is to construct a basis of the Krylov subspace of order k , $\mathcal{K}_k(A, r^{(0)})$, seek an approximate solution $x^{(k)}$, $k \in \mathbb{N}$ from this subspace, such as $x^{(k)} \in x^{(0)} + \mathcal{K}_k(A, r^{(0)})$ and satisfy the *Petrov-Galerkin* condition $r^{(k)} = b - Ax^{(k)} \perp \mathcal{K}_k(A, r^{(0)})$ where $x^{(0)}$ is the initial solution.

The main principle relies on a short recurrence for the approximate solution $x^{(k)}$ update step with a given starting point $x^{(0)}$:

$$x^{(k+1)} = x^{(k)} + \alpha^{(k)} p^{(k)}, \quad (3.28)$$

where $\alpha^{(k)} \in \mathbb{C}$ is the descent coefficient and $p^{(k)}$ represents the descent-director vector in \mathcal{K}_k , which verifies the orthogonality and conjugacy conditions, *i.e.*, for $i, j \in \mathbb{N}$, $i \neq j$, $p^{(i)}$ and $p^{(j)}$ are *A-conjugate* and *A-orthogonal*. That means that $p^{(i)T} A p^{(j)} = 0$, $i \neq j$. According to the recurrence relation (3.28), the residual vectors must satisfy the recurrence $r^{(k+1)} = r^{(k)} - \alpha^{(k)} A p^{(k)}$. Assuming that all residual vectors, $r_{k \geq 0}^{(k)}$, are non-zero, and if the $r_{k \geq 0}^{(k)}$ are to be orthogonal, then $(r^{(k+1)}, r^{(k)}) = (r^{(k)} - \alpha^{(k)} A p^{(k)}, r^{(k)}) = 0$. Therefore, descent-director vectors are expressed as

$$\alpha^{(k)} = \frac{(r^{(k)}, r^{(k)})}{(A p^{(k)}, r^{(k)})} \quad (3.29)$$

The next search direction $p^{(k+1)}$ is a linear combination of the residual $r^{(k+1)}$ and a scaled version of the current direction $p^{(k)}$. The search direction $p^{(k+1)}$ is expressed as follows

$$p^{(k+1)} = r^{(k+1)} + \beta^{(k)} p^{(k)}, \quad \beta^{(k)} \in \mathbb{C}, \quad (3.30)$$

where the initial descent $p^{(0)} = r^{(0)}$. According to (3.30), we have

$$\begin{aligned} (Ap^{(k)}, r^{(k)}) &= (Ap^{(k)}, p^{(k)} - \beta^{(k-1)}p^{(k-1)}) = (Ap^{(k)}, p^{(k)}) - \beta^{(k-1)}(Ap^{(k)}, p^{(k-1)}), \\ &= (Ap^{(k)}, p^{(k)}) - \beta^{(k-1)}(Ap^{(k)}, p^{(k-1)}) = (Ap^{(k)}, p^{(k)}), \end{aligned}$$

because the pairs of the sequence $\{p^{(k)}\}_{k \in \mathbb{N}}$ are *A-conjugate*, i.e., for $i, j \in \mathbb{N}$, $i \neq j$, $p^{(i)T}Ap^{(j)} = 0$, which gives $(Ap^{(k)}, p^{(k-1)}) = 0$.

Therefore, the relation (3.29) can be rewritten $\alpha^{(k)} = \frac{(r^{(k)}, r^{(k)})}{(Ap^{(k)}, p^{(k)})}$. The $\beta^{(k)}$ are chosen so that the descent vectors satisfy the orthogonality condition, i.e., $(Ap^{(k)}, p^{(k+1)}) = 0$. The relation (3.30) leads us to

$$\begin{aligned} (Ap^{(k)}, p^{(k+1)}) = 0, &\Leftrightarrow (Ap^{(k)}, r^{(k+1)} + \beta^{(k)}p^{(k)}) = 0, \\ \Leftrightarrow (Ap^{(k)}, r^{(k+1)}) + \beta^{(k)}(Ap^{(k)}, p^{(k)}) = 0 &\Leftrightarrow \beta^{(k)} = -\frac{(Ap^{(k)}, r^{(k+1)})}{(Ap^{(k)}, p^{(k)})} = -\frac{(Ap^{(k)}, r^{(k+1)})}{(Ap^{(k)}, r^{(k)})}, \end{aligned}$$

or, $Ap^{(k)} = -\frac{1}{\alpha^{(k)}}(r^{(k+1)} - r^{(k)}),$ so, $\beta^{(k)} = -\frac{(r^{(k+1)}, r^{(k+1)}) - (r^{(k+1)}, r^{(k)})}{(r^{(k+1)}, r^{(k)}) - (r^{(k)}, r^{(k)})}.$

Finally, the $\beta^{(k)}$ can be rewritten as follows $\beta^{(k)} = \frac{(r^{(k+1)}, r^{(k+1)})}{(r^{(k)}, r^{(k)})}$, because $(r^{(k+1)}, r^{(k)}) = 0$.

The main steps of the (left) preconditioned CG method are described in Algorithm 3.7.

Theorem 3.10 *In theory, i.e., in exact arithmetic, the conjugate gradient algorithm converges at most n iterations, i.e., it exists $k \in]0; n]$ such as $r^{(k)} = b - Ax^{(k)} = \varepsilon$, where ε is the tolerance threshold and k the final number of iterations of the CG.*

Nevertheless, in practice due to rounding errors, the CG method can take more than n iterations.

Conjugate Residual (CR) and Generalized Conjugate Residual (GCR)

Conjugate Residual (CR) When the matrix of the system (3.1) is only an *Hermitian* matrix, not symmetric positive-definite, the *Conjugate Residual (CR)* method, close to the CG method, is sometimes used. Nevertheless, the CR requires more numerical operations as well as more memory storage. Indeed, in addition of \mathbf{x} , \mathbf{r} , \mathbf{z} , $\mathbf{A}\mathbf{p}$, CR needs another vector $\mathbf{A}\mathbf{r}$. The CR method follows a similar construction and holds similar convergence properties as the CG method. However, the CG algorithm is often preferred over the CR method. In the CG algorithm, the descent vectors $p^{(k)}$ are *A-orthogonal*, i.e., *A-conjugate*, in the CR method, only $Ap^{(k)}$ are to be *orthogonal*, i.e., $p^{(k)}$ are *A^TA-orthogonal*. Algorithm 3.8 presents the main points of the (left) preconditioned CR method. The preconditioning matrix \mathbf{M}^{-1} must be symmetric. When A is Hermitian ($A = A^H$), the CR method is mathematically equivalent to the GMRES method [85] and the residual minimization property is based on the complex Euclidean norm. Reference [134] shows that the natural choice of complex symmetric linear systems is to consider the complex bilinear form $(x, y) = y^T x$ as the governing “inner product”.

Algorithm 3.7: Preconditioned Conjugate Gradient (P-CG) method

input : $A \in \mathbb{C}^{n \times n}$: initial matrix, $M \in \mathbb{C}^{n \times n}$: preconditioning matrix,
 b: right hand side vector, $\mathbf{x}^{(0)}$: initial solution,
 ε : tolerance threshold, K : maximum number of iterations
output : \mathbf{x} : solution vector
variable: k , convergence
variable: vector: \mathbf{r} , \mathbf{z} , \mathbf{Ap}
variable: scalar: ρ

```
1 // - initialization
2 convergence  $\leftarrow$  false;  $k \leftarrow 0$ 
3  $\mathbf{r}^{(0)} \leftarrow \mathbf{b} - A\mathbf{x}^{(0)}$ 
4 // - loop until convergence
5 while .not. convergence .AND.  $k < K$  do
6    $\mathbf{z}^{(k)} \leftarrow M^{-1}\mathbf{r}^{(k)}$ 
7    $\rho^{(k)} \leftarrow (\mathbf{r}^{(k)}, \mathbf{z}^{(k)})$ 
8   if  $k = 0$  then
9      $\mathbf{p}^{(k)} \leftarrow \mathbf{z}^{(k)}$ 
10  else
11     $\beta^{(k)} \leftarrow \rho^{(k)} / \rho^{(k-1)}$ 
12     $\mathbf{p}^{(k)} \leftarrow \mathbf{z}^{(k)} + \beta^{(k)} \times \mathbf{p}^{(k-1)}$ 
13  end
14   $\mathbf{Ap}^{(k)} \leftarrow A \times \mathbf{p}^{(k)}$ 
15   $\alpha^{(k)} \leftarrow \rho^{(k)} / (\mathbf{Ap}^{(k)}, \mathbf{p}^{(k)})$ 
16   $\mathbf{x}^{(k)} \leftarrow \mathbf{x}^{(k-1)} + \alpha^{(k)} \times \mathbf{p}^{(k)}$ 
17   $\mathbf{r}^{(k)} \leftarrow \mathbf{r}^{(k-1)} - \alpha^{(k)} \times \mathbf{Ap}^{(k)}$ 
18  convergence  $\leftarrow$  ( $\|\mathbf{r}^{(k)}\| < \varepsilon$ )
19   $k \leftarrow k + 1$ 
20 end
```

Algorithm 3.8: Preconditioned Conjugate Residual (P-CR) method

input : $A \in \mathbb{C}^{n \times n}$: initial matrix, $M \in \mathbb{C}^{n \times n}$: preconditioning matrix,
 b: right hand side vector, $\mathbf{x}^{(0)}$: initial solution,
 ε : tolerance threshold, K : maximum number of iterations
output : \mathbf{x} : solution vector
variable: k , convergence
variable: vector: \mathbf{r} , \mathbf{zAp} , \mathbf{Ap} , \mathbf{Ar}
variable: scalar: ρ

```
1 // - initialization
2 convergence  $\leftarrow$  false;  $k \leftarrow 0$ 
3  $\mathbf{r}^{(0)} \leftarrow \mathbf{b} - A\mathbf{x}^{(0)}$ 
4 // - loop until convergence
5 while .not. convergence .AND.  $k < K$  do
6    $\mathbf{Ar}^{(k)} \leftarrow A \times \mathbf{r}^{(k)}$ 
7    $\rho^{(k)} \leftarrow (\mathbf{r}^{(k)}, \mathbf{Ar}^{(k)})$ 
8   if  $k = 0$  then
9      $\mathbf{Ap}^{(k)} \leftarrow \mathbf{Ar}^{(k)}$ 
10  else
11     $\beta^{(k)} \leftarrow \rho^{(k)} / \rho^{(k-1)}$ 
12     $\mathbf{p}^{(k)} \leftarrow \mathbf{r}^{(k)} + \beta^{(k)} \times \mathbf{p}^{(k-1)}$ 
13     $\mathbf{Ap}^{(k)} \leftarrow \mathbf{Ar}^{(k)} + \beta^{(k)} \times \mathbf{Ap}^{(k-1)}$ 
14  end
15   $\mathbf{zAp}^{(k)} \leftarrow M^{-1}\mathbf{Ap}^{(k)}$ 
16   $\alpha^{(k)} \leftarrow \rho^{(k)} / (\mathbf{Ap}^{(k)}, \mathbf{zAp}^{(k)})$ 
17   $\mathbf{x}^{(k)} \leftarrow \mathbf{x}^{(k-1)} + \alpha^{(k)} \times \mathbf{p}^{(k)}$ 
18   $\mathbf{r}^{(k)} \leftarrow \mathbf{r}^{(k-1)} - \alpha^{(k)} \times \mathbf{zAp}^{(k)}$ 
19  convergence  $\leftarrow$  ( $\|\mathbf{r}^{(k)}\| < \varepsilon$ )
20   $k \leftarrow k + 1$ 
21 end
```

Generalized Conjugate Residual (GCR) Saad proposes in [69], a Generalized Conjugate Residual (GCR). The main keys of the GCR algorithm are exposed in Algorithm 3.9.

Algorithm 3.9: Precond. Generalized Conjugate Residual (P-GCR) method

input : $A \in \mathbb{C}^{n \times n}$: initial matrix, $M \in \mathbb{C}^{n \times n}$: preconditioning matrix,
b: right hand side vector, $\mathbf{x}^{(0)}$: initial solution,
 ε : tolerance threshold, K : maximum number of iterations
output : \mathbf{x} : solution vector
variable: k , convergence
variable: vector: \mathbf{r} , \mathbf{zAp} , \mathbf{Ap} , \mathbf{Ar} , \mathbf{zAr}
variable: scalar: ρ

```
1 // - initialization
2 convergence  $\leftarrow$  false;  $k \leftarrow 0$ 
3  $\mathbf{r}^{(0)} \leftarrow b - A\mathbf{x}^{(0)}$ 
4  $\mathbf{p}^{(0)} \leftarrow \mathbf{r}^{(0)}$ 
5 // - loop until convergence
6 while .not. convergence .AND.  $k < K$  do
7    $\mathbf{Ap}^{(k)} \leftarrow A \times \mathbf{p}^{(k)}$ 
8    $\mathbf{zAp}^{(k)} \leftarrow M^{-1} \mathbf{Ap}^{(k)}$ 
9    $\rho^{(k)} \leftarrow (\mathbf{Ap}^{(k)}, \mathbf{zAp}^{(k)})$ 
10   $\alpha^{(k)} \leftarrow (\mathbf{r}^{(k)}, \mathbf{zAp}^{(k)}) / \rho^{(k)}$ 
11   $\mathbf{x}^{(k)} \leftarrow \mathbf{x}^{(k-1)} + \alpha^{(k)} \times \mathbf{p}^{(k)}$ 
12   $\mathbf{r}^{(k)} \leftarrow \mathbf{r}^{(k-1)} - \alpha^{(k)} \times \mathbf{zAp}^{(k)}$ 
13   $\mathbf{Ar}^{(k)} \leftarrow A \times \mathbf{r}^{(k)}$ 
14   $\mathbf{zAr}^{(k)} \leftarrow M^{-1} \mathbf{Ar}^{(k)}$ 
15   $\mathbf{p}^{(k)} \leftarrow \mathbf{r}^{(k)}$ 
16  for  $i = 0$  to  $k$  do
17     $\beta_i^{(k)} \leftarrow -(\mathbf{zAr}^{(k)}, \mathbf{zAp}^{(i)}) / \rho^{(i)}$ 
18     $\mathbf{p}^{(k)} \leftarrow \mathbf{p}^{(k)} + \beta_i^{(k)} \times \mathbf{p}^{(i-1)}$ 
19  end
20  convergence  $\leftarrow$  ( $\|\mathbf{r}^{(k)}\| < \varepsilon$ )
21   $k \leftarrow k + 1$ 
22 end
```

Algorithm 3.10: Preconditioned Stabilized BiConjugate Gradient (Bi-CGSTAB)

input : $A \in \mathbb{C}^{n \times n}$: initial matrix, $M \in \mathbb{C}^{n \times n}$: preconditioning matrix,
b: right hand side vector, $\mathbf{x}^{(0)}$: initial solution,
 ε : tolerance threshold, K : maximum number of iterations
output : \mathbf{x} : solution vector
variable: k , convergence
variable: vector: \mathbf{r} , $\bar{\mathbf{r}}$, \mathbf{t} , \mathbf{v} , \mathbf{y} , \mathbf{p} , \mathbf{s}
variable: scalar: α , β , ρ , ω

```
1 // - initialization
2 convergence  $\leftarrow$  false;  $k \leftarrow 0$ 
3 { $\bar{\mathbf{r}}$ , an arbitrary vector, such that  $(\bar{\mathbf{r}}^{(0)}, \mathbf{r}^{(0)}) \neq 0$ }
4  $\rho^{(0)} \leftarrow \alpha = 1$ ;  $\omega^{(0)} \leftarrow 1$ 
5  $\mathbf{r}^{(0)} \leftarrow b - A\mathbf{x}^{(0)}$ ;  $\bar{\mathbf{r}}^{(0)} \leftarrow \mathbf{r}^{(0)}$ 
6 // - loop until convergence
7 while .not. convergence .AND.  $k < K$  do
8    $\rho^{(k+1)} \leftarrow (\bar{\mathbf{r}}^{(0)}, \mathbf{r}^{(k)})$ 
9    $\beta \leftarrow (\rho^{(k+1)} / \rho^{(k)}) (\alpha / \omega^{(k)})$ 
10   $\mathbf{p}^{(k+1)} \leftarrow \mathbf{r}^{(k)} + \beta \times (\mathbf{p}^{(k)} - \omega^{(k)} \times \mathbf{v}^{(k)})$ 
11   $\mathbf{y} \leftarrow M^{-1} \mathbf{p}^{(k+1)}$ ;  $\mathbf{v}^{(k+1)} \leftarrow A \times \mathbf{y}$ 
12   $\alpha \leftarrow \rho^{(k+1)} / (\bar{\mathbf{r}}^{(0)}, \mathbf{v}^{(k+1)})$ 
13   $\mathbf{s} \leftarrow \mathbf{r}^{(k)} - \alpha \times \mathbf{v}^{(k+1)}$ 
14   $\mathbf{z} \leftarrow M^{-1} \mathbf{s}$ ;  $\mathbf{t} \leftarrow A \times \mathbf{z}$ 
15   $\omega^{(k+1)} \leftarrow (M^{-1} \mathbf{t}, M^{-1} \mathbf{s}) / (M^{-1} \mathbf{t}, M^{-1} \mathbf{t})$ 
16   $\mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} + \alpha \times \mathbf{y} + \omega^{(k+1)} \mathbf{z}$ 
17  if  $\mathbf{x}^{(k+1)}$  is accurate enough then
18    convergence  $\leftarrow$  true
19  else
20     $\mathbf{r}^{(k+1)} = \mathbf{s} - \omega^{(k+1)} \times \mathbf{t}$ 
21  end
22   $k \leftarrow k + 1$ 
23 end
```

More iterative Krylov methods

For Hermitian positive-definite matrices, the classical CG method [133] is one of the most powerful iterative process for solving the system (3.1). For general non-Hermitian matrices, the CG method loses its robustness and effectiveness.

Therefore, in this thesis, we also discuss alternative iterative Krylov methods such as the *Stabilized BiConjugate Gradient (Bi-CGSTAB)*, the *Stabilized BiConjugate Gradient (l) (Bi-CGSTAB(l))*, the *Quasi Minimal Residual (QMR)*, the *Transpose-Free Quasi Minimal Residual (TFQMR)* and the *Bi-Conjugate Gradient Conjugate Residual (Bi-CGCR)*. Most of these methods are derived from the *BiConjugate Gradient (Bi-CG)* method [135] [69] [136], which is the “natural” generalization of the classical CG algorithm for Hermitian positive-definite matrices to general non-Hermitian linear systems.

Stabilized BiConjugate Gradient (Bi-CGSTAB) Prior to the Bi-CGSTAB method, the Conjugate Gradients-Squared (CG-S) method [137] has been identified as an attractive alternative to the Bi-Conjugate Gradient (Bi-CG) iterative method for solving certain non-symmetric linear systems. The CG-S method suffers from irregular convergence behavior [138] that may, in some cases, generate rounding errors, which impact drastic cancellation effects in the solution. Due to this remark, H. A. van der Vorst proposed [139] another variant of the Bi-CG method, a stabilized version of the CG algorithm, the stabilized BiConjugate Gradient (Bi-CGSTAB), which seems to overcome these negative effects. In exact arithmetic, both CG-S and Bi-CGSTAB break down everytime Bi-CG does. H. A. van der Vorst concluded in [139] that the convergence behavior of Bi-CGSTAB is much smoother and therefore gives much more accurate residual vectors. This method converges considerably faster than CG-S in the most cases [139]. H. A. van der Vorst and many others after [140] [141] [142] [143] [144], showed that Bi-CGSTAB with preconditioning is a very competitive method for solving relevant classes of non-symmetric linear systems. The pseudocode for the preconditioned Bi-CGSTAB iterative method is described in Algorithm 3.10. Reference [145] presents a parallel implementation of the GPU Bi-CGSTAB solver for the Poisson problem using CUDA.

Stabilized BiConjugate Gradient (l) (Bi-CGSTAB(l)) The Bi-CGSTAB(l) is a generalization of the Bi-CGSTAB presented by van der Vorst [139]. This method is presented by Sleijpen *et al.* [146] and then in [147]. In [146], Sleijpen *et al.* justify the motivation of this method by the fact that for a large class of equations, the convergence of Bi-CGSTAB stagnates for the typical type of equations, this has to do with the matrix having almost pure imaginary eigenvalues. Before [146], Gutknecht attempted, in a research report [148] (article published two years later [147]), to avoid this stagnation with Bi-CGSTAB2. Sleijpen *et al.* has concluded that Bi-CGSTAB(l) may be an attractive method and may be considered as a competitive algorithm to solve non-symmetric linear systems of equations.

For $l = 1$, the Bi-CGSTAB(l) algorithm computes exactly the same approximation of the solution $x^{(k)}$ as Bi-CGSTAB does. Reference [147] shows that Bi-CGSTAB(l) can be implemented in different ways. We adopted in this thesis the version presented in [146].

Quasi Minimal Residual (QMR) Due to the susceptible possibility of breakdowns (divisions by 0) and numerical instabilities of the Bi-CG algorithm, Freund and Nachtigal overcome the problems of Bi-CG by proposing a novel Bi-CG-like approach, the quasi-minimal residual method (QMR), in [149]. They present how Bi-CG iterates can be recovered stably from the QMR algorithm. It is shown in [149] that the QMR algorithm has smooth convergence curves and better numerical properties than the Bi-CG algorithm.

The QMR method requires matrix-vector multiplications with both the matrix A and its transpose A^T . The principle of the matrix transpose is simple, it consists in interchanging rows and columns. However, when the matrix is sparse, this operation can be complicated and costly in terms of computations. The operation is further complicated in parallel. One of the solutions is the TFQMR algorithm.

Transpose-Free Quasi Minimal Residual (TFQMR) The TFQMR is proposed in 1993 by R. W. Freund [150], for solving general non-singular non-Hermitian linear systems. The TFQMR method is closely related to the CG-S (QMR) algorithm. The main difference lies in the absence of the matrix transpose A^T computation, which justifies the name prefix of the

method “transpose-free”. The TFQMR can be implemented very simply by modifying only a few lines in the standard CG-S algorithm. Unlike the CG-S algorithm, the iterates of the TFQMR algorithm are characterized by a quasi minimization of the residual norm. Freund demonstrated in [150] that the TFQMR presents smooth convergence curves. However, the TFQMR can break down but it is very rare in practice. References [151] [152] give some results of convergence of the TFQMR method.

Bi-Conjugate Gradient Conjugate Residual (Bi-CGCR) In this thesis, we use the Bi-CGCR of M. Clemens [153] [154], which is a symmetric variant of the Bi-CG algorithm.

Algorithm and implementations The algorithms implemented in this dissertation are those presented by Youcef Saad in [69], which are based on the Lanczos algorithm for a class of non-symmetric systems [155] [156] [157]. However, in this thesis, we pay attention to optimizing both memory storages (limit the number of vectors) and avoiding unnecessary computations.

3.6.2 Implementation on a GPU

Data transfers (sending and receiving) between host and device are not negligible when dealing with GPU computing. It is essential to minimize data dependencies between CPU and GPU. In our implementation, we propose to send all input data from the host to the device just once, before starting the iterative routine. Nevertheless, each iteration of the iterative Krylov algorithm requires more than one calculation of the dot product (or norm) that implies data copy from the device to the host.

Numerical aspects of the Krylov methods

As said in Section 3.2, Page 41, in practice, iterative Krylov methods’ performance depends on the matrix conditioning, and efficient preconditioning techniques must be used to ensure fast convergence such as the ILU factorization [95] [96], domain decomposition methods [18] [97] [98], which involve various interface conditions [99], either based on a continuous optimization [158] [101] [102] using an approximation of Steklov-Poincaré operators, or an algebraic optimization [103] [104] using an approximation of the Schur complement matrix [105], etc. Designing the optimal preconditioner is a research problem in itself. All the mentioned preconditioning techniques are very efficient and are mandatory to achieve good performance.

Although these preconditioning strategies are implemented in our Alinea library, it is not easy to integrate them with the native Cusp library, which is necessary for an objective comparison of both libraries. The optimal preconditioner is outside the scope of this thesis. So we choose, for experiments, to use a diagonal preconditioner, which provides a pretty good preconditioning, if the matrices are not too ill-conditioned. Note that the basic diagonal preconditioner is available in all libraries (native Cusp and Alinea). For the sake of simplicity, we chose a preconditioning matrix M easy to compute and to invert. We have adopted M as the A diagonal which provides a relatively good preconditioning in most cases. This preconditioner is called the *Jacobi preconditioner* or *diagonal preconditioner*. The diagonal preconditioner is one of the simplest forms of preconditioning, which consists of the diagonal

of the matrix alone: if $i = j$, $m_{i,j} = a_{i,i}$ otherwise $m_{i,j} = 0$, and assuming $a_{i,i} \neq 0, \forall i$, we simply compute $m_{ij}^{-1} = \frac{1}{a_{i,i}}$ if $i = j$ otherwise $m_{ij}^{-1} = 0$:

$$M = \begin{pmatrix} a_{1,1} & 0 & 0 & \cdots & 0 \\ 0 & a_{2,2} & 0 & \cdots & 0 \\ 0 & 0 & \ddots & 0 & \vdots \\ \vdots & \vdots & 0 & \ddots & 0 \\ 0 & 0 & \cdots & 0 & a_{n,n} \end{pmatrix} \Leftrightarrow M^{-1} = \begin{pmatrix} \frac{1}{a_{1,1}} & 0 & 0 & \cdots & 0 \\ 0 & \frac{1}{a_{2,2}} & 0 & \cdots & 0 \\ 0 & 0 & \ddots & 0 & \vdots \\ \vdots & \vdots & 0 & \ddots & 0 \\ 0 & 0 & \cdots & 0 & \frac{1}{a_{n,n}} \end{pmatrix} \quad (3.31)$$

Our implementation computes the inverse of the diagonal on CPU and sends it to the GPU outside the iterative routine. Then the preconditioning step is done on the GPU with an element-wise multiplication operation. The iterative Krylov algorithm has been carried out on the host but all computing steps (DOT, NORM, AXPY, matrix-vector product) are performed on the device. In fact, at each iteration of the iterative Krylov algorithms, the computation of one or more linear algebra operations such as SpMV, vector additions, inner products or/and scalar-vector products are required. At the end of the algorithm, the vector result is copied from the device to the host. We implement a generic algorithm template and then only specialize operations according to the architecture (CPU or GPU). As a consequence, both CPU and GPU codes are similar, except that the GPU one performs operations on the graphics card by calling the associated kernel. At every conjugate gradient iteration, the preconditioning step is applied in order to improve the convergence rate of the method. The process is similar for all the other Krylov methods presented in this chapter.

As said above, the proposed algorithm implementations pay attention to minimizing the unnecessary data transfers between the CPU and the GPU, and they also take care of CPU management (copy, etc.). In the last chapter, we will discuss more complex preconditioning techniques, such as the domain decomposition methods. The ILU factorization preconditioning technique is effective as diagonal preconditioning as proved in [95]. However, for sparse matrices, the incomplete factorization step can be very expensive, as for the solver LU (see Chapter 3.2.2, Page 44).

3.6.3 Numerical results

Knowing the behavior and performance of linear algebra operations with matrices arising from the finite element discretization of different applications, we report numerical experiments of the presented Krylov methods, performed on both a CPU and a GPU device. We also evaluate their performance for different sparse storage formats. In this section, we study the same datasets as those presented in the numerical results of the sparse matrix-vector product. We follow the same plan as the SpMV part, *i.e.*, comparisons: sparse formats, the influence of the gridification, dataset matrices from a gravity equation, a realistic physical problem, complex number arithmetics, what about OpenCL?. All experimental results of the iterative Krylov methods presented here are obtained for the residual tolerance threshold 1×10^{-6} , an initial guess equal to zero. When no right-hand side is associated with the matrix, we consider a right-hand side vector filled with 1. The maximum number of iterations is fixed to 30000 for

all considered Krylov algorithms.

We consider the preconditioned Conjugate Gradient (P-CG) for symmetric positive-definite matrices, the preconditioned Generalized Conjugate Residual method (P-GCR) with $restart = 50$, the preconditioned Bi-Conjugate Gradient Conjugate Residual method (P-Bi-CGCR), the preconditioned Transpose-Free Quasi Minimal Residual (P-TFQMR), the preconditioned Bi-Conjugate Gradient Stabilized (P-Bi-CGSTAB) and its parametered version (P-Bi-CGSTAB(l)). The practical performance of the CG method is compared with the other methods.

Iterative Krylov comparisons: sparse formats

In this part, we report the numerical findings of the presented Krylov method, performed on a GPU device, with matrices described in Table 3.11.

Matrix	#iter.	Alinea	Alinea	Cusp	Alinea	Alinea
		CPU	GPU	GPU	GPU	GPU
		CSR	CSR	CSR	ELL	HYB
2cubes_sphere	24	0.13	0.04	0.04	0.04	0.04
cf2	2818	38.62	5.23	11.79	10.75	8.98
thermomech_dM	12	0.08	0.02	0.02	0.02	0.02
thermomech_TK	13226	42.55	14.32	18.7	18.32	15.32
qa8fm	29	0.11	0.04	0.04	0.04	0.03
Dubcova2	168	0.41	0.15	0.18	0.22	0.17
af_shell8	2815	93.3	21.2	18.00	13.84	13.56
finan512	15	0.03	0.01	0.02	0.02	0.02
thermal2	4453	198.49	32.16	26.45	30.8	27.46

TABLE 3.21: Execution time of the P-CG (seconds)

The execution times in seconds of the preconditioned conjugate gradient (P-CG) algorithm with the Cusp and Alinea libraries for the CSR format, are reported in columns three to five in Table 3.21. The times given in the table correspond to the global running time, which includes the communication time between the CPU and the GPU. As illustrated by this table with respect to the native Cusp iterative algorithm (column five), the in-house algorithm still outperforms the Cusp library for the CSR format applied to GPU (column four). The performance of the preconditioned Conjugate Gradient on GPU for ELL and HYB formats is illustrated in columns six and seven in Table 3.21. The comparison proves better performance for the HYB format.

Matrix	#iter	Alinea T.	Grid Alinea T.
			$\langle ntb, tw \rangle$
qa8fm	29	0.04	$\langle 64, 8 \rangle$
2cubes_sphere	24	0.04	$\langle 64, 8 \rangle$
thermomech_TK	17039	21.72	$\langle 64, 8 \rangle$
cf2	5938	12.05	$\langle 64, 8 \rangle$
thermomech_dM	12	0.03	$\langle 128, 8 \rangle$
thermal2	4552	32.27	$\langle 256, 8 \rangle$

TABLE 3.22: Running time of the P-CG (seconds) with auto-tuning

In Table 3.22 we report the running time for the preconditioned Conjugate Gradient on GPU by considering the best gridification given in Table 3.13 to ensure the effective sparse matrix-vector multiplication. Other Krylov methods of practical interest, such as P-TFQMR, P-Bi-CGCR, P-GCR(50) and P-Bi-CGSTAB, are presented in the following.

Matrix	P-TFQMR		P-Bi-CGCR		P-GCR, restart=50	
	#iter.	time (s)	#iter.	time (s)	#iter.	time (s)
2cubes_sphere	15	0.05	23	0.04	13	0.04
cf2	5284	19.53	4664	8.56	30000	176.38
thermomech_dM	6	0.03	12	0.02	23	0.09
thermomech_TK	90	0.19	14373	15.92	30000	123.75
qa8fm	25	0.06	28	0.03	50	0.22
Dubcova2	191	0.38	165	0.17	703	2.74
af_shell18	30000	448.78	2298	16.8	723	13.96
finan512	9	0.02	15	0.02	15	0.03
thermal2	112	1.68	4151	31.62	30000	441.37

TABLE 3.23: Number of iterations and execution times (in seconds) of P-GCR, P-Bi-CGCR and P-TFQMR Alinea algorithms for CSR format

Table 3.23 gives respectively the number of iterations (#iter.) and the execution time in seconds for the CSR format of preconditioned Transpose-Free Quasi Minimal Residual (P-TFQMR) in column 2-3, the preconditioned Bi-Conjugate Gradient Conjugate Residual method (P-Bi-CGCR) in column 4-5 and the preconditioned Generalized Conjugate Residual method (P-GCR) with a restart parameter equal to 50, in columns 6-7.

Matrix	CSR		ELL		HYB	
	#iter.	time (s)	#iter.	time (s)	#iter.	time (s)
2cubes_sphere	15	0.05	15	0.05	15	0.06
cf2	6285	24.98	5466	15.65	5026	17.69
thermomech_dM	6	0.02	6	0.02	6	0.02
thermomech_TK	30000	64.8	30000	61.73	30000	88.06
qa8fm	30000	80.37	30000	61.25	30000	61.41
Dubcova2	114	0.25	114	0.23	114	0.31
af_shell18	2540	41.32	3014	32.86	2842	31.09
finan512	10	0.02	15	0.05	10	0.03
thermal2	4006	54.29	30000	380.7	30000	346.16

TABLE 3.24: Execution time of the P-Bi-CGSTAB (seconds)

Table 3.24 presents the execution time in seconds for the preconditioned Bi-Conjugate Gradient Stabilized (P-Bi-CGSTAB) for CSR, ELL and HYB formats.

Table 3.25 illustrates the execution time in seconds for the CSR format for the preconditioned Bi-Conjugate Gradient Stabilized (L) (P-Bi-CGSTAB(L)) algorithm, where L denotes the stabilized parameter. The collected numerical results show that solving linear systems is less efficient than computing a simple SpMV because in the Krylov methods, several other operations take place. Anyway, as we can see in Table 3.25, when the parameter l varies for the P-Bi-CGSTAB(L) method, the number of iterations decreases and so does the computation time. It should be also noted that Alinea gives encouraging results for solving linear systems as seen from these tables.

Iterative Krylov: realistic gravity equation problem

Now we present numerical results of the iterative Krylov methods using gravity matrices, described in Table 3.14. Recall the workstation used, *Platform-1* consists of a workstation equipped with an Intel Core i7 920 2.67GHz processor, which has 8 cores composed of 4 physical cores and 4 logical cores, 5.8 GB RAM memory and two *NVIDIA GTX275* GPUs fitted with 895MB memory. Table 3.26 reports respectively the double precision execution times in seconds of the preconditioned BiConjugate Gradient Stabilized (Bi-

Matrix	#iter. time		#iter. time		#iter. time		#iter. time		#iter. time	
	L = 1		L = 2		L = 3		L = 4		L = 5	
2cubes_sphere	16	0.07	7	0.04	5	0.04	3	0.04	3	0.05
cfcd2	5950	19.44	3287	22.05	1705	17.68	1251	17.81	692	12.85
thermomech_dM	7	0.02	4	0.02	2	0.03	2	0.03	2	0.04
thermomech_TK	20	0.04	18	0.07	36	0.21	21	0.17	6	0.06
qa8fm	37	0.1	6	0.02	12	0.08	2	0.02	7	0.07
Dubcova2	115	0.22	57	0.22	39	0.21	29	0.23	25	0.25
af_shell18	2795	37.88	987	27.14	678	28.42	421	24.2	418	30.32
finan512	10	0.02	5	0.01	3	0.02	2	0.02	2	0.03
thermal2	4535	59.25	1969	53.03	1171	49.05	1070	62.23	775	58.22
	L = 6		L = 7		L = 8		L = 9			
2cubes_sphere	2	0.04	2	0.05	2	0.06	2	0.07		
cfcd2	590	13.4	587	16.07	450	14.5	457	17.08		
thermomech_dM	1	0.02	1	0.02	1	0.03	1	0.04		
thermomech_TK	8	0.1	5	0.08	3	0.06	8	0.17		
qa8fm	5	0.07	1	0.02	1	0.02	1	0.02		
Dubcova2	20	0.26	17	0.27	15	0.28	13	0.28		
af_shell18	330	29.26	254	26.7	244	29.85	233	32.71		
finan512	2	0.02	2	0.03	1	0.02	1	0.02		
thermal2	671	62.56	553	62.31	483	64.44	384	59.51		

TABLE 3.25: Execution time of the P-Bi-CGSTAB(l) (seconds) for CSR format

CGSTAB), the preconditioned BIConjugate Gradient Conjugate Residual (P-Bi-CGCR) and the preconditioned transpose free Quasi Minimal Residual (P-TFQMR) method. In Table 3.26, we collect the size of the problem in the first column, from column 2 to column 5 we give the number of iterations, the CPU time in seconds, the GPU time in seconds and the speed-ups CPU/GPU for the double precision execution times of P-Bi-CGSTAB. Those of P-Bi-CGCR and P-TFQMR are respectively given from column 6 to column 9 and from column 10 to 13. The speed-ups presented in these results are computed as follows: $ratio = CPU_time / GPU_time$.

h	P-Bi-CGSTAB				P-Bi-CGCR				P-TFQMR			
	#iter	CPU time (s)	GPU time (s)	CPU/GPU ratio	#iter	CPU time (s)	GPU time (s)	CPU/GPU ratio	#iter	CPU time (s)	GPU time (s)	CPU/GPU ratio
101,168	128	4.83	0.296	16	187	3.91	0.259	15	204	8.21	0.486	16
296,208	195	22.38	0.99	23	273	17.23	0.787	22	338	41.12	1.747	23
544,563	223	47.88	1.926	25	342	40.7	1.619	25	384	87.34	3.384	25
650,848	257	66.23	2.601	25	365	51.59	2.025	25	396	108.07	4.122	26
848,256	305	105.39	3.951	27	409	75.76	2.889	26	441	157.66	5.916	26
1,213,488	325	180.39	5.944	30	464	123.76	4.58	27	537	276.79	10.132	27
1,325,848	382	203.78	7.591	27	529	154.44	5.669	27	580	326.7	11.9	27

TABLE 3.26: Double precision CSR preconditioned Krylov methods: P-Bi-CGSTAB, P-Bi-CGCR, P-TFQMR

Table 3.26 clearly proves the effectiveness and robustness of the use of GPU compared to CPU for solving gravity equations with double precision number arithmetics. Despite GPUs good performance in terms of computing time for solving linear systems, as said and proved in the previous section, the energetic efficiency must not be neglected. We have performed the CG solver by setting the residual tolerance $\varepsilon = 10^{-6}$ for both Alinea and Cusp. Table 3.27 shows the number of iterations, the time in seconds, the electrical power in Watts, and the energy consumption by the GPU in Joules.

Figure 3.32(a) and Figure 3.32(b) respectively show the graphic data presentation of the execution times in seconds and the energy consumption in Joules, as in Table 3.27.

n	Alinea				Cusp		
	CPU time (s)	GPU time (s)	GPU P (W)	E (J)	GPU time (s)	GPU P (W)	E (J)
101168	203	0.313	100.201	31.363	0.471	110.271	51.888
296208	299	1.155	123.19	142.284	1.436	133.129	191.199
544563	371	2.46	133.564	328.568	3.003	140.889	423.019
650848	394	2.46	133.772	329.078	3.741	141.852	530.609
848256	435	4.336	137.843	597.618	5.25	145.828	765.595
1213488	491	6.872	141.169	970.115	8.396	146.889	1233.248
1325848	538	8.18	141.747	1159.492	9.945	150.403	1495.759

TABLE 3.27: Double precision CG CSR with gravitational potential matrices

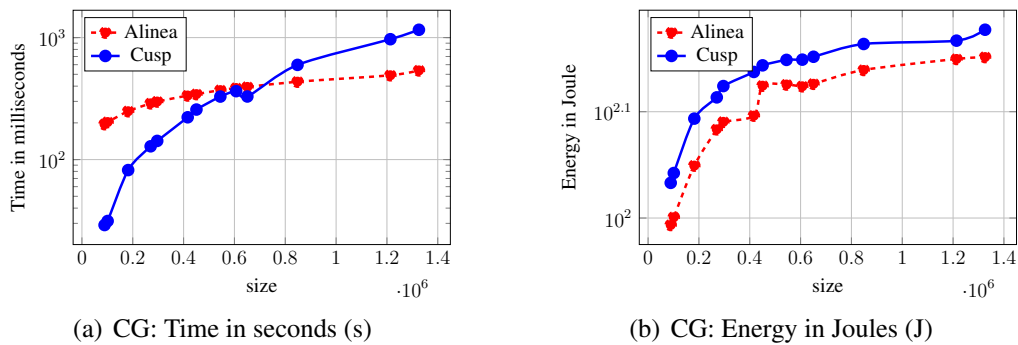


FIGURE 3.32: Conjugate Gradient (CG) CSR with gravitational potential matrices: Time in seconds (s) and Energy in Joules (J)

From Table 3.27, we can derive the dependency of the number of iterations on the size of problem,

$$\#iter = n^{0.35}, \quad (3.32)$$

The coefficient 0.35 is in agreement with the theoretical estimation of the condition number of the Poisson equation in three dimensions. This value leads to the complexity of the whole CG solver as $n^{1.35}$. We can conclude that the Alinea implementation is much more efficient than Cusp in terms of energy consumption, whereas both implementations, Alinea's and Cusp's follow the same numerical characteristic with the iteration number of the CG. The computational time of Alinea is 18% less than Cusp, and the consumed energy to get an approximate solution, 22% less, for the largest size of our experiment. The coefficient of prediction of the energy consumption also shows the greater efficiency of Alinea over Cusp.

	equation
Alinea Time (s)	$1.408 \times 10^{-2} + 4.287 \times 10^{-8} x^{1.35}$
Cusp Time (s)	$8.060 \times 10^{-2} + 5.231 \times 10^{-8} x^{1.35}$
Alinea Energy (J)	$-1.881 \times 10^1 + 6.156 \times 10^{-6} x^{1.35}$
Cusp Energy (J)	$-1.1773 \times 10^{-1} + 7.890 \times 10^{-6} x^{1.35}$

where x is the size of the problem.

TABLE 3.28: Predictions of computational time and energy consumption of CG solver with gravitational potential matrices

Iterative Krylov: complex number arithmetics

In both previous sections, we have demonstrated the effectiveness and robustness of linear algebra operations for complex number arithmetics. The aim of this experiments consists

in analyzing the impact of complex number arithmetics when solving of linear systems. So we present and evaluate numerical experiments performed on a set of acoustic matrices, listed in Table 3.18 (Audi3D) and Table 3.19 (Twingo3D), which arise from the modeling of acoustic phenomena within a car compartment. We give a comparison of CPU and GPU implementations for complex number arithmetics with double precision. In the following experiments, we consider a residual tolerance threshold of 1×10^{-9} , an initial guess of zero and a maximum number of iterations equal to 1000. Table 3.29 and Table 3.30 show respectively the speed-up obtained with our GPU implementation compared to the CPU implementation for Audi3D and Twingo3D: the first column presents the size of the mesh, the second column gives the number of iterations, the third and the fourth columns represents the CPU and GPU time in seconds, and the last column collects the speed-up.

problem	#iter	CPU time (s)	GPU time (s)	speed-up
<i>P-Bi-CGSTAB</i>				
Audi3D-1	21	0.01	0.030	0.33
Audi3D-2	53	0.24	0.106	2.26
Audi3D-3	94	4.01	0.703	5.71
Audi3D-4	183	85.70	9.209	9.31
<i>P-Bi-CGSTAB(8)</i>				
Audi3D-1	6	0.03	0.110	0.27
Audi3D-2	12	0.52	0.286	1.82
Audi3D-3	31	12.47	2.162	5.77
Audi3D-4	70	266.26	30.100	8.85
<i>P-TFQMR</i>				
Audi3D-1	24	0.02	0.040	0.50
Audi3D-2	52	0.27	0.113	2.40
Audi3D-3	99	4.71	0.755	6.24
Audi3D-4	214	102.17	10.786	9.47

TABLE 3.29: Audi3D - Speed-up of the CSR acoustic solver (complex number arithmetics with double precision): P-Bi-CGSTAB, P-Bi-CGSTAB(8) and P-TFQMR

problem	#iter	CPU time (s)	GPU time (s)	speed-up
<i>P-Bi-CGSTAB</i>				
Twingo3D-0	563	1.85	1.008	1.84
Twingo3D-1	1000	29.45	5.730	5.14
Twingo3D-2	1000	295.66	37.670	7.85
<i>P-Bi-CGSTAB(8)</i>				
Twingo3D-0	1000	31.2	20.970	1.49
Twingo3D-1	1000	273.81	54.630	5.01
Twingo3D-2	1000	2559.67	324.500	7.89
<i>P-TFQMR</i>				
Twingo3D-0	366	1.34	0.626	2.14
Twingo3D-1	954	30.4	5.438	5.59
Twingo3D-2	1000	318.93	38.090	8.37

TABLE 3.30: Twingo3D - Speed-up of the CSR acoustic solver (complex number arithmetics with double precision): P-Bi-CGSTAB, P-Bi-CGSTAB(8) and P-TFQMR

Figure 3.33 gives an illustration of the solution on the mesh, which consists of the real part (see Figure 3.33(a)) and the imaginary part (see Figure 3.33(b)) of the solution plotted on the mesh given in the previous chapter (see Figure 3.28, Page 77).

When we refine the mesh, *i.e.*, the size of the problem increases, the results become more accurate since more details in the car compartment are taken into account. As we can see in

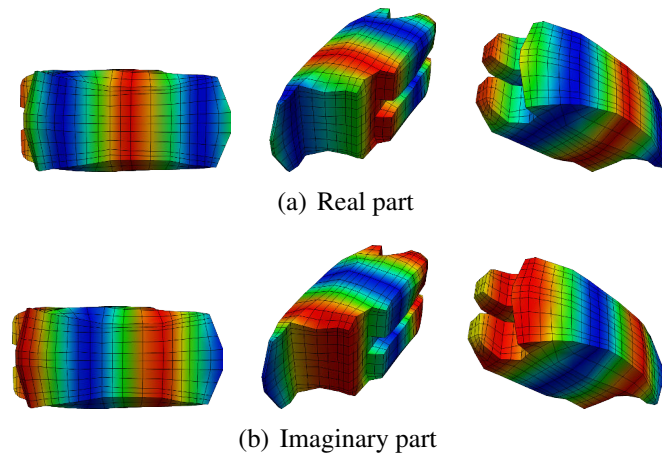


FIGURE 3.33: Illustration of the obtained solution for the Audi 3D, with $h = 0.033289$ (see Figure 3.28, Page 77)

Table 3.29 and Table 3.30 (also in Table 3.26 Table 3.27), for all our implementations the speed-up increases when the size of the problem increases. Nevertheless, when the size of the problem becomes too large for GPU memory, which is often very limited on most GPUs, other methods must be considered. Domain decomposition methods [17] [18] [97] [98] based on iterative algorithms are an alternative. Such methods have encountered strong success for the solution of coercive elliptic problems [159] [160] and are easy to implement on parallel computers. Using absorbing boundary transmission conditions on the interface between the sub-domains [99] is a key point to obtain a fast convergence of the domain decomposition algorithm, such as the Schwarz algorithm [161] [162] [163]. First works presented in references [164] [165] [166] consider Robin-type absorbing boundary transmission conditions on the interface, which have then been optimized with a continuous approach in [167], [100] [104] [168]. Further works have considered a discrete optimization of the interface conditions as first introduced in [105], and then in [103] [169] [170] [171] [172]. In the next chapter, we present how we have proceeded to efficiently implement domain decomposition methods on a multi-core-GPU system.

Iterative Krylov: what about OpenCL?

In the following, we collect numerical experiments of CG algorithms with OpenCL. We use the O -ring matrices associated with the mesh presented in Figure 3.30(b).

Figure 3.34(a) reports the execution times in seconds of the CG algorithm using O -ring matrices on CPU and GPU with OpenCL in single precision, and GPU with CUDA in both single and double precision. The results clearly show that GPU is more effective than CPU for a large size. The speed-up CPU/GPU is equal to 9 for the matrix of size 402, 445. The gain is even more important when the size of the problem increases. The zoom on the results for small-size matrices, given in Figure 3.34(b) shows that CPU is more efficient than GPU for small dimensions, due to the costly data transfer from the host to the device for the GPU version. When size increases the transfers are overloaded by the computations. For equivalent accuracy for the matrix of size 1, 607, 352, the CG method respectively converges in 630 and 629 iterations for GPU with CUDA and GPU with OpenCL in single precision. On the other side, it converges in 621 iterations for GPU with double CUDA. Single precision is faster than double precision on GPU in terms of computing time, but double precision is faster than single

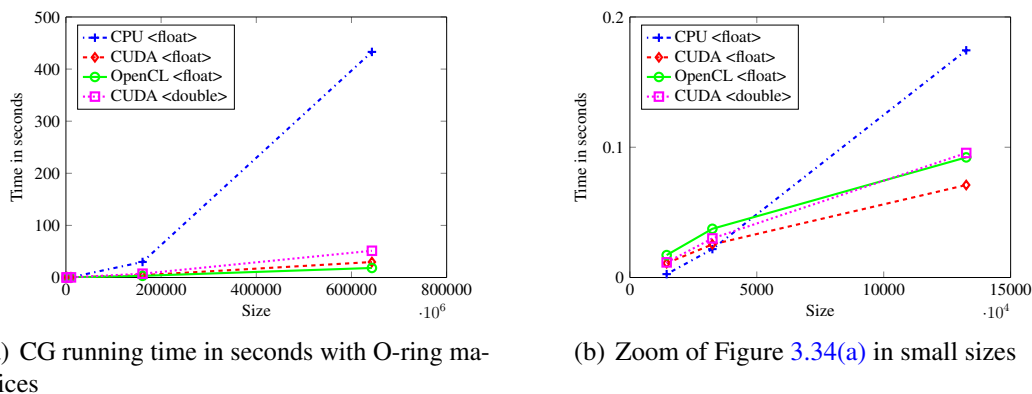


FIGURE 3.34: CG execution times in seconds with O-ring matrices and zoom at small sizes

precision in terms of the convergence of the algorithm. We also highlight the good results obtained by OpenCL over CUDA.

3.7 Alinea: an hybrid CPU/GPU library

In this section, we present an overview of the library we have implemented, Alinea, effective for computing advanced linear algebra and solving linear systems on both CPU and GPU.

3.7.1 Introduction

Since computers are evolving with an increasing number of processors (hundreds of thousands in high-end machines) and exascale computers are expected to have highly hierarchical architectures with nodes composed by multiple core processors, *i.e.*, CPUs and accelerators, *e.g.*, GPUs, the algorithms must be adapted to the evolution of hardware components and be optimized according to these features. This is the reason why we have decided to implement our own library in order to better optimize the algorithms according to the characteristics of hardware and the properties of the problem to be solved. Alinea stands for **A**dvanced **L**INEar **A**lgebra. Alinea is targeted as a scalable software for proposing effective linear algebra operations on both CPU and GPU platforms. It includes numerous algorithms for solving linear systems, with different matrix storage formats, with real and complex arithmetics in single and double precision, on both CPU and GPU devices. Alinea is devoted to simplifying the development of engineering and science problems on CPU and GPU by discharging most of the difficulties encountered when using these architectures, particularly with GPU. Alinea investigates and seeks the best way to effectively implement linear algebra operations and solver algorithms on both CPU and GPU. It also allows to write the same code for the CPU and GPU versions of an algorithm easily. The library is implemented in C++ language and proposes a simple C API interface. The main features of Alinea described in Table 3.31 which are applied to real and complex arithmetic numbers, simple and double precision.

We are dealing with huge matrix systems and therefore parallel computing becomes an important issue for optimization. The GPU version with CUDA or OpenCL, written in C++, has the task to load the methods into the GPU, to control the data transfer and to manage the memory.

type	function	example	reference
vector-vector	addition & multiplication	$w = u + \alpha v$	BLAS 1
vector-vector	scalar product	$p = \langle u, v \rangle$	BLAS 1
vector	norm	$q = \ u\ $	BLAS 1
matrix-vector	sparse product	$y = A * x$	BLAS 2
matrix-matrix	matrix product	$C = A * B$	BLAS 3
solver	direct	$x = LU(A, b)$	
solver	iterative	$x = CG(A, b)$	

TABLE 3.31: Alinea (*Advanced LINEar Algebra*) main levels

A summary of the general specifications of Alinea are described and reported in Table 3.31(a).

(a) Alinea specifications		(b) Diagram of Alinea levels
Name	Alinea	<pre> graph TD A[Algorithm template <T,U>] --> B[CPU] A --> C[GPU] B --> D[device chosen] C --> D D --> E[BLAS 1] D --> F[BLAS 2] D --> G[BLAS 3] D --> H[Solver] H --> I[direct] H --> J[iterative] J --> K[none preconditioner] J --> L[preconditioner] </pre>
Operating System	Linux, Windows	
Architectures	32 bits, 64 bits	
Language	C++	
Compiler	GNU/g++, clang++	
Precision	single (float), double	
Type	real, complex	
Device	CPU, GPU	
GPU language	CUDA, OpenCL	
BLAS Level	1, 2, 3	
Solvers	direct, iterative	
Tested compiler	gcc/g++, clang/clang++	

TABLE 3.32: Alinea specifications and diagram of the different levels

The diagram drawn in the figure of Table 3.31(b) presents the main levels of our template library. This concept allows to develop a much more modular code, easier to use for users or developers. Indeed, each block is independent and the use of templates allows a very intuitive use of the library, regardless of the architecture or the data type selected (real, complex, simple precision or double-precision). Therefore, the available C++ methods are identical. The template is designed to $\langle T, U \rangle$, where T is the type of value, *e.g.*, double or `std::complex<double>` and U the type of index, *e.g.*, int or unsigned int.

3.7.2 BLAS level-1

In this section, we describe the block that implements BLAS level-one functions on both CPU and GPU (using CUDA and OpenCL). Figure 3.35 reports the routines and functions of

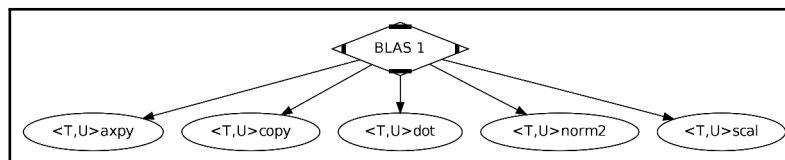


FIGURE 3.35: Diagram of the BLAS level-1

BLAS level-1, including *vector-vector* and *vector* operations. To use the vector module of the library, the lines of Listing 3.4 are mandatory.

```

1 // -- include part of library : Vector
2 // -- CPU and or GPU
3 #include <Vector.hpp>
4 //

```

LISTING 3.4: Vector usage

```

1 // -- include part of library : GpuDevice
2 #include <GpuDevice.hpp>
3 // -- if complex with GPU device
4 #include <GpuDevice/complex.hpp>

```

LISTING 3.5: GpuDevice usage

Listing 3.6 and Listing 3.7 present basic examples of vector declaration and usage on CPU and GPU respectively.

```

1 // -- allocate a vector on CPU
2 Vector<double,int> x_cpu( 4 );
3
4 // -- initialize the CPU vector
5 x_cpu(0) = 2; // x_cpu[0] = 2;
6 x_cpu(1) = 4;
7 x_cpu(2) = -2;
8 x_cpu(3) = 3;
9
10 // -- print x
11 x_cpu.WriteToStdout();

```

LISTING 3.6: CPU real vector storage

```

1 // -- include GPU environment
2 #include <GpuDevice.hpp>
3 // -- device environment (in-house device)
4 GpuDevice::Cuda::Kernel dev
5 = GpuDevice::Cuda::MakeDevice();
6 // -- initialize the card number 0
7 dev.Init( 0 ); // device #0
8 // -- declare a vector on GPU
9 Vector<double,int> x_gpu( 4, &dev );
10 // -- finalize GPU
11 dev.Finalize( );

```

LISTING 3.7: GPU real vector storage

Line 2 of Listing 3.6 declares a double precision (double) vector of size 4 on CPU memory, with an integer index (int). On the other hand, the GPU version described in Listing 3.7 requires the declaration of the type of device used first and its initialization. In Line 5 of Listing 3.7, the *CUDA* device is used. To use *OpenCL*, it suffices to replace “Cuda” by “OpenCL” at Line 5 of Listing 3.7.

Listing 3.8 and Listing 3.9 give corresponding versions of Listing 3.6 and Listing 3.7 for complex number arithmetics. These examples show the easy way to use templates.

```

1 #include <complex>
2 typedef std::complex<double> T;
3 // -- allocate a vector on CPU
4 Vector<T,int> x_cpu( 3 );
5
6 // -- initialize the CPU vector
7 x_cpu(0) = T(2,1);
8 x_cpu(1) = T(4,-1);
9 x_cpu(2) = T(-2,3);
10 x_cpu(3) = T(1,3);
11
12 // -- print x
13 x_cpu.WriteToStdout();

```

LISTING 3.8: CPU complex vector storage

```

1 // -- include GPU environment
2 #include <GpuDevice.hpp>
3 #include <GpuDevice/complex.hpp>
4 typedef stdmrg::complex<double> T;
5 // -- device environment (in-house device)
6 GpuDevice::Cuda::Kernel dev
7 = GpuDevice::Cuda::MakeDevice();
8 // -- initialize the card number 0
9 dev.Init( 0 ); // device #0
10 // -- declare a vector on GPU
11 Vector<T,int> x_gpu( 3, &dev );
12 // -- finalize GPU
13 dev.Finalize( );

```

LISTING 3.9: GPU complex vector storage

As we can see in Listing 3.9, `stdmrg::complex` is required when we deal with GPU. Listing 3.10 shows a complete example of vector usage on GPU. The code also illustrates CPU usage, in fact the vector is first created (Line 2 of Listing 3.10) and filled (Line 4 of Listing 3.10) on CPU, and then copied to GPU (Line 18 of Listing 3.10). After computation by the GPU, the result is copied back to CPU. Listing 3.11 presents the same procedure using *operators*. The example is given for real double precision (Line 2 of Listing 3.10) but it is also valid for simple precision by replacing `double` by `float`. Similarly, for the complex version, we replace `double` (`float`) by `stdmrg::complex<double>` (`stdmrg::complex<float>`). Moreover, CPU complex data can use standard `complex`: `std::complex<double>` (`std::complex<float>`).

```

1 // -- allocate a vector on CPU
2 Vector<double,int> x_cpu( 2 );
3 // -- initialize the CPU vector
4 x_cpu(0) = 2;
5 x_cpu(1) = 4;
6 // -- device environment (in-house device)
7 GpuDevice::Cuda::Kernel dev
8   = GpuDevice::Cuda::MakeDevice( );
9 // -- initialize the card number 1
10 dev.Init( 0 ); // device #1
11 // -- declare a vector on GPU
12 Vector<double,int> x_gpu( 3, &dev );
13 // -- copy from CPU to GPU
14 CopyTo( x_gpu, x_cpu );
15 // -- declare a vector on GPU
16 Vector<double,int> y_gpu( 2, &dev );
17 // -- copy GPU to GPU
18 CopyTo( y_gpu, x_gpu );
19 // -- compute y_gpu=2.0*x_gpu+y_gpu
20 Blap1::Saxpy( 2.0, x_gpu, y_gpu );
21 // -- compute the dot product of y
22 double d = Blap1::Dot( y_gpu, y_gpu );
23 // -- finalize GPU
24 dev.Finalize( );

```

LISTING 3.10: GPU usage, BLAS1 operations

```

1 // -- allocate a vector on CPU
2 Vector<double,int> x_cpu( 2 );
3 // -- initialize the CPU vector
4 x_cpu(0) = 2;
5 x_cpu(1) = 4;
6 // -- device environment (in-house device)
7 GpuDevice::Cuda::Kernel dev
8   = GpuDevice::Cuda::MakeDevice( );
9 // -- initialize the card number 1
10 dev.Init( 0 ); // device #1
11 // -- declare a vector on GPU
12 Vector<double,int> x_gpu( 3, &dev );
13 // -- copy from CPU to GPU
14 x_gpu = x_cpu;
15 // -- declare a vector on GPU
16 Vector<double,int> y_gpu( 2, &dev );
17 // -- copy GPU to GPU
18 y_gpu = x_gpu;
19 // -- compute y_gpu = 2.0 * x_gpu + y_gpu
20 y_gpu = 2.0 * x_gpu + y_gpu;
21 // -- compute the dot product of y
22 double d = Blap1::Dot( y_gpu, y_gpu );
23 // -- finalize GPU
24 dev.Finalize( );

```

LISTING 3.11: GPU usage, BLAS1 (Operator)

3.7.3 Matrix storage format

Alinea implements several conventional formats of matrix storage such as the Compressed Sparse Row (CSR) format that is probably the most popular, the COOrdinate (COO) format that is probably the most trivial, the ELLPACK (ELL) format, the HYBrid (HYB) format, the DIAgonal (DIA) format, the Symmetric Skyline format (SSK) and the Mapped format that uses *map* structure. In this section, we describe the block that implements BLAS level-one functions on both CPU and GPU (using CUDA and OpenCL). Figure 3.36 illustrates the main

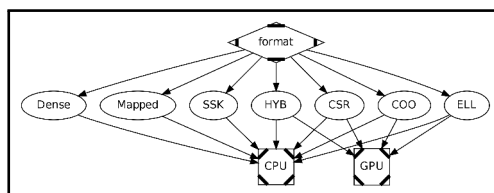


FIGURE 3.36: Diagram of the available matrix formats

matrix storage formats available in Alinea. The code given in Listing 3.12 is required when dealing with matrices.

```

1 // -- include part of library : Matrix<T,U>
2 #include <Matrix.hpp>

```

LISTING 3.12: Matrix usage

Listing 3.13 shows how `Matrix<T,U>` is declared on CPU and on GPU in Listing 3.14. Line 9 of Listing 3.13 (Line 7 of Listing 3.14 for GPU) describes an instantiation of a matrix with a given format name “*fmt*”: *coo*, *csr*, *etc.*. The way of declaring a matrix on GPU is similar to that of a vector.

Alinea provides various possibilities of allocation and matrix filling. Next, we present direct and explicit allocations. Direct allocation is the most trivial and intuitive mode. Listing 3.15 and Listing 3.16 give illustrations for CPU and GPU usage respectively. The comments given in these listings offer more explanations.

```

1
2
3
4
5
6 // -- declare an unknown matrix on CPU
7 Matrix<double,int> A;
8 // -- declare a matrix on CPU
9 Matrix<double,int> B( "fmt" );
10 // where fmt, the matrix format:
11 // dense1d, dense2d, mapped, ssk,
12 // coo, csr, ell, hyb, ...
13 //

```

LISTING 3.13: Matrix constructors (CPU)

```

1 // -- device environment (in-house device)
2 GpuDevice::Cuda::Kernel dev
3 = GpuDevice::Cuda::MakeDevice();
4 // -- initialize the card number 0
5 dev.Init( 0 ); // device #0
6 // -- declare an unknown matrix on GPU
7 Matrix<double,int> A( &dev );
8 // -- declare a matrix on GPU
9 Matrix<double,int> B( "fmt", &dev );
10 // where fmt, the matrix format:
11 // coo, csr, ell, hyb, ...
12 // -- finalize GPU
13 dev.Finalize( );

```

LISTING 3.14: Matrix constructors (GPU)

The second argument of ReadFromFile function at Line 15 of Listing 3.15 (Line 9 of Listing 3.16) corresponds to the format of the matrix. Alinea proposes a matrix market [173] and csv (Comma-separated values) format. Both ascii and binary modes are available. We have implemented a library, called *MatrixMarket*, which manages all the input/output of the “matrix market” matrices.

```

1
2
3 // -- declare a CPU coo matrix
4 Matrix<double,int> m_coo( "coo" );
5 // -- allocate the coo matrix
6 m_coo.Allocate( 3, 3 );
7 // -- fill out the coo matrix
8 m_coo(0,0) = 5.0;
9 m_coo(2,2) = 6.0;
10 m_coo(1,2) = -1.0;
11 m_coo(2,1) = 4.0;
12 // -- declare a CPU csr matrix
13 Matrix<double,int> m_csr( "csr" );
14 // -- read csr matrix from mtx file
15 m_csr.ReadFromFile( "f_name", "mtx" );

```

LISTING 3.15: Matrix direct allocation (CPU)

```

1 // -- device environment (in-house device)
2 GpuDevice::Cuda::Kernel dev
3 = GpuDevice::Cuda::MakeDevice();
4 // -- initialize the card number 0
5 dev.Init( 0 ); // device #0
6 // -- declare a CPU csr matrix
7 Matrix<double,int> m_csr( "csr" );
8 // -- read csr matrix from mtx file
9 m_csr.ReadFromFile( "f_name", "mtx" );
10 // -- declare a GPU csr matrix
11 Matrix<double,int> m_csr_gpu( "csr", &dev );
12 // -- copy from csr CPU to csr GPU
13 CopyTo( m_csr_gpu, m_csr );
14 // -- finalize GPU
15 dev.Finalize( );

```

LISTING 3.16: Matrix direct allocation (GPU)

As its name suggests, the explicit allocation consists in constructing a matrix by explicitly calling the constructor function of a given matrix. Examples for CPU and GPU are both given in Listing 3.17 and Listing 3.18.

```

1
2
3
4
5
6 // -- declare an unknown matrix
7 Matrix<T,U> A;
8 // -- set coo matrix from "mtx" file "f_name"
9 A.SetMatrix(
10   &NewMatrixCoo<T,U>("f_name","mtx"));
11 //
12 //

```

LISTING 3.17: Matrix explicit allocation (CPU)

```

1 // -- device environment (in-house device)
2 GpuDevice::Cuda::Kernel dev
3   = GpuDevice::Cuda::MakeDevice();
4 // -- initialize the card number 0
5 dev.Init( 1 ); // device #1
6 // -- declare an unknown matrix
7 Matrix<T,U> A;
8 // -- set coo matrix ("bmtx" for binary mtx)
9 A.SetMatrix(
10   &NewMatrixCoo<T,U>(&dev,"f_name","mtx"));
11 // -- finalize GPU
12 dev.Finalize( );

```

LISTING 3.18: Matrix explicit allocation (GPU)

When we focus on GPU Computing, data are first copied from CPU to GPU before computation, and then copied back from GPU to CPU in order to get results. An example of usage of copy routines is described in Listing 3.19. Furthermore, depending on methods, a storage format may be preferred. This is why Alinea has conversion functions between all formats, ConvertTo that is the global function defined by the library. Listing 3.20 gives an example of use.

```

1 // -- declare a CPU csr matrix
2 Matrix<double,int> m_csr( "csr" );
3 // -- read csr matrix from mtx file
4 m_csr.ReadFromFile( "f_name", "mtx" );
5 // -- declare a GPU csr matrix
6 Matrix<double,int> m_csr_gpu( "csr" );
7 // -- copy from csr CPU to csr GPU
8 CopyTo(m_csr_gpu, m_csr);

```

LISTING 3.19: Matrix copy (CPU↔GPU)

```

1 // -- declare a CPU csr matrix
2 Matrix<double,int> m_csr( "csr" );
3 // -- read csr matrix from mtx file
4 m_csr.ReadFromFile( "f_name", "mtx" );
5 // -- declare a CPU coo matrix
6 Matrix<double,int> m_coo( "coo" );
7 // -- convert csr to coo
8 ConvertTo( m_coo, m_csr );

```

LISTING 3.20: Matrix conversion (only CPU)

3.7.4 BLAS level-2 and level-3

The matrix-vector product is the most time-consuming operation. This operation is classified in the BLAS level-2 category. In this category we find operations such as the transpose matrix-vector product and the adjoint matrix-vector product. In addition, some algorithms

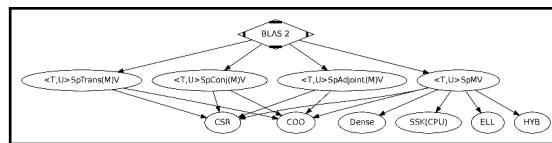


FIGURE 3.37: Diagram of the BLAS level-2

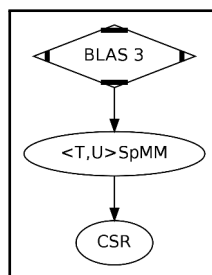


FIGURE 3.38: Diagram of the BLAS level-3

require the computation of the matrix-matrix product. Alinea provides such routines that manipulate matrices and perform operations between matrices. They are classified as BLAS level-3. Figure 3.37 and Figure 3.38 illustrate respectively the BLAS level-2 and level-3 modules. Listing 3.21 and Listing 3.22 present a complete example of CPU and GPU use. Listing 3.23 and Listing 3.24 demonstrate the use of BLAS level-3 functions on CPU and GPU respectively.

```

1
2
3
4
5
6 // -- allocate and initialize x vector on CPU
7 Vector<double,int> x( 3 );
8 // -- initialize the vector
9 for ( int i = 0; i < 3; i++ ) x(i) = i;
10
11
12
13
14 // -- declare a CPU csr matrix
15 Matrix<double,int> m_csr( "csr" );
16 // -- read csr matrix from mtx file
17 m_csr.ReadFromFile( "f_name", "mtx" );
18
19
20
21
22 // -- declare a vector on CPU
23 Vector<double,int> y( 3 );
24 // -- compute y = m_csr * x
25 Blap2::MatrixVectorProduct( y, m_csr, x );
26 //
27 //

```

LISTING 3.21: BLAS 2: computation on CPU

```

1 // -- device environment (in-house device)
2 GpuDevice::Cuda::Kernel dev
3 = GpuDevice::Cuda::MakeDevice();
4 // -- initialize with the powerful card
5 dev.Init ( );
6 // -- allocate vector on CPU
7 Vector<double,int> x_cpu( 3 );
8 // -- initialize the CPU vector
9 for ( int i = 0; i < 3; i++ ) x_cpu(i) = i;
10 // -- declare a vector on GPU
11 Vector<double,int> x_gpu( 3, &dev );
12 // -- copy vector from CPU to GPU
13 CopyTo( x_gpu, x_cpu );
14 // -- declare a CPU csr matrix
15 Matrix<double,int> m_csr( "csr" );
16 // -- read csr matrix from mtx file
17 m_csr.ReadFromFile( "f_name", "mtx" );
18 // -- declare a GPU csr matrix
19 Matrix<double,int> m_csr_gpu( "csr", &dev );
20 // -- copy from csr CPU to csr GPU
21 CopyTo( m_csr_gpu, m_csr );
22 // -- declare a vector on GPU
23 Vector<double,int> y_gpu( 3, &dev );
24 // -- compute y_gpu = m_csr * x_gpu
25 y_gpu = m_csr_gpu * x_cpu;
26 // -- finalize GPU
27 dev.Finalize ( );

```

LISTING 3.22: BLAS 2: computation on GPU

```

1
2
3
4
5
6 // -- declare CPU coo matrices
7 Matrix<double,int> m_coo( "coo" );
8 Matrix<double,int> m_cool( "cool" );
9 // -- read coo matrix from mtx file
10 m_coo.ReadFromFile( "f_name", "mtx" );
11 // -- copy between matrices
12 m_cool.Copy( m_coo );
13 // -- m_cool = m_coo * m_cool;
14 Blap3::MatrixHadProduct(m_coo, m_coo, m_cool);
15 //
16 //

```

LISTING 3.23: BLAS 3: simple usage on CPU

```

1 // -- device environment (in-house device)
2 GpuDevice::Cuda::Kernel dev
3 = GpuDevice::Cuda::MakeDevice();
4 // -- initialize with the powerful card
5 dev.Init ( );
6 // -- declare GPU csr matrices
7 Matrix<double,int> m_csr( "csr", &dev );
8 Matrix<double,int> m_csr1( "csr", &dev );
9 // -- read csr matrix from mtx file
10 m_csr.ReadFromFile( "f_name", "mtx" );
11 // -- copy between matrices
12 m_csr1.Copy( m_csr );
13 // -- m_csr = m_csr * m_csr1
14 Blap3::MatrixHadProduct(m_csr, m_csr, m_csr1);
15 // -- finalize GPU
16 dev.Finalize ( );

```

LISTING 3.24: BLAS 3: simple usage on GPU

3.7.5 Solvers

Figure 3.39 details the main block of the solver module. Alinea implements several direct solvers including the LU for CSR matrices, LDLt and Cholesky for symmetric skyline matrices, etc. The LDLt solver for instance consists of two main routines, which are

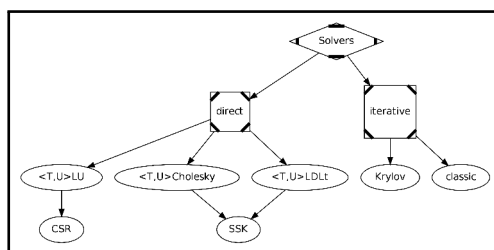


FIGURE 3.39: Diagram of the solvers

DirectSolver :: LDLt performing the LDLt decomposition and DirectSolver :: ForwardBackwardLDLt solving the triangular systems with forward/backward substitutions. The code given in Listing 3.25 and Listing 3.26 correspond respectively to “Solver<T,U>” usage and “Preconditioner <T,U>” usage, with the same rule as the other containers.

```

1 // -- include part of library : Solver
2 // -- CPU and GPU (direct and iterative)
3 #include <Solver.hpp>
4 // -- if only direct solvers
5 #include <DirectSolver.hpp>
  
```

LISTING 3.25: Solver usage

```

5 // -- include part of library : Preconditioner
6 // -- CPU and GPU
7 //
8 #include <Preconditioner.hpp>
9 //
  
```

LISTING 3.26: Preconditioner usage

Listing 3.27 presents the LDLt solver when the matrix is stored in CSR format. For the Cholesky solver, it suffices to change *LDLt* by *Cholesky* in Listing 3.27.

```

1 // -- declare a CPU csr matrix
2 Matrix<double,int> m_csr( "csr" );
3 // -- read csr matrix from mtx file
4 m_csr.ReadFromFile( "f_name", "mtx" );
5 // -- declare a CPU ssk matrix
6 Matrix<double,int> m_ssk( "ssk" );
7 // -- convert csr to ssk
8 ConvertTo( m_ssk, m_csr );
9 // rhs vector
10 Vector<T,U> v_rhs( m_ssk.GetNumRows() );
11 // -- assign v_rhs to 1
12 v_rhs = 1.;
13 // -- ldl decomposition
14 Matrix<T,U> ldl_m( "ssk" );
15 DirectSolver::LDLt( ldl_m, m_ssk );
16 // solution vector
17 Vector<T,U> v_x( m_ssk.GetNumRows() );
18 // forward backward resolution
19 DirectSolver::ForwardBackwardLDLt( v_x, ldl_m, v_rhs );
  
```

LISTING 3.27: Direct solver simple usage (CPU)

The diagrams of Figure 3.40 and Figure 3.41 describe the iterative Krylov methods and the diagrams Figure 3.42 and Figure 3.43 show iterative classical solvers.

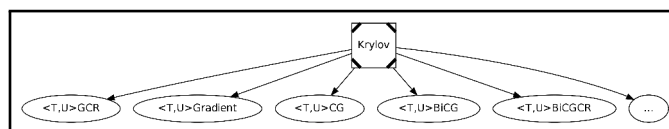


FIGURE 3.40: Diagram of the iterative Krylov Solvers (1)

Knowing how to handle the linear algebra operations easily, we now present how simple the use of iterative Krylov methods proposed by Alinea is. One of the most expensive operations

of GPU versions is the data transfer (sending and receiving) between the host (CPU) and the device (GPU). Alinea pay attention to sending the input data from the host (CPU) to the device

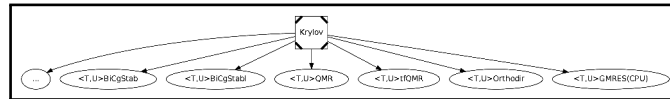


FIGURE 3.41: Diagram of iterative Krylov Solvers (2)

(GPU) once before starting the iterative routine. The iterative Krylov algorithm is executed on the host (CPU), but all computing steps (matrix-vector multiplication, ...) are performed on the device (GPU). At the end of the algorithm, the solution vector is copied from the device to the

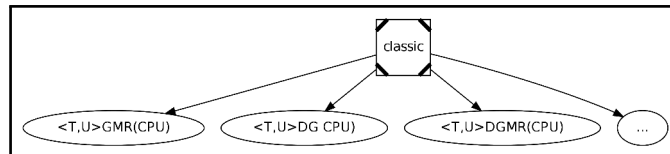


FIGURE 3.42: Diagram of the stationary iterative Solvers (3)

host. Alinea proposes the main Krylov methods on both CPU and GPU. Listing 3.28 gives an

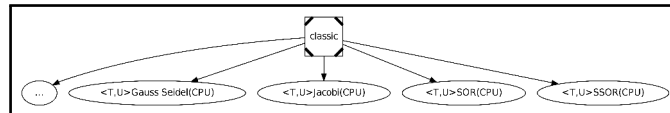


FIGURE 3.43: Diagram of the stationary iterative Solvers (4)

example of usage of iterative solvers. This example is also valid for the LU direct solver.

<pre> 1 2 3 4 5 6 // -- declare a csr matrix on CPU 7 Matrix<double,int> m_csr("csr"); 8 // -- read csr matrix from mtx file 9 m_csr.ReadFromFile("f_name", "mtx"); 10 11 // -- declare a vector on CPU 12 Vector<double,int> v_rhs; 13 // -- read vector from mtx file 14 v_rhs.ReadFromFile("f_v_name", "mtx"); 15 16 17 18 19 20 21 22 // -- declare a solver 23 Solver<double,int> solver; 24 // -- solver properties 25 const int m_it = 100; const double r_thres = 1e-6; 26 solver.SetSolver(&MakeSolverCg<T,U>(m_it, r_thres)); 27 // -- m_csr: matrix of the solver 28 solver.Setup(m_csr); 29 // -- set diagonal preconditioner 30 solver.SetPrecond(MakePrecondDiag<T,U>()); 31 // -- solve m_csr * v_x = v_rhs 32 Vector<double,int> v_x(v_rhs.GetSize()); 33 // -- solve 34 solver.Solve(v_x, v_rhs); 35 // 36 // </pre>	<pre> 1 // -- device environment (in-house device) 2 GpuDevice::Cuda::Kernel dev 3 = GpuDevice::Cuda::MakeDevice(); 4 // -- initialize with the powerful card 5 dev.Init (); 6 // -- declare a csr matrix on CPU 7 Matrix<double,int> m_csr("csr"); 8 // -- read csr matrix from mtx file 9 m_csr.ReadFromFile("f_name", "mtx"); 10 // -- declare a csr matrix on GPU 11 Matrix<double,int> m_csr_gpu("csr", &dev); 12 // -- copy matrix from CPU to GPU 13 CopyTo(m_csr_gpu, m_csr); 14 // -- declare a vector on CPU 15 Vector<double,int> v_rhs; 16 // -- read vector from mtx file 17 v_rhs.ReadFromFile("f_v_name", "mtx"); 18 // -- declare a vector on GPU 19 Vector<double,int> v_rhs_gpu(&dev); 20 // -- copy vector from CPU to GPU 21 CopyTo(v_rhs_gpu, v_rhs); 22 // -- declare a solver 23 Solver<double,int> solver; 24 // -- solver properties 25 const int m_it = 100; const double r_thres = 1e-6; 26 solver.SetSolver(&MakeSolverCg<T,U>(m_it, r_thres)); 27 // -- m_csr_gpu: matrix of the solver 28 solver.Setup(m_csr_gpu, m_csr); 29 // -- set diagonal preconditioner 30 solver.SetPrecond(MakePrecondDiag<T,U>(&dev)); 31 // -- solve m_csr * v_x = v_rhs 32 Vector<double,int> v_x(v_rhs.GetSize(), v_rhs.GetDev()); 33 // -- solve 34 solver.Solve(v_x, v_rhs); 35 // -- finalize GPU 36 dev.Finalize (); </pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

LISTING 3.28: CPU iterative solvers usage

LISTING 3.29: CPU iterative solvers usage

Solvers differ on the input of the function SetSolver at Line 26 of Listing 3.28 (Line 26 of Listing 3.29 for GPU). The given example consists of a Conjugate Gradient (Cg) solver. If the preconditioner is applied, Line 30 of Listing 3.28 (Line 30 of Listing 3.29 for GPU) if necessary. For the GPU version, when the preconditioner is required, the setup step, Line 28 of Listing 3.28 (Line 28 of Listing 3.29 for GPU), must also give the CPU matrix in the second parameter. In fact, the inverse of the diagonal is performed once on CPU, and then copied to GPU. After that, the element wise product is performed on GPU.

3.7.6 Conclusion

In this section, we have presented Alinea, which stands for *Advanced LINEar Algebra*, a library well suited to hybrid CPU/GPU computing. Alinea is targeted as a scalable software for proposing effective linear algebra operations on both CPU and GPU platforms, using CUDA and OpenCL languages. Alinea proposes a new vision of GPU computing that allows to harness the power of GPUs easily, without the programming complexity of classical APIs such as CUDA and OpenCL. This library, which I developed during my thesis, includes basic and advanced linear algebra operations, direct solvers and iterative solvers both on CPU and GPU. Numerical results presented in this thesis confirm the robustness of these hybrid algorithms that should pave to exascale hybrid methods. Alinea proves to be a well-balanced compromise in terms of programming and efficiency regarding computing performance and

energy consumption.

Alinea: An Advanced Linear Algebra Library for Massively Parallel Computations on Graphics Processing Units.

3.8 Conclusion

In this chapter, we have proposed different ways to efficiently compute linear algebra operations in order to implement effective iterative Krylov methods for solving large and sparse linear systems on GPU with different precisions of computations. We have explained how to handle data transfers and memory management. We have compared the performances of *Alinea*, the in-house library, against existing scientific linear algebra libraries for GPU with CUDA. The experiments have been performed on a set of large-size sparse matrices for several engineering and scientific problems. In order to ensure even greater efficiency, the auto-tuning of the gridification on the GPU architecture has been performed to obtain faster implementation. Auto-tuning proves that gridification strongly impacts the performance of algorithms. The experiments also exhibit that the tuning of the parameters depends on the objective of the algorithm, in fact the dot product features will differ from those of the SpMV.

We have described how to implement efficient iterative Krylov methods on GPU by using our gathered experience. Different Krylov methods are then developed and compared for different data matrix storage formats. The experiments, performed for several matrices, confirm the performance of our proposed implementation. The results demonstrate the robustness, competitiveness and efficiency of our own implementation compared to the existing libraries. We have also compared two famous and used accelerators: CUDA and OpenCL. A comparison of *Alinea*, a library developed by the author, and *Cusp*, an open-source library, has shown the importance of optimization for energy consumption; *Alinea* outperforms *Cusp* with 30% less energy consumption for the same computations. A prediction on energy consumption based on the complexity of the CG method is on a par with the actual energy consumption. This methodology could be applied to other kinds of linear solvers and numerical applications.

In order to study the scalability of the implementations, we plan to analyze in the next chapter, parallel iterative algorithms, such as the sub-structuring method on a cluster of multi-core-GPUs. In parallel, for each processor we will have to decide on the data distribution of matrix and vector, and then establish an effective communication scheme by taking into account the sparsity pattern of the matrix, in order to minimize the overall execution time. We will pay a special attention to parallel asynchronous algorithms.

Introduction to parallel linear system solvers

” *The essence of mathematics lies in its freedom.*

— Georg Cantor
(Mathematician)

4.1 Introduction

EXASCALE computers are expected to have highly hierarchical architectures with nodes composed by multiple core processors (CPU) and accelerators (GPU) [8]. Distributed computing constantly gains in importance and has become an important tool in common scientific research work. Nowadays, the scientific community is committed to demonstrating that parallel computing has a key role in engineering and applied science simulations. Parallel computations are fundamental and ubiquitous in numerical analysis and its large application areas, when we deal with the problem of large data size. Usually, solving partial differential equations by numerical methods such as the finite element method leads to linear systems with large and sparse matrices. Parallel iterative methods have become indispensable for solving large sparse linear systems. Indeed, parallel computers provide the resources and computing power for solving these systems in a high-performance computing context. However, the different programming levels generate new difficulties and algorithms issues.

In this thesis, we focus on algorithms in parallel distributed-memory computing. In distributed-memory computing, the communication system represents a significant portion of the total execution cost of an algorithm. Most applications in parallel computing involve synchronous algorithms. Nevertheless, these algorithms become inefficient when dealing with a large number of processors. One solution is to use asynchronous algorithms. In general, asynchronous algorithms require a greater number of iterations before the convergence than synchronous algorithms. However, asynchronous iterative algorithms can significantly reduce the overall execution times by eliminating idle times due to synchronizations. Let us consider, in the remainder of this chapter, a system, $\mathcal{PS}_{system}\{p\}$ (Parallel System), capable of executing p processes of execution of \mathcal{PS} . Let $\mathcal{PS} = \{P_1 = 1, \dots, P_p = p\}$ be the set of the processors. The implementation of parallel iterative solvers on the parallel platform $\mathcal{PS}_{system}\{p\}$ requires first for each processor to decide the data distribution of the matrix and the vector, *i.e.*, the partitioning of the data of the linear system on $\mathcal{PS}_{system}\{p\}$. Then to establish an effective communication scheme by taking into account the sparsity pattern of the matrix, in order to minimize the overall execution time.

In this chapter the problem space is supposed euclidean of the form \mathbb{K}^n where \mathbb{K} is \mathbb{R} or \mathbb{C} . \mathbb{K}^n denotes a n -dimensional linear space composed of n coordinates of type \mathbb{K} .

My contribution in this chapter was to develop and implement a code for partitioning data among processors in order to control and adapt the splitting to the studied parallel algorithms. This choice allowed us to be free and implement parallel algorithms as we like.

I have proposed a way to partition correctly a sparse matrix for sub-structuring methods.

The structure of this chapter is as follows. For convenience and thoroughness, Section 4.2 gives an overview of the different classes of parallel iterative methods for readers not familiar with parallel and distributed computing. Given that the first step in parallel processing, one also necessary for optimization, consists in distributing data on the cluster processors, in Section 4.3 we shortly describe how data are distributed among processors for different splitting strategies. In particular, we present a smart process to partition a sparse matrix for parallel sub-structuring methods. Section 4.4 is devoted to a state-of-the-art theory of parallel synchronous and asynchronous iterations. We present the classical model and models with total communication in particular.

Keywords — Distributed computing, SISC, SIAC, AIAC, Synchronous Iterations, Synchronous Communications, Asynchronous Iterations, Asynchronous Communications, Flexible communications, Matrix partitioning, Band-row splitting, Substructuring splitting, Graph partitioning, Line-graph

4.2 Classification of parallel iterative methods

In this section, we review the state-of-the-art parallel iterative methods. Effective interprocessor communication is one of the most important and most challenging problems associated with massively parallel computing. The performance of algorithms strongly depends on the management of interprocessor communication. Naturally, except for a rare parallel method, the local computations of each process in parallel solver algorithms have dependencies between them. Therefore, synchronization points must be added from the moment data are exchanged between cooperating processes. The main difference between the algorithms is based on the choice of the synchronization points.

In this section, we are interested in understanding the different existing choices. We will also give a short review of asynchronous algorithms. The parallel iterative methods for solving the linear system (3.1) are classified into different categories depending on both the nature of the scheme (synchronous, asynchronous) of iterations, and communications. A classification has been defined based on these criteria that leads to three classes. The three classes, which be presented in the following, are *Synchronous Iterations and Synchronous Communications (SISC)*, *Synchronous Iterations and Asynchronous Communications (SIAC)*, and *Synchronous Iterations and Asynchronous Communications (AIAC)*. These categories have *flexible communication* versions.

4.2.1 Iterative methods SISC

The most classical and popular scheme for parallel iterative algorithms is *synchronous iterations* (SI). This consists of an algorithm where a new iteration is started only when all the data from the previous iteration has been received. When coupled with *synchronous communication* (SC), the system synchronizes data exchanges, which means that data can only be exchanged when the source and destination processes have terminated their computation.

Thus, all processes begin calculating the same iteration at the same time and perform the exchanges of shared data at the end of each iteration through global synchronous communications. In this case, communication and computation are totally separate. This scheme is called *Synchronous Iterations and Synchronous Communications (SISC)*. Figure 4.1 presents

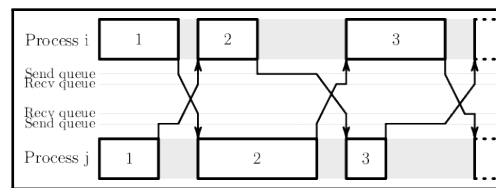


FIGURE 4.1: Example of the execution scheme of the parallel iterative “*Synchronous Iterations and Synchronous Communications (SISC)*” with 2 processors

an example of execution scheme of parallel iterative SISC. In Figure 4.1, the arrows and the bold rectangles represent respectively the communications and the computations, and the light gray box corresponds to the time where the process is at rest, *i.e.*, without computations (waiting communication). The parallel solvers with the SISC scheme have exactly the same number of iterations as their sequential versions. In addition, in exact arithmetic, the residual histories of sequential and SISC algorithms are identical. Thus, the convergence results of parallel SISC algorithms are easy to determine. In fact, they are closely related to those of sequential algorithms. In general, synchronous communications are often a weak link in the algorithm with respect to performance [174] [175] [176]. With this scheme, the fastest processes often remain idle, waiting for the others. In theory, this scheme is advantageous for an homogeneous set of processes which have identical features in terms of speed and synchronization. Figure 4.2 gives another variant of the SISC scheme where the computation

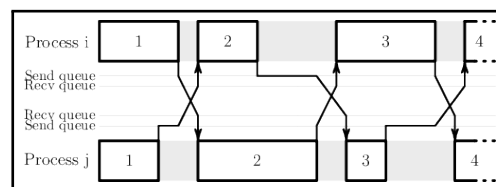


FIGURE 4.2: Example of the execution scheme of another variant of the parallel iterative “*SISC*” with 2 processors

starts once all the data has been received, even if the sending is not completed. This requires more memory since an additional buffer is needed, *i.e.*, one for the computation in progress and another for the sending.

4.2.2 Iterative methods SIAC

The *Synchronous Iterations and Asynchronous Communications (SIAC)* class is a proposed solution to overcome the handicap of SISC algorithms, *i.e.*, the synchronization of the communications. This scheme is often used for slow interconnection networks and/or heterogeneous platforms. The main idea behind the SIAC scheme consists in preserving a synchronize iterative scheme (SI) while data are exchanged asynchronously between processes (AC), which allows an overlapping of the computations with the communications. The conservation of the *SI* (Synchronous Iterations) scheme ensures the same overall behavior, and thus the same convergence conditions as the SISC scheme. Therefore, the convergence conditions of SIAC are identical to those of its sequential counterparts. With the *SC* (Asynchronous

Communications) scheme, the idle time of processes between two successive iterations is reduced. Figure 4.3 shows an example of execution scheme of parallel iterative SIAC. In this case, communication and computation are overlapped when possible. Indeed, a process can send data to its dependency as soon as they are ready to be used in the computations of the next iteration. This allows to reduce the waiting time for receiving data between two successive iterations. An example of execution scheme of parallel iterative SIAC with *flexible send*

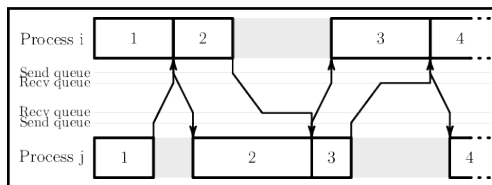


FIGURE 4.3: Example of the execution scheme of the parallel iterative “*Synchronous Iterations and Asynchronous Communications (SIAC)*” with 2 processors

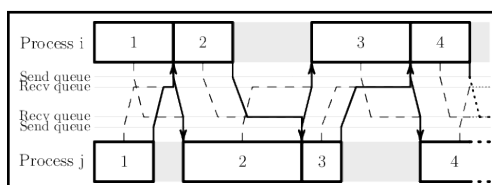


FIGURE 4.4: Example of the execution scheme of the parallel iterative “*Synchronous Iterations and Asynchronous Communications (SIAC)*” with flexible send, with 2 processors

is given in Figure 4.4. In the *flexible send* scheme, a part of the data can be sent before the completion of the computation. Flexible communication is illustrated in Figure 4.4 by a dashed line, which means that data are partially incomplete. This allow to have better overlap between communication and computation and reduce the delay between two iterations. It allows to use a lower bandwidth interconnection than the standard SIAC for the same performance. The concept of flexible communication allows to model efficient asynchronous iterations on parallel computers.

4.2.3 Iterative methods AIAC

The previous parallel iterative synchronous subclasses, SISC and SIAC, are the most used for solving large sparse linear systems. They use the same mathematical model and have the same convergence results as their sequential counterparts. They are often simply called parallel iterative algorithms, synchronous being omitted. Their popularity is probably due to the fact that they are relatively easy to implement, in contrast to parallel iterative asynchronous subclasses. Their global convergence behaviors are easy to control and determine as sequential iterative algorithms, due to the synchronization between the successive iterations. However, these previous methods have another obstacle related to the *SI* (Synchronous Iterations). The last class called *Synchronous Iterations and Asynchronous Communications (AIAC)* aims to suppress that obstacle. In this type of parallel scheme, the processes become independent. In fact, each process executes its own iterations without taking into account the progress of the execution of the other processes. The principle is simple: when a process has finished one iteration, it starts a new one immediately. The latest available data is used for the computation and the data are sent asynchronously. When new data arrives, the previous one is discarded, even if it has never been read. Also the sending of some data may be skipped if the previous

send process is not finished. Figure 4.5 gives an example of execution scheme of parallel

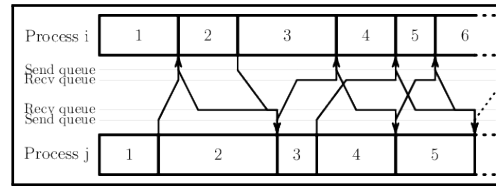


FIGURE 4.5: Example of the execution scheme of the parallel iterative “*Synchronous Iterations and Asynchronous Communications (AIAC)*” with 2 processors

iterative AIAC. In this case, the iterations are not synchronized anymore. The computation never waits and simply uses the last data available. This allows for a complete overlap of computation and communication, and suppresses wasted time. Some processes may be faster in their calculations and performs more iterations than others. However, the asynchronous AIAC solvers have more strict convergence conditions their synchronous counterparts (SISC, SIAC). In general, the AIAC solvers perform more iterations than the synchronous solvers, SISC and SIAC. An example of execution scheme of parallel iterative AIAC with *flexible send and receive*

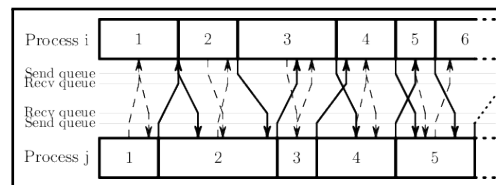


FIGURE 4.6: Example of the execution scheme of the parallel iterative “*Synchronous Iterations and Asynchronous Communications (AIAC)*” with flexible send and receive, with 2 processors

is drawn in Figure 4.6. As we can see in Figure 4.6, partial data are sent before the end of the iteration, and data received are used as soon as they are received. In Figure 4.4, the dashed line describes flexible communication, which shows that data are partially incomplete. The iterative methods AIAC were first called *chaotic relaxations* [177] [178], but are now usually designated by *asynchronous iterations*. In the state-of-the-art, three main schemes are used for asynchronous iterations: *partially asynchronous iterations (P-AI)*, *totally asynchronous iterations (T-AI)*, and *flexible asynchronous iterations (F-AI)*.

The first scheme, *partially asynchronous iterations (P-AI)*, is based on the hypothesis that the communication time is bounded and that each process performs at least one iteration for each given period. In *totally asynchronous iterations (T-AI)*, there are few or almost no constraints on iterations and communications, except that they must never stop. In 1989, D.P. Bertsekas and J.N. Tsitsiklis [179] proposed a mathematical model of these methods, which has resulted in several convergence theorems widely applicable. In [179], a general convergence theorem based on a set of imbricated boxes has been used as the basis of large convergence results in several domains. In this thesis, this is this scheme and theoretical model in use.

Sometimes the “*total*” feature can be a strong condition for the convergence of some algorithms. When these algorithms do not converge under *totally asynchronous iterations*, they can converge with *partially asynchronous iterations* assumptions, as shown by D.P. Bertsekas and J.N. Tsitsiklis [179], J.N. Tsitsiklis, D.P. Bertsekas and M. Athans [174], and J.N. Tsitsiklis [180]. R. De Leone [181] presents a unified treatment for partially and totally asynchronous parallel successive overrelaxation (SOR) algorithms for the linear comple-

mentarity problem. The last scheme, *flexible asynchronous iterations (F-AI)*, more general than *totally asynchronous iterations*, is also more recent and was first introduced by J.-C. Miellou, D. El Baz, and P. Spitéri [182]. Several studies have deepened the analysis of this scheme [183] [184] [185].

4.2.4 Towards asynchronous algorithms

In the context of massively parallel computing, the asynchronous parallel algorithms currently present a great interest in the scientific community for solving extremely large linear systems. New programming paradigms of parallel methods should be defined and evaluated with respect to state-of-the-art scientific methods. As said before, in this thesis we pay a particular attention to asynchronous iterative algorithms. Therefore we will start with a review of the state-of-the-art asynchronous algorithms.

Asynchronous algorithms have a long history. These methods were first introduced in 1969 by D. Chazan and W. Miranker [177] for solving linear systems using *chaotic relaxation* methods. Two years later, in 1971, J.D.P. Donnelly [178] generalized two results of [177] concerning *periodic chaotic relaxation*. These methods have been widely studied since. The studies of Chazan and Miranker, and of Donnelly were generalized in 1975 by J.-C. Miellou [186] for solving large-size non-linear algebraic systems. In 1978, G.M. Baudet [187] presented a class of asynchronous iterative methods for solving a system of equations. In [177] [178] [186], the motivation for defining chaotic relaxation was to account for the implementation of iterative methods on a multi-core system so as to reduce the communication and synchronization between processes with a dependency. However, in their studies, communication occurs only at the end of each update. This translates a lack of flexibility in communications. Over the years, several studies of asynchronous convergence analysis have appeared. One can mention M. Charnay [188], P. Comte [189], J. Jacquemard [190], M.B El Tarazi [191] [192] [193], D.P. Bertsekas and J.N. Tsitsiklis [179], D. El Baz [194] for the solution of non-linear systems of equations $Fx = z$ via parallel asynchronous algorithms, and J.M. Bahi [195]. J.-C. Miellou *et al.* [182] proposed in 1998 a new class of parallel asynchronous iterative algorithms for solving non-linear systems of equations with flexible communication between processors. The convergence analysis have been performed with partial order technics. This new class corresponds to flexible AIAC described in Figure 4.6. The convergence analysis of these methods is based on monotone operators with respect to a partial ordering. Recently in 2005, D. El Baz, A. Frommer, and P. Spitéri [196], introduced a new formulation of parallel asynchronous algorithms with flexible communication where the convergence analysis is based on contracting operators. They propose global convergence results for asynchronous iterations with flexible communication based in a contraction context. The asynchronous algorithms have been used to solve diverse numerical problems:

- **linear systems** (D.J. Evans and W. Deren [197], R. Bru, V. Migallón, and J. Penadés [198])
- **non-linear fixed point and optimization problems** (J.-C. Miellou [186], G.M. Baudet [187], P. Spitéri [199], Z. Bai, D. Wang, and D. Evans [200], D. El Baz [201], J.M. Bahi, J.-C. Miellou, and K. Rhofir [202], J. Arnal, V. Migallón, and J. Penadés [203])
- **network flow problems** (D.P. Bertsekas and D. El Baz [204], E. D. Chajakis and S. A. Zenios [205], D. El Baz [206], D. El Baz, P. Spitéri, J.-C. Miellou and D. Gazen [183])

- *Markov systems* (J. Bernussou, F. Le Gall, and G. Authie [207], D. El Baz [208] and J.M. Bahi [209])

These asynchronous algorithms have also been used in domain decomposition methods. One can cite the work of P. Spitéri, J. Miellou, and D. El Baz [210] [211]. In this thesis, they are analyzed only in the context of iterative linear system solvers, *i.e.*, only linear fixed point problems are considered. In this thesis, we aim to solve linear partial differential equations discretized with Finite Element Methods (FEM). Among the first asynchronous algorithms used to solve these problems are those using splitting and contracting operators such as Jacobi's, Richardson's, the Successive Over Relaxation (SOR) method, etc. L. Lei analyzes the convergence of asynchronous iterations using fixed point iterations with an arbitrary splitting form, in [212]. The fixed point problems for asynchronous algorithms have also been used in multi-splitting methods [213] [214] [215] [185] [211] [216] [217], which were pioneered by O'Leary and R. E. White [218] in 1985. The multi-splitting method is a generalization of the additive Schwarz and splitting methods.

Asynchronous analysis for iterative Krylov methods remains a challenge in the domain, and they have not been solved to converge with asynchronous iterations. Asynchronous iterations have been unpopular compared to synchronous iterations due to limited knowledge about them and their complexity both in terms of algorithm and implementation. With the advent of massively parallel machines, heterogeneous platforms and GPUs, asynchronous algorithms have taken a more important place in parallel computing. They have become increasingly widespread in diverse applications [217] [219] [220] [220] [221]. Asynchronous algorithms ensure a complete overlap of computations and communications, and the fault tolerance become easier to design. Thus, there is no time wasted on synchronizing cooperating processors. In addition, there is no time wasted on waiting for messages. The most favorable environments are particularly the distributed clusters in which the communication networks are often heterogeneous with some great differences between the networks performances. Asynchronous algorithms have been implemented in diverse environments: from workstations, supercomputers to grids, and with shared and distributed memory. There are various communication APIs such as MPI, PVM, TCP/IP, Infiniband, etc. Recently, several libraries have appeared in order to ease the development of asynchronous algorithms: Jace [222] [223] for grid applications, JACEP2P [224] for executing parallel iterative asynchronous applications on volatile distributed architectures, CRAC [225] an environment dedicated to designing efficient asynchronous iterative algorithms for a grid architecture. In this thesis, we focus on *MPI* communication API.

Considering the AIAC scheme, where the iterations are asynchronous, it appears that the last obstacle is based on the convergence detection and the halting procedure. Compared to synchronous iterations, where the detection of convergence requires just an allreduce operation, asynchronous iterations are a more complex problem. They present a serious obstacle, which is research problem in itself. In contrast to sequential iterations and synchronous iterations, there are no global iterations. Each process needs only update data and computations locally, while respecting all the asynchronous properties (in exact arithmetic). Several methods keep the same algorithms as in synchronous iterations. They are complete computer science. Most of them are based on simultaneous local convergence for each process. These methods are mostly

empirical. They consist in considering the synchronous iterations as *macro-iterations*. They may perform a lot of unnecessary iterations before detecting the convergence. These methods have generated further research, which have been oriented mostly toward two versions: those *centralized* (D.P. Bertsekas *et al.* [179], K. Blathras *et al.* [226]) and those *decentralised* (J.M. Bahi [227] [216] *et al.* [228]). In [179] and [229], D.P. Bertsekas and J.N. Tsitsiklis have demonstrated the convergence results of the algorithms they have proposed. Unfortunately, there is no way to answer the following question: what is a global asynchronous number of iterations? (what is the number of iterations in an algorithm with asynchronous iterations?). The mathematical model of asynchronous convergence requires some additional conditions on the communication delay. The idea behind it is to match asynchronous communications with synchronous communications. This hypothesis enlarges the number of communications needed.

B.F. Beidas *et al.* [230] have assumed communication delays among the processors to be stochastic, with a Markovian character. In 1996, D. El Baz [231] proposed a variant of proof that reduces the overhead. A general algorithm exchanging many small messages, is introduced and demonstrated by S.A. Savari *et al.* [232]. D. El Baz presents another approach to terminating asynchronous iterative algorithms in [201], which requires few additional communications. This method uses the sequence of sets from the convergence theorem of D.P. Bertsekas *et al.* [179]. D. El Baz has demonstrated that this method can be applied with success to all asynchronous iterative algorithms satisfying the conditions of D.P. Bertsekas [179] [233].

The implementation of asynchronous algorithms is summarized by the implementation of the convergence detection and the halting procedure. A good analysis of the convergence ensures more accurate termination detection. Further research into this analysis has also been carried out: those taking into account roundoff errors [234] [235] [236], and those based on stochastic behaviors [237] [230] [238] [239].

4.3 Graph and Matrix partitioning

Undoubtedly, parallelism is the future of computing. As said above, the main step in parallel processing consists in distributing the data on the cluster processors, which is commonly called parallel distributed computing. In this section, we describe how data are distributed among processors for different splitting strategies: band-row, band-column, and sub-structuring splitting. The distribution of data is accomplished as a preprocessing step, independently from the solver code. The data such as matrix, right hand-side, vector solution and local to global, are written into a file, and will be input for the solver code. The matrix is read from and written into the matrix market file [173] [240].

My contribution was to develop and implement a code for partitioning data in order to control and adapt the splitting to the studied parallel algorithms. This choice has allowed us to be free and implement parallel algorithms as we like. The sub-structuring splitting uses METIS software [241] for partitioning graphs. Then, we have implemented the partitioning into sub-structures.

To illustrate the advantages and disadvantages of these types of partitioning, we will focus on the matrix-vector product, which is the most expensive operation in linear algebra, as demonstrated in the previous chapter.

4.3.1 Band-row splitting

The partition of the equation set leads to allocating each processor a band of rows corresponding to the block of the processed vectors. In Figure 4.7(a) where an example of band-row splitting is given, these terms are located in a colorful area. The band-row splitting approach consists in partitioning the matrix A of size $n \times n$ into horizontal band matrices. Each processor is in charge of the management of a band-row matrix of size $N_p \times n$ and the associated unknown vector x of size $N_p \times 1$, as drawn in Figure 4.7(a). This method of partitioning by band-row allows to exhibit a sufficient degree of properly balanced parallelism. This implies assigning all processors, a block of equally sized rows, containing approximately the same number of non-zero coefficients. Unfortunately, it suffers from a major lack of granularity for implementation on a distributed memory system. The band-row partitioning algorithm of a

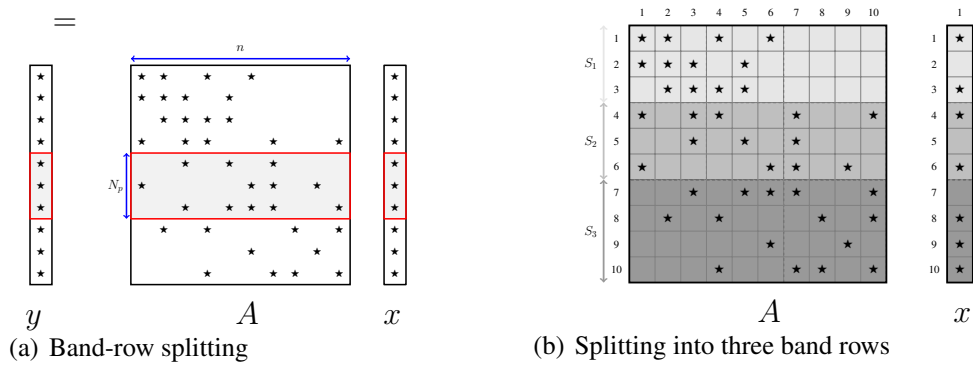


FIGURE 4.7: Example of band-row splitting of a matrix

given matrix for a given processor p is presented in Algorithm 4.1. However, in practice,

Algorithm 4.1: Band-row partitioning for the p – th band, with CSR matrix

input : n (size of the matrix),
 $A.r_ptr$ (index of first entry of each row), $A.cols_n$ (column numbers), $A.coef$ (non-zero values)
input : N_b (number of row-bands), $0 < N_b \leq n$; p (processor number), $1 < p \leq N_b$
output : A_b : band matrix (CSR)
variable : n_r_band : number of rows per band, r_d : r_ptr displacement
variable : nz_d : $cols_n$ and $coef$ displacement, nz_c : $cols_n$ and $coef$ count

```

1  $n\_r\_band \leftarrow n/N_b$ ;  $N_p \leftarrow n\_r\_band$ 
2 if  $p = N_b - 1$  then  $N_p \leftarrow n - p \times n\_r\_band$ 
3  $r_d \leftarrow i \times n\_r\_band$ ;  $nz_d \leftarrow A.r\_ptr[r_d]$ ;  $nz_c \leftarrow A.r\_ptr[r_d + N_p] - A.r\_ptr[r_d]$ 
4 // Allocate band-row matrix  $A_b(N_p \times n, nz_c)$ 
5  $A_b.r\_ptr[:] \leftarrow A.r\_ptr[r_d + :]$  // -- copy row indices
6  $A_b.r\_ptr[:] \leftarrow A_b.r\_ptr[:] - A_b.r\_ptr[1]$  // -- shift row indices to local
7  $A_b.r\_ptr[N_p + 1] \leftarrow nz_c$  // -- copy cols numb and coef
8  $A_b.cols\_n[:] \leftarrow A.[nz_d + :]$ ;  $A_b.coef[:] \leftarrow A.[nz_d + :]$ 

```

i.e., in the implementation, there are two main strategies to store data into each processor with band-row splitting.

Algorithm 4.2: Band-column partitioning for the $p - th$ band, with CSR matrix

```
input   :  $n$  (size of the matrix),  
           $A.r\_ptr$  (index of first entry of each row),  $A.cols\_n$  (column numbers),  $A.coef$  (non-zero values)  
input   :  $N_b$  (number of column-bands),  $0 < N_b \leq n$ ,  $p$  (processor number),  $1 < p \leq N_b$   
output  :  $A_b$ : band matrix (CSR)  
variable:  $n\_c\_band$ : number of columns per band,  $col_d$ : column indices ( $r\_ptr$ ) displacement  
variable:  $nz_d$ :  $cols\_n$  and coef displacement,  $nz_c$ :  $cols\_n$  and coef count  
  
1  $n\_c\_band \leftarrow n/N_b$ ;  $N_p \leftarrow n\_c\_band$   
2 if  $p = N_b - 1$  then  $N_p \leftarrow n - p \times n\_c\_band$   
3  $col_d \leftarrow i \times n\_c\_band$   
4  $nz_d \leftarrow A.r\_ptr[r_d]$   
5 // Recover  $nz_c$  of the band  
6 // Allocate band-column matrix  $A_b(n \times N_p, nz_c)$   
7  $nz_c \leftarrow A.r\_ptr[r_d + N_p] - A.r\_ptr[r_d]$   
8  $c \leftarrow 1$   
9 for  $i \leftarrow 1 : n$  do  
10 |  $A_b.r\_ptr[i] \leftarrow c$   
11 | for  $k = A_b.r\_ptr[i] : A_b.r\_ptr[i + 1]$  do  
12 | | if  $nz_d \leq A.cols\_n[k] < nz_d + N_p$  then  
13 | | |  $A_b.cols\_n[c] \leftarrow A.cols\_n[k] - nz_d$ ;  $A_b.val[c] \leftarrow A.val[k]$   
14 | | |  $c \leftarrow c + 1$ ;  
15 | | end  
16 | end  
17 end
```

Naive splitting The first splitting is the naive one, which consists in storing both matrices and vectors without any other informations. The principle of the local matrix-vector product with row-band splitting consists in multiplying the local band row matrix by the global (*gathered*) vector. This implies data exchange between all processors (*MPI_ALLGATHER*). To perform the matrix-vector product, each processor needs to receive and/or send missing information of the vector x from/to cooperating processors. According to the sparsity of the matrix, *i.e.*, the distribution of non-zero values, the processor may receive and/or send unnecessary information which overload the communication. This operation is very expensive for large-size matrices. The communications dominate the computations for large-size matrices. One solution to overcome this problem is to take into account the sparsity pattern of the local matrix, in order to send/receive only the necessary data from/to cooperating processors.

Sparsity Pattern splitting In this technique, in addition to the *naive splitting* data, we store the information of dependencies from local to cooperating processors and from cooperating to local processors, according to the sparsity pattern of the local matrix. Each processor has both a list of receive and a list of send dependencies, which keeps data exchange to a minimum. Figure 4.7(b) gives an example of splitting a sparse matrix into three band rows. Table 4.0(b) gives the corresponding list of dependencies for **sending** to cooperating processors of the splitting described in Figure 4.7(b). The corresponding list of dependencies for **receiving** from cooperating processors of the splitting described in Figure 4.7(b) is reported in Table 4.0(a).

The dependency nodes drawn in gray color in Table 4.1 correspond to zero values in vector x . In fact, we can remove these nodes from the list of dependencies. In this thesis, the band-row splitting with the “*Sparsity Pattern*” technique takes into account both the sparsity of the matrix and the vector. However, this technique does not guarantee a perfect load balance. One solution for a perfect load-balance consists in partitioning the weighted graph (graph where the weight of the vertex v_i associated with the row i is its number of non-zero values)

(a) Receiving dependencies			(b) Sending dependencies		
Local proc.	Recv. proc	List Dependency nodes	Local proc.	Send. proc	List Dependency nodes
1	← 2	4 - 5 - 6	1	→ 2	1 - 3
2	← 1	1 - 3	1	→ 3	2 - 3
2	← 3	7 - 9 - 10	2	→ 1	4 - 5 - 6
3	← 1	2 - 3	2	→ 3	4 - 5 - 6
3	← 2	4 - 5 - 6	3	→ 2	7 - 9 - 10

TABLE 4.1: List of **receiving** and **sending** dependencies of the splitting described in Figure 4.7(b).

associated with the matrix in k parts with the minimal amount of edges cut. This solution leads to finding a good distribution of the sparse matrix on the parallel processors. The concept consists in storing the i^{th} row of the matrix on processor j if the vertex v_i is in the i^{th} sub-part. Then, we have equal weight in each band of the splitting. The bands may be composed with non-contiguous rows.

Matrix-vector product The processor that will perform the matrix-vector product for a band-row has only the corresponding terms of the vector x , the colored area in Figure 4.7(a). In order to carry out the sparse matrix-vector, this process needs all the terms of the vector x . The first step, therefore, consists in collecting the terms that lacking from the colored area in Figure 4.7(a). As it is the same for all processors, it will therefore be necessary to reconstruct the full vector x on each processor. This operation corresponds to a classic collective exchange, where each is both a transmitter and a receiver. In this work, instead of using the collective operation, *MPI_Allgather*, including the message passing library (MPI), we use the equivalent *Send/Recv*, with a *left-right* ordering of sending and receiving. For the processor p , the *left-right* ordering consists in respectively sending and receiving to and from $k = p - 1$, $k = p + 1$, $k = p - 2$, $k = p + 2$, $k = p - 3$, $k = p + 3$, ..., if $k > 0$. This process is described in Figure 4.8. The number of arithmetical operations requires to

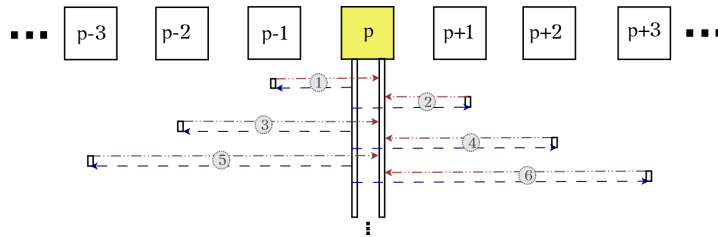


FIGURE 4.8: Send/Recv ordering of the processor p

perform the local sparse matrix-vector multiplication, which is approximately $\frac{K \times n}{s}$, where s is the number of processors, n the dimension of the matrix, and K the average number of non-zero coefficients per row. On the other hand, the total number of terms of the vector x to recover before performing the product is approximately $\frac{(s - 1) \cdot n}{s}$, if the local matrix has non-zero values in almost all columns. The amount of data is not small compared with the number of arithmetic operations. Optimizing communications consists in finding a way to drastically limit the number of external values of vector x , located on the others processors, and is necessary to compute the product by the matrix.

Basic linear algebra operations The computation of the dot product is a relatively simple operation. Each processor performs a local dot product, *i.e.*, multiplies its elements and sums them, from their two local vectors. Finally, the local sums are added using *MPI_ALLREDUCE* with the *MPI_SUM* operation. Then each processor has the global dot product. Operations such as an addition of vectors, the element wise product, etc. do not change compared to the sequential code. For the GPU version, local operations have been performed on a graphics card.

4.3.2 Band-column splitting

As for band-row splitting, the band-column approach consists in partitioning the matrix A into vertical band matrices. Each processor is in charge of the management of a band-column matrix of size $n \times N_p$. The associated unknown vector x of size $N_p \times 1$ is splitting into horizontal band vectors as in band-row splitting as described in Figure 4.9(a).

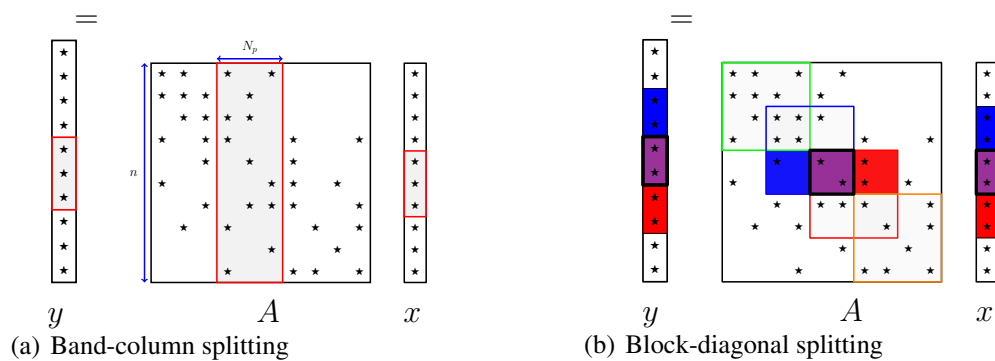


FIGURE 4.9: Example of the band-column and block-diagonal splitting of a matrix

The Algorithm 4.2 describes the column-row partitioning procedure of a given matrix for a given processor p . According to the structure of the CSR format, the computation of the number of non-zero values of each band requires a particular calculation, unlike row partitioning. At line 8 of the Algorithm 4.2, we recover the number of non-zero values computed outside the routine. All non-zero values of all processors are stored in an independent array, which is built using the same test process described at line 14 of Algorithm 4.2.

Matrix-vector product Unlike the band-row splitting sparse matrix-vector multiplication, the SpMV for band-column splitting avoids the exchange of the vector x . However, an *MPI_ALLREDUCE* (*MPI_SUM*) is required, in order to assemble the vector $y = Ax$. Note that basic operations are the same as for band-row splitting. For the GPU version, local operations have been performed on a graphics card. For the matrix-vector product, the results are first sent to CPU before applying the same procedure as in CPU.

4.3.3 Block-diagonal splitting

When the product is performed by the matrix, the product of the diagonal block requires only local terms of the vector x . In contrast, off-diagonal coefficients require the corresponding terms of the vector x . The diagonal blocks are thick black lines in Figure 4.9(b). The optimal splitting is the one that partitions the mesh into sub-structures of the same size, in order to balance the load with the smallest possible boundary to limit data transfers. Substructures

should be as spherical as possible topologically, since it is the sphere that has the smaller outer surface.

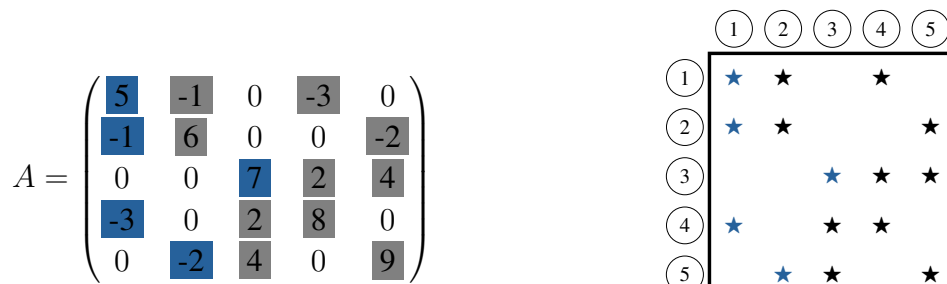
4.3.4 Mesh and Graph Coloring with METIS

In this thesis, we have used METIS software to partition a directed graph and partition a finite element mesh. Graph partitioning is a common preprocessing step in many high-performance parallel applications on distributed and shared-memory architectures. The problem consists in partitioning the vertices of a graph in k into approximately equal parts, such that the number of edges connecting vertices in different parts is minimized. For instance, the solution of a sparse linear system, $Ax = b$, by parallel iterative methods leads to a problem of graph partitioning. The partitioning of the graph corresponding to the matrix A , significantly reduces the amount of communication required by the computation of a sparse matrix-vector, which is the key step in each iteration of iterative methods [242]. G. Karypis and V. Kumar [243] have showed that the partition produced by METIS is consistently 10% to 50% better than those produced by spectral partitioning algorithms [244] [245], and 5% to 15% better than those produced by Chaco multilevel [245] [246].

METIS [241] is a set of algorithms for partitioning graphs and finite element meshes. It uses multilevel recursive-bisection, multilevel k -way, and multi-constraint partitioning schemes. The version of the METIS library (API) used is *v.5.1* (ParMETIS *v.4.0* for parallel version). ParMetis is a parallelization of Metis using MPI. Let us first get some definitions used in the graph theory [247] [248] [249] [250] [251].

Definition 4.1 (Graph) In graph theory, a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a collection of points, \mathcal{V} , called vertices and lines connecting some (possibly empty) subset of them, $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$, called edges.

Assuming that matrices, vertices, and edges are indexed from 1, the sparse matrix A described in Figure 4.10(a) will be used to illustrate the terms of this section. Figure 4.10(b) draws the pattern of non-zero values of the matrix A . For the sake of generality, we discuss graph theory without taking into account the symmetry of the matrix given in Figure 4.10(a). Whenever it concerns a particular process for symmetric matrices, we will clearly state it.



(a) Example of a symmetric matrix A , **nnz** represents the first non-zero on the row

(b) Non-zero pattern of a matrix A

FIGURE 4.10: Example of non-zero pattern of a sparse matrix A

Definition 4.2 (Cardinality of a partition) The number of elements in \mathcal{V} and \mathcal{E} are respectively called “cardinality” of \mathcal{V} and \mathcal{E} . They are respectively noted $|\mathcal{V}|$ and $|\mathcal{E}|$.

Definition 4.3 (Directed Graph (DiGraph)) A “directed graph” (or digraph) is a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where the edges have a direction associated with them, *i.e.*, all the edges are directed from one vertex to another.

Definition 4.4 (Undirected Graph) An “undirected graph” is a digraph where the edges are bidirectional.

Notation 4.5 Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph. Let us note $v_i \in \mathcal{V}$ the node labeled i . Let us note $e_{(v_i, v_j)} = \{v_i, v_j\} \in \mathcal{E}$ the edge that connects the vertices v_i and v_j with tail v_i and head v_j . If the graph is undirected, the tail v_i and head v_j are not significant. If $i = j$, we can note simply $e_{(v_i)}$ instead of $e_{(v_i, v_i)}$. For the sake of simplicity in writing, we note (i) instead of (v_i) , $e_{(i, j)}$ instead of $e_{(v_i, v_j)}$ and $e_{(i)}$ instead of $e_{(v_i)}$. If the graph is undirected, we will note $e_{(\min(i, j), \max(i, j))}$ instead of $e_{(i, j)}$ and $e_{(j, i)}$ (for example, $e_{(2, 1)} \rightarrow e_{(\min(1, 2), \max(1, 2))} \rightarrow e_{(1, 2)}$).

Notation 4.5 is summarized in Figure 4.11.

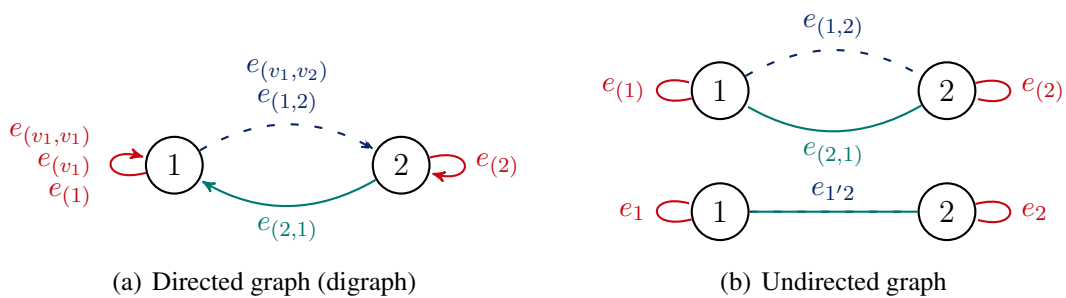


FIGURE 4.11: Notation of a directed graph and an undirected graph

Definition 4.6 (Adjacency Matrix for a Graph) An “adjacency matrix” of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, also called connection matrix of \mathcal{G} , is a matrix $n \times n$ where n is the number of vertices, such that $(v_i, v_j) = 1$ if the vertices (nodes) v_i and v_j are adjacent, *i.e.*, $\{v_i, v_j\} \in \mathcal{E}$ otherwise $(v_i, v_j) = 0$.

Figure 4.12 presents the directed graph of the matrix given in Figure 4.10 and its corresponding adjacency matrix (see Figure 4.13(b)).

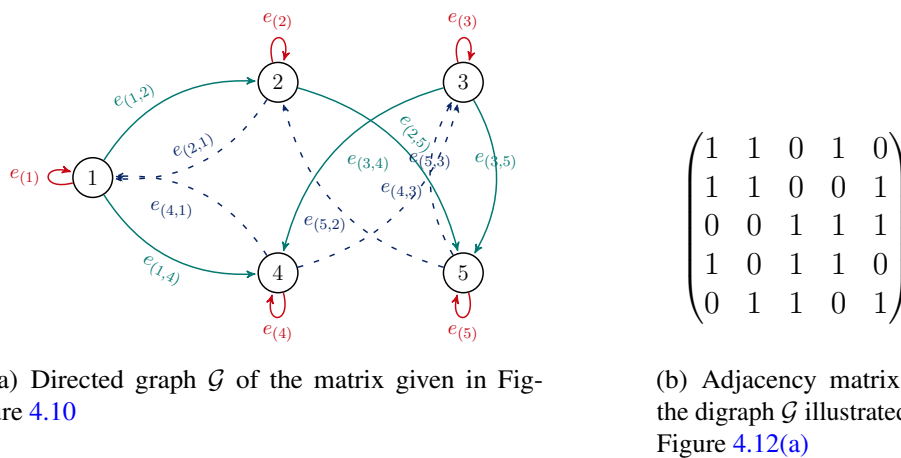


FIGURE 4.12: Example of directed graph and its corresponding adjacency matrix

Remark 4.7 The adjacency matrix of a digraph is not necessarily symmetric. That of Figure 4.12(a) given in Figure 4.13(b) is symmetric because of the symmetry of the initial matrix described in Figure 4.10.

Definition 4.8 (Adjacency Matrix of an Undirected Graph) For an undirected graph, the adjacency matrix is symmetric. Therefore, the adjacency matrix of the graph of symmetric matrix is symmetric.

Figure 4.13 presents the undirected graph of the matrix described in Figure 4.10 and its corresponding adjacency matrix (see Figure 4.13(b)).

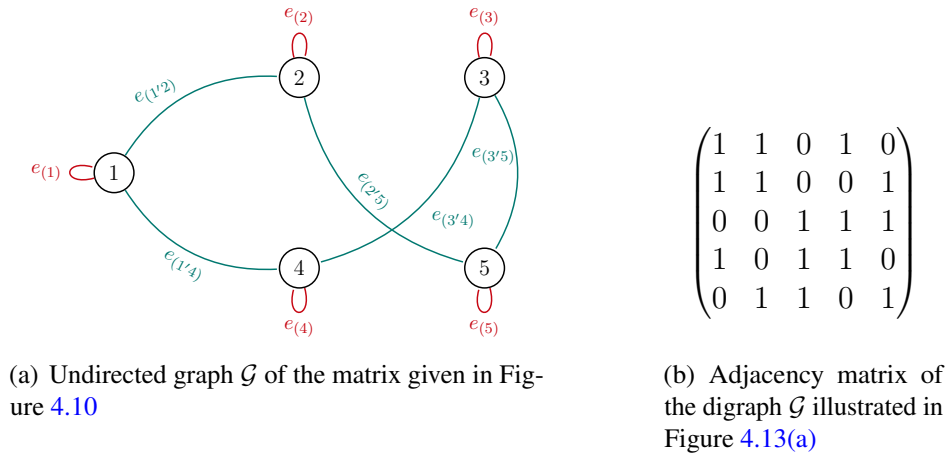


FIGURE 4.13: Example of undirected graph and its corresponding adjacency matrix

Definition 4.9 (Partitioning of graph) A partitioning of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a set \mathcal{S} of subsets S_1, S_2, \dots, S_k of \mathcal{V} such that

$$\text{for } i, j \in \{1, \dots, k\}, \quad \bigcup S_i = \mathcal{V} \quad \text{and} \quad S_i \cap S_j = \emptyset, \quad i \neq j.$$

An individual subset S_i is sometimes called a partition.

Definition 4.10 (Cardinality of a partition) The number of elements in S_i is called “cardinality” of S_i and is noted $|S_i|$.

Definition 4.11 (Cut-edge of a partition) A “cut-edge” of a partition is an edge $e_{v_i, v_j} = \{v_i, v_j\}$ such that $v_i \in S_i$ and $v_j \in S_j$ such as $S_i \neq S_j$.

The number of edges-cut (or edge-cut) of the partitioning corresponds to the number of edges of the graph cut by a partitioning.

Definition 4.12 (Graph partition problem) The “graph partitioning problem” is defined, given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, such that it is possible to partition \mathcal{G} into smaller components with specific properties, *i.e.* given a number of partition k the graph partitioning problem is to find k -way partitioning \mathcal{P} such that the balance of \mathcal{P} and the edge cut of \mathcal{P} are minimized.

Proposition 4.13 *The goal of graph partitioning is to find a partitioning where the number of cut-edges is minimized and each S_i has roughly the same size. In terms of parallel computing, we are interested in evening the amount of work on each processor (load balancing) and minimizing the total amount of communication.*

All of the graph partitioning routines in METIS take as input the *adjacency structure* of the graph and the weights of the vertices and edges (if any). The adjacency structure of the graph is stored using the compressed storage format (CSR) (see Section 3.2.1, Page 43). The adjacency list of the vertex i is stored in the array *adjncy* starting at index $xadj[i]$ and ending at (but not including) index $xadj[i + 1]$. In the METIS structure we store both $e_{(i,j)}$ and $e_{(j,i)}$ for each edge $e_{(i,j)}$. Figure 4.14 shows the adjacency structure or the CSR format of the graph

$$\begin{aligned} xadj &= 1 / 4 / 7 / 10 / 13 / 16, \\ adjncy &= \mathbf{1} / 2 / 4 / \mathbf{1} / 2 / 5 / \mathbf{3} / 4 / 5 / \mathbf{1} / 3 / 4 / \mathbf{2} / 3 / 5 \end{aligned}$$

FIGURE 4.14: An example of the CSR format for storing sparse graph given in Figure 4.13(a)

described in Figure 4.12(a). As a result, from a CSR data structure, we can easily create the data structure required for METIS routines, *i.e.*, *xadj* (row pointers) and *adjncy* (column indices). For detailed information on the adjacency structure, the readers are invited to look up page 23 of the METIS 5.1.x (API C/C++) manual [241].

Remark 4.14 METIS software cuts edges by respecting the Proposition 4.13.

When we partition a graph or mesh, METIS does not guarantee that the partitions are contiguous. In most cases, the partitions will be contiguous. As consequence, the number of iterations may be impacted when the number separation set varies. However, there exists an option which allows to insist in obtaining contiguous parts: *METIS_OPTION_CONTIG*. Let us note that this option just ‘*insists*’ but does not fully ensure a partitioning with only contiguous parts.

4.3.5 Sub-structuring splitting

The splitting techniques presented previously (band-row, band-column, etc.) are undoubtedly simple and easy to control but present disadvantages concerning the Proposition 4.13, *i.e.*, in terms of *load balancing* and the *minimization of the total amount of communication* due to the sparsity of the matrix. Therefore, the communications risk to dominate the computations of large-size matrices. One solution to overcome this problem is to take into account the sparsity pattern of the local matrix, *i.e.*, the *graph of the matrix*. In terms of communication, this approach will allow, for instance, to only send/receive necessary data from/to cooperating processors. The sub-structuring splitting approach proposed here is based on this solution. In this subsection, we present how we proceed to prepare data for sub-structuring methods (see Section 5.2, Page 142).

Definition 4.15 (Graph of a matrix) The “graph of a matrix” $A \in \mathbb{K}^{n \times n}$ is a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $|\mathcal{V}| = n$, *i.e.*, \mathcal{G} has n vertices v_1, v_2, \dots, v_n such that there is an edge between v_i and v_j ($i \neq j$) if $a_{ij} \neq 0$.

Definition 4.16 (Weighted graph of a matrix) The weight of a vertex v_i of the graph of the matrix $A \in \mathbb{K}^{n \times n}$ is defined as the number of non-zero values of the i^{th} row of A . This quantity will be noted $\omega(v_i)$ and we have $\omega(v_i) = d(v_i) + 1$, where $d(v_i)$ is the degree of the vertex v_i .

Remark 4.17 The “graph of a matrix” $A \in \mathbb{K}^{n \times n}$ is a representation of a sparse matrix. The graph is weighted by the number of non-zero values per row if necessary.

Remark 4.18 (Data dependency) Each edge $e_{(i,j)}$ of the graph of the matrix $A \in \mathbb{K}^{n \times n}$ symbolizes a data dependency between the i^{th} row and j^{th} row of the matrix, and thus a possible communication.

In order to optimize, there is communication only if row i and row j present a dependency Remark 4.18 and also if the corresponding vector element x_j is non-zero ($a_{ij}x_j$). The solution of the “graph partitioning problem” (see Definition 4.12, Page 119) leads to a good distribution of the sparse matrix on the parallel processors. A good partitioning helps to achieve perfect load balance. The communication cost of a sub-domain is a function of its edge-cut as well as the number of neighboring sub-domains it share edges with [252] [253]. For the sake of clarity, we consider first an undirected graph. The objective is to partition a matrix into consistent sub-structures for sub-structuring methods. As said in Remark 4.18, each edge $e_{(i,j)}$ of the graph \mathcal{G} of the matrix $A \in \mathbb{K}^{n \times n}$ means that there is a data dependency between the i^{th} row and j^{th} row of A . Whereas if we partition the graph directly into k partition sets using METIS, we will have k independent partitions, which lead to k independent sub-matrices. This procedure implies a loss of informations concerning the dependencies between rows. As a direct consequence, communication will not be done correctly. Indeed, some required send/receive will not occur. Without loss of generality and for the sake of clarity, we illustrate the partitioning for $k = 2$ and $k = 3$. For example, with the undirected graph described in Figure 4.13(a), and via the adjacency structure given in Figure 4.14, METIS cuts the edges $e_{(2'5)}$ and $e_{(3'5)}$ for $k = 2$ and the edges $e_{(2'5)}$ (part #1,part #2), $e_{(2'5)}$ (part #2,part #3) and $e_{(3'5)}$ (part #1,part #3) for $k = 3$. The partition produced by METIS for 2 and 3 sub-parts is respectively presented in Figure 4.15(a) and Figure 4.15(b).

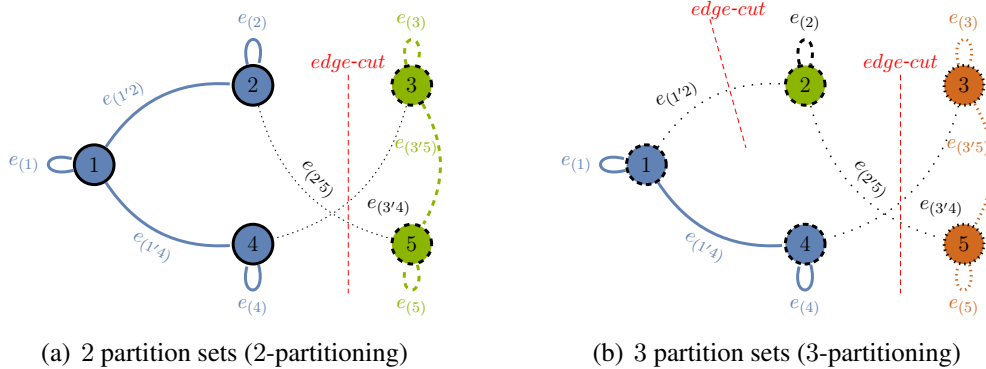


FIGURE 4.15: Direct partitioning by METIS (API C/C++) using CSR structure of the graph of the matrix given in Figure 4.13(a)

Remark 4.19 Notice that METIS does not take into account self-edges, $e_{(i,i)}$, when partitioning.

The solution we propose for correctly partitioning the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ without losing information consists in partitioning the *line-graph* of \mathcal{G} , $\mathcal{L}(\mathcal{G})$, rather than partition the graph \mathcal{G} directly as in Figure 4.15. The line-graph is induced from the original graph.

Definition 4.20 (Line-graph of undirected graph) The line-graph, $\mathcal{L}(\mathcal{G})$, of a undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a graph such that each vertex of $\mathcal{L}(\mathcal{G}) = (\mathcal{L}(\mathcal{V}), \mathcal{L}(\mathcal{E}))$ corresponds to an edge of \mathcal{G} , and there is $e_{(a,b)}$ an edge from v_a to v_b of $\mathcal{L}(\mathcal{G})$ if the corresponding edges, $e_{(a_i,a_j)}$ and $e_{(b_i,b_j)}$ have a vertex in common in \mathcal{G} [254] [255]. The edges of $\mathcal{L}(\mathcal{G})$ have the form $e_{e_{(i,j)},e_{(i,k)}}$ where $e_{(i,j)}$ and $e_{(i,k)}$ are edges in \mathcal{G} and nodes in $\mathcal{L}(\mathcal{G})$.

Notation 4.21 Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph. Let $e_{(i,j)}, e_{(k,l)} \in \mathcal{E}$ an edge of \mathcal{G} . Let $\mathcal{L}(\mathcal{G}) = (\mathcal{L}(\mathcal{V}), \mathcal{L}(\mathcal{E}))$ be the line-graph of \mathcal{G} . For the sake of clarity, in the line-graph $\mathcal{L}(\mathcal{G})$ we will note $e_{(i,j)} \in \mathcal{L}(\mathcal{V})$ and $\hat{e}_{e_{(i,j)},e_{(k,l)}}$ the edge that connects the vertices $e_{(i,j)}$ and $e_{(k,l)}$ of $\mathcal{L}(\mathcal{G})$. For an undirected line-graph, we will note $\hat{e}_{e_{(\min(i,k),j)},e_{(\max(i,k),l)}}$.

Figure 4.16 illustrates Notation 4.21.

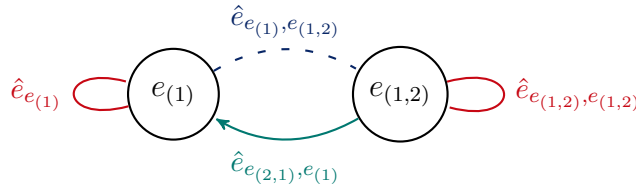


FIGURE 4.16: Notation of the line-graph with $e_{(1)} \in \mathcal{E}$ and $e_{(1,2)} \in \mathcal{E}$. Note that the directed edge $\hat{e}_{e_{(1,2)},e_{(1,2)}}$ is for the illustration.

Remark 4.22 The edges in the original graph \mathcal{G} contain all the edge information in the line-graph $\mathcal{L}(\mathcal{G})$.

Remark 4.23 An undirected graph leads necessarily to an undirected line-graph.

In the case of our matrix partitioning, we remove the self-edges, $e_{(i,i)}$, when we construct the line-graph. Anyway, METIS does not take into account edges, $e_{(i,i)}$, when partitioning a graph. They are virtual but exist. They will be taken into account when we construct the sub-matrices. Figure 4.17 illustrates the line-graph of the undirected graph given in Figure 4.13(a). The next

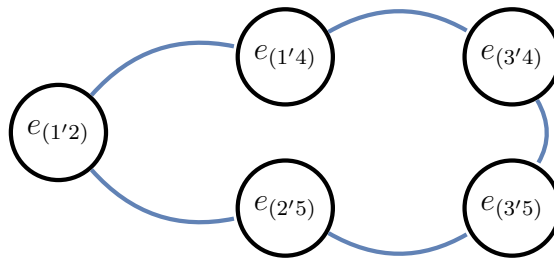
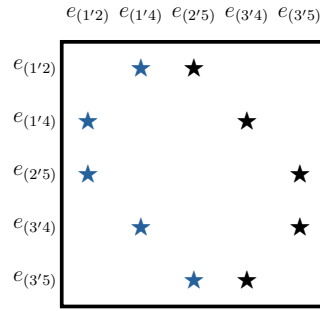


FIGURE 4.17: Line-graph $\mathcal{L}(\mathcal{G})$ of the undirected graph given in Figure 4.13(a)

step consists in partitioning the obtained line-graph, which is illustrated in Figure 4.17. Let us consider the following numbering: $1 \rightarrow e_{(1'2)}$, $2 \rightarrow e_{(1'4)}$, $3 \rightarrow e_{(2'5)}$, $4 \rightarrow e_{(3'4)}$, $5 \rightarrow e_{(3'5)}$, where $l \rightarrow e_{(i'j)}$ means that the node $e_{(i'j)} \in \mathcal{L}(\mathcal{V})$ is the node $\#l$. Considering the previous numbering, we give the corresponding adjacency structure in Figure 4.19. Figure 4.18 gives the adjacency matrix of the line-graph described in Figure 4.17 and illustrates its corresponding non-zero pattern. Notice that an edge $e_{(i,j)} \in \mathcal{E}$ in the original graph (see Figure 4.13(a)) corresponds to a vertex in $\mathcal{L}(\mathcal{G})$. Thus, when METIS partitions the graph, the cut-edges will correspond to the cut-nodes of the original graph. Figure 4.20 gives the partitioning given by METIS of the line-graph described in Figure 4.17 for 2 sub-parts. By the proposed partitioning

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

(a) Adjacency matrix (with self-loops removed)



(b) Non-zero pattern of the adjacency matrix given in Figure 4.18(a)

FIGURE 4.18: Adjacency matrix of the line-graph described in Figure 4.17 and the corresponding non-zero pattern

$$\begin{aligned} xadj &= 1 / 3 / 5 / 7 / 9 / 11 \\ adjncy &= \mathbf{2} / 3 / \mathbf{1} / 4 / \mathbf{1} / 5 / \mathbf{2} / 5 / 3 / 4 \end{aligned} \tag{4.1}$$

FIGURE 4.19: The CSR format for storing the sparse line-graph given in Figure 4.17

procedure, a vertex (node) v_i of \mathcal{G} (edge in $\mathcal{L}(\mathcal{G})$) can be in different partitions. In this case, the vertex is called an *interface node*.

Definition 4.24 (Interface node) Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph. Let $\mathcal{L}(\mathcal{G}) = (\mathcal{L}(\mathcal{V}), \mathcal{L}(\mathcal{E}))$ be the line-graph of \mathcal{G} . Let \mathcal{S} be a set of subsets S_1, S_2, \dots, S_s of $\mathcal{L}(\mathcal{V})$ as defined in Definition 4.9. Let \mathcal{S} be the partition of $\mathcal{L}(\mathcal{G})$. A node $i \in \mathcal{V}$ is considered interface if and only if there exists an edge-cut $\hat{e}_{e(i'j), e(i'l)} \in \mathcal{L}(\mathcal{E})$, $e(i'j) \in S_p \subset \mathcal{L}(\mathcal{E})$ and $e(i'l) \in S_q \subset \mathcal{L}(\mathcal{E})$ such that $S_p \neq S_q$.

For example, in the *twofold*-partitioning (see Figure 4.20(a), Page 123), there are 2 cut-edges : $\hat{e}_{e(1'4), e(3'4)}$ and $\hat{e}_{e(2'5), e(3'5)}$, and in the *threefold*-partitioning (see Figure 4.20(b), Page 123), there are 3 cut-edges : $\hat{e}_{e(1'4), e(3'4)}$, $\hat{e}_{e(2'5), e(3'5)}$ and $\hat{e}_{e(3'4), e(3'5)}$. Each edge-cut leads to an interface node. To construct the sub-domains it suffices to convert the line-graph of each partition

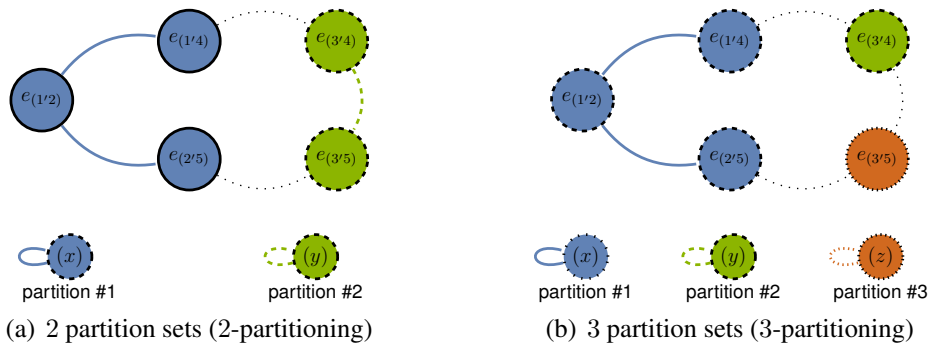


FIGURE 4.20: METIS (API C/C++) 2-partitioning and 3-partitioning of the line-graph described in Figure 4.17 using CSR structure given in Figure 4.19

and also to keep the information concerning the cut-edges of the line-graph, *i.e.*, the interface nodes.

Remark 4.25 Let $\mathcal{L}(\mathcal{G}) = (\mathcal{L}(\mathcal{V}), \mathcal{L}(\mathcal{E}))$ be a line-graph. A vertex $e_{(i'j)} \in \mathcal{L}(\mathcal{V})$ in the line-graph corresponds to an edge in \mathcal{G} and an edge $\hat{e}_{e_{(i'j)}, e_{(k'l)}} \in \mathcal{L}(\mathcal{E})$ is a vertex in \mathcal{G} . Considering Notation 4.21 and according to Definition 4.20, an edge of the line-graph $\mathcal{L}(\mathcal{G})$ has the form $\hat{e}_{e_{(i'j)}, e_{(i'l)}} \in \mathcal{L}(\mathcal{E})$, and gives the node i in \mathcal{G} .

Considering Definition 4.24 and Remark 4.25, from the METIS partitioning of $\mathcal{L}(\mathcal{G})$ described in Figure 4.20, we have obtained the sub-domains illustrated in Figure 4.21. The 2

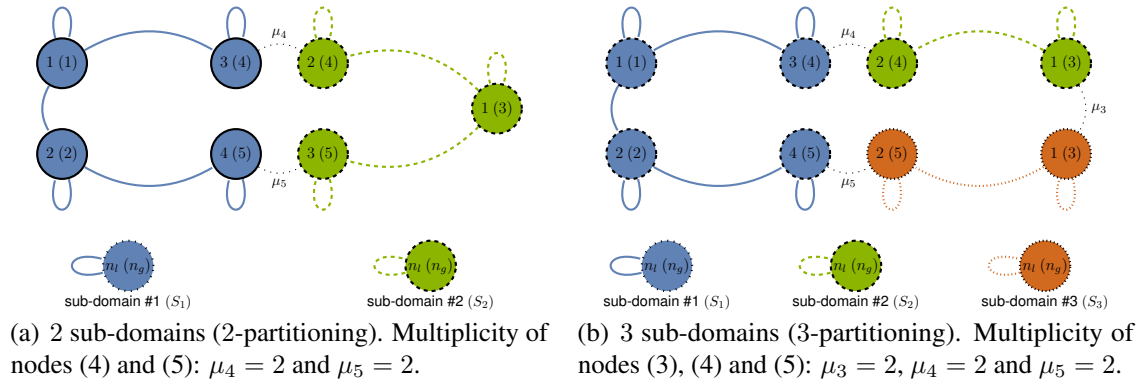


FIGURE 4.21: Sub-domains obtained from the METIS (API C/C++) 2-coloring and 3-coloring of the line-graph given in Figure 4.20. $n_l (n_g)$ stands for n_l : local node number, n_g : global node number

cut-edges $\hat{e}_{e_{(1'4)}, e_{(3'4)}}$ and $\hat{e}_{e_{(2'5)}, e_{(3'5)}}$ in Figure 4.20(a) lead respectively to the interface nodes (4) and (5). As we can see in Figure 4.21(a), the node (4) is split into 2 nodes: $3(4) \in S_1$ and $2(4) \in S_2$, and the node (5) is also split into 2 nodes: $4(5) \in S_1$ and $3(5) \in S_2$. Similarly, in Figure 4.21(b), we have three interface nodes: (3), split into $1(3) \in S_2$ and $1(3) \in S_3$, from the cut-edge $\hat{e}_{e_{(3'4)}, e_{(3'5)}}$, (4), split into $3(4) \in S_1$ and $1(4) \in S_2$, from the cut-edge $\hat{e}_{e_{(1'4)}, e_{(3'4)}}$ and (5), split into $4(5) \in S_1$ and $2(5) \in S_3$, from the cut-edge $\hat{e}_{e_{(2'5)}, e_{(3'5)}}$ (see Figure 4.20(b), Page 123). In the following, we consider non-interface nodes as *internal nodes*.

Knowing the partitioning, *i.e.*, the internal and interface nodes of each sub-domain, we now present how sub-structure (sub-domain) matrices are constructed. The self-edges have been removed before the partitioning (see Remark 4.19, Page 121), but must be taken into account for the sub-domains given in Figure 4.21, and therefore in the construction of the sub-matrices (we have considered that all diagonal values are non-zero). Moreover, each interface node, ' $n_l (n_g)$ ', is associated with a number called *multiplicity* and noted $\mu_{\#n_g}$ where n_g is the global number of the node and n_l the local number, and an edge $e(n_l^1 (n_g^1), n_l^2 (n_g^1))$ of two interface nodes ' $n_l^1 (n_g^1)$ ' and ' $n_l^2 (n_g^2)$ ' is associated with an *interface edge multiplicity*, noted $\mu_{\#e(n_g^1, n_g^2)}$.

Definition 4.26 (Interface node multiplicity μ_e) The multiplicity of an interface node ' $i_l (i_g)$ ' (i_l local number, i_g global number) is the number of associated sub-domains.

We define the associated weight as follows: $\omega_{\mu_{\#i_g}} = \frac{1}{\mu_{\#i_g}}$. In the following, $\{i_l (i_g), \mu_{\#i_g}\}$ and $\{i_l (i_g), \omega_{\mu_{\#i_g}}\}$ will denote the interface node associated with the multiplicity $\mu_{\#i_g}$ and associated with the weight $\omega_{\mu_{\#i_g}}$ respectively.

Definition 4.27 (Interface edge multiplicity μ) Let us define $\mu_{\#e(i_g^1, i_g^2)}$ the multiplicity of the edge, $e(i_g^1, i_g^2)$, formed by two interface nodes ' $i_l^1 (i_g^1)$ ' and ' $i_l^2 (i_g^2)$ '. The multiplicity $\mu_{\#e(i_g^1, i_g^2)}$ is defined as the number of sub-domains containing the edge $e(i_g^1, i_g^2)$. Note that if $e(i_g^1, i_g^2)$ is in only one sub-domain then $\mu_{\#e(i_g^1, i_g^2)} = 1$.

The notions of node multiplicity and edge multiplicity are crucial. In fact, values at the interface nodes are weighted by the associated weight. Let us illustrate the process with a simple example. Let $M \in \mathbb{K}^{n \times n}$ be a square symmetric matrix, $\mathcal{S} = \{S_1, S_2, S_3\}$ be the partition sets of the graph associated with M , and $\{a_l^1 (a_g), \mu_{\#a_g}\} \in S_1$, $\{a_l^2 (a_g), \mu_{\#a_g}\} \in S_2$, $\{a_l^3 (a_g), \mu_{\#a_g}\} \in S_3$ an interface node a_g split into S_1, S_2 , and S_3 , and $\{b_l^1 (b_g), \mu_{\#b_g}\} \in S_1$, $\{b_l^2 (b_g), \mu_{\#b_g}\} \in S_2$ another interface node b_g split into S_1, S_2 . We have $\mu_{\#a_g} = 3$ and $\mu_{\#b_g} = 2$. In the sub-matrices $M_{S_1}, M_{S_2}, M_{S_3}$, we have the following contributions:

- $M_{S_1}(a_l^1, a_l^1) = \frac{M(a_g, a_g)}{3}$, $M_{S_2}(a_l^2, a_l^2) = \frac{M(a_g, a_g)}{3}$ and $M_{S_3}(a_l^3, a_l^3) = \frac{M(a_g, a_g)}{3}$, such that $M(a_g, a_g) = M_{S_1}(a_l^1, a_l^1) + M_{S_2}(a_l^2, a_l^2) + M_{S_3}(a_l^3, a_l^3)$,
- $M_{S_1}(b_l^1, b_l^1) = \frac{M(b_g, b_g)}{2}$, $M_{S_2}(b_l^2, b_l^2) = \frac{M(b_g, b_g)}{2}$, such that $M(b_g, b_g) = M_{S_1}(b_l^1, b_l^1) + M_{S_2}(b_l^2, b_l^2)$,
- if the nodes a_l^i and b_l^i are not linked in S_i then $M_{S_i}(a_l^i, b_l^i) = M_{S_i}(b_l^i, a_l^i) = 0$, otherwise

$$M_{S_i}(a_l^i, b_l^i) = M_{S_i}(b_l^i, a_l^i) = \frac{M(a_g, b_g)}{\mu_{\#e(a_g, b_g)}}$$

Proposition 4.28 Let $\{a_l^k (a_g), \mu_{\#a_g}\} \in S_k$ and $\{b_l^k (b_g), \mu_{\#b_g}\} \in S_k$ two interface nodes where S_k is one of the sub-domain. We have

$$\begin{cases} M_{S_k}(a_l^k, a_l^k) = \frac{M(a_g, a_g)}{\mu_{\#a_g}} \text{ and } M_{S_k}(b_l^k, b_l^k) = \frac{M(b_g, b_g)}{\mu_{\#b_g}}, \\ M_{S_k}(a_l^k, b_l^k) = \frac{M(a_g, b_g)}{\mu_{\#e(a_g, b_g)}} \text{ and } M_{S_k}(b_l^k, a_l^k) = \frac{M(b_g, a_g)}{\mu_{\#e(b_g, a_g)}}. \end{cases} \quad (4.2)$$

such that

$$\begin{cases} M(a_g, a_g) = M_{S_k}(a_l^k, a_l^k) + \sum_{j=1, j \neq k}^s M_{S_j}(a_l^j, a_l^j), & s = \mu_{\#a_g} \\ M(a_g, b_g) = M_{S_k}(a_l^k, b_l^k) + \sum_{j=1, j \neq k}^s M_{S_j}(a_l^j, b_l^j), & s = \mu_{\#e(a_g, b_g)}, \\ M(b_g, b_g) = M_{S_k}(b_l^k, b_l^k) + \sum_{j=1, j \neq k}^s M_{S_j}(b_l^j, b_l^j), & s = \mu_{\#b_g} \\ M(b_g, a_g) = M_{S_k}(b_l^k, a_l^k) + \sum_{j=1, j \neq k}^s M_{S_j}(b_l^j, a_l^j). & s = \mu_{\#e(a_g, b_g)}. \end{cases} \quad (4.3)$$

where $\{a_l^i (a_g), \mu_{\#a_g}\} \in S_i$ and $\{b_l^j (b_g), \mu_{\#b_g}\} \in S_j$. When the matrix is symmetric: $M_{S_k}(a_l^k, b_l^k) = M_{S_k}(b_l^k, a_l^k)$.

Considering Proposition 4.28 and according to the *twofold*-partitioning given in Figure 4.21(a), we obtained the sub-matrices illustrated in Figure 4.22 for 2 sub-domains. And from the

threefold-partitioning given in Figure 4.21(b), we have the sub-matrices described in Figure 4.23 for 3 sub-domains.

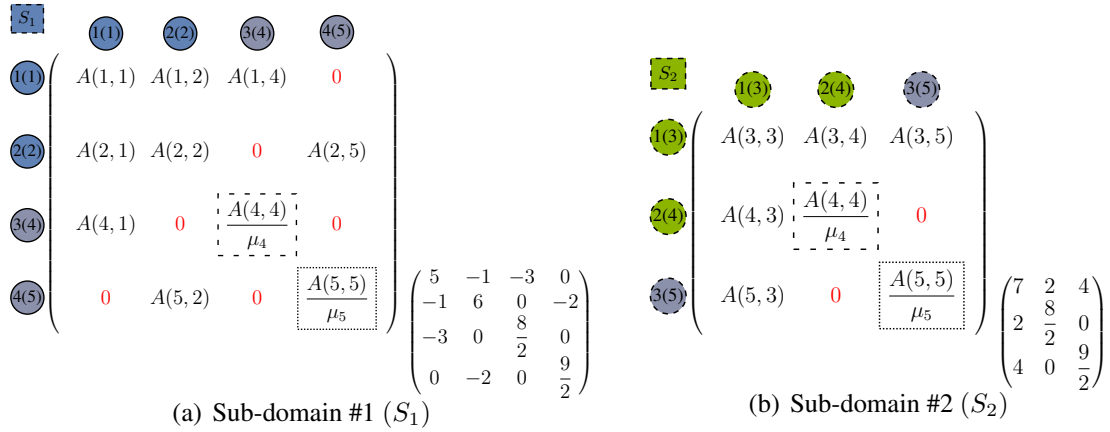


FIGURE 4.22: Sub-domain matrices of the 2-partitioning described in Figure 4.21(a)

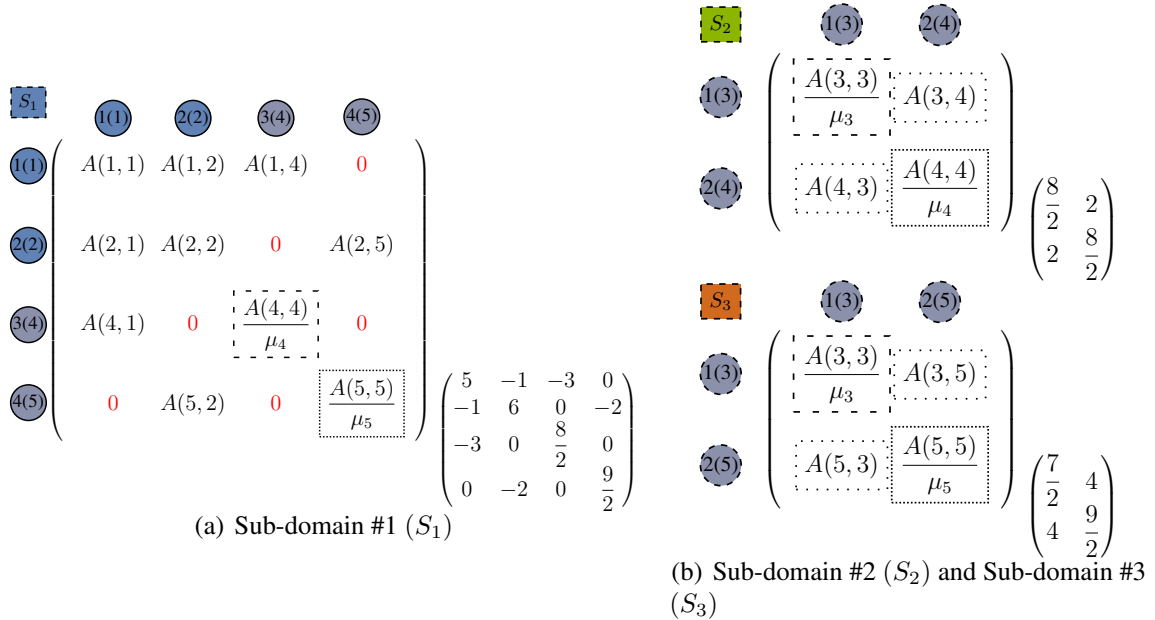


FIGURE 4.23: Sub-domain matrices of the 3-partitioning described in Figure 4.21(b)

Remark 4.29 The presented procedure for partitioning can be extended to a directed graph. The crucial point with a digraph is to take into account the definition of adjacency in a directed line-graph. Thus, in the following, we give the definition of a line-digraph, $\mathcal{L}^d(\mathcal{G})$ of a directed graph \mathcal{G}^d .

Definition 4.30 (Directed line-graph) The directed line-graph, $\mathcal{L}^d(\mathcal{G})$, of a directed graph \mathcal{G} is a graph such that each vertex of $\mathcal{L}^d(\mathcal{G})$ corresponds to an edge of \mathcal{G} , and there is $e_{(a,b)}$ an edge from v_a to v_b of $\mathcal{L}^d(\mathcal{G})$ if the corresponding edges, $e_{(a_i,a_j)}$ and $e_{(b_i,b_j)}$ form a length-two directed graph path from $e_{(a_i,a_j)}$ to $e_{(b_i,b_j)}$ in \mathcal{G} [254] [255].

In order to avoid deadlock, the list of neighboring interfaces is re-ordering using the algorithm proposed by D.J.A. Welsh and M.B. Powell [256] for coloring the graph optimally.

4.3.6 Evaluation of the partitioning

To have an idea of the preprocessing step, we evaluate such data partitioning. The evaluation has been performed in *Platform-1*, equipped with an Intel Core i7 920 2.67GHz processor, which has eight cores composed of four physical cores and four logical cores, and 5.8 GB RAM memory. To evaluate our analysis on large-scale engineering problems, we used a set of matrices from the University of Florida repository (see Table 3.11, Page 68). The time in seconds (s) used for partitioning into row-bands, column-bands, and sub-structures is reported in Table 4.2. The first column gives the name of the matrix. In the second column (**metis**), the sub-structuring splitting using METIS software for coloring the graph is collected. The band column (**band-c**) splitting is given in the third column. The last column gives the band row (**band-r**) splitting.

Matrix	2-partitioning			4-partitioning			8-partitioning		
	metis	band-c	band-r	metis	band-c	band-r	metis	band-c	band-r
qa8fm	3.16	1.15	1.20	4.6	1.19	1.11	6.64	1.28	1.51
2cubes_sphere	5.4	2.49	2.49	5.14	2.52	2.49	6.08	2.58	2.49
thermomech_TK	1.58	1.03	1.0	2.83	1.03	1.0	1.88	1.07	1.0
cfid2	6.74	3.72	2.06	8.28	2.52	2.04	10.84	2.25	2.61
thermal2	12.53	6.98	6.53	15.72	7.37	7.63	17.33	8.01	7.75
af_shell8	68.05	26.93	27.65	86.18	28.32	51.36	104.41	28.72	27.38
finan512	0.83	0.44	0.42	1.01	0.45	0.42	1.21	0.48	0.42

TABLE 4.2: Execution time used for partitioning (s)

For the sub-structuring splitting, see Section 5.4, Page 174.

4.4 Mathematical model

Considering the parallel platform, $\mathcal{PS}_{system}\{p\}$, is capable of performing p processes of execution of \mathcal{PS} . Let E be a *Banach space*, defined as a finite product of Banach spaces:

$$E = \prod_{s=1}^p E^{\{s\}} \subset \prod_{s=1}^p \mathbb{K}^{n^{\{s\}}} \quad (4.4)$$

where $s \in \{1, \dots, p\}$ and $p \in \mathbb{N}^*$. The decomposition of any vector $X \in E$ can be written as the following sequence from $X = (x^{\{1\}}, \dots, x^{\{s\}}, \dots, x^{\{p\}})$, where $x^{\{s\}} \in E^{\{s\}}, \forall s \in \{1, \dots, p\}$. Each $E^{\{s\}}$ ($s \in \{1, \dots, p\}$) is a Banach space with respect to the norm $|\cdot|_i$. The canonical vectorial norm $c(\cdot)$ of $X \in E$ is a subset of \mathbb{K}^{p*} , and is defined as follows

$$\forall X \in E, c(X) = (|x^{\{1\}}|^{\{1\}}, \dots, |x^{\{s\}}|^{\{s\}}, \dots, |x^{\{p\}}|^{\{p\}}). \quad (4.5)$$

Let \mathcal{T} be a linear mapping from $\mathcal{D}(\mathcal{T}) \subset E$ to E , whose domain of definition is $\mathcal{D}(\mathcal{T})$, $\mathcal{D}(\mathcal{T}) \neq \emptyset, \mathcal{T} : \mathcal{D}(\mathcal{T}) \subset E \mapsto E$.

Let us consider the *fixed point* problem: $x^* = \mathcal{T}(x^*), x^* \in \mathcal{D}(\mathcal{T})$. Considering the decomposition of the space E , the function \mathcal{T} can be decomposed as follows:

$$\mathcal{T}(x) = (\mathcal{T}^{\{1\}}(x), \dots, \mathcal{T}^{\{s\}}(x), \dots, \mathcal{T}^{\{p\}}(x)). \quad (4.6)$$

Each set $\mathcal{T}^{\{s\}}$ is a function of $\mathcal{D}(\mathcal{T})$ with values in $E^{\{s\}} \cap \mathcal{D}(\mathcal{T})$. In this chapter, $\mathcal{T}^{(k)}$, $\mathcal{T}^{\{k\}}$ and \mathcal{T}^k will denote respectively a linear matrix or a function depending on the iteration k , depending on the process k and the k^{th} power of \mathcal{T} . So, $\mathcal{T}^{\{k\}(k)}$ and $\mathcal{T}^{\{k\}^k}$ denote respectively the linear matrix or the function associated with the process k depending on the iteration k and the k^{th} power of $\mathcal{T}^{\{k\}}$. For the sake of generality, we consider in the following linear and non-linear functions. However, we will clearly state the appropriate function if necessary. Before discussing the convergence analysis, we will recall basic definitions.

Definition 4.31 Let $p \in [1, +\infty[$. We define the l_p norms in \mathbb{K}^n by $\|x\|_p = \left(\sum_{s=1}^n |x_s|^p \right)^{1/p}$.

The particular l_2 and l_∞ norms are respectively called *Euclidean norm* and *maximum norm*.

They are respectively defined as follows $\|x\|_2 = \sqrt{\sum_{s=1}^n |x_s|^2}$, $\|x\|_\infty = \max_{1 \leq s \leq n} \|x_s\|$.

Definition 4.32 A sequence of vectors $(x^{(k)})_{k \in \mathbb{N}}$ converges to a vector x^* if each component $x_i^{(k)}$ converges to x_i^* , then $\lim_{k \rightarrow \infty} x^{(k)} = x^* \Leftrightarrow \forall i \in \{1, \dots, n\}, \lim_{k \rightarrow \infty} x_i^{(k)} = x_i^*$, which proved that $\lim_{k \rightarrow \infty} x^{(k)} = x^* \Leftrightarrow \lim_{k \rightarrow \infty} \|x^{(k)} - x^*\| = 0$ for any arbitrary norm.

Now we introduce the tools which will help us model the parallelism in mathematical terms. First, we model the behavior of synchronous algorithms, followed by the chaotic behavior of asynchronous algorithms.

4.4.1 Synchronous iterations

Let us consider a problem defined on a set X which has an unique solution x^* . As synchronous algorithms use the same mathematical model and have the same convergence results as their sequential counterparts, the prerequisites are the same as those presented in Section 3.2, Page 41. From a mathematical point of view, an iterative algorithm is defined as a sequence of successive functions $(\mathcal{T}^{(k)})_{k \in \mathbb{N}}$ from an arbitrary initial guess of the solution, where $\mathcal{T} : X \mapsto X$.

The mathematical formulation of the algorithm, with an initial guess x_0 , is given as

$$\begin{cases} x^{(0)} = x_0 \\ x^{(k+1)} = \mathcal{T}^{(k)}(x^{(k)}), \quad k \in \mathbb{N}^* \end{cases} \quad (4.7)$$

We called *iteration* an application of a function $\mathcal{T}^{(k)}$, $k \in \mathbb{N}$ and $x^{(k)}$ the corresponding approximation of the solution x^* . The convergence is detected when an accurate approximation of the solution has been found, with a given tolerance threshold. We say that the algorithm has converged. When \mathcal{T} is constant, *i.e.*, $\forall k \in \mathbb{N}, \mathcal{T}^{(k)} = \mathcal{T}$, the algorithm is stationary such as the Jacobi iterative method. Under this condition, the algorithm becomes a fixed point problem. A *parallel iterative algorithm* is an iterative algorithm where the applications of functions $\mathcal{T}^{(k)}$, $k \in \mathbb{N}$, are performed in parallel. Let us define $X^{\{s\}} \subset E$, a set associated with the process $s \in \mathcal{PS}$. In most cases, $X^{\{s\}}$ is a subset of X . In addition to the variables of

a sequential algorithm, some artificial ones may be used for the needs of the parallel algorithm.

Let \mathcal{P} be a linear (or non-linear) mapping from $E = \prod_{s=1}^p E^{\{s\}}$ to E ,

$$\mathcal{P} : X^{\{1\}} \times \dots \times X^{\{s\}} \times \dots \times X^{\{p\}} \mapsto X. \quad (4.8)$$

where $s \in \{1, \dots, p\}$. From the mapping (4.8) we extract the problem variable from the parallel problem, such that each process s is associated with a sequence of functions $\mathcal{T}^{\{s\}(k)}$, $k \in \mathbb{N}$, with

$$\mathcal{T}^{\{s\}(k)} : X^{\{1\}} \times \dots \times X^{\{s\}} \times \dots \times X^{\{p\}} \mapsto X^{\{s\}}. \quad (4.9)$$

Definition 4.33 (Parallel synchronous algorithm) A classical synchronous parallel algorithm constructs recursively for each process $s \in \{1, \dots, p\}$ a sequence of iterations $(x^{(k)})_{k \in \mathbb{N}}$, from an initial guess solution $x^{\{s\}(0)} = x_0$, with $x^{\{s\}(k)} \in X^{(k)}$:

$$x^{\{s\}(k+1)} = \mathcal{T}^{\{s\}(k)} \left(x^{\{1\}(k)}, \dots, x^{\{s\}(k)}, \dots, x^{\{p\}(k)} \right), \quad k \in \mathbb{N}^*. \quad (4.10)$$

At the iteration $k \in \mathbb{N}$, the approximate solution of the problem is expressed as follows

$$x^{(k)} = \mathcal{P} \left(x^{\{1\}(k)}, \dots, x^{\{s\}(k)}, \dots, x^{\{p\}(k)} \right).$$

In the synchronous algorithm (4.10), at iteration $k \in \mathbb{N}$, each process s locally performs the calculations associated with its function $\mathcal{T}^{\{s\}(k)}$ and then shares the result with cooperating processes. The functions $(\mathcal{T}^{\{s\}})_{s \in \{1, \dots, p\}}$ are considered independent. The exchange between cooperating processes is only carried out at the end of the local application of the functions in order to prepare the next iteration. Indeed, the iterations are performed synchronously (**SISC** or **SIAC**). In general, for the sake of optimization, a given process $s_1 \in \{1, \dots, p\}$ will send its result to process $s_2 \in \{1, \dots, p\}$, $s_1 \neq s_2$ only if its associated function $\mathcal{T}^{\{s_1\}}$ (or $\mathcal{T}^{\{s_1\}(k)}$, $k \in \mathbb{N}$, for non-linear functions) depends on its s_2^{th} component. With the same procedure, and for the sake of comparison, we now give the sequential version of the algorithm (4.10). Let us define $\widehat{\mathcal{T}}$ a mapping from $\mathcal{D}(\widehat{\mathcal{T}}) \subset E$ to E ,

$$\begin{aligned} \widehat{\mathcal{T}}^{(k)} : \widehat{X} &\mapsto \widehat{X} \\ \widehat{x} &\mapsto \widehat{\mathcal{T}}^{(k)}(\widehat{x}) \end{aligned} \quad (4.11)$$

where $\widehat{X} = X^{\{1\}} \times \dots \times X^{\{s\}} \times \dots \times X^{\{p\}}$ and $\widehat{x} = (x^{\{1\}}, \dots, x^{\{s\}}, \dots, x^{\{p\}})$. Hence,

$$\widehat{\mathcal{T}}^{(k)}(\widehat{x}) = \left(\mathcal{T}^{\{1\}(k)} \left(x^{\{1\}}, \dots, x^{\{s\}}, \dots, x^{\{p\}} \right), \dots, \mathcal{T}^{\{p\}(k)} \left(x^{\{1\}}, \dots, x^{\{s\}}, \dots, x^{\{p\}} \right) \right).$$

An equivalent sequential algorithm to the synchronous parallel algorithm (4.10) is obtained by applying \widehat{X} and $\widehat{\mathcal{T}}$ into the iterations given in (4.7). In general, the set \widehat{X} is different to the set X . This equivalent version may need more variables compared to the pure sequential algorithm (see Algorithm 3.1, Page 49). However, this algorithm presents a constraint, which compels the function \mathcal{P} to be surjective, *i.e.*, $X \subset \widehat{X}$ (all values of the problem domain can be represented). Let remark that if $\widehat{X} = X$, the function \mathcal{P} is the identity.

Proposition 4.34 (Identical behavior, sequential and synchronous) *Both the sequential and parallel algorithms have the same mathematical model and convergence results.*

Considering the Proposition 4.34, to demonstrate the convergence of the parallel model, it suffices to prove the sequential version.

Algorithm 4.3: Algorithm of sequential iterative methods

input : \mathcal{T} : function of the iterative algorithm, $x^{(0)} \in \mathcal{D}(\mathcal{T})$: initial arbitrary solution,
 ε : tolerance threshold, K : maximum number of iterations
output : x : solution
variable: $k, convg$

```

1  $convg \leftarrow false; k \leftarrow 0$ 
2 while .not. convg .OR.  $k < K$  do
3    $x^{(k+1)} \leftarrow \mathcal{T}^{(k)}(x^{(k)})$  //  $\mathcal{T}^{(k)} = \mathcal{T}$  (for stationary)
4    $convg \leftarrow (\|x^{(k)} - x^{(k-1)}\| \leq \varepsilon)$ 
5    $k \leftarrow k + 1$ 
6 end

```

Theorem 4.35 (Convergence results with contractive application) *Suppose that \mathcal{T} is a contraction for a norm $\|\cdot\|$ and α is its constant of contraction. The Algorithm 4.3 generates a convergent sequence $(x^{(k)})_{k \in \mathbb{N}}$ whose limit is the unique fixed point x^* of \mathcal{T} , i.e.*

$$\lim_{k \rightarrow \infty} x^{(k)} = x^* \quad \text{where } x^* = \mathcal{T}(x^*). \quad (4.12)$$

Moreover, we have the estimation (for this norm) at the iteration k ,

$$\|x^{(k)} - x^*\| \leq \frac{\alpha}{1 - \alpha} \|x^{(k)} - x^{(k-1)}\|. \quad (4.13)$$

In addition, if \mathcal{T} is α -Lipschitzian, the approximation error at the iteration k is bounded by

$$\|x^{(k)} - x^*\| \leq \alpha^k \|x_0 - x^*\|. \quad (4.14)$$

This important theorem gives a sufficient condition for the convergence of sequential stationary iterative algorithms, and thus for the parallel synchronous algorithm. For more developments references: Y. Saad [69] and J.M. Bahi *et al.* [216]. The theorem also marks the uniqueness of the fixed point of a mapping \mathcal{T} .

4.4.2 Totally asynchronous iterations (T-AI)

In general, except for rare cases, asynchronous iterative algorithms can work as synchronous algorithms with minor changes, including the addition of synchronization points. On the contrary, a synchronous iterative algorithm is often hard to write in asynchronous iterations because of their special conditions. Some synchronous algorithms, such as parallel iterative Krylov methods have no answer with asynchronous schemes. The study of these methods in an asynchronous environment remains a real challenge. Nevertheless, there are some synchronous algorithms that can work with asynchronous iterations without any change. In the following, we will describe the parallel model and the chaotic behavior of asynchronous algorithms mathematically. We are interested in the stationary fixed point algorithms, which correspond to a constant function \mathcal{T} , i.e., $\forall k, \mathcal{T}^{(k)} = \mathcal{T}$.

Definition 4.36 (Steering) A *steering* Σ of the algorithm is a sequence $(\sigma^{(k)})_{k \in \mathbb{N}}$ of non-empty subsets of $\{1, \dots, p\}$, such as

$$\forall s \in \{1, \dots, p\}, \{k \in \mathbb{N} \mid s \in \sigma^{(k)}\} \text{ is infinite set,} \quad (4.15)$$

$$\text{i.e., } \text{card} \{k \in \mathbb{N} \mid s \in \sigma^{(k)}\} = +\infty. \quad (4.16)$$

The *steering* of the algorithm represents which processes update their components at the iteration k .

Definition 4.37 (A sequence of delay of processes) A *sequence of delays* \mathcal{R} is a sequence $(r^{(k)})_{k \in \mathbb{N}}$, such as

$$\forall k \in \mathbb{N}, r^{(k)} = (r^{\{1\}(k)}, \dots, r^{\{j\}(k)}, \dots, r^{\{p\}(k)}) \in \mathbb{N}^p, \quad j \in \{1, \dots, p\}. \quad (4.17)$$

For every $s, i \in \{1, \dots, p\}$, let us define a function $\tau^{\{s\}} : \mathbb{N} \rightarrow \mathbb{N}^p$, of components $\tau^{\{s|i\}} : k \mapsto k - r^{\{s|i\}(k)}$. For processes $s, i \in \{1, \dots, p\}$, $(\tau^{\{s|i\}(k)})_{k \in \mathbb{N}}$ is a sequence of integers, such as

$$\forall k \in \mathbb{N}, \tau^{\{s\}(k)} = (\tau^{\{s|1\}(k)}, \dots, \tau^{\{s|j\}(k)}, \dots, \tau^{\{s|p\}(k)}) \in \mathbb{N}^p, \quad j \in \{1, \dots, p\}. \quad (4.18)$$

$\tau^{\{s|i\}(k)}$ represents the iteration number of the data coming from process i and available on process s at iteration k and satisfies the following conditions:

$$\forall k \in \mathbb{N}, \forall s, i \in \{1, \dots, p\}, 0 \leq \tau^{\{s|i\}(k)} \leq k, \quad (4.19)$$

and

$$\forall k \in \mathbb{N}, \lim_{k \rightarrow \infty} \tau^{\{s|i\}(k)} = +\infty. \quad (4.20)$$

The mathematical model of the asynchronous algorithms presented in this thesis follows the model of D.P. Bertsekas and J.N. Tsitsiklis [179] (also see J.-C. Miellou [186] and G.M. Baudet [187]).

For more developments, proofs and more convergence results, see [177] [186] [187] [179] [216].

Definition 4.38 (Parallel asynchronous algorithm) Let a process $s \in \{1, \dots, p\}$. Let $X^{(0)} = (x^{\{s\}(0)}) \in \mathcal{D}(\mathcal{T})$ a given initial solution, Σ a sequence of steering and \mathcal{R} a sequence of delays. A classical asynchronous parallel algorithm constructs recursively for each process $s \in \{1, \dots, p\}$ a sequence of iterations $(x^{\{s\}(k)})_{k \in \mathbb{N}}$, from an initial guess solution $x^{\{s\}(0)} = x_0$, with $x^{\{s\}(k)} \in X^{(k)}$:

$$\begin{cases} x^{\{s\}(0)} & = x_0^{\{s\}} \\ x^{\{s\}(k+1)} & = \begin{cases} \mathcal{T}^{\{s\}}(x^{\{1\}(\tau^{\{s|1\}(k)})}, \dots, x^{\{i\}(\tau^{\{s|i\}(k)})}, \dots, x^{\{p\}(\tau^{\{s|p\}(k)})}) & \text{if } s \in \sigma^{(k)} \\ x^{\{s\}(k)} & \text{if } s \notin \sigma^{(k)} \end{cases} \end{cases} \quad (4.21)$$

On the one hand, the sequence of *steering* Σ (4.15) gives the order of calculations in parallel. The indices in $\sigma^{(k)}$ consists of the components relaxed in parallel at the iteration k . On the other hand, the sequence of *delays* \mathcal{R} (4.17) provides information on the availability

of data, which allows to model both the absence of synchronization between cooperating processes and the delays due to communication latencies. The condition (4.16) on steering Σ implies that no process will definitively stop refreshing its corresponding components. The condition (4.20) means that old data are cleaned out of the process, *i.e.*, new data will always be furnished to the process, and no process will have a piece of data blocked at one iteration. Thus, the refreshment of the components of the iterated vector always use recent data. The condition (4.19) corresponds to the fact that data used at the time k must have been generated before time k , *i.e.*, time is ascending. When conditions (4.16) and (4.20) are satisfied, we talk about a *totally asynchronous algorithm*. For *partially asynchronous algorithms*, the conditions given in Definition 4.37 become the following:

$$\forall k \in \mathbb{N}, \forall s, i \in \{1, \dots, p\}, \exists \gamma \in \mathbb{N}, \gamma \geq 0, \quad k - \gamma + 1 \leq \tau^{\{s|i\}(k)} \leq k, \quad (4.22)$$

and

$$\forall s \in \{1, \dots, p\}, \tau^{\{s|s\}(k)} = k. \quad (4.23)$$

The condition (4.23) means that old data are cleaned out from the process after at most γ iterations. The condition (4.22) ensures that the own calculated components of a process are never obsolete.

Remark 4.39 For all iterations $k \in \mathbb{N}$ and all processes $s, i \in \{1, \dots, p\}$, if $\tau^{\{s|i\}(k)} = k$, *i.e.*, $r^{\{s|i\}(k)} = 0$, *i.e.*, we consider a sequence of zero delays [257], then the algorithm (4.21) describes synchronous parallel algorithms. In this context, the conventional sequential relaxation methods are modeled by specific choices of steering Σ :

- For the *Jacobi* method: $\forall k \in \mathbb{N}, \sigma^{(k)} = \{1, \dots, p\}$,
- For the *Gauss-Seidel* method: $\forall k \in \mathbb{N}, \sigma^{(k)} = 1 + (k \bmod p)$.

Convergence results In the following, we present some convergence results for asynchronous algorithms.

Assumption 4.40 Let $E = \prod_{s=1}^p E^{\{s\}} \subset \prod_{s=1}^p \mathbb{K}^{n_{[s]}}$. $\forall i \in \{1, \dots, p\}, \forall k \in \mathbb{N}, \exists E^{\{s\}(k)} \subset E^{\{i\}}$, a sequence of nested sets, such as $E^{\{i\}(k+1)} \subset E^{\{i\}(k)}$ and $\mathcal{T}(E^{(k)}) \subset E^{(k+1)}$, where $E^{(k)} = \prod_{i=1}^p E^{\{i\}(k)}$.

We give in Assumption 4.41 another equivalent approach of the conditions of Assumption 4.40.

Assumption 4.41 There exists a sequence of non-empty sets $(\widehat{X}^{(k)})_{k \in \mathbb{N}}$ such as, $\forall k \in \mathbb{N}, \widehat{X}^{(k+1)} \subset \widehat{X}^{(k)}$, with $\widehat{X}^{(0)} \subset \widehat{X}$, such that satisfies the following conditions:

- $\forall k \in \mathbb{N}, \hat{x} \in \widehat{X}^{(k)}, \widehat{\mathcal{T}}(\hat{x}) \in \widehat{X}^{(k+1)}$.
- For each $k \in \mathbb{N}$, there exists a set $X^{\{s\}(k)} \subset X^{\{s\}}, s \in \{1, \dots, p\}$, such that

$$\widehat{X}^{(k)} = X^{\{1\}(k)} \times \dots \times X^{\{i\}(k)} \times \dots \times X^{\{p\}(k)}.$$

Considering the first condition, if $(y^{(k)})_{k \in \mathbb{N}}$ is a sequence such that for each $k, y^{(k)} \in \widehat{X}^{(k)}$, then every limit point of $(y^{(k)})_{k \in \mathbb{N}}$ is a fixed point of $\widehat{\mathcal{T}}$.

Theorem 4.42 (Convergence results of parallel asynchronous iterations) Under Assumption 4.40 (Assumption 4.41) and under hypothesis (4.16) and (4.20), from any initial guess $x^{(0)} \in E^{(0)}$ ($\hat{x}^{(0)} \in \hat{X}^{(0)}$), the limit points of the sequence $(x^{(k)})_{k \in \mathbb{N}}$ ($(\hat{x}^{(k)})_{k \in \mathbb{N}}$) produced by an asynchronous iterative algorithm (4.21) are the solution of a fixed point problem (4.12) (fixed point of \hat{T}).

D. Chazan and W. Miranker [177] have shown that the sufficient conditions of the Theorem 4.42 are necessary when \mathcal{T} is a linear mapping and $n_i = 1$ for each $i \in \{1, \dots, p\}$. Convergence results have been proven by D. Chazan and W. Miranker [177] for linear systems, J.-C. Miellou [186], G.M Baudet [187], and M.N. El Tarazi [191] [192] for contracting operators and J.-C. Miellou [186] and D.P. Bertsekas [233] for monotone iterations.

Definition 4.43 (Weighted maximum norm) A weighted maximum norm $\|\cdot\|_\infty^w$ is defined in such a way that,

$$\forall x \in E, \quad \|x\|_\infty^w = \max_{1 \leq i \leq p} \frac{|x^{\{i\}}|_i}{w_i}. \quad (4.24)$$

where $|\cdot|_i$ is a norm defined in $E^{\{i\}}$ and w is a positive vector, such as

$$w = (w_1, \dots, w_i, \dots, w_p) \in \mathbb{K}^p$$

with $w_i > 0, \forall i \in \{1, \dots, p\}$. And thus, for T a $n \times n$ matrix,

$$\|T\|_\infty^w = \max_{1 \leq i \leq p} \frac{1}{w_i} \sum_{j=1}^n |t_{ij}| w_j. \quad (4.25)$$

Theorem 4.44 (Convergence with weighted maximum norm) If \mathcal{T} has a fixed point and if \mathcal{T} is a contraction mapping in x^* with respect to the weighted maximum norm $\|\cdot\|_\infty^w$, i.e.

$$\exists \alpha \in]0, 1[, \quad \forall x \in \mathcal{D}(\mathcal{T}), \quad \|\mathcal{T}(x) - \mathcal{T}(x^*)\|_\infty^w \leq \alpha \|x - x^*\|_\infty^w \quad (4.26)$$

then the sequence $(x^{(k)})_{k \in \mathbb{N}}$ generated by a parallel asynchronous algorithm (4.21) converges to the unique fixed point x^* of \mathcal{T} with respect to the weighted maximum norm $\|\cdot\|_\infty^w$.

The Theorem 4.44 is the asynchronous equivalent of Theorem 4.35. J.-C. Miellou [186] has demonstrated this theorem.

The weights w_i are obtained by the *Perron-Frobenius* theorem (Theorem 4.47) in case of monotone operators. Let us fix the following notations. Let $v \in \mathbb{R}^n$ and $w \in \mathbb{R}^n$ two vectors, $A \in \mathbb{R}^{n \times n}$, and β noted as a scalar but $\beta \in \mathbb{R}^n$.

- $v > w$ means $\forall i \in \{1, \dots, n\}, v_i > w_i$,
- $v \geq w$ means $\forall i \in \{1, \dots, n\}, v_i \geq w_i$,
- $v \geq \beta$ means $\forall i \in \{1, \dots, n\}, v_i \geq \beta$,
- $A \geq \beta$ means $\forall i, j \in \{1, \dots, n\}, A_{i,j} \geq \beta$; if $\beta = 0 \in \mathbb{R}^n$, A is a *non-negative matrix*.

Proposition 4.45 If an $n \times n$ matrix is irreducible and if some non-negative vector $w \geq 0$, such as $Tw = 0$, then $w = 0$.

Proof. This is a direct consequence of (4.25). □

Proposition 4.46 *If T and N are $n \times n$ matrices and $w \geq 0$ a non-negative vector, then the following are equivalent:*

1. *If $T \geq 0$, then $\|T\|_\infty^w = \|Tw\|_\infty^w$.*

Proof. By (4.25), $\|T\|_\infty^w = \max_i \frac{1}{w_i} \sum_{j=1}^n t_{ij} w_j = \max_i \frac{1}{w_i} (Tw)_i = \|Tw\|_\infty^w$. □

2. $\| \|T\|_\infty^w = \|T\|_\infty^w$

Proof. This is a direct consequence of (4.25). □

3. *Let $T \geq 0$. Then, for any scalar $\zeta > 0$, $\|T\|_\infty^w \leq \zeta$ iff $Tw \leq \zeta w$.*

Proof. Using 1., $\|T\|_\infty^w \leq \zeta$ iff $\|Tw\|_\infty^w \leq \zeta \Leftrightarrow Tw \leq \zeta w$. □

4. $\rho(T) \leq \|T\|_\infty^w$.

Proof. True for any norm $\|\cdot\|$. □

5. *If $T \geq N \geq 0$, then $\|T\|_\infty^w \geq \|N\|_\infty^w$.*

Proof. This is a direct consequence of (4.25). □

Theorem 4.47 (Perron-Frobenius Theorem) *Let T be a $n \times n$ non-negative matrix ($T \geq 0$). Then*

- *If T is irreducible, then $\rho(T)$ is an eigenvalue of T and there exists some $w \in \mathbb{R}^n$, $w > 0$ such that $Tw = \rho(T)w$. Moreover, such a w is unique within a scalar multiple, i.e., if some v also satisfies $Mv = \rho(T)v$, then for some scalar α , $v = \alpha w$. Finally, $\|T\|_\infty^w = \rho(T)$.*
- *$\rho(T)$ is an eigenvalue of T and there exists some $w \geq 0$, $w \neq 0$, such as $Tw = \rho(T)w$.*
- *$\forall \varepsilon > 0$, there exists some $w > 0$ such that $\rho(T) \leq \|T\|_\infty^w \leq \rho(T) + \varepsilon$.*

The proof of the Theorem 4.47 is given in the subsection 2.6 (*Convergence Analysis of Classical Iterative Methods*), pp.148–150, of the book by D.P. Bertsekas and J.N. Tsitsiklis [179], Part 1 (*Synchronous Algorithms*), Section 2 (*Algorithms for Systems of Linear Equations and Matrix Inversion*). Considering the *Perron-Frobenius Theorem* (Theorem 4.47), we give the following corollaries and propositions.

Corollary 4.48 *If T is a $n \times n$ non-negative matrix, then the following are equivalent:*

1. $\rho(T) < 1$.

Proof. Using 2 and 3., then $\|T\|_\infty^w \leq \zeta < 1$, and by 4. of Proposition 4.46, $\rho(T) < 1$. □

2. *There exists some $w > 0$ such that $\|T\|_\infty^w < 1$.*

Proof. Using 1., by Theorem 4.47 (*Perron-Frobenius*), let $\varepsilon \rightarrow 0^+$, there exists some w such that $\|T\|_\infty^w \leq \rho(|T|) + \varepsilon < 1$ □

3. *There exist some $\zeta < 1$ and $w > 0$ such that $Tw \leq \zeta w$.*

Proof. See 3. of Proposition 4.46. □

Corollary 4.49 *Given any $n \times n$ matrix T , there exists some $w > 0$ such that $\|T\|_\infty^w < 1$ is and only is $\rho(|T|) < 1$.*

Proof. By Theorem 4.47 (Perron-Frobenius), for any $\varepsilon > 0$, there exists some w such that

$$\rho(T) \leq \|T\|_\infty^w = \||T|\|_\infty^w \leq \rho(|T|) + \varepsilon$$

Since w was arbitrary, we can choose $\varepsilon \rightarrow 0$ to obtain $\rho(T) < \rho(|T|)$. \square

Corollary 4.50 For any $n \times n$ matrix T , $\rho(T) \leq \rho(|T|)$.

Proof. By Theorem 4.47 (Perron-Frobenius), for any $\varepsilon > 0$, there exists some w such that

$$\rho(T) \leq \|T\|_\infty^w = \||T|\|_\infty^w \leq \rho(|T|) + \varepsilon$$

Since w was arbitrary, we can choose $\varepsilon \rightarrow 0$ to obtain $\rho(T) < \rho(|T|)$. \square

exist some $\zeta < 1$ and $w > 0$ such that $Tw \leq \zeta w$.

Corollary 4.51 For any $n \times n$ matrix T , exist some $\gamma \in \mathbb{R}$, $\zeta \in \mathbb{R}$ and $w \in \mathbb{R}^n$ such that

$$0 < \gamma < \zeta < 1, w > 0 \quad \text{and} \quad |T|w \leq \gamma w.$$

Proof. By Theorem 4.47 (Perron-Frobenius) on $|T|$, for any $\varepsilon > 0$, there exists some w such that

$$\rho(|T|) \leq \||T|\|_\infty^w \leq \rho(|T|) + \varepsilon.$$

Let us consider $\varepsilon = \frac{1 - \rho(|T|)}{2}$ and an appropriate w which satisfies the previous. We can then

choose, $\gamma = \||T|\|_\infty^w$ and $\zeta = 1 - \frac{1 - \gamma}{2}$. By (4.25), $\gamma = \||T|\|_\infty^w = \max_i \frac{1}{w_i} \sum_{j=1}^n |t_{ij}| w_j = \max_i \frac{1}{w_i} (|T|w)_i$.

Thus, $\forall i, \gamma \geq \frac{1}{w_i} (|T|w)_i$. Finally, $|T|w \leq \gamma w < \zeta w$ ($\gamma < \zeta$ and $w > 0$). \square

The following proposition stems from Theorem 4.47.

Corollary 4.52 Let $T \in \mathbb{K}^{n \times n}$ a square matrix and $c \in \mathbb{K}^n$ a vector. $\mathcal{F} : x \mapsto Tx + c$ is a contraction mapping with respect to a weighted maximum norm $\|\cdot\|_\infty^w$ if and only if $\rho(|T|) < 1$.

Definition 4.53 (\mathcal{Z} -matrix) A matrix $T \in \mathbb{K}^{n \times n}$ is said to be a \mathcal{Z} -matrix if all of its off-diagonal entries (if any) are non-negative, i.e.

$$\forall i, j \in \{1, \dots, n\}, A_{i,i} > 0 \text{ and } A_{i,j} \leq 0 \text{ for } i \neq j \quad (4.27)$$

Proposition 4.54 Let A be a \mathcal{Z} -matrix, then the following properties are equivalent:

- There exists a non-negative vector u ($u \geq 0$) such that $Au > 0$.
- There exists a positive vector u ($u > 0$) such that $Au > 0$.
- The matrix A is non-singular and $(A^{-1})_{i,j} \geq 0, \forall i, j \in \{1, \dots, n\}$
- The spectral radius of the Jacobi matrix associated to A is strictly less than 1, i.e., $\rho(I - D^{-1}A) < 1$

Definition 4.55 (\mathcal{M} -matrix) A matrix $T \in \mathbb{K}^{n \times n}$ is said to be a \mathcal{M} -matrix if T is a \mathcal{Z} -matrix that satisfies Proposition 4.54.

Proposition 4.56 *The Jacobi iterative algorithm converges for any \mathcal{M} -matrix.*

Proposition 4.57 *The Jacobi matrix associated with a \mathcal{Z} -matrix is non-negative. For \mathcal{Z} -matrices, the asynchronous convergence conditions are similar to synchronous ones. Moreover, if the matrix A of the linear system $Ax = b$ (3.1) is a \mathcal{Z} -matrix, then the associated asynchronous Jacobi algorithm converges if and only if the properties of Proposition 4.54 are verified.*

Before presenting the sub-structuring methods in the next section, we first give some properties and convergence results of splitting methods. A particular attention will be paid to the *Jacobi method*.

4.4.3 Jacobi method, a stationary iterative method based on splittings

The Jacobi method is a stationary iterative method based on splittings of the matrix $A \in \mathbb{K}^{n \times n}$, presented in Section 3.2.3, Page 48. Let

$$A = M - N, \quad (4.28)$$

be a splitting of A , such that M is a non-singular matrix.

Definition 4.58 (Iterative splitting algorithm) The iterative algorithm associated with the splitting (4.28) is defined by

$$\begin{cases} u^{(0)} \text{ given,} \\ u^{(k+1)} = M^{-1}Nu^{(k)} + M^{-1}b, \quad \forall k \in \mathbb{N}. \end{cases} \quad (4.29)$$

Let $T = M^{-1}N$ and $c = M^{-1}b$. The matrix $|T|$ is defined such that $|T| = (|T_{i,j}|)$, for $i, j \in \{1, \dots, n\}$.

Remark 4.59 The speed of the algorithm is determined by the quantity $\rho(T)$, where $\rho(T)$ denotes the spectral radius of the matrix T . More accurately, the bounded value of the error between the exact solution and the approximate solution at iteration k is expressed by:

$$\forall \varepsilon > 0, \quad \|u^{(k)} - u^*\| \leq (\rho(T) + \varepsilon)^k \|u^{(0)} - u^*\| \quad (4.30)$$

For a detailed proof of (4.30), see Section 3.2, Page 41 and J.M Bahi *et al.* book [216].

Theorem 4.60 *The sequential and parallel synchronous algorithms (4.29) converge if and only if the spectral radius $\rho(T) < 1$.*

Proof. (see Section 3.2.3, Page 46) □

Theorem 4.61 *The asynchronous associated with algorithm (4.29) converges if and only if the spectral radius $\rho(|T|) < 1$.*

The Theorem 4.61 stems from D. Chazan and W. Miranker [177].

Definition 4.62 (Jacobi iterations) The Jacobi algorithm is obtained by considering M and N in algorithm (4.29) such as $M = D$ is the diagonal part of A (non-singular), $N = -(L + U)$ where L and U are the strictly lower and upper triangular of A , *i.e.*

$$\begin{cases} u^{(0)} \text{ given,} \\ u^{(k+1)} = D^{-1} (b - (L + U)u^{(k)}), \quad \forall k \in \mathbb{N} \end{cases} \quad (4.31)$$

$$\begin{cases} u_i^{(0)} \text{ given,} \\ u_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^n a_{ij} u_j^{(k)} \right), \quad i = \{1, \dots, n\}, \quad k \in \mathbb{N}. \end{cases} \quad (4.32)$$

Definition 4.63 (Gauss-Seidel iterations) The Gauss-Seidel algorithm is obtained by considering M and N in algorithm (4.29) such that $M = D + L$ and $N = -U$, *i.e.*

$$\begin{cases} u^{(0)} \text{ given,} \\ u^{(k+1)} = (D + L)^{-1} (b - Uu^{(k)}), \quad \forall k \in \mathbb{N} \end{cases} \quad (4.33)$$

Figure 4.24 shows the scheme of the Jacobi method (4.31). Considering the algorithm (4.32),

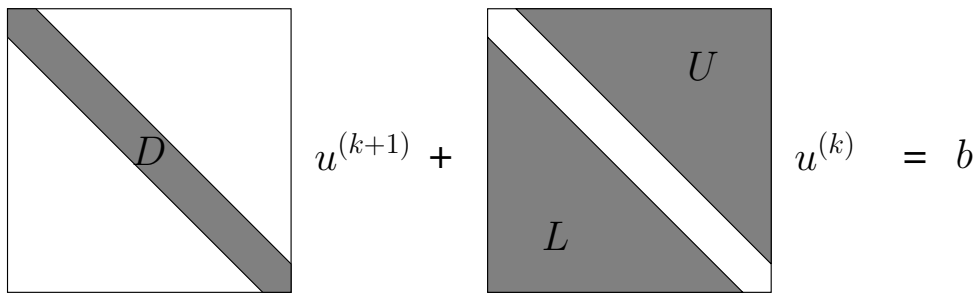


FIGURE 4.24: Design of the Jacobi algorithm

the computations of the components $u_i^{(k)}$, $i \in \{1, \dots, n\}$ are independent, which signifies that their update can be performed in parallel.

Algorithm 4.4: Jacobi method: element updates version

input : n : size of the matrix,
 A : $n \times n$ square matrix, b : right-hand side vector, $u^{(0)}$: initial guess,
 ε : tolerance threshold, K : maximum number of iterations

output : u : solution vector

- 1 Choose an initial guess $u^{(0)}$ to the solution
- 2 $k \leftarrow 0$
- 3 **while** *Loop until convergence* **do**
- 4 **for** $i = 1 : n$ **do**
- 5 $\sigma \leftarrow 0$
- 6 **for** $j = 1 : n$ **do**
- 7 **if** $i \neq j$ **then**
- 8 $\sigma \leftarrow \sigma + a_{ij} * u_j^{(k)}$
- 9 **end**
- 10 **end**
- 11 **end**
- 12 $u_i^{(k+1)} \leftarrow \frac{1}{a_{ii}}(b_i - \sigma)$
- 13 **if** $\|u^{(k+1)} - u^{(k)}\| \leq \varepsilon$ **then**
- 14 $k \leftarrow k + 1$
- 15 **break**;
- 16 **end**
- 17 $k \leftarrow k + 1$
- 18 **end**

Algorithm 4.5: Jacobi method: vectorial version

input : n : size of the matrix,
 A : $n \times n$ square matrix, b : right-hand side vector,
 $u^{(0)}$: initial guess,
 ε : tolerance threshold, K : maximum number of iterations

output : u : solution vector

- 1 Choose an initial guess $u^{(0)}$ to the solution
- 2 $k \leftarrow 0$
- 3 *Compute D^{-1} // - D^{-1} is computed once before the iterations*
- 4 **while** *Loop until convergence* **do**
- 5 $q \leftarrow Au^{(k)}$
- 6 // Compute $r \leftarrow b - Au^{(k)}$
- 7 $r \leftarrow b - q$
- 8 // - *Compute $\|r\| \equiv \|u^{(k+1)} - u^{(k)}\| \|D^{-1}\|$*
- 9 $\|r\| \leftarrow \|b - Au^{(k)}\|$
- 10 **if** $\|r\| \leq \varepsilon$ **then**
- 11 $k \leftarrow k + 1$
- 12 **break**;
- 13 **end**
- 14 $u^{(k+1)} \leftarrow u^{(k)} + D^{-1}r$
- 15 $k \leftarrow k + 1$
- 16 **end**

Algorithm 4.4 and Algorithm 4.5 present respectively the *element updates version* and *vectorial version*. Unlike the Gauss-Seidel algorithm, the Jacobi algorithm converges more slowly, but is *fully parallelizable*.

Proposition 4.64 (Jacobi iterations (vectorial form)) *The jacobi iteration given in algorithm (4.31) can be rewritten in the following form*

$$\begin{cases} u^{(0)} \text{ given,} \\ u^{(k+1)} = D^{-1} (b - Au^{(k)}) + u^{(k)}, \quad \forall k \in \mathbb{N} \end{cases} \quad (4.34)$$

The vectorial form given in algorithm (4.32) can also be written as

$$\begin{cases} u_i^{(0)} \text{ given,} \\ u_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^n a_{ij} u_j^{(k)} \right) + u_i^{(k)}, \quad i = \{1, \dots, n\}, \quad k \in \mathbb{N}. \end{cases} \quad (4.35)$$

Proof.

$$\begin{aligned} u^{(k+1)} &= D^{-1} (b - (L + U)u^{(k)}) = D^{-1} (b - (A - D)u^{(k)}) \\ &= D^{-1} (b - Au^{(k)}) + u^{(k)} \end{aligned}$$

$$\begin{aligned} \text{Similarly, } u_i^{(k+1)} &= \frac{1}{a_{ii}} \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^n a_{ij} u_j^{(k)} \right) = \frac{1}{a_{ii}} \left(b_i - \left(\sum_{j=1}^n a_{ij} u_j^{(k)} - a_{ii} u_i^{(k)} \right) \right) \\ &= \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^n a_{ij} u_j^{(k)} \right) + u_i^{(k)}, \quad \forall i = \{1, \dots, n\}. \end{aligned}$$

□

Proposition 4.65 *The convergence of the Jacobi method is ensured if the matrix A is diagonally dominant:*

$$\forall i \in \{1, \dots, p\}, |a_{ii}| \geq \sum_{\substack{j=1 \\ j \neq i}}^p |a_{ij}|. \quad (4.36)$$

The convergence is also ensured for \mathcal{Z} -matrices, which are matrices with positive diagonal elements and negative off-diagonal elements.

Proposition 4.66 (Jacobi error vector) *Considering the Proposition 4.64, the error vector $\zeta^{(k)} = u^{(k+1)} - u^{(k)}$ of the Jacobi algorithm can be expressed as follows: $\zeta^{(k)} = u^{(k+1)} - u^{(k)} = D^{-1} (b - Au^{(k)})$.*

4.5 Conclusion

In this chapter, we have introduced the different classes of parallel iterative methods for readers not familiar with parallel and distributed computing. Different splitting strategies allowing data distribution among processors are also described. In particular, we have presented a smart process to partition a sparse matrix for parallel sub-structuring methods. In order to

better understand the mechanism and the model of parallel algorithms, we have presented the mathematical models of parallel synchronous and asynchronous algorithms.

Implementation of parallel linear system solvers

” *The only source of knowledge is experience.*
— Albert Einstein
(Physicist)

5.1 Introduction

Knowing more about synchronous and asynchronous iterative methods and how to prepare data for parallel algorithms, we now focus on the study of sub-structuring method, a domain decomposition method without overlap.

My contribution to this chapter has been to develop, implement and test innovative algorithms of parallel synchronous and asynchronous sub-structuring methods.

My contribution was also to integrate GPU Computing in parallel algorithms with synchronous and asynchronous iterations.

We give proof of the convergence of asynchronous sub-structuring methods.

We also give some theorems concerning fault-tolerance in parallel processing and we study the behavior of the algorithm when some iterations are penalized.

The remainder of this chapter is organized as follows. The first section, Section 5.2 describes and discusses sub-structuring methods. The sub-structuring method is first presented in the synchronous case. Then a particular attention is paid to the asynchronous case. In this section, we give a proof of the convergence of the asynchronous sub-structuring methods. In Section 5.2, we also evaluate the behavior of parallel algorithms in terms of fault-tolerance and iteration penalization. Section 5.3 studies the theoretical speed-up of fully parallelizable iterative methods with synchronous and asynchronous iterations. This section aims to analyze the behavior of asynchronous algorithms. *In Section 5.3, my contribution is the theorems for theoretical speed-up of synchronous and asynchronous parallelizable iterative methods.* Section 5.4 reports numerical results and shows the advantage of parallel sub-structuring methods, particularly when they are associated with asynchronous iterations with enough processes. The numerical results highlight the effectiveness and robustness of these methods on a platform of multi-core-GPUs. Section 5.5 concludes the chapter.

Keywords — Parallel Synchronous, Parallel Asynchronous, Convergence detection, Iterative algorithms, Sub-structuring methods, Jacobi method, Domain Decomposition Method, DDM, Schwarz

5.2 Sub-structuring methods

The sub-structuring method is the precursor of non-overlapping domain decomposition methods [16] [17] [18] [19]. The sub-structuring method is based on the decomposition of the original structure into several sub-structures. The term *sub-structuring* is a way to describe a general method allowing to decompose a splitting among sub-domains sharing a common interface. This method is most often used as a way to reduce the number of unknowns in the linear system by eliminating the interior unknowns. In this section, we present a convergence theorem for asynchronous sub-structuring methods. The convergence results are derived from the convergence of the associated sequential fixed point problem and the splitting of the matrix between the sub-domains. Consider again the system of linear equations,

$$Au = f, \quad (5.1)$$

where A is a $n \times n$ square non-singular matrix, f and u represent respectively the right-hand side and the solution vector we are looking for. Let us consider the field \mathbb{K} , usually \mathbb{R} or \mathbb{C} .

The remainder of this section is organized as follows. We start by giving a complete and basic illustration of sub-structuring methods in order to better understand their main steps and properties. A general mathematical model of synchronous and asynchronous sub-structuring methods are then presented. The proof of the asynchronous convergence of the sub-structuring algorithm is also presented. The next section relates some important points concerning the halting procedure and the convergence detection of asynchronous algorithms. The last section describes how we proceed to implement sub-structuring methods on a system of multi-core-GPUs. Some approaches of this section are inspired by the chapter presented in reference [258].

5.2.1 Overview of the principle of sub-structuring methods

In order to illustrate the sub-structuring method, we consider a problem stemming from the finite element discretization of an elliptic partial differential problem. To simplify the analysis, we consider the Laplace equation. However, the analysis can be carried out for any coercive elliptic problem. The model problem for the unknown u , in a bounded domain Ω with homogeneous Dirichlet boundary conditions on the boundary $\partial\Omega = \Gamma$ can be expressed as: for $f \in L^2(\Omega)$, find $u \in H^1(\Omega)$ such that $-\nabla^2 u = f$ in Ω and $u = 0$ on Γ . An equivalent variational formulation of this problem can be formulated as: for $f \in L^2(\Omega)$, find $u \in H_0^1(\Omega)$ such that $\forall v \in H_0^1(\Omega)$, $\int_{\Omega} \nabla u \cdot \nabla v = \int_{\Omega} f \cdot v$. This problem is well posed, *i.e.*, it has one and only one solution. After a Galerkin discretization with finite elements and a choice of nodal basis, the linear system (5.1), $Au = f$, is obtained, where f denotes the right-hand side, u the unknown and A the stiffness matrix which is sparse. The following shows the main steps for performing a matrix-vector product using a splitting into sub-structures.

Matrix splitting

In practice, mesh partitioning is a crucial step of the finite element method. A finite element matrix is associated with a finite element mesh and the elements of the matrix are correlated with the interaction of the basis functions defined in the elements of the mesh. The total

matrix is calculated as an assembly of elementary matrices. Let us consider a global domain Ω partitioned into two sub-domains without overlap Ω_1 and Ω_2 , with a shared interface Γ as drawn in Figure 5.1. When a suitable numbering of the degrees of freedom is harnessed, the

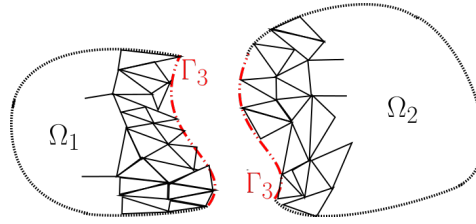


FIGURE 5.1: Splitting into two sub-domains

stiffness matrix of the initially considered model problem can be written as in the following matrix:

$$A = \begin{pmatrix} A_{[11]} & 0 & A_{[13]} \\ 0 & A_{[22]} & A_{[23]} \\ A_{[31]} & A_{[32]} & A_{[33]} \end{pmatrix} \quad (5.2)$$

It is formulated considering the case where the set of nodes numbered 1 and 2 are respectively associated to the sub-domains Ω_1 and Ω_2 . The last set of nodes numbered 3 corresponds to the interface nodes of both sub-domains. It is possible to split the original system (5.1) into blocks

$$\begin{pmatrix} A_{[11]} & 0 & A_{[13]} \\ 0 & A_{[22]} & A_{[23]} \\ A_{[31]} & A_{[32]} & A_{[33]} \end{pmatrix} \begin{pmatrix} u_{[1]} \\ u_{[2]} \\ u_{[3]} \end{pmatrix} = \begin{pmatrix} f_{[1]} \\ f_{[2]} \\ f_{[3]} \end{pmatrix} \quad (5.3)$$

where $u = (u_{[1]}, u_{[2]}, u_{[3]})^t$ is the unknown vector and $f = (f_{[1]}, f_{[2]}, f_{[3]})^t$ is the right-hand side. The blocks $A_{[13]}$ and $A_{[23]}$ are respectively the transpose matrix of $A_{[31]}$ and $A_{[32]}$, and the blocks $A_{[11]}$ and $A_{[22]}$ are symmetric positive-definite if A was symmetric positive-definite. By assigning the different sub-domains at distinct processors, the local matrices can be formulated in parallel as follows:

$$A_1 = \begin{pmatrix} A_{[11]} & A_{[13]} \\ A_{[31]} & A_{[33]}^{\{1\}} \end{pmatrix} \quad \text{and} \quad A_2 = \begin{pmatrix} A_{[22]} & A_{[23]} \\ A_{[32]} & A_{[33]}^{\{2\}} \end{pmatrix}. \quad (5.4)$$

The blocks $A_{[33]}^{\{1\}}$ and $A_{[33]}^{\{2\}}$ denote the interaction between the nodes on the interface Γ , respectively integrated in sub-domains Ω_1 and on Ω_2 , *i.e.*

$$A_{[33]} = A_{[33]}^{\{1\}} + A_{[33]}^{\{2\}} \quad (5.5)$$

In practice, the sub-domains Ω_1 and Ω_2 respectively know the set of nodes (1, 3) and (2,3).

Matrix-vector product

As described in the previous chapter, iterative Krylov algorithms require performing one or more multiplication(s) of the matrix A by a descent direction vector $u = (u_{[1]}, u_{[2]}, u_{[3]})^t$ at

each iteration. With the splitting into two sub-domains, the global matrix-vector multiplication can be written as follows:

$$\begin{aligned} \begin{pmatrix} y_{[1]} \\ y_{[2]} \\ y_{[3]} \end{pmatrix} &= \begin{pmatrix} A_{[11]} & 0 & A_{[13]} \\ 0 & A_{[22]} & A_{[23]} \\ A_{[31]} & A_{[32]} & A_{[33]} \end{pmatrix} \begin{pmatrix} u_{[1]} \\ u_{[2]} \\ u_{[3]} \end{pmatrix} \\ &= \begin{pmatrix} A_{[11]}u_{[1]} + A_{[13]}u_{[3]} \\ A_{[22]}u_{[2]} + A_{[23]}u_{[3]} \\ A_{[31]}u_{[1]} + A_{[32]}u_{[2]} + A_{[33]}u_{[3]} \end{pmatrix} \end{aligned}$$

Considering the local matrices' described in equation (5.4), we can independently compute both local matrix-vector products as follows

$$\begin{aligned} \begin{pmatrix} y_{[1]} \\ y_{[1]}^{\{1\}} \\ y_{[3]} \end{pmatrix} &= \begin{pmatrix} A_{[11]}u_{[1]} + A_{[13]}u_{[3]} \\ A_{[31]}u_{[1]} + A_{[33]}^{\{1\}}u_{[3]} \end{pmatrix} \\ \begin{pmatrix} y_{[2]} \\ y_{[3]}^{\{2\}} \end{pmatrix} &= \begin{pmatrix} A_{[22]}u_{[2]} + A_{[23]}u_{[3]} \\ A_{[32]}u_{[2]} + A_{[33]}^{\{2\}}u_{[3]} \end{pmatrix} \end{aligned}$$

Since $A_{[33]} = A_{[33]}^{\{1\}} + A_{[33]}^{\{2\}}$, and $y_{[3]} = y_{[3]}^{\{1\}} + y_{[3]}^{\{2\}}$. According to this last remark, SpMV can be calculated in two steps:

- calculate the local matrix-vector multiplication in each sub-domain
- assemble, the local contributions on the interface

The first step involves only local data. The second requires the exchange of data between processes dealing with sub-domains with a common interface. In order to assemble interface values of neighboring sub-domains, each processor responsible for a sub-domain must know the description of its interfaces.

Exchange at the interfaces

When a sub-domain Ω_i has several neighboring sub-domains, we denote Γ_{ij} the interface between Ω_i and Ω_j as described in (see Figure 5.2, Page 144). An interface is identified by

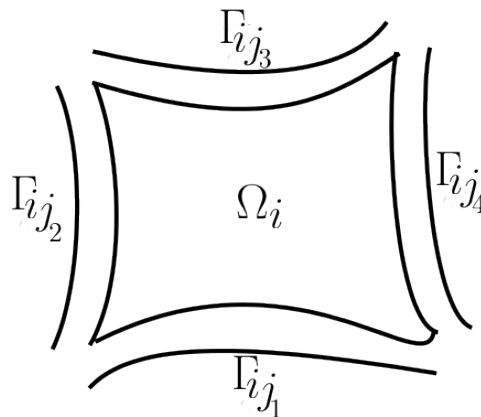


FIGURE 5.2: Sub-structure interface description

its neighboring sub-domains and the equations associated with its nodes. The interface is evaluated from its sub-domains using the sparse matrix-vector product. This computation is in two steps for each neighboring sub-domain (Algorithm 5.1): *collect* the values of the local vector $y = Au$ for all interface nodes, and then *send* this list to the y vector of the interface equation. The next step consists in updating these changes to all neighboring

Algorithm 5.1: Construct inner buffer and send to neighboring

input : n_s : number of interface nodes
 $number_of_neighboring$: number of neighboring sub-domains
 y : values on the whole sub-domain
 $list_s$: list of the interface nodes
output : $buffer_s$: sending buffer
variable s, i

```

1 for  $s \leftarrow 1 : number\_of\_neighboring$  do
2   | for  $i = 1 : n_s$  do
3   |   |  $buffer_s(i) = y(list_s(i))$ 
4   |   end
5   |   Send  $buffer_s$  to neighbour(s)
6 end
```

sub-domains at interfaces equations. First, the contributions of the array containing the result of the matrix-vector product at the interface are received, and then values on the corresponding interface nodes are updated. This process is described in Algorithm 5.2. In GPU code, the construction of the inner buffer is carried out on CPU before sending it to the neighboring sub-domain. When an equation is shared by several interfaces, the node value of the local

Algorithm 5.2: Receiving interface results and updating interface equations

input : n_s : number of interface nodes
 $number_of_neighboring$: number of neighboring sub-domains
 y : values on the whole sub-domain
 $list_s$: list of the interface nodes
output : $buffer_s$: receiving buffer
variable s, i

```

1 for  $s \leftarrow 1 : number\_of\_neighboring$  do
2   | Receive  $buffer_s$  from neighbour(s) for  $i = 1 : n_s$  do
3   |   |  $y(list_s(i)) = y(list_s(i)) + buffer_s(i)$ 
4   |   end
5 end
```

vector $y = Au$ in question is sent to all interfaces to which it belongs. For any number of sub-domains, the mechanism of interface exchange and update is similar to those previously presented. In GPU code, the procedure of exchange has been performed on CPU and then the assembled vector is copied back to GPU, before continuing the algorithm.

The use of the sub-structuring approach in iterative algorithms is inherently parallel and makes it an excellent candidate for implementation on parallel computers. Indeed, we can distribute the sub-domains over all available processors and thus compute the matrix-vector products locally, independently and in parallel, and use the distributed memory in order to limit the memory usage. As explained previously, after the computing of the local matrix-vector multiplications, they required to be assembled along the interface. The key ingredient of the data is the local matrix C that arises from the finite element discretization. With this approach, each node i only needs to store C_i , the corresponding local matrix to the sub-domain Ω_i , which

is only a fraction of the original matrix. The dot product requires that each processor compute a weighted combination of the interface contributions in order to update its own data. After that, an *MPI_ALLREDUCE* is required to compute the global inner product. Transfers between CPU and GPU at each iteration can decrease the performance of the exchanges' algorithms.

The iterative sub-structuring method introduces only two new steps, which reside in data exchange. They consist firstly in sharing the contributions of the local computed SpMV at the interface. Each machine requires to know the list of nodes along the interface and the number of neighboring sub-domains. Secondly, in assembling results over the cluster in order to piece together the local scalar product. This action, realized with MPI, is independent from the splitting. Finally, another advantage of this algorithm is that it can easily be generalized for n sub-domains. This approach however, presents two disadvantages. The first drawback arises from a computational point of view. The granularity, *i.e.*, the number of operations to be performed by the processors compared to the amount of data received or sent by the processors may be weak. Indeed, here the granularity is proportional to the number of nodes in the sub-domains compared to the number of nodes on the interface. The number of operations depend on the first parameter and the data transfer depends on the second parameter. If a lot of sub-domains are used, the interface size will not be small compared to the local sub-problem size. This means that the processors realize few computing operations (a local matrix-vector product) and a lot of communications. The second and more important drawback is an algorithmic one. The classical parallel preconditioners per sub-domain are based on an incomplete factorization of the local matrices. Such preconditioners are less and less efficient when the number of sub-domain increases.

5.2.2 Matrix-based splitting

Let us consider the parallel platform, $\mathcal{PS}_{system}\{p\}$, is capable of executing p processes of execution of \mathcal{PS} and the original domain is decomposed in p sub-domains.

The system of linear algebraic equations (5.1) ($Au = f$, $A \in \mathbb{K}^{n \times n}$ is a sparse matrix, $f \in \mathbb{K}^n$ the right-hand side and $u \in \mathbb{K}^n$ represents the solution vector we are looking for) obtained from a finite element discretization, which has a special form thanks to the appropriate ordering of the unknowns

$$\underbrace{\begin{pmatrix} A_{[ii]}^{\{1\}} & 0 & \cdots & A_{[ib]}^{\{1\}} \\ & \ddots & & \vdots \\ 0 & & A_{[ii]}^{\{s\}} & A_{[ib]}^{\{s\}} \\ \vdots & & & \vdots \\ A_{[bi]}^{\{1\}} & \cdots & A_{[bi]}^{\{s\}} & \cdots & A_{[bi]}^{\{p\}} & A_{[bb]}^{\{p\}} \end{pmatrix}}_A \underbrace{\begin{pmatrix} u_{[i]}^{\{1\}} \\ \vdots \\ u_{[i]}^{\{s\}} \\ \vdots \\ u_{[i]}^{\{p\}} \\ u_{[b]} \end{pmatrix}}_u = \underbrace{\begin{pmatrix} f_{[i]}^{\{1\}} \\ \vdots \\ f_{[i]}^{\{s\}} \\ \vdots \\ f_{[i]}^{\{p\}} \\ f_{[b]} \end{pmatrix}}_f \quad (5.6)$$

Let us describe the notations used in (5.6):

- b denotes the parts corresponding to the interface between sub-domains,
- i denotes the parts corresponding to the interior between sub-domains,

- $A_{[ii]}^{\{s\}}$, where $s \in \{1, \dots, p\}$, is the sub-matrix of internal nodes of matrix A on the sub-domain s ,
- $A_{[ib]}^{\{s\}}$ and $A_{[bi]}^{\{s\}}$, where $s \in \{1, \dots, p\}$, denote respectively the interactions between the interface (i) and the interior (b) of the sub-domain s ,
- $A_{[bb]}$ is the interface sub-matrix,
- $u_{[i]}^{\{s\}}$ and $f_{[i]}^{\{s\}}$, where $s \in \{1, \dots, p\}$, are respectively sub-vectors of the internal nodes of the solution and the right-hand side on the s^{th} sub-domain,
- $u_{[b]}$ and $f_{[b]}$ are respectively sub-vectors of the solution and the right-hand side on the interface.

Matrix $A_{[bb]}$ and vector $f_{[b]}$ contain contributions from the sub-domains

$$A_{[bb]} = \sum_{s=1}^p A_{[bb]}^{\{s\}} \quad \text{and} \quad f_{[b]} = \sum_{s=1}^p f_{[b]}^{\{s\}}. \quad (5.7)$$

A spatial partitioning of the domain conforming to the finite element discretization is given in Figure 5.1.

The splitting of the system (5.6) is performed following a renumbering of the rows so that the degree of freedom (*dof*) of internal nodes comes out first and is ranged by sub-domains. The splitting is such that two cooperating sub-domains s_1 and s_2 interact only via their interface nodes. Considering the special form (5.6) of the system of linear algebraic equations (5.1), which thanks to appropriate renumbering of the unknowns, the associated matrix $T = M^{-1}N$ and the vector $c = M^{-1}f$ of iterative splitting algorithm (see eq. (4.29), Page 136) can be rewritten in the form

$$T = \begin{pmatrix} T_{[ii]}^{\{1\}} & 0 & \cdots & T_{[ib]}^{\{1\}} \\ & \ddots & & \vdots \\ 0 & T_{[ii]}^{\{s\}} & & T_{[ib]}^{\{s\}} \\ \vdots & & \ddots & \vdots \\ T_{[bi]}^{\{1\}} & \cdots & T_{[bi]}^{\{s\}} & \cdots & T_{[ii]}^{\{p\}} & T_{[ib]}^{\{p\}} \\ & & & & T_{[bi]}^{\{p\}} & T_{[bb]} \end{pmatrix} \quad \text{and} \quad c = \begin{pmatrix} c_{[i]}^{\{1\}} \\ \vdots \\ c_{[i]}^{\{s\}} \\ \vdots \\ c_{[i]}^{\{p\}} \\ c_{[b]} \end{pmatrix} \quad (5.8)$$

After the splitting of the matrix T and vector c described in (5.8), on each subdom $s \in \{1, \dots, p\}$, these contributions can be rewritten in the form

$$T^{\{s\}} = \begin{pmatrix} T_{[ii]}^{\{s\}} & T_{[ib]}^{\{s\}} \\ T_{[bi]}^{\{s\}} & T_{[bb]} \end{pmatrix} \quad \text{and} \quad c^{\{s\}} = \begin{pmatrix} c_{[i]}^{\{s\}} \\ c_{[b]} \end{pmatrix} \quad (5.9)$$

The matrix $T_{[bb]}$ and vector $c_{[b]}$ contain contributions from the sub-domains:

$$T_{[bb]} = \sum_{s=1}^p T_{[bb]}^{\{s\}} \quad \text{and} \quad c_{[b]} = \sum_{s=1}^p c_{[b]}^{\{s\}}. \quad (5.10)$$

The splitting (5.9) becomes more attractive if we can build for each sub-domain $s \in \{1, \dots, p\}$, the sub-matrix $T^{\{s\}}$ from the sub-matrix $A^{\{s\}}$ of the original matrix:

$$A^{\{s\}} = \begin{pmatrix} A_{[ii]}^{\{s\}} & A_{[ib]}^{\{s\}} \\ A_{[bi]}^{\{s\}} & A_{[bb]}^{\{s\}} \end{pmatrix} \quad (5.11)$$

This is the case for the Jacobi method. Considering the special form of the matrix A (5.6) and the splitting of the Jacobi method (see Definition 4.62, Page 136), where $M = D$ is the diagonal part of A (non-singular) and $N = -(L + U)$ with L and U are the strictly lower and upper triangular of A , we can rewrite M and N as follows

$$M = \begin{pmatrix} M_{[ii]}^{\{1\}} & & 0 & \cdots & & 0 \\ & \ddots & & & & \vdots \\ 0 & & M_{[ii]}^{\{s\}} & & & 0 \\ \vdots & & & \ddots & & \vdots \\ & & & & M_{[ii]}^{\{p\}} & 0 \\ 0 & \cdots & 0 & \cdots & 0 & M_{[bb]} \end{pmatrix}, N = \begin{pmatrix} N_{[ii]}^{\{1\}} & & 0 & & & N_{[ib]}^{\{1\}} \\ & \ddots & & & & \vdots \\ 0 & & N_{[ii]}^{\{s\}} & & & N_{[ib]}^{\{s\}} \\ \vdots & & & \ddots & & \vdots \\ & & & & N_{[ii]}^{\{p\}} & N_{[ib]}^{\{p\}} \\ N_{[bi]}^{\{1\}} & \cdots & N_{[bi]}^{\{s\}} & \cdots & N_{[bi]}^{\{p\}} & N_{[bb]} \end{pmatrix} \quad (5.12)$$

The matrix $N_{[bb]}$ is the opposite of the off-diagonal values of $A_{[bb]}$, *i.e.*

$$N_{[bb]} = -A_{[bb]}. \quad (5.13)$$

With the splitting (5.12), the terms of the sub-matrix $T^{\{s\}}$ described in (5.9) can be expressed as follows:

$$\begin{cases} T_{[ii]}^{\{s\}} = M_{[ii]}^{\{s\}-1} N_{[ii]}^{\{s\}}, \\ T_{[ib]}^{\{s\}} = M_{[ii]}^{\{s\}-1} N_{[ib]}^{\{s\}}, \end{cases} \quad \text{and} \quad \begin{cases} T_{[bi]}^{\{s\}} = M_{[bb]}^{-1} N_{[bi]}^{\{s\}}, \\ T_{[bb]}^{\{s\}} = M_{[bb]}^{-1} N_{[bb]}^{\{s\}}. \end{cases} \quad (5.14)$$

where $N_{[bb]}^{\{s\}} = -A_{[bb]}^{\{s\}}$. Given that $A_{[bb]} = \sum_{s=1}^p A_{[bb]}^{\{s\}}$ Similarly,

$$c_{[i]}^{\{s\}} = M_{[bb]}^{-1} f_{[i]}^{\{s\}} \quad \text{and} \quad c_{[b]}^{\{s\}} = M_{[bb]}^{-1} f_{[b]}^{\{s\}}. \quad (5.15)$$

For each sub-domain $s \in \{1, \dots, p\}$, the computation of the terms $T_{[bi]}^{\{s\}}$ and $T_{[bb]}^{\{s\}}$, and $c_{[i]}^{\{s\}}$ and $c_{[b]}^{\{s\}}$, requires the assembly matrix $M_{[bb]}$ on the interface, which is the diagonal part of the interface matrix $A_{[bb]}$ of A . As described in (5.16), the matrix $A_{[bb]}$ is computed as:

$A_{[bb]} = \sum_{s=1}^p A_{[bb]}^{\{s\}}$. Let remark that the full value of $f_{[b]} = \sum_{s=1}^p f_{[b]}^{\{s\}}$ is not needed, in fact each sub-domain has only its own part $f_{[b]}^{\{s\}}$.

According to the equalities (5.13), i.e., $N_{[bb]} = -A_{[bb]}$, and (5.16), i.e., $A_{[bb]} = \sum_{s=1}^p A_{[bb]}^{\{s\}}$, after some substitutions, we can have

$$N_{[bb]} = \sum_{s=1}^p N_{[bb]}^{\{s\}} \quad \text{and} \quad T_{[bb]} = \sum_{s=1}^p T_{[bb]}^{\{s\}}. \quad (5.16)$$

5.2.3 Synchronous sub-structuring algorithm

Let us consider the following notations. Let $T \in \mathbb{K}^{n \times n}$ be a square matrix and $c \in \mathbb{K}^n$ a vector. Let us define $n^{\{s\}}$ the number of equations of the matrix $T^{\{s\}}$ of the sub-domain $s \in \{1, \dots, p\}$. Let us recall the fixed point mapping function $\widehat{\mathcal{T}}$ described in (4.11) and (4.12) (see Section 4.4, Page 127)

$$\begin{aligned} \widehat{\mathcal{T}} : \widehat{X} &\mapsto \widehat{X} \\ \widehat{u} &\mapsto \widehat{\mathcal{T}}(\widehat{u}) \end{aligned} \quad (5.17)$$

where $\widehat{X} = X^{\{1\}} \times \dots \times X^{\{s\}} \times \dots \times X^{\{p\}}$ and $\widehat{u} = (u^{\{1\}}, \dots, u^{\{s\}}, \dots, u^{\{p\}})^T \in \widehat{X}$. Let us define $u^{\{s\}} \in X^{\{s\}}$, $s \in \{1, \dots, p\}$ such that the piece of the solution

$$u^{\{s\}} = \begin{pmatrix} u_{[i]}^{\{s\}} & u_{[b]}^{\{s\}} \end{pmatrix}^T \quad (5.18)$$

is computed by the process $s \in \{1, \dots, p\}$. The mapping function $\widehat{\mathcal{T}}$ can be expressed as follows

$$\widehat{\mathcal{T}}(\widehat{u}) = \left(\widehat{\mathcal{T}}^{\{1\}}(\widehat{u}), \dots, \widehat{\mathcal{T}}^{\{s\}}(\widehat{u}), \dots, \widehat{\mathcal{T}}^{\{p\}}(\widehat{u}) \right)^T.$$

with

$$\widehat{\mathcal{T}}^{\{s\}}(\widehat{u}) = T^{\{s\}} \begin{pmatrix} u_{[i]}^{\{s\}} \\ u_{[b]}^{\{s\}} \end{pmatrix} + c^{\{s\}}. \quad (5.19)$$

where $u_{[b]} = \sum_{j=1}^p u_{[b]}^{\{j\}}$ must be assembled between the sub-domains on the interface. In general, the interface is small compared to the internal nodes. Therefore, the exchange required between the cooperating processors is also small. The solution u of the system of linear algebraic equations (5.1) can finally be calculated from \widehat{u} as follows:

$$u = \mathcal{P}(\widehat{u}) = \left(u_{[i]}^{\{1\}}, \dots, u_{[i]}^{\{s\}}, \dots, u_{[i]}^{\{p\}}, u_{[b]} = \sum_{j=1}^p u_{[b]}^{\{j\}} \right)^T. \quad (5.20)$$

Definition 5.1 (Parallel sub-structuring algorithm) The synchronous parallel sub-structuring algorithm constructs recursively for each process $s \in \{1, \dots, p\}$ a sequence of iterations $(u^{(k)})_{k \in \mathbb{N}}$, from an initial guess solution $u^{\{s\}(0)} = u_{[0]}^{\{s\}}$, with $u^{\{s\}(k)} \in X^{\{k\}}$:

$$\begin{cases} u^{\{s\}(0)} = u_{[0]}^{\{s\}} \\ u^{\{s\}(k+1)} = \widehat{\mathcal{T}}^{\{s\}}(\widehat{u}) = \widehat{\mathcal{T}}^{\{s\}}(u^{\{1\}(k)}, \dots, u^{\{s\}(k)}, \dots, u^{\{p\}(k)}), \quad k \in \mathbb{N}^*. \end{cases} \quad (5.21)$$

At the iteration $k \in \mathbb{N}$, the approximate solution of the problem is expressed as follows

$$u^{(k)} = \mathcal{P} \left(u_{[i]}^{\{1\}(k)}, \dots, u_{[i]}^{\{s\}(k)}, \dots, u_{[i]}^{\{p\}(k)}, u_{[b]}^{(k)} = \sum_{j=1}^p u_{[b]}^{\{j\}(k)} \right)^T.$$

Lemma 5.2 (Convergence of parallel sub-structuring algorithms) *Considering the Proposition 4.34 (see Proposition 4.34, Page 129), as for the sequential algorithm, the algorithm (5.21) converges if and only if the spectral radius $\rho(T) < 1$.*

Proof. Considering the equation (5.9) “(5.14)” and given that $u^{\{s\}} = \left(u_{[i]}^{\{s\}} \quad u_{[b]}^{\{s\}} \right)^T$ (5.18), after substitution in the algorithm (5.21), we have the two following equalities:

$$\begin{pmatrix} u_{[i]}^{\{s\}(k+1)} \\ u_{[b]}^{\{s\}(k+1)} \end{pmatrix} = \begin{pmatrix} T_{[ii]}^{\{s\}} & T_{[ib]}^{\{s\}} \\ T_{[bi]}^{\{s\}} & T_{[bb]}^{\{s\}} \end{pmatrix} \begin{pmatrix} u_{[i]}^{\{s\}(k)} \\ u_{[b]}^{(k)} \end{pmatrix} + \begin{pmatrix} c_{[i]}^{\{s\}} \\ c_{[b]}^{\{s\}} \end{pmatrix}$$

on each process $s \in \{1, \dots, p\}$. Hence,

$$\begin{cases} u_{[i]}^{\{s\}(k+1)} = T_{[ii]}^{\{s\}} \times u_{[i]}^{\{s\}(k)} + T_{[ib]}^{\{s\}} \times u_{[b]}^{(k)} + c_{[i]}^{\{s\}} \\ u_{[b]}^{\{s\}(k+1)} = T_{[bi]}^{\{s\}} \times u_{[i]}^{\{s\}(k)} + T_{[bb]}^{\{s\}} \times u_{[b]}^{(k)} + c_{[b]}^{\{s\}} \end{cases}$$

Then, after summation over all the processes

$$\begin{aligned} \sum_{s=1}^p u_{[b]}^{\{s\}(k+1)} &= \sum_{s=1}^p \left(T_{[bb]}^{\{s\}} \times u_{[b]}^{(k)} + c_{[b]}^{\{s\}} \right) \\ &= \sum_{s=1}^p T_{[bb]}^{\{s\}} \times u_{[b]}^{(k)} + u_{[b]}^{(k)} \underbrace{\sum_{s=1}^p T_{[bb]}^{\{s\}}}_{T_{[bb]} \text{ (5.10)}} + \underbrace{\sum_{s=1}^p c_{[b]}^{\{s\}}}_{c_{[b]} \text{ (5.10)}} \\ &= \sum_{s=1}^p T_{[bi]}^{\{s\}} \times u_{[i]}^{\{s\}(k)} + T_{[bb]} u_{[b]}^{(k)} + c_{[b]} \\ &= u_{[b]}^{(k+1)} \end{aligned} \tag{5.22}$$

Finally,

$$\begin{pmatrix} u_{[i]}^{\{1\}(k+1)} \\ \vdots \\ u_{[i]}^{\{s\}(k+1)} \\ \vdots \\ u_{[i]}^{\{p\}(k+1)} \\ u_{[b]}^{(k+1)} \end{pmatrix} = \begin{pmatrix} T_{[ii]}^{\{1\}} & 0 & \cdots & T_{[ib]}^{\{1\}} \\ & \ddots & & \vdots \\ 0 & & T_{[ii]}^{\{s\}} & T_{[ib]}^{\{s\}} \\ \vdots & & & \vdots \\ & & & T_{[ii]}^{\{p\}} & T_{[ib]}^{\{p\}} \\ T_{[bi]}^{\{1\}} & \cdots & T_{[bi]}^{\{s\}} & \cdots & T_{[bi]}^{\{p\}} & T_{[bb]} \end{pmatrix} \begin{pmatrix} u_{[i]}^{\{1\}(k)} \\ \vdots \\ u_{[i]}^{\{s\}(k)} \\ \vdots \\ u_{[i]}^{\{p\}(k)} \\ u_{[b]}^{(k)} \end{pmatrix} + \begin{pmatrix} c_{[i]}^{\{1\}} \\ \vdots \\ c_{[i]}^{\{s\}} \\ \vdots \\ c_{[i]}^{\{p\}} \\ c_{[b]} \end{pmatrix}$$

More simply,

$$u^{(k+1)} = Tu^{(k)} + c, \quad k \in \mathbb{N}^*, \quad \text{and given } u^{(0)}.$$

This last is equivalent to the sequential algorithm (see eq. (4.7), Page 128). Therefore, the parallel sub-structuring algorithm has the same condition, *i.e.*, *iff the spectral radius* $\rho(T) < 1$. \square

Algorithm 5.3 describes the main steps of the algorithm of parallel synchronous sub-structuring iterative methods given in (5.21). In terms of implementation, the summation of

Algorithm 5.3: Algorithm of the iterative parallel synchronous sub-structuring methods

input : $s \in \{1, \dots, p\}$
 $T^{\{s\}}$: sub-matrix of the sub-domain s , $c^{\{s\}}$: sub-vector of the sub-domain s (5.9),
 $u^{\{s\}(0)} = (u_{[i]}^{\{s\}(0)} \quad u_{[b]}^{\{s\}(0)})$: initial solution of whole sub-domain s ,
 $u_{[i]}^{\{s\}(0)}$ (internal) and $u_{[b]}^{\{s\}(0)}$ (interface) from $u^{\{s\}(0)}$,
 ε : tolerance threshold, K : maximum number of iterations

output : $u^{\{s\}} = (u_{[i]}^{\{s\}} \quad u_{[b]}^{\{s\}})$: local solution

variable: $k, j, convg$

variable: $u_{[b]}$

```

1 convg ← false; k ← 0
2 // -- allocate  $u^{\{s\}}$ 
3 while .not. convg .OR. k < K do
4   // -- assembly of sub-domain  $s$  along the interface
5    $u_{[b]}^{(k)} \leftarrow \sum_{j=1}^p u_{[b]}^{\{j\}(k)}$ 
6   // -- local computation of  $u^{\{s\}(k+1)} = \widehat{\mathcal{T}}^{\{s\}}(\hat{u})$  (5.21)
7    $u_{[i]}^{\{s\}(k+1)} = T_{[ii]}^{\{s\}} \times u_{[i]}^{\{s\}(k)} + T_{[ib]}^{\{s\}} \times u_{[b]}^{(k)} + c_{[i]}^{\{s\}}$ 
8    $u_{[b]}^{\{s\}(k+1)} = T_{[bi]}^{\{s\}} \times u_{[i]}^{\{s\}(k)} + T_{[bb]}^{\{s\}} \times u_{[b]}^{(k)} + c_{[b]}^{\{s\}}$ 
9   convg ← DetectConvergence( $u^{\{s\}}$ ,  $\varepsilon$ )
10  k ← k + 1
11 end
```

$u_{[b]}^{\{s\}(k)}$ on the interface, at Line 5 of Algorithm 5.3 is performed using the *MPI_ALLREDUCE* function:

$$MPI_ALLREDUCE(u_{[b]}^{(k)}, u_{[b]}^{\{s\}(k)}, MPI_SUM).$$

Due to the pattern of $u_{[b]}^{\{s\}(k)}$, which contains most of the zero values, this choice is not the most efficient way to perform the summation $\sum_{j=1}^p u_{[b]}^{\{j\}(k)}$ on the interface. In fact, only non-zero values of $u_{[b]}^{\{s\}(k)}$ correspond to the interface nodes of the sub-domain s . In general, a small number of interface nodes is associated with a given cooperating sub-domain. In practice, the summation on the interface is performed only between sub-domains which share an interface. We have also implemented a special function using (MPI_)*send/receive* for performing this assembly operation, which only exchanges and updates the interface with the sub-domains s_1 sharing an interface with the sub-domain s_2 . The main points of the algorithm are given in Algorithm 5.1 and (see Algorithm 5.2, Page 145). In case of synchronous iterations, the convergence detection and the halting procedure, Line 9 of Algorithm 5.3, is easy to control. For example, it suffices to perform an *MPI_ALLREDUCE*

$$MPI_ALLREDUCE(\&global_norm, local_norm, MPI_MAX)$$

in order to get the maximum of $\|u^{\{s\}(k+1)} - u^{\{s\}(k)}\|_\infty$ over all processes, where *local_norm* and *global_norm* denote respectively the local $\|u^{\{s\}(k+1)} - u^{\{s\}(k)}\|_\infty, s \in \{1, \dots, p\}$ and global norm $\max_{1 \leq s \leq p} \|u^{\{s\}(k+1)} - u^{\{s\}(k)}\|_\infty$.

Particular case: internal unknowns are hidden Unlike the method of Jacobi, where we have a particular splitting $M = D$ and $N = -(L + U)$, with some methods such as Gauss-Seidel's, the sub-matrix $T^{\{s\}}$ (5.9) cannot be built for each sub-domain $s \in \{1, \dots, p\}$ directly from the sub-matrix $A^{\{s\}}$ of the original matrix described in (5.11). The idea behind the general case consists in performing an exact solve in the sub-domain and iterate only on the interface. We consider the sub-structuring finite element where the internal unknowns are hidden. To describe the general case, we focus on the Gauss-Seidel method defined in Definition 4.63. As before, on the interface, the splitting is

$$A_{[bb]} = M_{[bb]} - N_{[bb]} \quad (5.23)$$

Considering the splitting given in (5.6), the splitting of the matrix $A = M - N$ can be rewritten in the form

$$A = \underbrace{\begin{pmatrix} A_{[ii]}^{\{1\}} & 0 & \cdots & 0 \\ & \ddots & & \vdots \\ 0 & A_{[ii]}^{\{s\}} & & 0 \\ \vdots & & \ddots & \vdots \\ & & & A_{[ii]}^{\{p\}} \\ A_{[bi]}^{\{1\}} & \cdots & A_{[bi]}^{\{s\}} & \cdots & A_{[bi]}^{\{p\}} & M_{[bb]} \end{pmatrix}}_M - \underbrace{\begin{pmatrix} 0 & 0 & \cdots & 0 & -A_{[ib]}^{\{1\}} \\ & \ddots & & \vdots & \vdots \\ 0 & 0 & & -A_{[ib]}^{\{s\}} \\ \vdots & & \ddots & 0 & \vdots \\ & & & 0 & -A_{[ib]}^{\{p\}} \\ 0 & \cdots & 0 & \cdots & 0 & N_{[bb]} \end{pmatrix}}_N \quad (5.24)$$

The inverse of the square matrix $M \in \mathbb{K}^{n \times n}$ can be explicitly computed

$$M^{-1} = \begin{pmatrix} A_{[ii]}^{\{1\}-1} & 0 & 0 & \cdots & 0 \\ 0 & \ddots & & & \vdots \\ 0 & 0 & A_{[ii]}^{\{s\}-1} & 0 & 0 \\ \vdots & 0 & 0 & \ddots & \vdots \\ & & 0 & 0 & A_{[ii]}^{\{p\}-1} & 0 \\ \cdots & \cdots & -M_{[bb]}^{-1} A_{[bi]}^{\{s\}} A_{[ii]}^{\{s\}-1} & \cdots & \cdots & M_{[bb]} \end{pmatrix} \quad (5.25)$$

Hence,

$$T = M^{-1}N = \begin{pmatrix} 0 & 0 & \cdots & 0 & -A_{[ii]}^{\{1\}-1} A_{[ib]}^{\{1\}} \\ & \ddots & & \vdots & \vdots \\ 0 & 0 & & -A_{[ii]}^{\{s\}-1} A_{[ib]}^{\{s\}} \\ \vdots & & \ddots & 0 & \vdots \\ & & & 0 & -A_{[ii]}^{\{p\}-1} A_{[ib]}^{\{p\}} \\ 0 & \cdots & 0 & \cdots & 0 & T_{[bb]} \end{pmatrix} \quad (5.26)$$

where

$$T_{[bb]} = M_{[bb]}^{-1} N_{[bb]} + \sum_{s=1}^p M_{[bb]}^{-1} A_{[bi]}^{\{s\}} A_{[ii]}^{\{s\}-1} A_{[ib]}^{\{s\}} \quad (5.27)$$

Or from (5.23) $A_{[bb]} = M_{[bb]} - N_{[bb]}$, i.e., $N_{[bb]} = M_{[bb]} - A_{[bb]}$, so we have,

$$T_{[bb]} = I - M_{[bb]}^{-1} \underbrace{\left(A_{[bb]} - \sum_{s=1}^p A_{[bi]}^{\{s\}} A_{[ii]}^{\{s\}-1} A_{[ib]}^{\{s\}} \right)}_{\text{Shur complement of } A \text{ on the interface}} \quad (5.28)$$

Considering the definition of $T_{[bb]}$ (5.10), i.e., $T_{[bb]} = \sum_{s=1}^p T_{[bb]}^{\{s\}}$, the splitting of $T_{[bb]}$ can be written

$$T_{[bb]}^{\{s\}} = I^{\{s\}} - M_{[bb]}^{-1} \left(A_{[bb]}^{\{s\}} - A_{[bi]}^{\{s\}} A_{[ii]}^{\{s\}-1} A_{[ib]}^{\{s\}} \right) \quad (5.29)$$

such as

$$A_{[bb]} = \sum_{s=1}^p A_{[bb]}^{\{s\}} \quad \text{and} \quad I = \sum_{s=1}^p I^{\{s\}}. \quad (5.30)$$

In practice, in addition to its local sub-matrix $A^{\{s\}}$, each sub-domain $s \in \{1, \dots, p\}$, requires only the assembly matrix $M_{[bb]}$ on the interface. At each iteration we must solve a system in each sub-domain in order to compute $A_{[ii]}^{\{s\}-1}$.

Proposition 5.3 *Under the previous conditions,*

$$\rho(T) = \rho(T_{[bb]}).$$

The convergence condition is reduced to $T_{[bb]}$ matrix $\rho(T_{[bb]}) < 1$.

5.2.4 Asynchronous sub-structuring algorithm

The asynchronous sub-structuring mathematical model is close to the multi-splitting model [218]. However, in case of sub-structuring, the sum on the interface is not weighted. Therefore, the sum of errors on the interface can not be bound by the maximum number of errors on the sub-domain interface values. Under these conditions, the classical multi-splitting convergence results cannot be used in the case of asynchronous iterations. The condition of convergence of the asynchronous multi-splitting method (see Theorem 4.61, Page 136) is necessary but not sufficient for sub-structuring. In contrast, we can use the convergence results for asynchronous splitting methods on the global fixed point and then conclude the convergence for the sub-structuring method.

This theorem was first proposed by C. Venet and F. Magoulès in [258].

Theorem 5.4 (Convergence of parallel asynchronous sub-structuring algorithms) *Let $T \in \mathbb{K}^{n \times n}$ and $c \in \mathbb{K}^n$ the square matrix and vector defined in Section 5.2.3, Page 149 such that*

$$\rho(|T|) < 1 \quad \text{and} \quad |T_{[bb]}| = \sum_{s=1}^p |T_{[bb]}^{\{s\}}|. \quad (5.31)$$

Under these conditions (5.31), the parallel associated asynchronous sub-structuring algorithm converges.

In the rest of this subsection, we give the proof of the Theorem 5.4.

Proof. Let $s \in \{1, \dots, p\}$. Let T be a $n \times n$ matrix. Let us choose $\gamma \in \mathbb{R}$, $\zeta \in \mathbb{R}$ and $w \in \mathbb{R}^n$ such that Corollary 4.51 ((see Corollary 4.51, Page 135)) is satisfied. The splitting of w can be rewritten

$$w = \left(w_{[i]}^{\{1\}}, \dots, w_{[i]}^{\{s\}}, \dots, w_{[i]}^{\{p\}}, w_{[b]} \right)^T \quad (5.32)$$

Let us define $w^{\{s\}}, \phi^{\{s\}} \in \mathbb{R}^{n^{\{s\}}}$ for each sub-domain $s \in \{1, \dots, p\}$ such that

$$w^{\{s\}} = \left(w_{[i]}^{\{s\}} \quad w_{[b]}^{\{s\}} \right)^T \quad (5.33)$$

and

$$\phi^{\{s\}} = \frac{1}{\zeta} \left(\left(|T_{[bi]}^{\{s\}}| \quad |T_{[bb]}^{\{s\}}| \right) \begin{pmatrix} w_{[i]}^{\{s\}} \\ w_{[b]} \end{pmatrix} + \frac{\zeta - \gamma}{p} w_{[b]} \right) \quad (5.34)$$

Given that $0 < \gamma < \zeta < 1$ and $w > 0$, then $\forall s > 0$, $\phi^{\{s\}} > 0$. We define for each sub-domain the quantity

$$u_{[b]}^{\{s\}*} = \left(T_{[bi]}^{\{s\}} \quad T_{[bb]}^{\{s\}} \right) \begin{pmatrix} u_{[i]}^{\{s\}*} \\ u_{[b]}^* \end{pmatrix} + c_{[b]}^{\{s\}}, \quad (5.35)$$

where

$$u_{[b]}^* = \sum_{s=1}^p u_{[b]}^{\{s\}*}. \quad (5.36)$$

Similarly,

$$u_{[i]}^{\{s\}*} = \left(T_{[ii]}^{\{s\}} \quad T_{[ib]}^{\{s\}} \right) \begin{pmatrix} u_{[i]}^{\{s\}*} \\ u_{[b]}^* \end{pmatrix} + c_{[i]}^{\{s\}}. \quad (5.37)$$

Let us define the quantity $\Delta^{(k)} = \zeta^k \Delta$ such that

$$\Delta = \max \left(\|u_0 - u^*\|_\infty^w, \max_{1 \leq s \leq p} \|u_{[b]}^{\{s\}(0)} - u_{[b]}^{\{s\}*} \|_\infty^{\phi^{\{s\}} \right) \quad (5.38)$$

and for each sub-domain s the sets $X_{[i]}^{\{s\}}$ (space of the degrees of freedom) and $X_{[b]}^{\{s\}}$ (space of the interface) such that

$$\begin{aligned} X_{[i]}^{\{s\}(k)} &= \{u_{[i]}^{\{s\}} \in X_{[i]}^{\{s\}}, |u_{[i]}^{\{s\}} - u_{[i]}^{\{s\}*}| \leq \Delta^{(k)} w_{[i]}^{\{s\}}\}, \\ X_{[b]}^{\{s\}(k)} &= \{u_{[b]}^{\{s\}} \in X_{[b]}^{\{s\}}, |u_{[b]}^{\{s\}} - u_{[b]}^{\{s\}*}| \leq \Delta^{(k)} \phi^{\{s\}}\}, \end{aligned} \quad (5.39)$$

with $X^{\{s\}(k)} = X_{[i]}^{\{s\}(k)} \times X_{[b]}^{\{s\}(k)}$. So, for each sub-domain s , the set $X^{\{s\}}$ is such that $X^{\{s\}} = X_{[i]}^{\{s\}} \times X_{[b]}^{\{s\}}$. We have $X^{\{s\}(k)} \subset X^{\{s\}}$.

Let $X_{[b]}^{(k)}$ be a set such that

$$X_{[b]}^{(k)} = \{u_{[b]} \in X_{[b]}, |u_{[b]} - u_{[b]}^*| \leq \Delta^{(k)} w_{[b]}\}. \quad (5.40)$$

Let us recall the set \widehat{X} described in (see eq. (4.11), Page 129):

$$\widehat{X} = X^{\{1\}} \times \dots \times X^{\{s\}} \times \dots \times X^{\{p\}} \quad (5.41)$$

Then, we define the set $X^{(k)} \subset \widehat{X}$ as follows

$$\widehat{X}^{(k)} = X^{\{1\}(k)} \times \dots \times X^{\{s\}(k)} \times \dots \times X^{\{p\}(k)} \quad (5.42)$$

The proof of the asynchronous convergence will be based on the following statement (based on the nested box theorem by D.P. Bertsekas [233]):

$$\forall k \in \mathbb{N}, \hat{u} \in \widehat{X}^{(k)} \Rightarrow \widehat{\mathcal{T}}(\hat{u}) \in \widehat{X}^{(k+1)}.$$

Let us fix k such that $\hat{u} \in \widehat{X}^{(k)}$, so

$$\forall s, u_{[i]}^{\{s\}} \in X_{[i]}^{\{s\}} \text{ and } u_{[b]}^{\{s\}} \in X_{[b]}^{\{s\}}.$$

The sum at the interface is satisfied $u_{[b]} = \sum_{s=1}^p u_{[b]}^{\{s\}} \in \widehat{X}_{[b]}$ (5.40).

First, let us prove that $u_{[b]} \in X_{[b]}$. It suffices to demonstrate that $|u_{[b]} - u_{[b]}^*| \leq \Delta^{(k)} w_{[b]}$.

$$\begin{aligned} |u_{[b]} - u_{[b]}^*| &\leq \sum_{s=1}^p |u_{[b]}^{\{s\}} - u_{[b]}^{\{s\}*}| \\ &\leq \sum_{s=1}^p \Delta^{(k)} \phi^{\{s\}} = \Delta^{(k)} \sum_{s=1}^p \phi^{\{s\}} \quad \text{by (5.39), } u_{[b]}^{\{s\}} \in X_{[b]}^{\{s\}} \end{aligned}$$

Or

$$\begin{aligned} \sum_{s=1}^p \phi^{\{s\}} &= \sum_{s=1}^p \frac{1}{\zeta} \left((|T_{[bi]}^{\{s\}}| \quad |T_{[bb]}^{\{s\}}|) \begin{pmatrix} w_{[i]}^{\{s\}} \\ w_{[b]}^{\{s\}} \end{pmatrix} + \frac{\zeta - \gamma}{p} w_{[b]} \right) \\ &= \frac{1}{\zeta} \sum_{s=1}^p \left(|T_{[bi]}^{\{s\}}| w_{[i]}^{\{s\}} + |T_{[bb]}^{\{s\}}| w_{[b]} + \frac{\zeta - \gamma}{p} w_{[b]} \right) \\ &= \frac{1}{\zeta} \left(\sum_{s=1}^p |T_{[bi]}^{\{s\}}| w_{[i]}^{\{s\}} + w_{[b]} \sum_{s=1}^p |T_{[bb]}^{\{s\}}| + \sum_{s=1}^p \frac{\zeta - \gamma}{p} w_{[b]} \right) \end{aligned}$$

By the Theorem 5.4, we have $\sum_{s=1}^p |T_{[bb]}^{\{s\}}| = |T_{[bb]}|$, so

$$\sum_{s=1}^p \phi^{\{s\}} = \frac{1}{\zeta} \left(\sum_{s=1}^p |T_{[bi]}^{\{s\}}| w_{[i]}^{\{s\}} + |T_{[bb]}| w_{[b]} + (\zeta - \gamma) w_{[b]} \right)$$

In addition, according to Corollary 4.51 (see Corollary 4.51, Page 135), we have

$$\sum_{s=1}^p |T_{[bi]}^{\{s\}}| w_{[i]}^{\{s\}} + |T_{[bb]}| w_{[b]} \leq \gamma w,$$

because it is a restriction of $|T| w$ to $X_{[b]}$. Hence,

$$\sum_{s=1}^p \phi^{\{s\}} \leq \frac{1}{\zeta} (\gamma w + (\zeta - \gamma) w_{[b]}) \leq w_{[b]}.$$

Finally, $|u_{[b]} - u_{[b]}^*| \leq \Delta^{(k)} w_{[b]}$, so by (5.40), we have $u_{[b]} \in X_{[b]}$.

Now, let us prove that $\widehat{\mathcal{T}}(\hat{u}) \in \widehat{X}^{(k+1)}$.

It suffices to demonstrate that for all processes $s \in \{1, \dots, p\}$, $\widehat{\mathcal{T}}^{\{s\}}(\hat{u}) \in X^{\{s\}(k+1)}$.

- First, let us show that $\forall s \in \{1, \dots, p\}$, $(T_{[ii]}^{\{s\}} \quad T_{[ib]}^{\{s\}}) \begin{pmatrix} u_{[i]}^{\{s\}} \\ u_{[b]} \end{pmatrix} + c_{[i]}^{\{s\}} \in X_{[i]}^{\{s\}(k+1)}$:

$$\begin{aligned} \left| (T_{[ii]}^{\{s\}} \quad T_{[ib]}^{\{s\}}) \begin{pmatrix} u_{[i]}^{\{s\}} \\ u_{[b]} \end{pmatrix} + c_{[i]}^{\{s\}} - u_{[i]}^{\{s\}*} \right| &= \left| (T_{[ii]}^{\{s\}} \quad T_{[ib]}^{\{s\}}) \begin{pmatrix} u_{[i]}^{\{s\}} - u_{[i]}^{\{s\}*} \\ u_{[b]} - u_{[b]}^* \end{pmatrix} \right| && \text{by (5.37)} \\ &\leq (|T_{[ii]}^{\{s\}}| \quad |T_{[ib]}^{\{s\}}|) \begin{pmatrix} |u_{[i]}^{\{s\}} - u_{[i]}^{\{s\}*}| \\ |u_{[b]} - u_{[b]}^*| \end{pmatrix} \\ &\leq (|T_{[ii]}^{\{s\}}| \quad |T_{[ib]}^{\{s\}}|) \begin{pmatrix} \Delta^{(k)} w_{[i]}^{\{s\}} \\ \Delta^{(k)} w_{[b]} \end{pmatrix} \\ &\leq \Delta^{(k)} (|T_{[ii]}^{\{s\}}| w_{[i]}^{\{s\}} + |T_{[ib]}^{\{s\}}| w_{[b]}) \end{aligned}$$

Or $\sum_{s=1}^p |T_{[ii]}^{\{s\}}| w_{[i]}^{\{s\}} + |T_{[ib]}^{\{s\}}| w_{[b]} \leq \zeta w$ because it is a restriction of $|T| w$ to $X_{[i]}^{\{s\}}$, so

$$\left| (T_{[ii]}^{\{s\}} \quad T_{[ib]}^{\{s\}}) \begin{pmatrix} u_{[i]}^{\{s\}} \\ u_{[b]} \end{pmatrix} + c_{[i]}^{\{s\}} - u_{[i]}^{\{s\}*} \right| \leq \Delta^{(k)} \zeta w_{[i]}^{\{s\}} = \Delta^{(k+1)} w_{[i]}^{\{s\}}, \quad \mathbf{QED.}$$

- Second, let us show that $\forall s \in \{1, \dots, p\}$, $(T_{[bi]}^{\{s\}} \quad T_{[bb]}^{\{s\}}) \begin{pmatrix} u_{[i]}^{\{s\}} \\ u_{[b]} \end{pmatrix} + c_{[b]}^{\{s\}} \in X_{[b]}^{\{s\}(k+1)}$:

$$\begin{aligned} \left| (T_{[bi]}^{\{s\}} \quad T_{[bb]}^{\{s\}}) \begin{pmatrix} u_{[i]}^{\{s\}} \\ u_{[b]} \end{pmatrix} + c_{[b]}^{\{s\}} - u_{[b]}^{\{s\}*} \right| &= \left| (T_{[bi]}^{\{s\}} \quad T_{[bb]}^{\{s\}}) \begin{pmatrix} u_{[i]}^{\{s\}} - u_{[i]}^{\{s\}*} \\ u_{[b]} - u_{[b]}^* \end{pmatrix} \right| && \text{by (5.35)} \\ &\leq (|T_{[bi]}^{\{s\}}| \quad |T_{[bb]}^{\{s\}}|) \begin{pmatrix} |u_{[i]}^{\{s\}} - u_{[i]}^{\{s\}*}| \\ |u_{[b]} - u_{[b]}^*| \end{pmatrix} \\ &\leq (|T_{[bi]}^{\{s\}}| \quad |T_{[bb]}^{\{s\}}|) \begin{pmatrix} \Delta^{(k)} w_{[i]}^{\{s\}} \\ \Delta^{(k)} w_{[b]} \end{pmatrix} \\ &\leq \Delta^{(k)} (|T_{[bi]}^{\{s\}}| w_{[i]}^{\{s\}} + |T_{[bb]}^{\{s\}}| w_{[b]}) \\ &\leq \Delta^{(k)} \left(\zeta \phi^{\{s\}} - \frac{\zeta - \gamma}{p} w_{[b]} \right) && \text{by (5.34)} \\ &\leq \Delta^{(k)} \zeta \phi^{\{s\}} = \Delta^{(k+1)} \zeta \phi^{\{s\}}, \quad \mathbf{QED.} \end{aligned}$$

Finally, with these two points, we can conclude that

$$\forall s \in \{1, \dots, p\}, \widehat{\mathcal{T}}^{\{s\}}(\hat{u}) \in X^{\{s\}(k+1)}.$$

In addition, the following results can be stated:

- $\lim_{k \rightarrow +\infty} \Delta^{(k)} = \lim_{k \rightarrow +\infty} \zeta^k \Delta = 0$.
- $\lim_{k \rightarrow +\infty} \widehat{X}^{(k)} = \{\widehat{u}^*\}$.
- $\forall k \in \mathbb{N}, \widehat{X}^{(k+1)} \subset \widehat{X}^{(k)}$ (see Assumption 4.41, Page 132).
- $\forall k \in \mathbb{N}, \widehat{X}^{\{s\}(k+1)} \subset \widehat{X}^{\{s\}(k)}$ (see Assumption 4.41, Page 132).

With an appropriate choice of Δ (5.38), we can find an initial solution so that $u^* \in \widehat{X}^{(0)}$. Under these conditions, the Theorem 5.4 guarantees the convergence of parallel asynchronous sub-structuring algorithms. \square

Algorithm 5.4 shows the main points for the algorithm of parallel asynchronous sub-structuring iterative methods.

Algorithm 5.4: Algorithm of the parallel iterative asynchronous sub-structuring methods

```

input   :  $s \in \{1, \dots, p\}$ 
            $T^{\{s\}}$ : sub-matrix of the sub-domain  $s$ ,  $c^{\{s\}}$ : sub-vector of the sub-domain  $s$  (5.9),
            $u^{\{s\}(0)} = (u_{[i]}^{\{s\}(0)} \quad u_{[b]}^{(0)})$ : initial solution of whole sub-domain  $s$ ,
            $u_{[i]}^{\{s\}(0)}$  (internal) and  $u_{[b]}^{\{s\}(0)}$  (interface) from  $u^{\{s\}(0)}$ ,
            $\varepsilon$ : tolerance threshold,  $K$ : maximum number of iterations
output  :  $u^{\{s\}} = (u_{[i]}^{\{s\}} \quad u_{[b]}^{\{s\}})$ : local solution
variable:  $k, j, convg$ 
variable:  $buff\{s_1, data\}$ : received buffer from  $buff.s_1$ , which contains  $buff.data$ 
variable:  $u_{[b]}$ 

1  $convg \leftarrow false; k \leftarrow 0$ 
2 foreach process  $j$  do
3   | // -- allocate  $u_{[b]}^{\{j\}}$ 
4   |  $u_{[b]}^{\{j\}(k)} \leftarrow 0$  // -- initialize to zero
5 end
6 // -- allocate  $u_{[b]} \leftarrow u_{[b]}^{(0)}$ 
7 while .not. convg .OR.  $k < K$  do
8   | // -- update the nodes on the interface
9   |  $u_{[b]}^{\{s\}(k+1)} = T_{[bi]}^{\{s\}} \times u_{[i]}^{\{s\}(k)} + T_{[bb]}^{\{s\}} \times u_{[b]}^{(k)} + c_{[b]}^{\{s\}}$ 
10  | foreach process  $j$  do
11  |   | // -- asynchronous non blocking send
12  |   |  $SEND(j, u_{[b]}^{\{s\}(k+1)})$ 
13  | end
14  |  $u_{[i]}^{\{s\}(k+1)} = T_{[ii]}^{\{s\}} \times u_{[i]}^{\{s\}(k)} + T_{[ib]}^{\{s\}} \times u_{[b]}^{(k)} + c_{[i]}^{\{s\}}$ 
15  | foreach message buff received do
16  |   | // -- continuous receive (always listening)
17  |   |  $u_{[b]}^{\{buff.s_1\}(k)} \leftarrow buff.data$  // -- RECEIVE(buff)
18  | end
19  | // -- assembly of sub-domain  $s$  along the interface
20  |  $u_{[b]}^{(k)} \leftarrow \sum_{j=1}^p u_{[b]}^{\{j\}(k)}$ 
21  |  $convg \leftarrow DetectConvergence(u^{\{s\}}, \varepsilon)$ 
22  |  $k \leftarrow k + 1$ 
23 end

```

The exchange at the interface of the asynchronous version differs to synchronous by the absence of waiting and synchronization points ((see Algorithm 5.1, Page 145) and (see Algorithm 5.2, Page 145)).

Remark 5.5 The total interface can be split into small interfaces, each involving only two neighboring sub-domains. This allows to transfer less data between the processes and reduces the number of operations to perform $\sum_{j=1}^p u_{[b]}^{\{j\}^{(k)}}$ (Line 20 of Algorithm 5.4).

In this asynchronous algorithm (Algorithm 5.4), receptions are non-blocking whereas they were blocking in the synchronous version (Algorithm 5.3). Therefore, after the sending, a process takes the last version of its neighbors' messages if new messages have arrived since the previous iteration. In Algorithm 5.4, restarting a new iteration without receiving any new messages leads to the same computation. In this way, to improve this algorithm and avoid to perform unnecessary iterations, we can choose to verify if a new message has arrived at each iteration. Otherwise it would probably be better to wait for a few micro seconds and then to test again if a new message has arrived. According to the number of neighbors, it may be judicious to wait for the reception of a given number of messages.

In case of asynchronous iterations, the convergence detection and the halting procedure, Line 21 of Algorithm 5.4 , is one of the main points of the algorithm.

In the following, we give a brief overview of the existing convergence detectors.

5.2.5 Convergence detection

The implementation of an asynchronous iterative algorithm may seem easier to achieve compared to a synchronous algorithm since there is no more synchronization. However, the convergence detection is different and requires a special attention. J.M Bahi *et al.* [228] show that according to the dedicated architecture, a centralized mechanism can either be used for a parallel architecture or a cluster with a high-speed network with quite a limited number of processors. On the other hand, with a distributed cluster or a grid with a large number of processors, this mechanism is completely inconceivable. Two versions are proposed by J.M Bahi *et al.* [228], which use a centralized version or a decentralized version in order to detect the convergence.

At the time of the implementation of the first algorithm developed by the team of F. Magoulès, the only publicly available library to help programs using asynchronous algorithms was Jace's [222] [223], which is programmed in Java and uses only TCP communication to exchange data. Jace was not suitable for our studies. There was also CRAC [225] (*a Grid Environment to Solve Scientific Applications with Asynchronous Iterative Algorithms*) in C++ proposed by R. Couturier *et al.* in 2007.

In order to easily program asynchronous algorithms, our team had the idea to develop an appropriate communication library. This library, named *Jack*, was first developed by C. Venet (PhD student of our team) and F. Magoulès [258]. This library is written on the top of MPI [259] and facilitates sending and receiving data asynchronously. In the asynchronous mode, the messages are placed in a queue until the communication medium is free to send the messages. If the tunnel is busy, *i.e.*, there is already a message inside the queue with the same tag and destination, the new message overwrites the previous one. This choice avoids having to put the new message at the end of the queue. In this way, the message in the queue awaiting

to be sent will be the updated one. Concerning the reception, the messages are placed in a receiving queue. As for the sending, if the tunnel is busy, *i.e.*, there is already a message inside the queue with the same tag and source, then the most recent message overwrites the previous one, instead of being put at the end of the queue. By this way, the message received will be the last sent.

C. Venet and F. Magoulès [258] have added a function to detect the global convergence of iterative algorithms. They use the convergence detection presented by J.M Bahi *et al.* in 2005 [227]. The global convergence is based on the estimation of local convergence. During my work, I added an extension based on the decentralized version proposed by J.M Bahi *et al.* [228] in 2008. For more details on the Jack library, the readers are referred to Section 5.1 (*Presentation of JACK architecture*) of [258].

5.2.6 Implementation on a cluster of multi-core-GPUs

In case of multi-core/multi-GPUs, the mechanism is identical to a multi-core system except that all local computations are performed on the GPU card associated with the process. The update of the solution at interface nodes require data transfers between CPU and GPU, which is the main drawback of the GPU version. At each iteration, each process copies its results from GPU to CPU, and then the solution is updated at interface nodes on CPU. Finally, the updated solution is copied back to GPU for the next iteration. The multi-core/GPU algorithm takes advantages of the effectiveness of the GPU computation in each process. The algorithm is mostly efficient when the sub-problems are also large in size. Given the computational power of GPUs, communication is the key point of the algorithm. The efficiency of this algorithm depends on the effectiveness of the communication latencies whenever synchronous. The asynchronous model may be an effective solution to improve the algorithm. In summary, all parallel and intensive computations on local data are performed on GPUs and reduction operations to compute global results are carried out by CPUs.

5.2.7 Fault-tolerance and iteration penalty behavior

In this subsection we aim to analyze the behavior of parallel algorithms in terms of *fault-tolerance* and in terms of iteration penalization. We consider synchronous iterations. The concept of *iteration penalization* consists in stopping the evolution of the solution during a certain time, *i.e.*, the penalized process keeps the current solution during the idle state. In this analysis, the *fault-tolerance* consists in restarting the solution to the corresponding initial guess and in stopping sending and receiving from cooperating processors. This enables us to simulate the breakdown of a process. This concept simulates asynchronous iterations. In this analysis, we give some heuristic for understanding the behavior of chaotic iterations. We consider that only one process is penalized (idle state or breakdown). We do not pay attention to communication and synchronization. Let us note that process synchronization is the main difference between synchronous and asynchronous algorithms.

Considering the parallel platform, $\mathcal{PS}_{system}\{p\}$, capable of executing p processes of execution of \mathcal{PS} . Let the original domain Ω be decomposed in p sub-domains: $\{S^{\{1\}}, \dots, S^{\{p\}}\}$.

Theorem of delay (iteration penalization)

Proposition 5.6 Let $(i, j) \in \{1, \dots, p\}^2$, $i \neq j$, such as the sub-domain $S^{\{i\}}$ has a delay, $S^{\{j\}}$ another one without delay. Their number of iterations satisfy the following $N_{[para]}^{\{i\}} \leq N_{[para]}^{\{j\}}$ where para is synchronous (sync) or asynchronous (async). In synchronous case, there is no delay. So, from Proposition 5.6, we have $N_{[sync]}^{\{i\}} = N_{[sync]}^{\{j\}} = N_{[seq]}$, where $N_{[seq]}$ is the number of iterations of the algorithm.

Definition 5.7 Let p -algorithm be an algorithm which converges synchronously with the Jacobi method. Let $i \in \{1, \dots, p\}$, such that $S^{\{i\}}$ is the only sub-domain with delay, i.e., $S^{\{i\}}$ presents an idle state. Let n be the n^{th} iteration of the algorithm. Let us define n^{r-} the iteration where the idle state (pause) is activated for the processor associated with $S^{\{i\}}$ and n^{r+} the iteration where the idle state ends. Let us define τ the pause duration expressed as: $\tau = n^{r+} - n^{r-}$. The sub-domain in idle state does not update its local solution. The other sub-domains continue to update their solutions normally. This scheme models chaotic iterations (basic asynchronous scheme). The asynchronous scheme is simulated here by the non-updating of the solution.

Theorem 5.8 (Theorem of delay) Let \mathcal{S} be a synchronous p -algorithm which converges in $N_{[\mathcal{S}]} = N_{[seq]}$ iterations and \mathcal{C} a chaotic (asynchronous) p -algorithm which converges with Definition 5.7 in $N_{[\mathcal{C}]}$ iterations.

The number of iterations of the sub-domain with delay, $S^{\{i\}}$, of the chaotic algorithm, \mathcal{C} , is largely inferior to that of the pure synchronous algorithm, \mathcal{S} , i.e.

$$\begin{cases} N_{[\mathcal{C}]}^{\{j\}} = N_{[\mathcal{C}]} > \max_j N_{[\mathcal{S}]}^{\{j\}} = N_{[\mathcal{S}]}, & \forall j \in \{1, \dots, p\} \setminus \{i\} \\ N_{[\mathcal{C}]}^{\{i\}} \ll N_{[\mathcal{S}]}^{\{i\}} = N_{[\mathcal{S}]} \end{cases} \quad (5.43)$$

where $i \in \{1, \dots, p\}$, $S^{\{i\}}$, is the only sub-domain with delay.

Proof. Let us show that $N_{[\mathcal{C}]}^{\{j\}} = N_{[\mathcal{C}]} > \max_j N_{[\mathcal{S}]}^{\{j\}} = N_{[\mathcal{S}]}$, $\forall j \in \{1, \dots, p\} \setminus \{i\}$.

The notation $S \xrightarrow[\text{para}]{c.v.} n$ means the sub-domain S converges in n iterations.

$\forall q \in \{1, \dots, p\}$, sub-domain $S^{\{q\}}$ converges in $N_{[\mathcal{S}]}^{\{q\}} = N_{[\mathcal{S}]}$ iterations for pure synchronous algorithms, i.e. $S^{\{i\}} \xrightarrow[\text{sync}]{c.v.} N_{[\mathcal{S}]}^{\{q\}} = N_{[\mathcal{S}]}$. For an asynchronous algorithm (chaotic synchronous)

with $S^{\{i\}}$ the only sub-domain with delay (n^{r-}, n^{r+}), we have $S^{\{i\}} \xrightarrow[\text{async}]{c.v.} N_{[\mathcal{C}]}^{\{i\}}$ and $S^{\{j\}} \xrightarrow[\text{async}]{c.v.} N_{[\mathcal{C}]}^{\{j\}} = N_{[\mathcal{C}]}$.

In chaotic iterations, all sub-domains without delay will need at least $N_{[\mathcal{S}]}$ iterations to converge. Nevertheless, the sub-domain $S^{\{i\}}$ has a delay, therefore cooperating processors ($S^{\{i\}}(l)$) will need to perform additional iterations, n^+ , to converge due to the use of the old solution from the idle processor. So, we have $S^{\{l\}} \xrightarrow[\text{async}]{c.v.} N_{[\mathcal{S}]}^{\{l\}} + n^+$. In conclusion, if the algorithm converges with a sub-domain presenting a delay, the other sub-domains will require more than the number of iterations in a pure synchronous case. For the idle processor, we have $S^{\{i\}} \xrightarrow[\text{async}]{c.v.} N_{[\mathcal{C}]}^{\{i\}} = N_{[\mathcal{C}]} - ((n^{r+} - n^{r-}))$. \square

- If $N_{[\mathcal{C}]} \gg n^{r-}$ then $N_{[\mathcal{C}]}^{\{i\}} \ll N_{[\mathcal{C}]}$.
- If $n^{r+} \gg n^{r-}$ then a steady state appears and the asynchronous algorithm loses its interest.

- If $n^{r^-} \geq N_{[S]}$, the algorithm is equivalent to a pure synchronous one.
- If $n^{r^-} + \tau = n^{r^+} \geq N_{[S]}$ then $N_{[C]}^{\{i\}} = N_{[S]}$.

This theorem helps observe the behavior of the convergence of the asynchronous algorithm. In the chaotic algorithm defined in Definition 5.7, the number of iterations of non-idle processors are identical, which is due to the synchronization of our chaotic model. In general, the number of iterations of the asynchronous algorithm is different in each processor and differs at each execution.

Fault-tolerance Now, we simulate *fault-tolerance* by restarting the solution to the corresponding initial guess. Moreover, at the breakdown point, *i.e.*, when the solution is reset, the sending and receiving from the broken processor to and from the cooperating processors are disabled. This allows us to simulate the breakdown of a process.

Theorem 5.9 (Fault-tolerance) *Let p -algorithm be an algorithm which converges synchronously with the Jacobi method. Let $i \in \{1, \dots, p\}$, such that $S^{\{i\}}$ is the only sub-domain breakdown. When the processor i is woken up and has finished its current iteration after a time t_i , it benefits from the solution computed by the other processors $j \in \{1, \dots, p\}$, $j \neq i$ during t_i , and thus the processor i performs less iterations than execution without breakdown.*

Proposition 5.10 (Fault-tolerance) *Let p -algorithm be an algorithm and an execution \mathcal{E} that satisfies Theorem 5.9. Let $i \in \{1, \dots, p\}$, such that $S^{\{i\}}$ is the only sub-domain breakdown. Let $N_{[S]}$ be the number of iterations obtained from the execution of the pure synchronous algorithm. Let $N_{[C]}^{\{i\}}$ be the number of iterations obtained from the execution \mathcal{E} . Let us suppose that processor i breaks down at the k -th iteration. Then, the processor i performs*

$$N_{[C]}^{\{i\}} = N_{[C]}^{\{j\}} - k, \quad \forall j \in \{1, \dots, p\} \setminus \{i\}. \quad (5.44)$$

Let us remark that $N_{[C]}^{\{j\}} = N_{[C]}^{\{l\}}$, $\forall j, l \in \{1, \dots, p\} \setminus \{i\}$ for the given scheme. In a real asynchronous scheme, usually, the number of iterations of each processor is different. Moreover, in general, asynchronous algorithms require a greater number of iterations before the convergence than in the synchronous case.

The numerical analysis of the two presented theorems will be given in the numerical result section (see Section 5.4, Page 174).

5.3 Theoretical speed-up of the fully parallelizable iterative method

In this section, my contribution is the theorem for the theoretical speed-up of synchronous and asynchronous parallelizable iterative methods.

5.3.1 Background and motivations

Unless otherwise indicated, the results of speed-ups presented in this thesis do not take into account the initialization (input, ...) and output writing times. In practice, in a standard implementation, these operations are sequential. When the total program running time is used, the limits of the parallelization become clear. In parallel programming, the *speed-up* measures

how much a parallel algorithm is faster than the corresponding sequential algorithm. Let us define $\mathcal{T}(p)$ the time required to finish an algorithm being executed on p processes, which will be denoted as p -algorithm and $\mu \in [0; 1]$ the proportion of the parallel algorithm that is strictly sequential. The time of the p -algorithm is expressed as

$$\mathcal{T}(p) = \mathcal{T}(1) \left(\mu + \frac{1}{p}(1 - \mu) \right). \quad (5.45)$$

The theoretical speed-up $\mathcal{S}(p)$, which corresponds to the ratio between the execution time of a serial algorithm and those of parallel algorithms executed on a system of p processes of execution, can be written as follows:

$$\mathcal{S}(p) = \frac{\mathcal{T}(1)}{\mathcal{T}(p)} = \frac{1}{\mu + \frac{1}{p}(1 - \mu)}. \quad (5.46)$$

Remark 5.11 The speed-up is ideal for $\mu = 0$ ensures to be linear, *i.e.*, $\mathcal{S}(p) = p$. Let us assume that $\mathcal{T}(p) \geq \mathcal{T}(1)$, where the equality is obtained when the sequential and parallel algorithm are equivalent in terms of executing times, *i.e.*, $\mathcal{T}(p) = \mathcal{T}(1)$, which means $\mathcal{S}(p) = 1$ with $\mu = 1$.

Definition 5.12 The efficiency of a p -algorithm, $\mathcal{E}(p)$, is a metric performance defined as

$$\mathcal{E}(p) = \frac{\mathcal{S}(p)}{p} = \frac{\mathcal{T}(1)}{p\mathcal{T}(p)} = \frac{1}{1 + \mu(p - 1)}. \quad (5.47)$$

As with any optimization method, the first step is to determine the critical parts of a code, those for which the benefit is potentially greater.

Theorem 5.13 (General Amdahl's law) *Amdahl's law states that the maximum speed-up that can be obtained on a system using p processes is*

$$\mathcal{S}(p) = \frac{1}{(1 - \gamma) + \frac{\gamma}{p}}, \quad (5.48)$$

where $\gamma \in [0; 1]$ is the fraction of the algorithm that can be parallelized and $(1 - \gamma)$ those cannot be parallelized, *i.e.*, stay sequential.

Remark 5.14 Let us remark that (5.48) is obtained for $\mu = 1 - \gamma$ in (5.46).

Theorem 5.15 (Amdahl's law, global speed-up from local speed-up) *Amdahl's law propounds that the global speed-up, \mathcal{S}_g , of an algorithm with a local speed-up, \mathcal{S}_l on a proportion $\gamma \in [0; 1]$ of the serial execution time is:*

$$\mathcal{S}_g = \frac{1}{(1 - \gamma) + \frac{\gamma}{\mathcal{S}_l}}. \quad (5.49)$$

Remark 5.16 For a partially parallel algorithm being executed on n thread(s) of execution with an efficiency λ , the local speed-up is $\mathcal{S}_l(n) = \lambda n$ and (5.49) gives:

$$\mathcal{S}_g(n) = \frac{1}{(1 - \gamma) + \frac{\gamma}{\lambda n}}. \quad (5.50)$$

Remark 5.17 *Amdahl's law* clearly shows that the maximum speed-up is limited by $\frac{1}{1 - \gamma}$, *i.e.*, the part of the algorithm that can be made parallel. When we add new process to the program, the gain from the addition of processes is smaller.

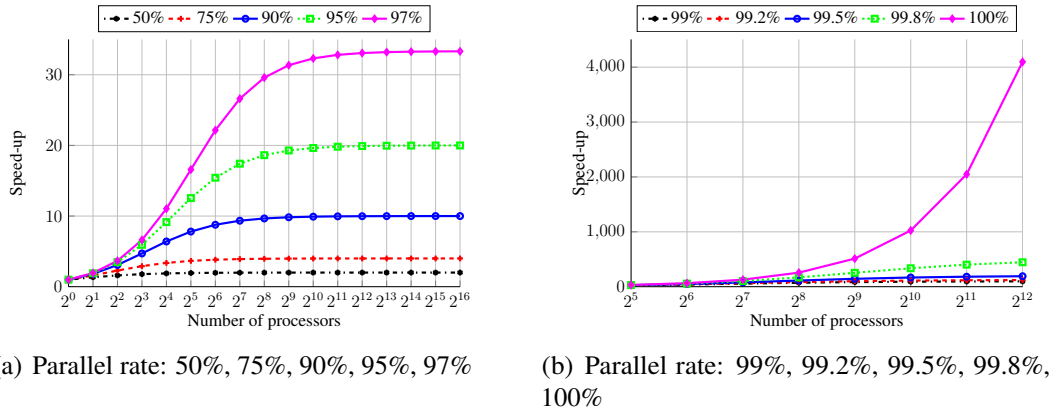


FIGURE 5.3: Speed-up upon the number of processes

Figure 5.3(a) and Figure 5.3(b) illustrate the speed-up of an algorithm executed in parallel on a system using several processes. According to *Amdahl's law*, the global speed-up of the algorithm is limited by the portion strictly sequential. As we can see in Figure 5.3(a), when the portion that can be parallelized consists of 97% of the algorithm, the theoretical maximum speed-up achievable is 33x.

Figure 5.4 and Figure 5.5 present the optimal speed-up of the program based on the proportion of the part that can be parallelized.

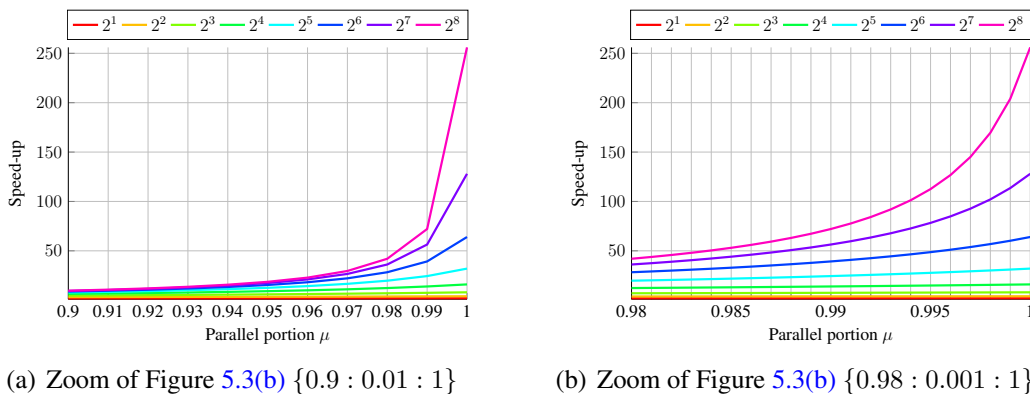
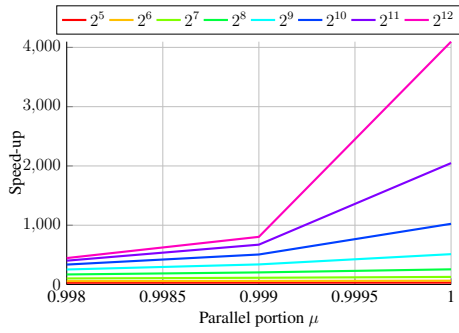
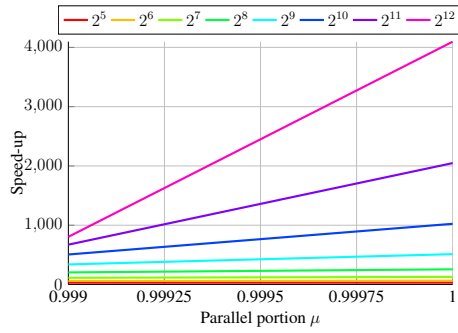


FIGURE 5.4: Speed-up upon the parallel portion of an algorithm with $2^1, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7, 2^8$ processes

Figure 5.5 shows that when a program is fully (or very close) parallelizable ($\mu \rightarrow 100\%$), the speed-up becomes linear based upon the number of processes.



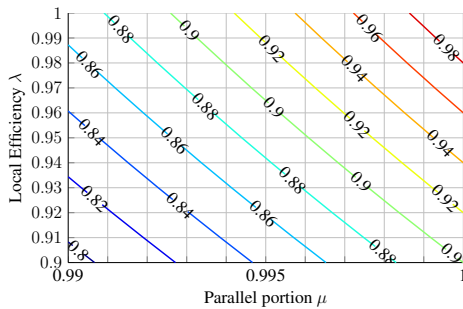
(a) Zoom of Figure 5.3(a) $\{0.998 : 0.0005 : 1\}$



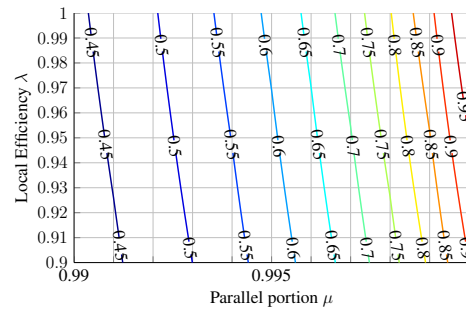
(b) Zoom of Figure 5.3(a) $\{0.999 : 0.00025 : 1\}$

FIGURE 5.5: Speed-up upon the parallel portion of an algorithm with $2^5, 2^6, 2^7, 2^8, 2^9, 2^{10}, 2^{11}, 2^{12}$ processes

Figure 5.6(a), Figure 5.6(b), Figure 5.7(a) and Figure 5.7(b) describe the global efficiency of an algorithm respectively using a system of 16, 128, 256 and 512 processes based on the proportion μ of the fraction of the algorithm that is parallelized and the efficiency λ of the parallel part.

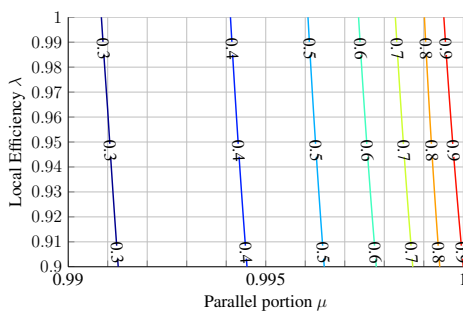


(a) 2^4 (16) processes

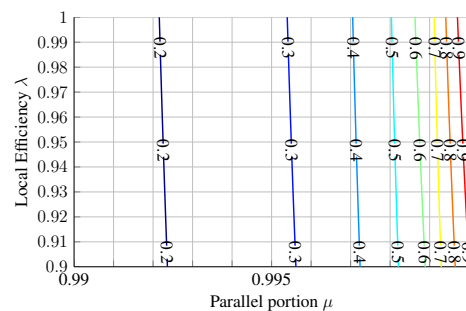


(b) 2^7 (128) processes

FIGURE 5.6: Global efficiency with 16 (a) and 128 (b) processes upon parallel proportion μ and parallel local efficiency λ



(a) 2^8 (256) processes



(b) 2^9 (512) processes

FIGURE 5.7: Global efficiency with 256 (a) and 512 (b) processes upon parallel proportion μ and parallel local efficiency λ

In Figure 5.6 and Figure 5.7 the point $(0.99, 0.9)$ represents a program with only 1% of sequential execution and 90% of efficiency of the remaining portion, which is sufficient to diminish the global efficiency to 80%, 45%, 30% and 20% for 16, 128, 256 and 512 processes respectively. Figure 5.6 and Figure 5.7 clearly show that an application must be fully

parallelizable in order to have a good scalability. When the application is fully parallelizable ($\mu = 1$), the scalability is good. For given examples, the efficiency is higher than 90% for 16, 128, 256 and 512 processes. With a given portion of sequential execution, the global efficiency of a program depends on the efficiency of the remaining portion, *i.e.*, it increases (decreases) when the efficiency of the remaining portion also increases (decreases).

In order to effectively use a lot of processes, the initialization (input, ...) and the output result gathering must also be performed in parallel. Sub-structuring methods [260] [172] (see Chapter 5.2, Page 142) and domain decomposition methods [16] [18] [19] [100] (see Chapter 6, Page 201) are two solutions which allow the whole code to be parallel. The problem is divided into sub-parts (sub-domains, sub-structures) executed in parallel such that the input (loading) and output (result gathering) can be performed in parallel.

5.3.2 Speed-up of the fully parallelizable iterative method in parallel computing

In this subsection, we study the performance of parallel synchronous and asynchronous iterative algorithms, compared to a sequential algorithm. An analysis of the theoretical speed-up is the subject of this part. However, the asynchronous methods will be the subject of a particular study. For the sake of generality, we give an analysis valid for all iterative methods that require at most four different operations: vector updates (saxpy), the element-wise product, the dot product (or inner product) and the matrix-vector product. In this analysis we only consider iterative methods where the iterations can be entirely parallelizable, *e.g.*, the Jacobi method. Before starting the analysis of the theoretical speed-up, we detail the computation complexities of the different operations of the algorithm.

Let us consider $A \in \mathbb{K}^{n \times n}$ a square matrix. Let $\mathcal{PS}_{system}\{p\}$ be a system capable of executing p processes of execution of \mathcal{PS} .

Let us define \mathcal{C}_{op} and \mathcal{C}_{op}^{para} the complexity of the operation “ op ” respectively on sequential and parallel computing, and n_{op} the number of “ op ” operations to be performed in the algorithm. In the following, we consider “ $op = mvp$ ” for the matrix-vector product, “ $op = dot$ ” for the dot product operation, “ $op = upd$ ” for the update operations and “ $op = ewp$ ” for the element-wise product. In the matrix-vector product, we have to perform $\sum_j a_{ij}x_j$, $i = 1 : n$ and in dot product we compute $\sum_i a_i b_i$, $i = 1 : n$. At each sum of these two operations, we have to carry out two arithmetic operations: one addition and one multiplication.

Definition 5.18 (Complexity of the sequential matrix-vector product) The complexity of the sequential matrix-vector multiplication is $\mathcal{C}_{mvp} = c_{mvp}.n$, where c_{mvp} is a constant that depends on the structure of the matrix and the number of arithmetic operations.

Definition 5.19 (Complexity of the sequential dot product) The computation costs of the dot product is $\mathcal{C}_{dot} = c_{dot}.n$, where c_{dot} is a constant.

Definition 5.20 (Complexity of the sequential element-wise product) Computation of the element-wise multiplication operation $z_i = x_i.y_i$, $i = 1 : n$ in sequential, which requires one multiplication for each element by the element operation, is $\mathcal{C}_{ewp} = c_{ewp}.n$, where c_{ewp} is a constant.

Definition 5.21 (Complexity of the sequential Saxpy) The complexity of the update operation (Saxpy), $y_i = \alpha * x_i + y_i$, $i = 1 : n$ is $\mathcal{C}_{upd} = c_{upd} \cdot n$, where c_{upd} is a constant.

Definition 5.22 (Complexity of the parallel element-wise product and Saxpy) The element-wise product and update operations are performed locally by each process without any exchange (communication) operations. The parallel complexity of these two operations are respectively evaluated by $\mathcal{C}_{ewp}^{para} = \frac{\mathcal{C}_{ewp}}{p} = \frac{c_{ewp} \cdot n}{p}$ and $\mathcal{C}_{upd}^{para} = \frac{\mathcal{C}_{upd}}{p} = \frac{c_{upd} \cdot n}{p}$.

In this analysis, we consider the class of parallel iterative methods that require communications only on the matrix-vector product and the dot product at most. We note the costs of these exchanges by $\epsilon_{mvp}^{para}(p)$ and $\epsilon_{dot}^{para}(p)$ respectively for the matrix-vector product and the dot product. The behavior of these amounts depends on the parallel mode: synchronous or asynchronous. For example, in a synchronous mode, the $\epsilon_{dot}^{para}(p)$ strongly relates to the behavior of the *MPI_ALLREDUCE* (or equivalent), when the *Message Passing Interface* (MPI) is considered. A detailed study will be given subsequently in order to analyze these two quantities.

Definition 5.23 (Complexity of the parallel matrix-vector product) The parallel computation costs of the matrix-vector product operation is estimated by

$$\mathcal{C}_{mvp}^{para} = \frac{\mathcal{C}_{mvp}}{p} + \epsilon_{mvp}^{para}(p) = \frac{c_{mvp} \cdot n}{p} + \epsilon_{mvp}^{para}(p). \quad (5.51)$$

Definition 5.24 (Complexity of the parallel dot product) The parallel computation costs of the dot product operation is estimated by

$$\mathcal{C}_{dot}^{para} = \frac{\mathcal{C}_{dot}}{p} + \epsilon_{dot}^{para}(p) = \frac{c_{dot} \cdot n}{p} + \epsilon_{dot}^{para}(p). \quad (5.52)$$

In the following, we consider that the constants c_{upd} , c_{ewp} , c_{dot} and c_{mvp} take into account the number of operations performed respectively for each operation.

Proposition 5.25 (Sequential complexity) *The theoretical model of the complexity of the sequential algorithm is*

$$\mathcal{C}^{seq} = c_{seq} \times n \times N, \quad (5.53)$$

where n is the number of equations, N is the total number of iterations of the sequential algorithm and $c_{seq} = c_{upd} + c_{ewp} + c_{dot} + c_{mvp} \in \mathbb{R}^+$ is a constant, and c_{upd} , c_{ewp} , c_{dot} , c_{mvp} are the coefficients of linear complexity respectively for update (saxpy), element-wise product, dot product and matrix-vector product operations.

Proof. After summing up all the contributions, we obtain the theoretical complexity of the sequential algorithm

$$\begin{aligned} \mathcal{C}^{seq} &= N(\mathcal{C}_{upd} + \mathcal{C}_{ewp} + \mathcal{C}_{dot} + \mathcal{C}_{mvp}) \\ &= N(c_{upd} \cdot n + c_{ewp} \cdot n + c_{dot} \cdot n + c_{mvp} \cdot n) \\ &= N \times n(c_{upd} + c_{ewp} + c_{dot} + c_{mvp}) = c_{seq} \times N \times n \end{aligned}$$

where $c_{seq} = c_{upd} + c_{ewp} + c_{dot} + c_{mvp}$. □

Proposition 5.26 (Parallel complexity) *The theoretical model of the complexity of the parallel algorithm is*

$$\mathcal{C}^{para} = N^{para} \times \left(\frac{c_{para} \cdot n}{p} + \left(\epsilon_{dot}^{para}(p) + \epsilon_{mvp}^{para}(p) \right) \right) \quad (5.54)$$

where n is the number of equations, N^{para} is the number iterations of the parallel algorithm and $c_{para} = c_{seq} = c_{upd} + c_{ewp} + c_{dot} + c_{mvp} \in \mathbb{R}^+$ is a constant.

Proof. After summing up all the computation costs, we obtain the theoretical complexity of the parallel algorithm

$$\begin{aligned} \mathcal{C}^{para} &= N^{para} (\mathcal{C}_{upd}^{para} + \mathcal{C}_{ewp}^{para} + \mathcal{C}_{dot}^{para} + \mathcal{C}_{mvp}^{para}) \\ &= N^{para} \left(\frac{c_{upd} \cdot n}{p} + \frac{c_{ewp} \cdot n}{p} + \left(\frac{c_{dot} \cdot n}{p} + \epsilon_{dot}^{para}(p) \right) + \left(\frac{c_{mvp} \cdot n}{p} + \epsilon_{mvp}^{para}(p) \right) \right) \\ &= N^{para} \times \left(\frac{n}{p} (c_{upd} + c_{ewp} + c_{dot} + c_{mvp}) + \left(\epsilon_{dot}^{para}(p) + \epsilon_{mvp}^{para}(p) \right) \right) \\ &= N^{para} \times \left(\frac{c_{para} \cdot n}{p} + \left(\epsilon_{dot}^{para}(p) + \epsilon_{mvp}^{para}(p) \right) \right), \end{aligned}$$

where $c_{para} = c_{seq} = c_{upd} + c_{ewp} + c_{dot} + c_{mvp}$ and N^{para} the number iterations of the parallel algorithm. \square

Remark 5.27 Considering the Proposition 5.26, for the synchronous case, we have $N^{para} = N^s = N$ and the asynchronous case will consider $N^{para} = N^a$.

Lemma 5.28 (Parallel and sequential relation complexity) *The theoretical model of the complexity of the parallel algorithm (\mathcal{C}^{para}) can be written based on the sequential complexity (\mathcal{C}^{seq})*

$$\mathcal{C}^{para} = N^{para} \left(\frac{1}{p} \frac{\mathcal{C}^{seq}}{N} + \left(\epsilon_{dot}^{para}(p) + \epsilon_{mvp}^{para}(p) \right) \right). \quad (5.55)$$

Similarly,

$$\mathcal{C}^{seq} = p \frac{N}{N^{para}} \left(\mathcal{C}^{para} - N^{para} \left(\epsilon_{dot}^{para}(p) + \epsilon_{mvp}^{para}(p) \right) \right), \quad (5.56)$$

where N is the total number iterations of sequential algorithm, N^{para} those of the parallel algorithm, i.e., $N^{para} = N^s = N$ or $N^{para} = N^a$. $\epsilon_{mvp}^{para}(p)$ and $\epsilon_{dot}^{para}(p)$ are respectively the latencies related to the parallel system for the matrix-vector product and the dot product.

Proof. Using both the sequential (5.53) and parallel (5.54) complexity and considering the assumption $c_{para} = c_{seq}$, we have

$$\begin{aligned} \mathcal{C}^{para} &= N^{para} \times \left(\frac{c_{para} \cdot n}{p} + \left(\epsilon_{dot}^{para}(p) + \epsilon_{mvp}^{para}(p) \right) \right) \\ &= N^{para} \left(\frac{1}{p} \frac{\mathcal{C}^{seq}}{N} + \left(\epsilon_{dot}^{para}(p) + \epsilon_{mvp}^{para}(p) \right) \right) \end{aligned}$$

and

$$\begin{aligned}
C^{para} &= N^{para} \times \left(\frac{c_{para} \cdot n}{p} + (\epsilon_{dot}^{para}(p) + \epsilon_{mvp}^{para}(p)) \right) \\
p \frac{C^{para}}{N^{para}} &= c_{para} \cdot n + p (\epsilon_{dot}^{para}(p) + \epsilon_{mvp}^{para}(p)) \\
p \frac{C^{para}}{N^{para}} &= \frac{C^{seq}}{N} + p (\epsilon_{dot}^{para}(p) + \epsilon_{mvp}^{para}(p)) \\
\Leftrightarrow C^{seq} &= p \frac{N}{N^{para}} (C^{para} - N^{para} (\epsilon_{dot}^{para}(p) + \epsilon_{mvp}^{para}(p)))
\end{aligned}$$

□

Knowing the sequential and parallel complexity enunciated respectively in (5.53) and (5.54), we now compute the speed-up $\mathcal{S}^{para|seq}$ that can be obtained by executing an algorithm on the system $\mathcal{P}\mathcal{S}_{system}\{p\}$, which is capable of executing p processes of execution, by

$$\mathcal{S}^{para|seq}(p) = \frac{C^{seq}}{C^{para}} \quad (5.57)$$

where C^{seq} and C^{para} are specified in units of time. We can consider C^{seq} and C^{para} as the number of arithmetic operations of the sequential and the parallel algorithm respectively. In the rest of the analysis, let us define the constant $\xi \in \mathbb{R}^+$ such that $\xi = c_{seq} = c_{para}$.

Theorem 5.29 (Speed-up of parallel algorithm upon sequential algorithm) *Let a given sequential algorithm with a complexity C^{seq} and the parallel version with a complexity C^{para} . Let us assume that the sequential and parallel algorithms converge. The theoretical speed-up $\mathcal{S}^{para|seq}$ of the parallel algorithm regarding the sequential algorithm is given by*

$$\mathcal{S}^{para|seq}(p) = \frac{N}{N^{para}} \cdot \frac{p}{1 + \frac{p}{\xi \cdot n} (\epsilon_{dot}^{para}(p) + \epsilon_{mvp}^{para}(p))} \quad (5.58)$$

where n is the number of equations, $\xi \in \mathbb{R}^+$ is the constant related to the coefficients of linear complexity of the four operations, and N and N^{para} are the number of iterations of the sequential algorithm and the parallel algorithm respectively, and $\epsilon_{dot}^{para}(p)$ and $\epsilon_{mvp}^{para}(p)$ are the latencies of the matrix-vector product and the dot product respectively, related to the parallel system.

Proof.

$$\begin{aligned}
\mathcal{S}^{para|seq}(p) &= \frac{C^{seq}}{C^{para}} \\
&= \frac{\xi \cdot n \times N}{N^{para} \times \left(\frac{\xi \cdot n}{p} + (\epsilon_{dot}^{para}(p) + \epsilon_{mvp}^{para}(p)) \right)} \\
&= \frac{N}{N^{para}} \times \frac{\xi \cdot n}{\xi \cdot n + p (\epsilon_{dot}^{para}(p) + \epsilon_{mvp}^{para}(p))} \\
&= \frac{N}{N^{para}} \cdot \frac{p}{1 + \frac{p}{\xi \cdot n} (\epsilon_{dot}^{para}(p) + \epsilon_{mvp}^{para}(p))}
\end{aligned}$$

□

Theorem 5.30 (Speed-up of asynchronous algorithm upon synchronous algorithm) *Let us assume that the asynchronous algorithm converges. The theoretical speed-up $\mathcal{S}^{a|s}$ of the parallel asynchronous algorithm regarding the synchronous algorithm is*

$$\mathcal{S}^{a|s}(p) = \frac{N^s}{N^a} \times \frac{1 + \frac{p}{\xi \cdot n} (\epsilon_{dot}^s(p) + \epsilon_{mvp}^s(p))}{1 + \frac{p}{\xi \cdot n} (\epsilon_{dot}^a(p) + \epsilon_{mvp}^a(p))} \quad (5.59)$$

where n is the number of equations, $\xi \in \mathbb{R}^+$ is the constant related to the coefficients of linear complexity of the four operations, N^s is the unique number of iterations of the parallel synchronous algorithm, N^a is the number of iterations of the parallel asynchronous algorithm (not unique, different in each process), and $(\epsilon_{mvp}^s(p), \epsilon_{dot}^s(p))$ and $(\epsilon_{mvp}^a(p), \epsilon_{dot}^a(p))$ are the latencies of the matrix-vector product and the dot product respectively, related to the parallel synchronous and asynchronous system.

Proof.

$$\begin{aligned} \mathcal{S}^{a|s}(p) = \frac{C^s}{C^a} &= \frac{N^s \times \left(\frac{\xi \cdot n}{p} + (\epsilon_{dot}^s(p) + \epsilon_{mvp}^s(p)) \right)}{N^a \times \left(\frac{\xi \cdot n}{p} + (\epsilon_{dot}^a(p) + \epsilon_{mvp}^a(p)) \right)} \\ &= \frac{N^s}{N^a} \times \frac{1 + \frac{p}{\xi \cdot n} (\epsilon_{dot}^s(p) + \epsilon_{mvp}^s(p))}{1 + \frac{p}{\xi \cdot n} (\epsilon_{dot}^a(p) + \epsilon_{mvp}^a(p))} \end{aligned}$$

□

5.3.3 Estimation of the costs of data exchange

The development of this part is based on studies presented in references [261] [262] [263] [264]. Now, we aim to give more details regarding the costs of data exchange of the matrix-vector product $\epsilon_{mvp}^{para}(p)$ and dot product $\epsilon_{dot}^{para}(p)$ due to the parallel system. In this analysis, we will study both the synchronous and the asynchronous case. We first propose to estimate the costs of data exchange operations.

Definition 5.31 Let $\mathcal{PS} = \{P_1 = 1, \dots, P_p = p\}$ be the set of the processors, a system capable of executing $p \in \mathcal{P}$ processes of execution. The *communication latency* (t_l) is the time that spends a process $P_i \in \mathcal{P}$ when it sends, receives or sends/receives a message to/from a process $P_j \in \mathcal{P}$, $P_i \neq P_j$. This time (t_l) is proportional to the size (number of items) of the message, m , which is defined as

$$t_l(m, p) = t_s + m \times t_t + \lfloor \frac{m}{p} \rfloor \times t_p, \quad (5.60)$$

where t_s is the *startup latency*, m is the size of the message, t_t is the *transmission latency*, and t_p is the *packetization latency*. This packetization latency is the time needed to create and fill packets of data for transfer over the network protocol. The startup time take into account the

fixed cost of the system call and those of recurring expenses during initialization. As proved in [261], the transmission latency (t_t) is generally inferior to the startup latency (t_s).

Remark 5.32 For asynchronous communication, the *startup latency* and the *transmission latency* are considered null, i.e., $t_s = 0$ and $t_t = 0$. The *communication latency* (t_l) is then given by

$$t_l(m, p) = \lfloor \frac{m}{p} \rfloor \times t_p. \quad (5.61)$$

The communication latency strongly depends on the properties of the program. Therefore, the parallel mode, synchronous or asynchronous, highly impacts the communication latency. When we deal with message transferring, we can no longer neglect the network throughput (τ) at which the network can distribute data. This metric is also called bandwidth and is often measured in *Mbits* per second. It is assumed that the network has a finite capacity.

Definition 5.33 The *network throughput (bandwidth)*, τ , is related to the communication latency (t_l) by the following formula: $\tau = 10^{-6} \cdot \frac{m}{t_l} = \frac{10^{-6}}{\frac{t_s}{m} + t_t}$. In the limit, as n tends to

infinity, which means that the size of the message is infinite, the maximum bandwidth tends to $\frac{10^{-6}}{t_t}$.

Definition 5.34 Let us consider a system capable of executing $p \in \mathcal{P}$ processes of execution. The costs of data exchange of the dot product $\epsilon_{dot}^{para}(p)$ is defined by

$$\epsilon_{dot}^{para}(p) = 2(p-1) \times t_l(1, p) = 2(p-1)(t_s + t_t), \quad (5.62)$$

whether we consider that the reduction is equivalent to $(p-1)$ sends and $(p-1)$ receives. For $p \geq 2$, $\frac{1}{p} = 0$, so $\lfloor \frac{1}{p} \rfloor \times t_p = 0$.

Remark 5.35 Considering Remark 5.32, we have $\epsilon_{dot}^{para}(p) = 0$ for asynchronous communication. In practice, in asynchronous mode, the dot product may not be computed. Another approach must be used for the halting procedure (cf. convergence detection).

Definition 5.36 Let us consider a system capable of executing $p \in \mathcal{P}$ processes of execution. The costs of data exchange of the matrix-vector product $\epsilon_{mvp}^{para}(p)$ is given by

$$\epsilon_{mvp}^{para}(p) = \sum_{i=1}^{card(u)} t_l(card(u_i), p) = \sum_{i=1}^k t_s + card(u_i) \times t_t + \lfloor \frac{card(u_i)}{p} \rfloor \times t_p, \quad (5.63)$$

where $u = \{u_1, \dots, u_k\}$ is the set of messages, $card(u)$ the number of exchanges performed during the computation and $card(u_i)$ the size of the i^{th} message.

5.3.4 Synchronous vs Sequential

Let N_q^s be the number of iterations of the process $q \in \mathcal{PS}_{system}\{p\}$ of a parallel synchronous algorithm. The following describes the unicity of the number of iterations in each process in a parallel synchronous algorithm.

$$\forall (r, q) \in \mathcal{PS} \times \mathcal{PS}, \quad N_r^s = N_q^s. \quad (5.64)$$

Definition 5.37 Let us define N^s as the unique number of iterations of a parallel synchronous algorithm such as

$$\forall q \in \mathcal{PS}, \quad N^s = N_q^s. \quad (5.65)$$

Remark 5.38 The parallel synchronous algorithm has the same number of iterations as the sequential algorithm, *i.e.*

$$\forall q \in \mathcal{PS}, \quad N_q^s = N^s = N. \quad (5.66)$$

Lemma 5.39 (Parallel synchronous speed-up upon sequential) *The theoretical speed-up S^s and its corresponding efficiency \mathcal{E}^s can be had by executing the parallel iterative algorithm, which solves linear systems $Ax = b$ with $A \in \mathbb{R}^{n \times n}$, on a system capable of executing $p \in \mathcal{P}$ processes of execution synchronously are*

$$S^s(p) = \frac{p}{1 + \frac{p}{\xi \cdot n} (\epsilon_{dot}^s(p) + \epsilon_{mvp}^s(p))} \text{ and } \mathcal{E}^s(p) = \frac{1}{1 + \frac{p}{\xi \cdot n} (\epsilon_{dot}^s(p) + \epsilon_{mvp}^s(p))}, \quad (5.67)$$

where n is the number of equations, p the number of processes, $\xi \in \mathbb{R}^+$ the constant related to the coefficients of linear complexity of the four operations, and $\epsilon_{mvp}^s(p)$, $\epsilon_{dot}^s(p)$ the latencies of the matrix-vector product and the dot product respectively related to a parallel synchronous system.

Proof. This is a direct consequence of Theorem 5.29 and considering the Remark 5.38. \square

5.3.5 Asynchronous vs Sequential

The asynchronous version of the studied iterative algorithm such as Jacobi's is obtained by simply removing from the synchronous version all synchronization points, and ensuring that the messages can be received at any point in the iteration. For example, we remove all reduction functions such as `MPI_ALLREDUCE`, which requires a synchronization of the whole set of processes. In case of asynchronous algorithms, each process has a different number of iterations. Indeed, from one execution to the other, the order of messages will change and the number of iterations to reach the convergence will also change, *i.e.*, $N_q^a(i) \neq N_q^a(j)$ in general for a process $q \in \mathcal{PS}$ and i, j two different executions. The asynchronous model is non-deterministic. In general, asynchronous algorithms require a greater number of iterations before the convergence than in the synchronous case. However, in some cases, the asynchronous version requires fewer iterations to converge than its synchronous counterpart, as proved in [265]. J.M Bull *et al.* [265] show that this behavior can be explained in terms of the presence or absence of oscillations in the sequence of error vectors in the synchronous version, and that removing the synchronization point can damp the oscillations.

Definition 5.40 Let N_q^a be the number of iterations of the process $q \in \mathcal{PS}$ of a parallel asynchronous algorithm. Let us note first that this assumption implies the following hypothesis:

$$\forall (r, q) \in \mathcal{PS} \times \mathcal{PS}, \quad r \neq q, N_r^a \neq N_q^a \quad (5.68)$$

and

$$\forall q \in \mathcal{PS}, \quad N_q^a(i^{th} \text{ run}) \neq N_q^a(j^{th} \text{ run}) \text{ for } i \neq j. \quad (5.69)$$

Definition 5.41 To analyse the asynchronous speed-up, we define the total number of iterations (unique) of a parallel asynchronous algorithm such that it is proportionally equal to the total number of iterations of a parallel synchronous algorithm, which is expressed by $N^a = \delta \cdot N^s$, where $\delta \in \mathbb{R}^{+*}$ is the multiplier coefficient.

We can choose for instance, $\delta \in \mathbb{R}^{+*}$ such that the total number of iterations of a parallel asynchronous algorithm is proportional to the average number of iterations of all the processes, which can be formulated as $\delta = \frac{1}{N^s} \times \frac{1}{p} \sum_{q=1}^p N_q^a$, $q \in \mathcal{PS}$.

Lemma 5.42 (Parallel asynchronous speed-up upon sequential) *The theoretical speed-up S^a and its corresponding efficiency \mathcal{E}^a can be had by executing the parallel iterative algorithm, which solves linear systems $Ax = b$ with $A \in \mathbb{R}^{n \times n}$, on a system capable of executing $p \in \mathcal{PS}$ processes of execution asynchronously are*

$$S^a(p) = \frac{1}{\delta} \times \frac{p}{1 + \frac{p}{\xi \cdot n} (\epsilon_{dot}^a(p) + \epsilon_{mvp}^a(p))} \text{ and } \mathcal{E}^a(p) = \frac{1}{\delta} \times \frac{1}{1 + \frac{p}{\xi \cdot n} (\epsilon_{dot}^a(p) + \epsilon_{mvp}^a(p))}. \quad (5.70)$$

where n is the number of equations, p the number of processes, $\xi \in \mathbb{R}^+$ the constant related to the coefficients of linear complexity of the four operations, and $\epsilon_{mvp}^a(p)$, $\epsilon_{dot}^a(p)$ the latencies of the matrix-vector product and the dot product respectively related to a parallel asynchronous system.

Proof. This is a direct consequence of Theorem 5.29 and considering the Definition 5.41. \square

5.3.6 Asynchronous vs Synchronous

Lemma 5.43 (Parallel asynchronous speed-up upon synchronous) *The theoretical speed-up $S^{a|s}$ and its corresponding efficiency $\mathcal{E}^{a|s}$ can be had by executing the parallel iterative algorithm, which solves linear systems $Ax = b$ with $A \in \mathbb{R}^{n \times n}$, on a system capable of executing $p \in \mathcal{PS}$ processes of execution asynchronously is*

$$S^{a|s}(p) = \frac{1}{\delta} \cdot \frac{1 + \frac{p}{\xi \cdot n} (\epsilon_{dot}^s(p) + \epsilon_{mvp}^s(p))}{1 + \frac{p}{\xi \cdot n} (\epsilon_{dot}^a(p) + \epsilon_{mvp}^a(p))} \text{ and } \mathcal{E}^{a|s}(p) = \frac{1}{\delta \cdot p} \cdot \frac{1 + \frac{p}{\xi \cdot n} (\epsilon_{dot}^s(p) + \epsilon_{mvp}^s(p))}{1 + \frac{p}{\xi \cdot n} (\epsilon_{dot}^a(p) + \epsilon_{mvp}^a(p))} \quad (5.71)$$

where $\delta \in \mathbb{R}^{+*}$, n is the number of equations, p the number of processes, $\xi \in \mathbb{R}^+$ the constant related to the coefficients of linear complexity of the four operations, $(\epsilon_{mvp}^s(p), \epsilon_{dot}^s(p))$ and $(\epsilon_{mvp}^a(p), \epsilon_{dot}^a(p))$ the latencies of the matrix-vector product and the dot product respectively related to a parallel synchronous and asynchronous system.

Proof. This is a direct consequence of Theorem 5.30 and considering the Definition 5.41. \square

5.3.7 Application to Jacobi method

Let $A \in \mathbb{K}^{n \times n}$ be a square matrix that satisfies the Jacobi condition of convergence enunciated in Theorem 4.60 for synchronous algorithms and Theorem 4.61 for asynchronous algorithms. For the sake of clarity and simplicity, we use in this analysis the vector version of the Jacobi method (see Proposition 4.64, Page 139), which corresponds to the Algorithm 4.5.

The analysis is equivalent with those considering the Jacobi element updates version described in Definition 4.62 (Algorithm 4.4).

Proposition 5.44 *As we can observe in (4.35) (Algorithm 4.5), each iteration of the Jacobi method performs one matrix-vector product ($n_{mvp} = 1$), one element-wise product ($n_{ewp} = 1$) and two vector updates ($n_{upd} = 2$). Finally, a dot product ($n_{dot} = 1$) is performed to compute the residual of $D^{-1}(b - Ax^{(k)})$. In fact, $\varepsilon^{(k)} = \|x^{(k+1)} - x^{(k)}\| = \|D^{-1}(b - Ax^{(k)})\|$.*

Remark 5.45 The Jacobi method satisfies the assumption of the speed-up analysis given previously, *i.e.*, it is fully parallelizable and has four operations (Proposition 5.44).

In our implementation, the input and output of a parallel algorithms are local to each process. Indeed, the distribution of data is accomplished as a preprocessing step, independently from the solver code, as described in Section 4.3, Page 112.

Multi-splitting band-row method Each processor has to perform $(p - 1)$ sends and $(p - 1)$ receives of m items where m is the size of blocks. Therefore,

$$\epsilon_{mvp}^{para}(p) = \sum_{i=1}^{p-1} 2 * t_l(card(b_i), p) = \sum_{i=1}^{p-1} 2(t_s + card(b_i) \times t_t + \lfloor \frac{card(b_i)}{p} \rfloor \times t_p),$$

where $b = \{b_1, \dots, b_{p-1}\}$ is the set of blocks with $card(b) = p - 1$ and $card(b_i)$ the size of the i^{th} block. Considering Remark 5.32, we have

$$\epsilon_{mvp}^a(p) = \sum_{i=1}^{p-1} 2 \lfloor \frac{card(b_i)}{p} \rfloor \times t_p.$$

Considering Definition 5.34, in synchronous algorithms, the dot product latency is always calculated as follows

$$\epsilon_{dot}^s(p) = 2(p - 1)(t_s + t_t). \quad (5.72)$$

Sub-structuring method The sub-structuring Jacobi algorithm (Algorithm 5.3 and Algorithm 5.4) considered here only performs the four operations locally in each process without any exchange of data. Considering the Algorithm 5.3 and Algorithm 5.4, the dot product and the matrix-vector product are computed locally without any exchange. Moreover, at each iteration, the input data of the matrix-vector product needs to be updated by assembling the local contributions from all neighboring sub-domains on the interface. This scheme requires the exchange of data between processes dealing with sub-domains sharing a common interface. For the sake of simplicity on the notation, we consider that this scheme corresponds to the contribution $\epsilon_{mvp}^{para}(p)$. Without losing generality and for the sake of simplicity, we assume that the number of interfaces (neighboring, which share a common interface) is the same in each sub-domain. Let I be the number of interfaces. So, each sub-domain has to perform I sends and I receives, *i.e.*, $2I$ communications. In addition, the size of a sending and receiving process at a given interface is equal, we can conclude that

$$\epsilon_{mvp}^{para}(p) = \sum_{i=1}^{2I} t_l(card(u_i), p) = \sum_{i=1}^{2I} t_s + card(u_i) \times t_t + \lfloor \frac{card(u_i)}{p} \rfloor \times t_p, \quad (5.73)$$

where $u = \{u_1, \dots, u_k\}$ is the set of interfaces, $card(u)$ is the number of exchanges performed during the computation, and $card(u_i)$ the number of nodes of the corresponding interface.

5.4 Numerical results

This section reports the numerical experiments concerning the studies presented in this chapter, *i.e.*, the sub-structuring methods with synchronous and asynchronous iterations. We will also analyze their behaviors using CUDA accelerators. In this chapter, although we mainly focus on parallel synchronous and asynchronous stationary iterative methods, we will also give some results of the *Conjugate Gradient*, a non-stationary iterative Krylov method. A numerical analysis of the theoretical speed-up of the Jacobi method will also be presented. To be in line with previous chapters, we will give an analysis using the set of matrices from the University of Florida repository (see Table 3.11, Page 68). Nevertheless, the matrices given in Table 3.11 do not meet the conditions of convergence of the Jacobi method (see Theorem 4.60, Page 136), so we will introduce a novel set of data that satisfies the convergence conditions. The matrices have been chosen so that Jacobi's convergence condition for parallel asynchronous algorithms are satisfied (see Theorem 4.61, Page 136). The matrices used in this section are stored in Compressed Sparse Row (CSR) format.

5.4.1 Test cases

To illustrate the results of the chapter, the following equation in an heterogeneous 3D media is first considered:

$$(-\nabla \cdot (\mu \nabla)) u(x, y, z) = f(x, y, z), \quad (x, y, z) \in \Omega \quad (5.74)$$

in the domain $\Omega = [a_1, b_1] \times [a_2, b_2] \times [a_3, b_3]$, where $a_1, b_1, a_2, b_2, a_3, b_3 \in \mathbb{R}$. The equation (5.74) is called *non-linear Poisson equation*. The problems are obtained by the finite element method discretization on diverse meshes. In this study, the global domain is split in n layers, where $n = 1$ (*Poisson equation*) and $n = 4$. The coefficient μ is assumed to be constant per layer. The design of the finite element method consists in discretizing the equation (5.74) to build an elementary matrix in each layer by considering the associated constant μ and then assemble the global matrix. The method consists of hexahedron finite elements. The layers can be non-uniform. Note that when μ is considered constant, the equation (5.74) can be rewritten as

$$-\mu \nabla \cdot \nabla u = -\mu \Delta u = f, \quad (5.75)$$

because when μ is constant, we have $-\nabla \mu = 0$, so $-\nabla \mu \cdot \nabla u = 0$. The meshes used here are structured. Table 5.1 shows representative ranges of hydraulic conductivity values ($m \cdot s^{-1}$), μ , for common rocks and soils (adapted from P.A Domenico and F.W Schwartz 1990 [266], T.T. Eaton *et al.* [267], B. Suski [268]).

Global domain in 4 layers upon μ The first set of data consists of matrices obtained by the finite element discretization of the equation (5.74), using 4 layers with the following value of μ : 10, 100, 10, 1000 (see Table 5.1).

Material	hydraulic conductivity ($m.s^{-1}$), $K = \mu$		
	<i>bound inf</i>	...	<i>bound sup</i>
coarse gravel	10^3	...	10^4
sand, gravel	10^0	...	10^3
fine sand, silt	10^{-4}	...	10^0
clay, shale	10^{-9}	...	10^{-6}
limestone	10^0	...	10^2
sandstone	10^{-5}	...	10^1
altered chalk	10^0	...	10^2
unaltered chalk	10^{-9}	...	10^1
granite, gneiss	10^{-8}	...	10^{-4}

TABLE 5.1: Hydraulic conductivity ($m.s^{-1}$), μ , depending on the environment of the medium (material)

The domain Ω has the shape of a parallelepiped extending from 0 to 250,000 in the x -direction and y -direction, and from $-15,000$ m to 0 in the z -direction. The domain $\Omega = [0; 250,000] \times [0; 250,000] \times [-15,000; 0]$ is projected into $[0; 1] \times [0; 1] \times [0; 0.06]$. The projection conserves the geometry, the discretization topology (h remains uniform), the number of nodes and elements, and the layers and partitioning.

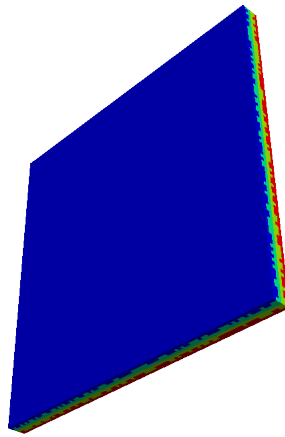


FIGURE 5.8: Example of the mesh with 4 layers

Figure 5.8 illustrates an instance of mesh with 4 layers.

Homogeneous Dirichlet boundary conditions are defined on the vertical faces (xz -plane and yz -plane), *i.e.*

$$\begin{cases} u(x, y, z) = 0, & \forall (x, y, z) \in \{0; 1\} \times [0; 1] \times [0, 0.06], \\ u(x, y, z) = 0, & \forall (x, y, z) \in [0; 1] \times \{0; 1\} \times [0, 0.06]. \end{cases} \quad (5.76)$$

The right-hand side is defined such that $f(x, y, z) = \cos(250 \times z)$.

Table 5.2 collects the main features of the two levels of meshes used for the finite element discretization of the equation (5.74) with 4 layers.

Table 5.3 reports the statistics of the layers of the level #1 and level #2 meshes respectively.

3D Laplace equation The second set of data consists of purely academic test cases. The matrices are obtained by the finite element discretization of the equation (5.74) with $\mu = 1$. Under this condition, the equation (5.74) corresponds to the classical Laplace equation (Poisson equation), *i.e.*

$$-\Delta u = f. \quad (5.77)$$

	original mesh		projected mesh	
	LEVEL #0	LEVEL #1	LEVEL #0	LEVEL #1
instance	Figure 5.8		Figure 5.8	
x -range	[0 : 2500 : 250,000]	[0 : 1250 : 250,000]	[0 : 0.01 : 1]	[0 : 0.005 : 1]
y -range	[0 : 2500 : 250,000]	[0 : 1250 : 250,000]	[0 : 0.01 : 1]	[0 : 0.005 : 1]
z -range	[-15,000 : 2500 : 0]	[-15,000 : 1250 : 0]	[0 : 0.01 : 0.06]	[0 : 0.005 : 0.06]
Number of nodes	71,407	525,213	71,407	525,213
Number of elements	60,000	480,000	60,000	480,000
Size of element	≈ 4330.12702	≈ 2165.06351	≈ 0.0173	≈ 0.0087
Number of layers	4	4	4	4
Total interface size	96,234	391,510	96,234	391,510

TABLE 5.2: Statistics of the original and projected meshes with 4 layers

	level #0 mesh				level #2 mesh			
	layer #1	layer #2	layer #3	layer #4	layer #1	layer #2	layer #3	layer #4
Number of nodes	29,921	29,846	29,806	29,951	29,921	29,846	29,806	29,951
Number of elements	15,066	14,934	14,918	15,082	15,066	14,934	14,918	15,082
Number of interface nodes	18,963	29,164	29,154	18,953	18,963	29,164	29,154	18,953

TABLE 5.3: Statistics of the level #0 and #1 of the meshes with 4 layers: *muluf_hexas100x100x6_0* (level #1) and *muluf_hexas100x100x6_1* (level #2)

Note that the Helmholtz equation does not converge with a classical splitting method such as Jacobi's. The domain has the shape of a cube extending from 0 to 1 in each direction (x -, y -, z -), *i.e.*, $\Omega = [0; 1] \times [0; 1] \times [0; 1]$. Homogeneous Dirichlet boundary conditions are defined on the whole boundary, *i.e.*

$$\begin{cases} u(x, y, z) = 0, & \forall (x, y, z) \in \{0; 1\} \times [0; 1] \times [0; 1], \\ u(x, y, z) = 0, & \forall (x, y, z) \in [0; 1] \times \{0; 1\} \times [0; 1], \\ u(x, y, z) = 0, & \forall (x, y, z) \in [0; 1] \times [0; 1] \times \{0; 1\}. \end{cases} \quad (5.78)$$

The right-hand side is defined such that $f(x, y, z) = \cos(x + y)$.

	CUBE-35937	CUBE-274624
instance	Figure 5.9(a)	Figure 5.9(b)
x -range	[0 : 0.03125 : 1]	[0 : 0.015625 : 1]
y -range	[0 : 0.03125 : 1]	[0 : 0.015625 : 1]
z -range	[0 : 0.03125 : 1]	[0 : 0.015625 : 1]
Number of nodes	35,937	274,624

TABLE 5.4: Statistics of the academic cube meshes: *luf_cube-35937* and *luf_cube-274624*

The set consists of two matrices of size 35,937 and 274,624 where the properties of the associated meshes are collected in Table 5.4. Figure 5.9 illustrates an instance of a finite element mesh of the CUBE-35937 (see Figure 5.9(a), Page 177) and the CUBE-274624 (see Figure 5.9(b), Page 177).

3D gravitational potential equation The third set of data is obtained from the finite element discretization of the gravitational potential equation. The gravitational potential of a density anomaly distribution is given as a particular case of the equation (5.77) with a right-hand side defined so that $f(x, y, z) = 4\pi G\delta\rho(x, y, z)$, where $\delta\rho$ is the density anomaly and G the gravitational constant. Let us remark that the domain Ω corresponds to the area of the

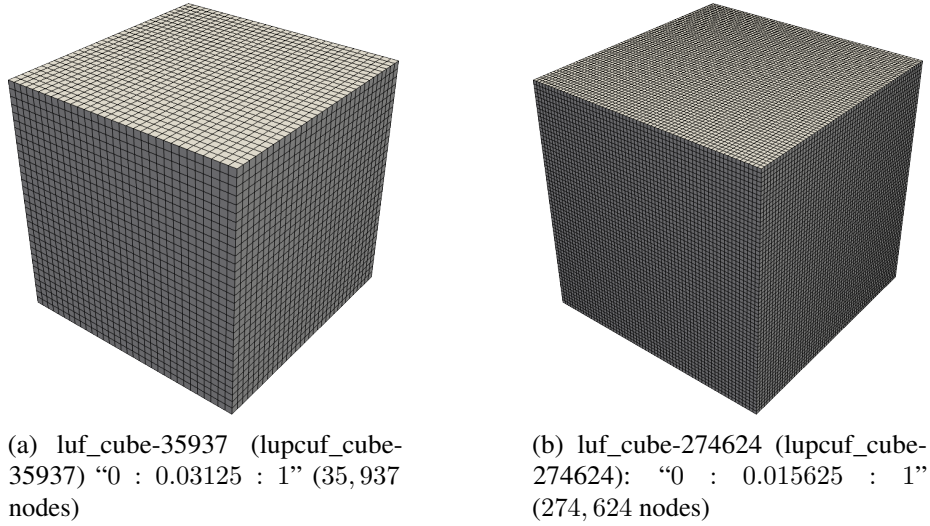


FIGURE 5.9: Finite element mesh examples of the cube

Chicxulub impact crater (see Section 3.5.3, Page 69). However, in the new discretization of the following mathematical modeling,

$$-\Delta u = 4\pi G\delta\rho, \quad (5.79)$$

we consider homogeneous Dirichlet boundary conditions defined on the vertical faces (xz -plane and yz - plane), *i.e.*

$$\begin{cases} u(x, y, z) = 0, & \forall (x, y, z) \in \{0; 1\} \times [0; 1] \times [0, 0.06], \\ u(x, y, z) = 0, & \forall (x, y, z) \in [0; 1] \times \{0; 1\} \times [0, 0.06] \end{cases} \quad (5.80)$$

The density anomalies $\delta\rho$ are those of the Chicxulub impact crater.

The properties and the mesh are closely similar to those presented in Table 5.2 and Figure 5.8, except that there is only one layer (whole domain). The matrices are named *gravi_hexas100x100x6_0* (level #1) and *gravi_hexas100x100x6_1* (level #2).

3D Heat equation Let us first get some definitions:

- $u(x, y, z, t)$: *temperature* at any point (x, y, z) and any time t , $[K]$
- $c(x, y, z)$: *specific heat* $[j/kg K]$ (assumed constant in time),
- $\rho(x, y, z)$: *mass density* $[kg/m^3]$ (assumed constant in time),
- $Q(x, y, z, t)$: *heat energy* generated per unit volume per unit time.

The specific heat, $c(x, y, z)$, of a material is the amount of heat energy that it takes to raise one unit of mass of the material by one unit of temperature, *i.e.*, the heat necessary to raise the temperature of a $1kg$ -mass by 1 Kelvin. The mass density, $\rho(x, y, z)$, is the mass per unit volume of the material. In our analysis, we assume that the specific heat, $c(x, y, z)$, the mass density, $\rho(x, y, z)$, and the heat energy, $Q(x, y, z, t)$, are considered constant, *i.e.*

$$c(x, y, z) = c, \quad \rho(x, y, z) = \rho \quad \text{and} \quad Q(x, y, z, t) = Q.$$

The *heat equation* in an isotropic and homogeneous media is considered:

$$\begin{cases} \frac{\partial u}{\partial t}(x, y, z, t) = k\Delta u(x, y, z, t) + \frac{Q}{c\rho}, & (x, y, z) \in \Omega \quad \text{and} \quad t \in \mathbb{R}^+, \\ u(x, y, z, 0) = f(x, y, z), & \forall (x, y, z), \\ u(x, y, z, t) = 0, & \forall (x, y, z) \quad \text{on} \quad \partial\Omega, \end{cases} \quad (5.81)$$

in the spatial domain $\Omega = [a_1, b_1] \times [a_2, b_2] \times [a_3, b_3]$ ($a_1, b_1, a_2, b_2, a_3, b_3 \in \mathbb{R}$) where $\partial\Omega$ denotes the closed surface of Ω , where $\Delta = \nabla^2$ is the Laplacian operator and $k = \frac{K_0}{c\rho}$ with K_0 is the thermal conductivity (considered constant). Here, there are no external sources, thus $Q = 0$, and then the equation (5.82) can be written as follows

$$\begin{cases} \frac{\partial u}{\partial t}(x, y, z, t) = k\Delta u(x, y, z, t), & (x, y, z) \in \Omega \quad \text{and} \quad t \in \mathbb{R}^+, \\ u(x, y, z, 0) = f(x, y, z), & \forall (x, y, z), \\ u(x, y, z, t) = 0, & \forall (x, y, z) \quad \text{on} \quad \partial\Omega, \end{cases} \quad (5.82)$$

where $k = \frac{K_0}{c\rho}$ ($[m^2/s]$) is the *thermal diffusivity*. The exact solution of the heat equation can be obtained using the technique of separation of variables. First, we assume that the solution will take the form,

$$u(x, y, z, t) = \psi(x, y, z)\phi(t), \quad (5.83)$$

and we substitute the equation (5.83) with the equation (5.82), by taking into account also the boundary conditions. We separate the equation (5.83) to get a function which only depends on t and another that only depends on the spatial domain (x, y, z) , and then introduce a separation constant λ . After that, the initial problem (5.82) leads to solve the following equations:

$$\frac{\partial \phi}{\partial t} = -k\lambda\phi \quad \text{and} \quad \begin{cases} \Delta\psi + \lambda\psi = 0, & \in \Omega, \\ \psi(x, y, z, t) = 0, & \forall (x, y, z) \quad \text{on} \quad \partial\Omega. \end{cases} \quad (5.84)$$

In this analysis we only solve the spatial problem,

$$\begin{cases} \Delta\psi + \lambda\psi = 0, & \in \Omega, \\ \psi(x, y, z, t) = 0, & \forall (x, y, z) \quad \text{on} \quad \partial\Omega, \end{cases} \quad (5.85)$$

with λ considered positive. The set of matrices is obtained from the finite element discretization in space of the problem (5.85). The time-dependent equation can really be solved at any time. The time equation can be solved using the finite difference discretization. For example, we can use an explicit Euler scheme, which gives

$$\frac{\phi_{n+1} - \phi_n}{\delta t} = -k\lambda\phi_n \quad (5.86)$$

where δt is the time step of the temporal discretization. Finally, at each time step n , we have to perform

$$\phi_{n+1} = \phi_n (1 - k\lambda\delta t). \quad (5.87)$$

In this analysis, for the sake of simplicity, the matrices used are obtained from the finite element discretization of the following space problem:

$$\Delta\psi + \eta\psi = f, \quad (5.88)$$

where η depends on $k = \frac{K_0}{c\rho}$ and the time discretization δt . The parameter η is fixed to 0.1 for cube matrices, and to 1 for the church matrix. The chosen η ensures the convergence of the Jacobi method. The right-hand side is defined so that $f(x, y, z) = \cos(x + y)$. The properties of the meshes used in the discretization are given in Table 5.5.

	CUBE-35937	CUBE-274624	CHURCH-484507
instance	Figure 5.9(a)	Figure 5.9(b)	Figure 5.12
x -range	[0 : 0.03125 : 1]	[0 : 0.015625 : 1]	[-24.4 : \approx 0.5 : 26.6]
y -range	[0 : 0.03125 : 1]	[0 : 0.015625 : 1]	[-59.9 : \approx 0.5 : 37.9]
z -range	[0 : 0.03125 : 1]	[0 : 0.015625 : 1]	[0.161 : \approx 0.5 : 39.2]
Number of nodes	35,937	274,624	484,507
Parameter η of (5.88)	1	1	0.1

TABLE 5.5: Statistics of the meshes used in the discretization of $\Delta\psi + \eta\psi = f$: *lupcuf_cube-35937*, *lupcuf_cube-274624* and *church-484507*

Figure 5.12 illustrates an instance of the mesh that models the Royaumont church (484,507 nodes). Computer-Aided Design (CAD) models of the Royaumont church are given in Figure 5.10 (exterior view) and Figure 5.11 (interior view). F. Magoulès, R. Cerise and P.



FIGURE 5.10: Computer-Aided Design (CAD) model of the Royaumont church (exterior view of the architecture)



(a) Interior view 1



(b) Interior view 2

FIGURE 5.11: CAD model of the Royaumont church (interior view of the architecture)

Callet [269] have designed an efficient methodology for sound holography within the church

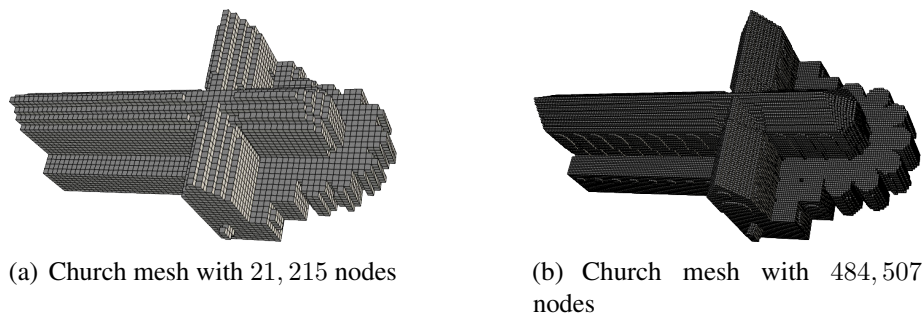


FIGURE 5.12: Finite element mesh examples of the church

of the Royaumont Abbey. The Royaumont Abbey is situated about 35 kilometers north of Paris. This is a royal church. The construction was demanded by King Saint Louis and his mother, Blanche de Castille, in 1228 and finished in 1235. During the sovereignty of Saint Louis, the Royaumont Abbey was one of the most important Cistercian places in Europe, inhabited by a maximum of 140 monks. For more detail, the readers are referred to the paper of F. Magoulès *et al.* [269]. F. Magoulès *et al.* [269] and G. Gbikpi-Benissan *et al.* [270] have studied the medieval glass rendering in terms of acoustics (Helmholtz equation) within the church of the Royaumont Abbey using the ray-tracing domain decomposition method (DDM). Given that the Helmholtz equation does not satisfy the convergence condition of Jacobi, we give an evaluation that helps analyze the heat within the church of the Royaumont Abbey.

Summary of test cases Table 5.6 collects the set of matrices obtained by the finite element $P1$ discretization of the test cases presented previously. The main properties of each matrix are also given: h the size of the matrix, nnz the number of non-zero values, $density$ the density that corresponds to the number of non-zero values divided by the total number of matrix coefficients, nnz/h the mean row density, max_row the maximal row density, $\sigma(nnz/n)$ or nnz/h_stddev the standard deviation of the mean row density (nnz/h) and $bandwidth$ the upper bandwidth, which is equal to the lower bandwidth in case of a symmetric matrix. The first and second pictures represent respectively the pattern of non-zero values and an histogram of the distribution of non-zero values per row for each matrix. The matrices are ranged in increasing number of non-zero values (nnz), from top to bottom, left to right.

5.4.2 Parallel test cases

The data of the parallel test cases consist of the splitting of the matrices presented in Table 5.6 according to the methods of data partitioning for distributed computing (see Section 4.3, Page 112).

Statistics of parallel test cases In this paragraph, we give and analyze some statistics concerning the matrix partitioning using the proposed sub-structuring splitting described in Section 4.3.5, Page 120. In this analysis, for the sake of simplicity, we give statistics for the matrices “*church-484507*” and “*lupcuf_cube-274625*” only. The other matrices have the same behavior concerning the partitioning. Figure 5.13, Figure 5.15 and Figure 5.14 describe respectively the number of nodes (# of nodes) in each sub-domain, the number of neighboring sub-domains and the number of interface nodes for the 32 and 64-partitioning of “*church-484507*”.

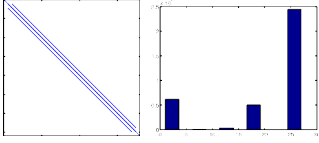
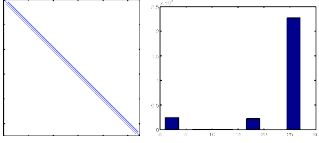
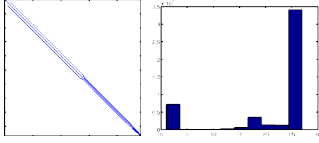
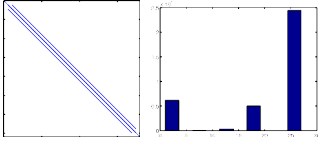
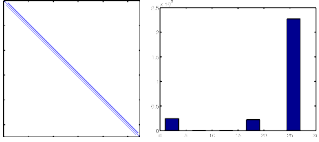
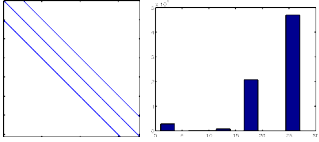
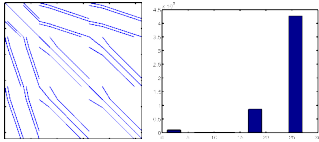
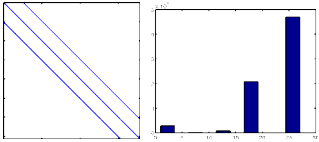
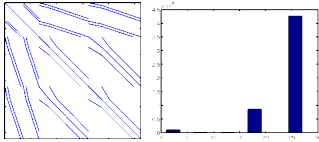
<p>“luf_cube-35937”</p> <p>$n = 35,937$ $nnz = 759,667$ $density = 0.059$ $nnz/n = 21.139$ $max_row = 27$ $\sigma(nnz/n) = 9.741$ $bandwidth = 1,123$</p>  <p>Structured FEM problem, 3D Laplace equation (5.77), cube $[0 : 0.03125 : 1] \times [0 : 0.03125 : 1] \times [0 : 0.03125 : 1]$.</p>	<p>“luf_cube-274625”</p> <p>$n = 274,625$ $nnz = 6,563,167$ $density = 0.009$ $nnz/n = 23.899$ $max_row = 27$ $\sigma(nnz/n) = 7.621$ $bandwidth = 4,291$</p>  <p>Structured FEM problem, 3D Laplace equation (5.77), cube $[0 : 0.015625 : 1] \times [0 : 0.015625 : 1] \times [0 : 0.015625 : 1]$.</p>	<p>“church-484507”</p> <p>$n = 484,507$ $nnz = 10,641,113$ $density = 0.005$ $nnz/n = 21.963$ $max_row = 27$ $\sigma(nnz/n) = 9.262$ $bandwidth = 11,341$</p>  <p>Structured FEM problem, 3D heat equation (5.88), church Figure 5.12.</p>
<p>“lupcuf_cube-35937”</p> <p>$n = 35,937$ $nnz = 759,717$ $density = 0.059$ $nnz/n = 21.140$ $max_row = 27$ $\sigma(nnz/n) = 9.742$ $bandwidth = 1,123$</p>  <p>Structured FEM problem, 3D heat equation (5.88), cube $[0 : 0.03125 : 1] \times [0 : 0.03125 : 1] \times [0 : 0.03125 : 1]$.</p>	<p>“lupcuf_cube-274625”</p> <p>$n = 274,625$ $nnz = 6,563,781$ $density = 0.009$ $nnz/n = 23.901$ $max_row = 27$ $\sigma(nnz/n) = 7.622$ $bandwidth = 4291$</p>  <p>Structured FEM problem, 3D heat equation (5.88), cube $[0 : 0.015625 : 1] \times [0 : 0.015625 : 1] \times [0 : 0.015625 : 1]$.</p>	<p>“gravi_hexas100x100x6_0”</p> <p>$n = 71,407$ $nnz = 1,656,131$ $density = 0.032$ $nnz/n = 23.193$ $max_row = 27$ $\sigma(nnz/n) = 6.174$ $bandwidth = 10,303$</p>  <p>Structured FEM problem, 3D gravitational potential equation (5.79), parallelepiped $[0 : 0.01 : 1] \times [0 : 0.01 : 1] \times [0 : 0.01 : 0.06]$.</p>
<p>“gravi_hexas100x100x6_1”</p> <p>$n = 525,213$ $nnz = 13,107,979$ $density = 0.005$ $nnz/n = 24.957$ $max_row = 27$ $\sigma(nnz/n) = 4.820$ $bandwidth = 463,803$</p>  <p>Structured FEM problem, 3D gravitational potential equation (5.79), parallelepiped $[0 : 0.005 : 1] \times [0 : 0.005 : 1] \times [0 : 0.005 : 0.06]$.</p>	<p>“muluf_hexas100x100x6_0”</p> <p>$n = 71407$ $nnz = 1,656,270$ $density = 0.032$ $nnz/n = 23.195$ $max_row = 27$ $\sigma(nnz/n) = 6.174$ $bandwidth = 10,303$</p>  <p>Structured FEM problem, non-linear Poisson equation in an heterogeneous 3D media (5.74), parallelepiped $[0 : 0.01 : 1] \times [0 : 0.01 : 1] \times [0 : 0.01 : 0.06]$ with 4 layers Figure 5.8.</p>	<p>“muluf_hexas100x100x6_1”</p> <p>$n = 525,213$ $nnz = 13,109,255$ $density = 0.005$ $nnz/n = 24.960$ $max_row = 27$ $\sigma(nnz/n) = 4.820$ $bandwidth = 463,803$</p>  <p>Structured FEM problem, non-linear Poisson equation in an heterogeneous 3D media (5.74), parallelepiped $[0 : 0.01 : 1] \times [0 : 0.01 : 1] \times [0 : 0.01 : 0.06]$ with 4 layers Figure 5.8.</p>

TABLE 5.6: Sketches of matrices obtained by the finite element discretization of the presented test cases

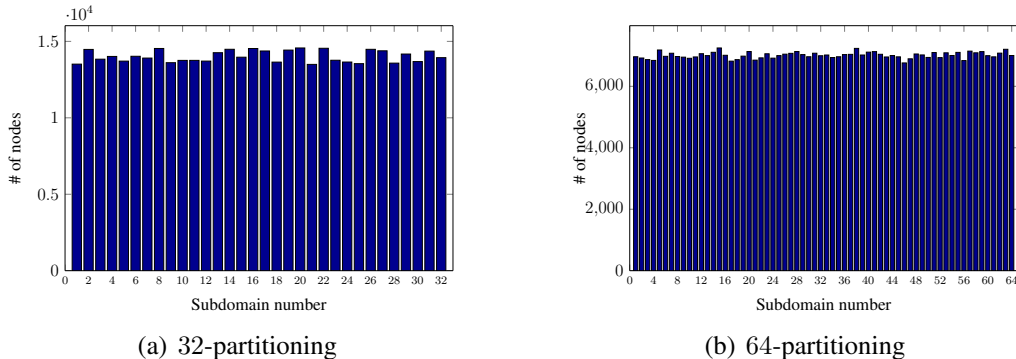


FIGURE 5.13: Number of nodes (# of nodes) in each sub-domain for 32 and 64-partitioning of the matrix “church-484507”

As we can see in Figure 5.13 and Figure 5.16, the partitioning is almost load-balanced. Figure 5.16, Figure 5.18 and Figure 5.17 give respectively the number of nodes (# of nodes)

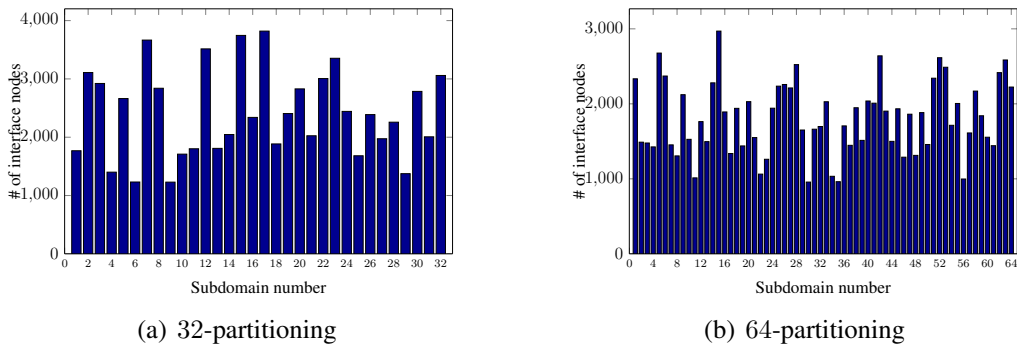


FIGURE 5.14: Number of interface nodes (# of interface nodes) in each sub-domain for 32 and 64-partitioning of the matrix “church-484507”

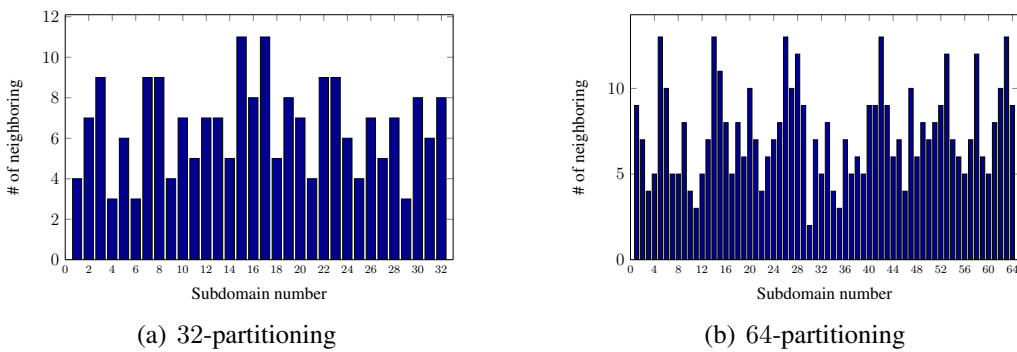


FIGURE 5.15: Number of neighboring (# of neighboring) in each sub-domain for 32 and 64-partitioning of the matrix “church-484507”

in each sub-domain, the number of neighboring sub-domains and the number of interface nodes for the 32 and 64-partitioning of “*lupcuf_cube-274625*”. The partitioning gives a good

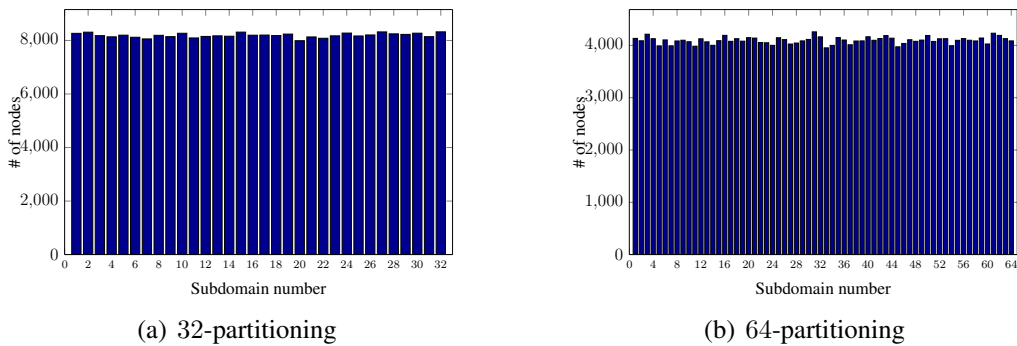


FIGURE 5.16: Number of nodes (# of nodes) in each sub-domain for the 32 and 64-partitioning of the matrix “*lupcuf_cube-274625*”

distribution of the sparse matrix on the parallel processors. The number of neighboring given in Figure 5.15 and Figure 5.18 estimates the number of communications (send/receive). Moreover, the number of interface nodes presented in Figure 5.14 and Figure 5.17 estimates the size of send/receive messages.

Figure 5.19 and Figure 5.20 present respectively the projection of the 32- and 64-partitioning of the “church-484507” and ‘*lupcuf_cube-274624*’ matrices into the associated meshes given in Figure 5.12 and Figure 5.9(b). As we can see in these figures, the partitioning of a matrix

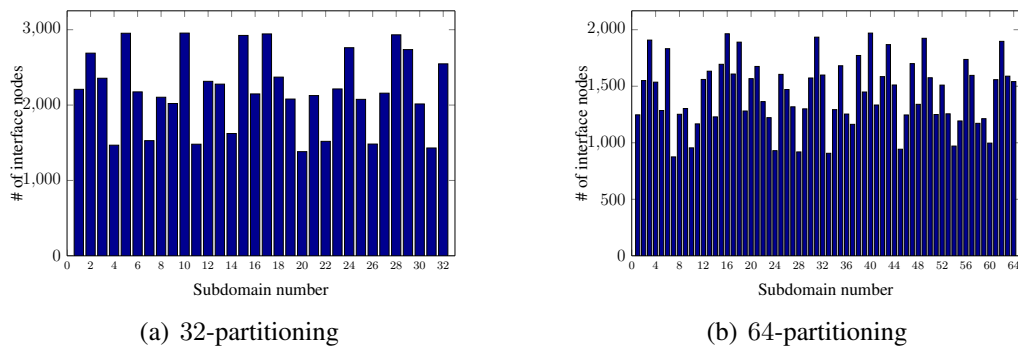


FIGURE 5.17: Number of interface nodes (# of interface nodes) in each sub-domain for the 32 and 64-partitioning of the matrix “lupcuf_cube-274625”

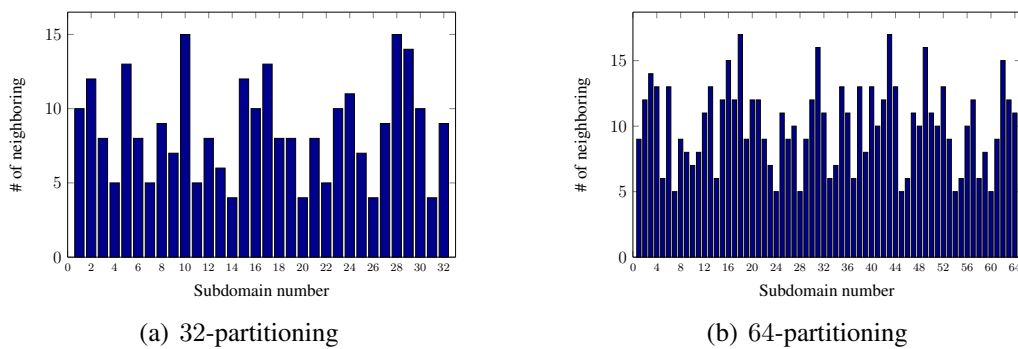


FIGURE 5.18: Number of neighboring (# of neighboring) in each sub-domain for the 32 and 64-partitioning of the matrix “lupcuf_cube-274625”

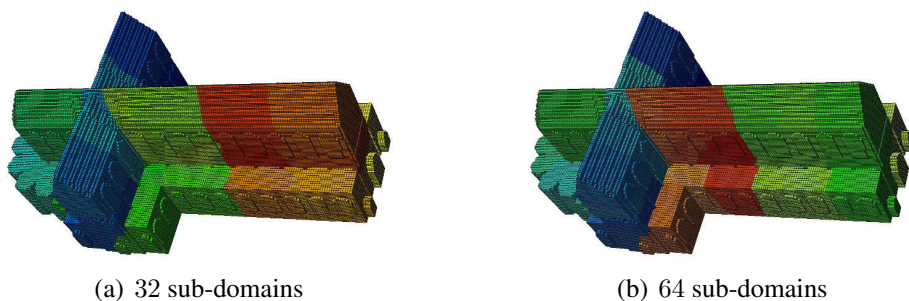


FIGURE 5.19: Projection of the 32- and 64-partitioning of the matrix “church-484507” into the associated mesh given in Figure 5.12

leads to regular sub-parts. The partitioning looks like what can be achieved by partitioning the mesh directly.

5.4.3 Jacobi’s numerical results using GPU

Before reporting the results of parallel synchronous and asynchronous algorithms, we first evaluate the behavior of Jacobi algorithm on GPU considering the test matrices described in Table 5.6. The GPU version will be compared to the CPU one. The implementation corresponds to the vector version of the Jacobi algorithm (see Proposition 4.64, Page 139) and (see Algorithm 4.5, Page 138). Moreover, we will compare the Jacobi results with the conjugate gradient (CG) method with and without preconditioner (P-). Let us remark that the matrices presented in Table 5.6 are symmetric positive-definite (necessary condition of the CG

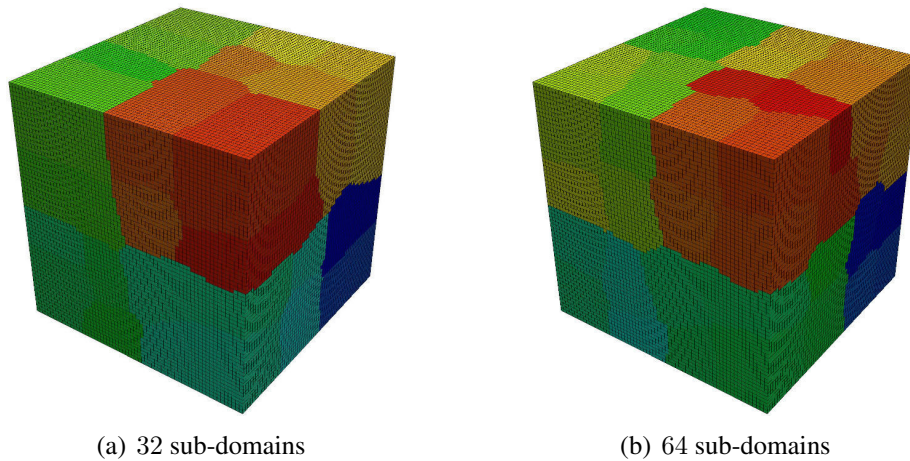


FIGURE 5.20: Projection of the 32- and 64-partitioning of the matrix “lupcuf_cube-274624” into the associated mesh given in Figure 5.9(b)

method). The diagonal (Jacobi) preconditioner is considered. As for the implementation of the preconditioning step in the iterative Krylov method (see Section 3.6.2, Page 86), in the Jacobi algorithm, we compute the inverse of the diagonal on CPU and send it to the GPU outside the iterative routine (Line 3 of Algorithm 4.5).

The numerical experiments have been performed on *Platform 4* (see Section 2.6, Page 35), which consists of a workstation equipped with an Intel Core i7930 running with 2.67GHz, which has 4 cores and 12 GB main memory and two *NVIDIA* graphics cards: a Tesla K20c (dev#0) with 4799GB memory and GeForce GTX 570 with 1279MB memory (dev#1). The GPU algorithms have been carried out on the Tesla K20c (dev#0). In the following results, the GPU codes use the best threading distribution of the CUDA grid (see Table 3.22, Page 88). For each simulation case, we ran 10 trials, and the presented times correspond to the average. The convergence criteria of the Jacobi experiments presented in Table 5.7 are defined by $\|b - Au^{(k)}\|_2 < \varepsilon$. where ε is the residual threshold (see Line 9 of Algorithm 4.5). Note that $\|b - Au^{(k)}\|_2 = \|u^{(k+1)} - u^{(k)}\|_2 / \|D^{-1}\|_2$ (see Proposition 4.66, Page 139). Table 5.7 reports the double precision execution times of the Jacobi algorithm in seconds on both CPU and GPU for $\varepsilon = 10^{-6}$, $\varepsilon = 10^{-8}$ and $\varepsilon = 10^{-10}$ respectively. In Table 5.7, we collect the name of the problem (see Table 5.6, Page 181) in the first column, from column 2 to column 5 we give the numbers of iterations, the CPU times in seconds, the GPU times in seconds and the speed-ups CPU/GPU for the double precision execution times with $\varepsilon = 10^{-6}$. Those considering $\varepsilon = 10^{-8}$ and $\varepsilon = 10^{-10}$ are respectively given from column 6 to column 9, and from column 10 to 13. The speed-ups presented in these results are computed as follows:

$$ratio = \frac{CPU_time}{GPU_time}.$$

The times given in the table correspond to the global running time, which includes the times of transfers between CPU and GPU.

Table 5.7 clearly confirms the results presented in Chapter 3 (see Section 3.6, Page 80) for the matrices described in Table 5.6. In general, the speed of convergence with the Jacobi iteration is slow compared to the iterative Krylov methods such as the CG. The CG without preconditioner and the Jacobi method perform approximately the same number of arithmetic operations. But the Jacobi method needs much more iterations compared to the CG. The

h	$\varepsilon = 10^{-6}$				$\varepsilon = 10^{-8}$				$\varepsilon = 10^{-10}$			
	#iter	CPU time (s)	GPU time (s)	ratio	#iter	CPU time (s)	GPU time (s)	ratio	#iter	CPU time (s)	GPU time (s)	ratio
<i>luf_cube-35937</i>	1,236	9.5	2.8	3.4	1,660	12.8	3.7	3.5	2,085	16.0	4.7	3.4
<i>luf_cube-274625</i>	4,937	298.4	16.5	18.1	6,635	400.1	22.2	18.0	8,334	502.0	27.9	18.0
<i>church-484507</i>	473	50.1	2.1	23.9	123	78.3	3.3	23.7	164	107.1	4.4	24.3
<i>lupcuf_cube-35937</i>	1,196	9.2	2.7	3.4	1,655	12.4	3.6	3.4	2,078	15.5	4.5	3.4
<i>lupcuf_cube-274625</i>	4,776	288.1	16.0	18.0	6,613	387.7	21.5	18.0	8,306	486.7	26.9	18.1
<i>gravi_hexas100x100x6_0</i>	18,220	265.7	43.3	6.1	24,440	356.3	58.1	6.1	30,661	447.9	72.8	6.2
<i>gravi_hexas100x100x6_1</i>	72,895	8,386.3	359.5	23.3	97,780	11,249.5	480.8	23.4	122,665	14,134.7	603.5	23.4
<i>muluf_hexas100x100x6_0</i>	16,702	243.8	39.6	6.2	23,554	343.7	55.8	6.2	30,406	443.5	71.9	6.2
<i>muluf_hexas100x100x6_1</i>	70,229	8,078.5	346.0	23.3	98,496	11,351.9	484.6	23.4	126,764	14,616.9	623.8	23.4

TABLE 5.7: Double precision CSR Jacobi method for $\varepsilon = 10^{-6}$, $\varepsilon = 10^{-8}$ and $\varepsilon = 10^{-10}$

speed-up increases with the number of iterations. The higher the number of iterations, the greater the speed-up. For a large-size matrix, the Jacobi method becomes rapidly unusable with CPU in terms of time computing. One solution can consist in seeking the appropriate iterative method according to the properties of the matrix. Otherwise, Table 5.7 shows that GPU can be used to accelerate the algorithm. In parallel algorithms, GPU can be used to accelerate the local computations. Table 5.7 also shows that when we seek to refine the solution by decreasing the residual threshold, ε , the CPU/GPU speed-up remains unchanged. Table 5.7 clearly show the value of GPU to accelerate the Jacobi algorithm for large sparse matrices. The convergence criteria of the CG and P-CG experiments presented respectively in Table 5.8 and Table 5.9 are defined by $\frac{\|b - Au^{(k)}\|_2}{\|b\|_2} < \varepsilon$. The performance of the CG and preconditioned CG (P-CG) on CPU and GPU for $\varepsilon = 10^{-6}$, $\varepsilon = 10^{-8}$ and $\varepsilon = 10^{-10}$ are respectively reported in Table 5.8 and Table 5.9.

h	$\varepsilon = 10^{-6}$				$\varepsilon = 10^{-8}$				$\varepsilon = 10^{-10}$			
	#iter	CPU time (s)	GPU time (s)	ratio	#iter	CPU time (s)	GPU time (s)	ratio	#iter	CPU time (s)	GPU time (s)	ratio
<i>luf_cube-35937</i>	55	0.4	0.3	1.6	66	0.5	0.3	1.9	78	0.6	0.4	1.6
<i>luf_cube-274625</i>	112	7.1	0.7	10.5	134	8.5	0.7	13.0	158	10.1	0.8	12.8
<i>church-484507</i>	57	6.3	0.3	18.7	77	8.5	0.5	15.5	99	10.9	0.6	18.5
<i>lupcuf_cube-35937</i>	55	0.4	0.3	1.6	66	0.6	0.2	2.4	78	0.6	0.3	2.2
<i>lupcuf_cube-274625</i>	112	7.1	0.6	12.1	133	8.4	0.8	11.0	157	9.9	0.9	10.8
<i>gravi_hexas100x100x6_0</i>	119	1.8	0.6	2.9	140	2.1	0.7	3.2	156	2.5	0.8	3.0
<i>gravi_hexas100x100x6_1</i>	234	28.4	1.5	18.6	268	32.5	1.8	18.2	305	37.0	2.0	18.6
<i>muluf_hexas100x100x6_0</i>	385	5.9	1.6	3.7	508	7.9	1.9	4.1	643	9.9	2.5	4.0
<i>muluf_hexas100x100x6_1</i>	803	97.4	5.2	18.6	1,069	132.5	7.0	19.0	1,350	164.1	8.8	18.6

TABLE 5.8: Double precision CSR CG method for $\varepsilon = 10^{-6}$, $\varepsilon = 10^{-8}$ and $\varepsilon = 10^{-10}$

As said above, Table 5.8 corroborates the fast convergence of the CG algorithm compared to the Jacobi method. The diagonal preconditioner accelerates the convergence as shown in Table 5.9.

h	$\varepsilon = 10^{-6}$				$\varepsilon = 10^{-8}$				$\varepsilon = 10^{-10}$			
	#iter	CPU time (s)	GPU time (s)	ratio	#iter	CPU time (s)	GPU time (s)	ratio	#iter	CPU time (s)	GPU time (s)	ratio
<i>luf_cube-35937</i>	55	0.5	0.3	1.5	66	0.6	0.4	1.5	78	0.7	0.3	2.1
<i>luf_cube-274625</i>	112	8.1	0.7	11.1	134	9.7	0.9	11.4	158	10.4	0.8	12.3
<i>church-484507</i>	55	0.5	0.2	2.1	66	0.6	0.3	2.1	78	0.7	0.3	2.1
<i>lupcuf_cube-35937</i>	112	7.3	0.7	9.8	133	8.7	0.9	9.9	157	10.3	0.9	11.2
<i>lupcuf_cube-274625</i>	57	6.5	0.4	17.0	77	8.9	0.5	16.6	99	11.3	0.7	15.3
<i>gravi_hexas100x100x6_0</i>	112	1.8	0.5	3.8	131	2.1	0.5	3.8	146	2.3	0.8	3.0
<i>gravi_hexas100x100x6_1</i>	225	28.2	1.7	17.0	258	32.3	1.8	17.7	293	42.2	2.1	20.1
<i>muluf_hexas100x100x6_0</i>	168	2.7	0.7	3.8	227	3.6	0.9	3.8	277	4.4	1.2	3.8
<i>muluf_hexas100x100x6_1</i>	353	44.2	2.5	17.8	473	59.2	3.3	17.9	566	71.0	4.0	17.9

TABLE 5.9: Double precision CSR preconditioned CG (P-CG) method for $\varepsilon = 10^{-6}$, $\varepsilon = 10^{-8}$ and $\varepsilon = 10^{-10}$

From Table 5.8 and Table 5.9, let us note that when we refine the solution, *i.e.*, decrease the residual threshold, the speed-up is not really constant compared to the Jacobi method (see Table 5.7, Page 185). Knowing the performance results of the CG and Jacobi methods for the studied matrices, we now focus on the evaluation and analysis of parallel synchronous and asynchronous algorithms.

5.4.4 Numerical analysis of the theoretical speed-up

In this section, we give an analysis of the different theorems of theoretical speed-up, presented in Section 5.3, Page 161. We vary the dependent parameters in order to better understand their influences. The analysis of the theoretical speed-up helps get an idea of the expected performance of parallel algorithm compared to the sequential one. The analysis given here also helps estimate the efficiency of the asynchronous algorithm.

Execution Profiles

In this analysis, we consider the Laplacian problem.

Parameters related to the parallel system According to the analysis given in the book of I. Foster (Section 3.4 *Scalability Analysis*, pp. 99) [271] and the studies described in the following papers [261] [272] [273] [274], with the *MPI* API, we can consider the proportion of the *communication latencies*, t_l , (see Definition 5.31, Page 169) as follows: $3/6$ for *startup latency* (t_s), $1/6$ for *transmission latency* (t_t), and $2/6$ for *packetization latency* (t_p), (see Figure 5.21).

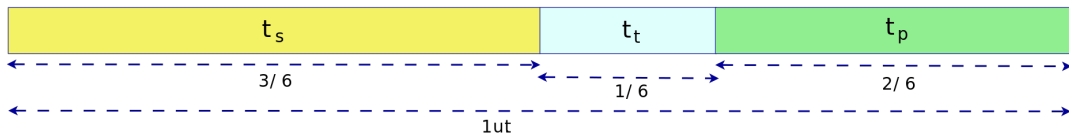


FIGURE 5.21: Proportion of *communication latencies*: $t_l(m, p) = t_s + m \times t_t + \lfloor \frac{m}{p} \rfloor \times t_p$

Parameters related to the Jacobi method The theoretical speed-ups are dependent on a constant $\xi = c_{para} = c_{seq} \in \mathbb{R}^+$, *i.e.*, $\xi = c_{upd} + c_{ewp} + c_{dot} + c_{mvp}$, where c_{upd} , c_{ewp} , c_{dot} , c_{mvp} are the coefficients of linear complexity respectively for the update (saxpy), the element-wise product, the dot product and the matrix-vector product operations (see Proposition 5.25, Page 166). In the Jacobi algorithm (see Algorithm 4.5, Page 138), we have to perform $n_{upd} = 2$ updates (saxpy), $n_{ewp} = 1$ element-wise product, $n_{dot} = 1$ dot product and $n_{mvp} = 1$ matrix-vector product (see Proposition 5.44, Page 173). In addition, update (saxpy), element-wise product, dot product and matrix-vector product carry out respectively $a_{upd} = 2$, $a_{ewp} = 1$ and $a_{dot} = 2$ arithmetic elementary operations. The ξ is then computed as follows:

$$\xi = n_{upd} \times a_{upd} + n_{ewp} \times a_{ewp} + n_{dot} \times a_{dot} + n_{mvp} \times a_{mvp}.$$

So, $\xi = 9$. This coefficient should also depend on the value of stencil associated with the finite element method. It gives an approximation of the number of non-zero values per line. For example, for a regular 3D grid with a $Q1$ finite element, the stencil value is equal to $9 \times 3 = 27$.

Analysis

The following numerical results illustrate the parallel *scalability* (scaling efficiency) of the Jacobi sub-structuring and band-row algorithms. We first analyze the theoretical scalability and then we evaluate real measurements. This parallel scalability indicates how efficient the algorithm is when using increasing numbers of parallel processors. We study how the theoretical speed-up varies with parameters such as the processor count and the problem size. In this analysis, we approximate the size of interfaces and sub-domains. These sizes strongly impact on the performance. In real cases, this step may decrease the performance compared to the theoretical model.

Strong Scaling In this case, we consider the *luf_cube-35937* problem where the size is equal to 35937. The first measurement consists in increasing the number of processors.

Figure 5.22(a) and Figure 5.22(b) show respectively the theoretical speed-up $\mathcal{S}^{para|seq}$ of the parallel sub-structuring algorithm (synchronous and asynchronous) regarding the sequential algorithm (see Theorem 5.29, Page 168) and $\mathcal{S}^{async|sync}$ of the asynchronous sub-structuring algorithm regarding the sub-structuring synchronous algorithm (see Theorem 5.30, Page 169) for the *luf_cube-35937* problem. In these figures, we vary the number of iterations of asynchronous algorithm, $N^a = \delta.N^s$ (see Definition 5.41, Page 171), where N^s is the number of iterations of the synchronous (sequential) algorithm, with $\delta = \{1.2, 1.4, 1.6, 2, 3\}$. In the legend of Figure 5.22, *sync* stands for synchronous and *async* δx means asynchronous with $N^a = \delta.N^s$.

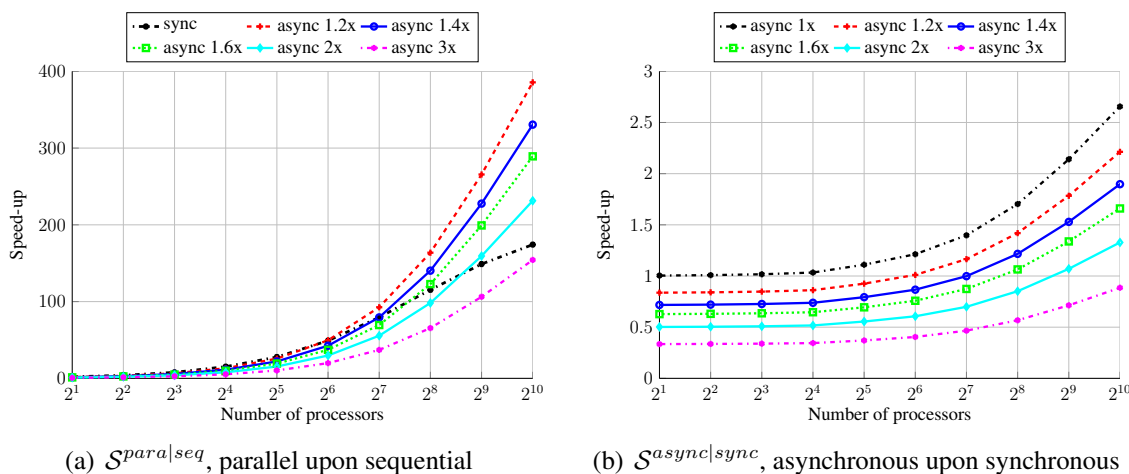


FIGURE 5.22: Theoretical speed-up $\mathcal{S}^{para|seq}$ and $\mathcal{S}^{async|sync}$ of sub-structuring Jacobi with *luf_cube-35937*

From Figure 5.22(a), the following observations can be made about the theoretical speed-up: the speed-up increases with increasing the number of processors for all performance models. In general, asynchronous algorithms require a greater number of iterations before the convergence than in the synchronous case. We can see in this figure that the asynchronous algorithm becomes more efficient compared to the synchronous one when we increase the number of processors. So, even if the former performs a greater number of iterations, with an acceptable large number of processors, we can find δ that asynchronous is more efficient. In contrast, the efficiency of the synchronous algorithms decreases fast with the number of processors. The asynchronous algorithm scales much more. This effect is particularly strong since the communication to computation ratio is high. This is due to low-speed

interconnection and fast iterations. These behaviors are confirmed in Figure 5.22(b). However, the sub-structuring algorithm presents the advantage of exchanging data only at the interface, which considerably optimizes the cost of communication, compared to band-row algorithms. Figure 5.23 gives the theoretical speed-up of the parallel band-row algorithm. The band-row

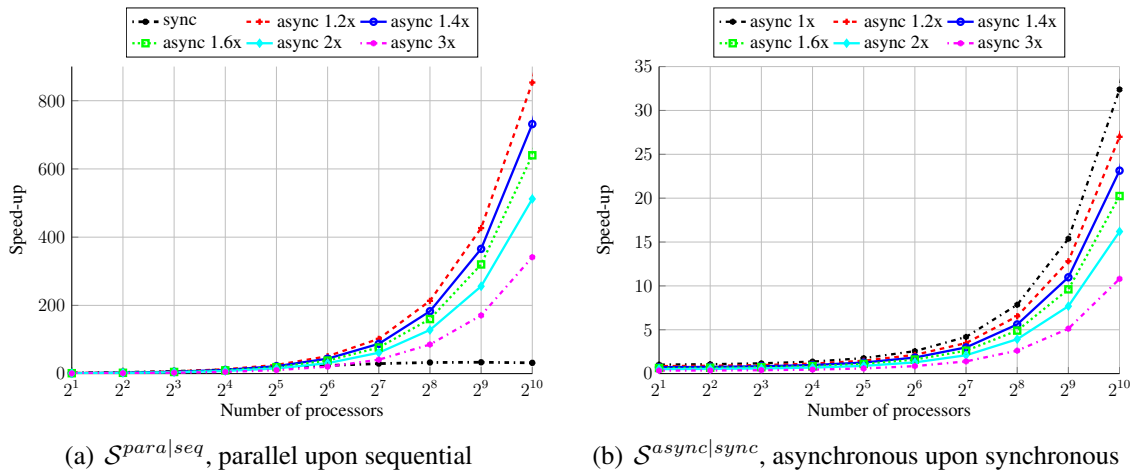


FIGURE 5.23: Theoretical speed-up $\mathcal{S}^{para|seq}$ and $\mathcal{S}^{async|sync}$ of band-row Jacobi with *luf_cube-35937*

algorithm requires a greater number of exchanges. As we can see in Figure 5.23(a) and Figure 5.23(b), the direct consequence is that the efficiency of the band-row synchronous algorithms decreases more quickly with the number of processors compared to sub-structuring algorithms. On the other hand, the asynchronous optimized is more efficient (as of 2^6), which is due to the removal of synchronization points. In a band-row algorithm, the impact of the increasing of δ in the efficiency of the asynchronous algorithm is weak compared to sub-structuring. The sub-structuring algorithms are faster than band-row. In the following, our study will concentrate on sub-structuring algorithms.

Limits of the efficiency of the asynchronous algorithm To evaluate these limits, we propose to vary the parameter δ , *i.e.*, the ratio $\frac{N^a}{N_s}$. Figure 5.24 reports the theoretical speed-up of sub-structuring of *luf_cube-35937* on the parameter δ . From Figure 5.24(a) and Figure 5.24(b),

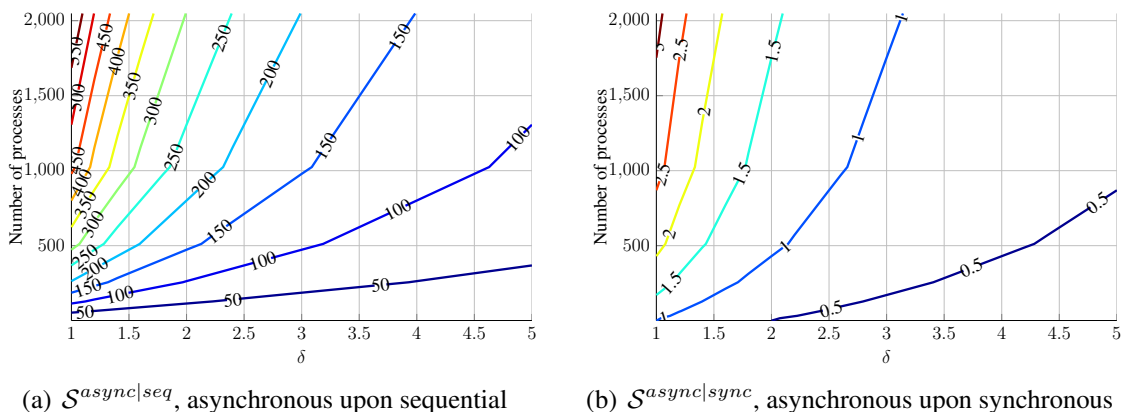


FIGURE 5.24: Theoretical speed-up $\mathcal{S}^{async|seq}$ and $\mathcal{S}^{async|sync}$ upon δ of sub-structuring with *luf_cube-35937*

the following observations can be made: speed-up increases with an increasing number of processors (as said above) and speed-up decreases with an increasing δ . For a fixed problem size,

if $\delta \gg 0$, *i.e.*, the asynchronous algorithm achieves many more iterations than the synchronous algorithm, then synchronous is more interesting. However, with a large number of processors, the asynchronous algorithm may become more efficient. This behavior can be expressed as follows: it exists (p, δ) , where p is the number of processors, such that asynchronous is more efficient than synchronous. In general terms, when δ increases, the algorithm is more effective with an increasing number of processors.

Scaled problem analysis A large number of processors is frequently used not only to solve fixed-size problems faster, but also to solve larger problems. Now, we evaluate the theoretical speed-up with a larger problem. We aim to study how the amount of performed computation is scaled with the number of processors. We consider the *luf_cube-274625* problem where the size is equal to 274625. In this case, the interface and sub-domain nodes increase. The size of messages becomes bigger. Figure 5.25(a) and Figure 5.25(b) report respectively the theoretical speed-up $\mathcal{S}^{para|seq}$ of the parallel sub-structuring algorithm (synchronous and asynchronous) regarding the sequential algorithm (see Theorem 5.29, Page 168) and the theoretical speed-up $\mathcal{S}^{async|sync}$ of the asynchronous sub-structuring algorithm regarding the sub-structuring synchronous algorithm (see Theorem 5.30, Page 169) for the *luf_cube-274625* problem. In these figures, we vary the number of iterations of the asynchronous algorithm, $N^a = \delta \cdot N^s$ (see Definition 5.41, Page 171), where N^s is the number of iterations of the synchronous (sequential) algorithm, with $\delta = \{1.2, 1.4, 1.6, 2, 3\}$. In the legend of Figure 5.22, *sync* stands for synchronous and *async* δx means asynchronous with $N^a = \delta \cdot N^s$. As we can see in

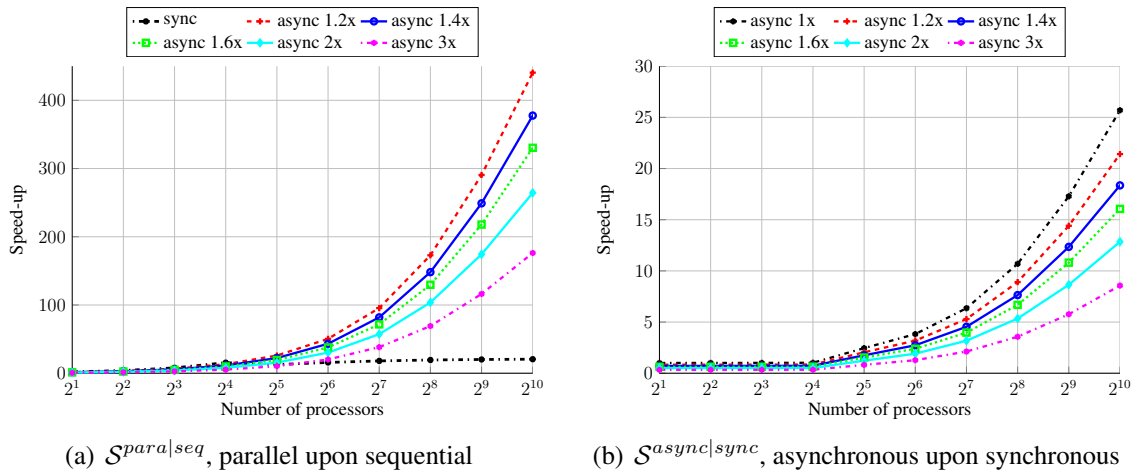


FIGURE 5.25: Theoretical speed-up $\mathcal{S}^{para|seq}$ and $\mathcal{S}^{async|sync}$ of the sub-structuring Jacobi with *luf_cube-274625*

Figure 5.25, the speed-up of the synchronous algorithms decreases very fast with the number of processors. Compared to the small problem, this is due to the increasing number of nodes at the interface. When the number of processors increases, the synchronous algorithm performs more exchanges. On the other hand, the asynchronous algorithm becomes more efficient. The asynchronous algorithm scales much more. Figure 5.26(a) and Figure 5.26(b) collect the theoretical speed-up of the sub-structuring of *luf_cube-274625* based on the parameter δ . As we can see in Figure 5.26, the limits concerning δ are higher. As a conclusion, considering a load-balanced problem with sufficient amount of data for each processor, asynchronous is efficient for large parallel computers. The asynchronous algorithm is highly scalable, mostly

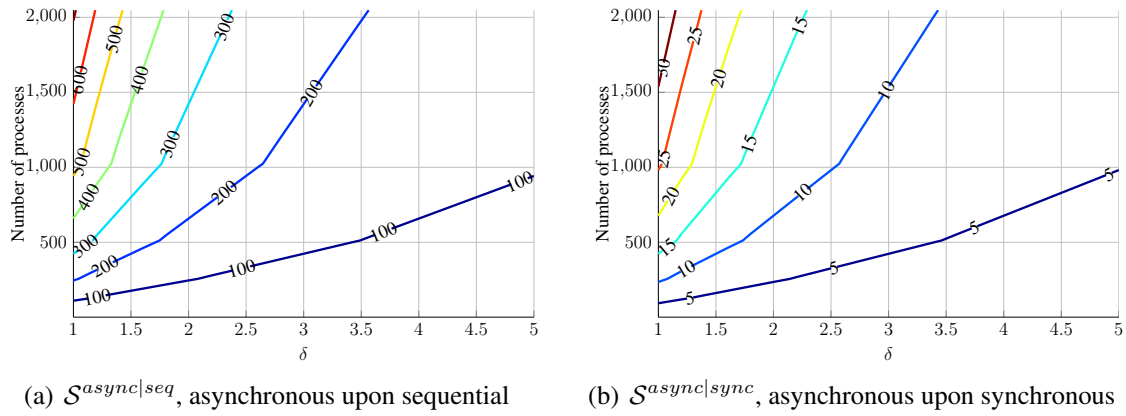


FIGURE 5.26: Theoretical speed-up $\mathcal{S}^{async|seq}$ and $\mathcal{S}^{async|sync}$ upon δ of sub-structuring with *luf_cube-274625*

when the amount of computation requires to increase. This is due to the fact that the amount of essential computations must increase at the same overhead rate.

Figure 5.27 shows the theoretical speed-up $\mathcal{S}^{sync|seq}$ and $\mathcal{S}^{async|seq}$ based on the degree of freedom. The two legends are to be used in the two graphs. The left one represents the problem sizes and the right one gives the number of processors. The size of the interfaces and the sub-problems are arbitrary. As we can see in these figures, the speed-up increases with an

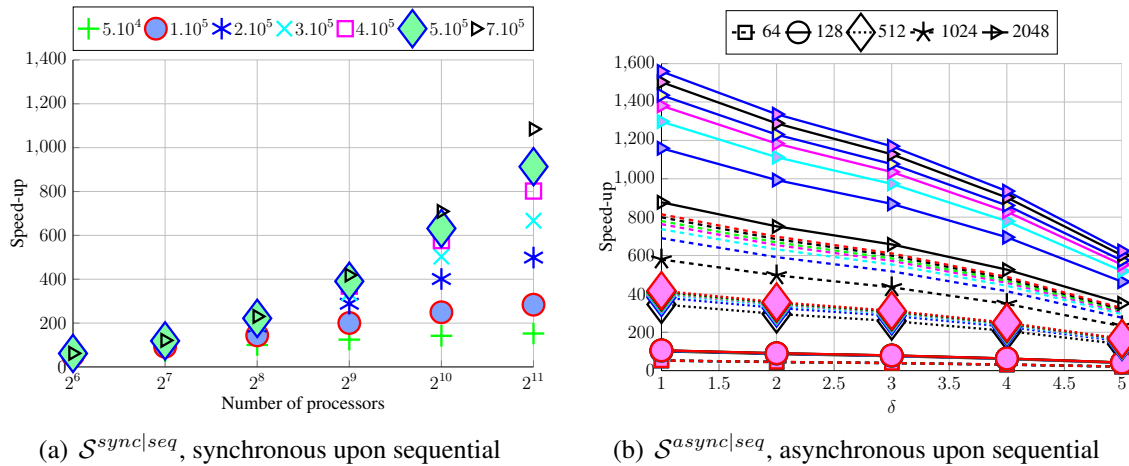


FIGURE 5.27: Sub-structuring theoretical speed-up $\mathcal{S}^{sync|seq}$ and $\mathcal{S}^{async|seq}$ upon degree of freedoms

increasing number of processors and increasing size.

When GPU is used to perform linear algebra operations, the speed-up may decrease with the number of iterations because of the constant copies between CPU/GPU at each iteration.

5.4.5 Parallel synchronous and asynchronous numerical results

On a cluster of multi-core-GPUs, all parallel and intensive computations on local data are performed on GPUs and reduction operations to compute global results are carried out by CPUs. The version of the MPI library used is OpenMPI (*OpenRTE*) 1.6.5.

Conjugate Gradient Here, we propose to evaluate and analyze the parallel sub-structuring CG method with synchronous iterations. Let us recall that the iterative Krylov methods, in

particular the CG, do not converge with asynchronous iterations. The influence of applying these techniques on the computation time of parallel processing is illustrated for the case of parallel conjugate gradient methods. In the following, we report the numerical results of the parallel sub-structuring Conjugate Gradient method. For each simulation case, we ran 100 trials with a residual tolerance threshold $\varepsilon = 1 \times 10^{-6}$, where an initial guess equals to zero. When no right-hand side is associated with the matrix, we consider a right-hand side vector filled with 1.

To be in line with previous chapters, the numerical experiments and benchmarks of the CG on a multi-core-GPUs system were performed with the set of matrices from the University of Florida repository (see Table 3.11, Page 68) and the same workstation, *Platform-1* (see Table 3.21, Page 88). The test cases are run on a single workstation since I did not have access to the cluster of a supercomputer at the time.

#subdoms	#subdom's number	2cubes_sphere		af_shell8		cfd2		Dubcova2	
		dof	nnz	dof	nnz	dof	nnz	dof	nnz
1	(0)	101,492	874,378	504,855	9,046,865	123,440	1,605,669	65,025	547,625
2	(0)	52,016	826,126	253,221	8,810,853	62,601	1,548,093	32,919	519,515
2	(1)	51,491	836,151	252,674	8,794,432	62,156	1,553,740	32,884	518,240
4	(0)	26,617	422,713	126,597	4,398,547	32,707	787,341	16,716	262,504
4	(1)	26,563	423,375	126,767	4,406,873	31,179	775,703	16,641	260,889
4	(2)	26,998	426,534	126,876	4,406,834	31,744	782,066	16,647	261,091
4	(3)	26,966	416,932	127,080	4,414,982	31,433	783,833	16,603	261,111
8	(0)	13,759	215,379	63,467	2,201,715	16,601	397,151	8,435	131,319
8	(1)	14,038	219,588	63,723	2,206,843	16,037	395,369	8,601	133,729
8	(2)	13,768	214,322	63,665	2,209,225	16,150	396,526	8,530	132,948
8	(3)	13,731	213,449	63,668	2,205,260	15,992	394,800	8,451	131,319
8	(4)	13,928	211,472	64,104	2,219,056	16,299	380,831	8,558	132,986
8	(5)	13,869	210,473	63,602	2,206,164	16,660	408,534	8,442	131,728
8	(6)	14,164	217,980	63,974	2,215,750	15,724	387,660	8,461	131,671
8	(7)	14,168	218,740	63,422	2,198,874	15,966	394,114	8,590	133,900
#subdoms	#subdom's number	finan512		qa8fm		thermomech_dM		thermomech_TK	
		dof	nnz	dof	nnz	dof	nnz	dof	nnz
1	(0)	74,752	335,872	66,127	863,353	204,316	813,716	102,158	406,858
2	(0)	37,436	298,814	33,248	831,710	102,158	711,558	51,134	356,060
2	(1)	37,395	298,479	33,268	832,512	102,158	711,558	51,179	355,965
4	(0)	18,713	149,273	17,092	419,714	51,116	355,948	25,630	177,754
4	(1)	18,713	149,273	17,328	423,986	51,200	356,086	25,702	178,598
4	(2)	18,713	149,273	17,226	421,230	51,173	356,313	25,672	178,354
4	(3)	18,713	149,273	17,217	422,921	51,150	355,746	25,660	178,392
8	(0)	9,369	74,649	9,067	218,487	25,693	178,523	12,871	89,213
8	(1)	9,369	74,649	9,032	217,326	25,650	178,332	12,903	89,315
8	(2)	9,369	74,649	8,995	214,621	25,664	177,978	12,907	89,383
8	(3)	9,369	74,649	8,997	214,955	25,643	178,209	13,027	90,187
8	(4)	9,369	74,649	8,949	214,675	25,677	178,115	12,842	88,912
8	(5)	9,369	74,649	8,899	214,329	25,680	178,428	12,917	89,491
8	(6)	9,369	74,649	9,080	217,594	25,673	178,385	12,870	89,184
8	(7)	9,369	74,649	9,146	219,042	25,612	178,094	12,895	89,097

TABLE 5.10: Degree of freedoms (dof) and non-zero values of the matrix of each sub-domain

Table 5.10 reports the degree of freedoms (dof) and non-zero values (nnz) of the matrix of each sub-domain. The first column gives the number of sub-domains (#subdoms), followed by the sub-domain number (#subdom's) in brackets.

A comparison of the results obtained for CPU clusters and GPU clusters is given in Table 5.11 for the sparse CSR preconditioned Conjugate Gradient (P-CG), and Table 5.12 represents the corresponding speed-up. To obtain these results, we execute the same algorithm 100 times with the same input data. The performance of our parallel P-CG solver for various sparse matrices are reported in columns 4 to 6 for CPU and in columns 4 to 10.

Both sequential CPU and GPU are given in columns three and seven, respectively. The first column lists the test cases' names and the second column gives the number of iterations

Matrix	#iter.	1CPU	2CPUs	4CPUs	8CPUs	GPU	2GPUs	4GPUs	8GPUs
2cubes_sphere	24	0.386	0.209	0.124	0.125	0.026	0.047	0.065	0.113
af_shell18	2,815	374.356	198.998	110.662	107.668	14.549	23.422	18.814	21.385
cfid2	2,818	71.078	38.114	21.657	22.111	3.730	5.904	8.280	12.186
Dubcova2	168	1.776	0.926	0.540	0.850	0.128	0.364	0.405	0.640
finan512	15	0.117	0.121	0.067	0.145	0.017	0.063	0.071	0.120
qa8fm	29	0.418	0.235	0.198	0.168	0.023	0.099	0.137	0.115
thermomech_dM	12	0.313	0.176	0.100	0.091	0.016	0.037	0.072	0.106
thermomech_TK	13,226	141.359	74.423	41.979	40.888	13.214	22.877	32.587	51.528

TABLE 5.11: Execution time for parallel sub-structuring P-CG (seconds) for CSR format

Matrix	#iter.	1CPU	2CPUs	4CPUs	8CPUs	GPU	2GPUs	4GPUs	8GPUs
2cubes_sphere	24	1.0	1.8	3.1	3.1	15.1	8.2	5.9	3.4
af_shell18	2815	1.0	1.9	3.4	3.5	25.7	16.0	19.9	17.5
cfid2	2818	1.0	1.9	3.3	3.2	19.1	12.0	8.6	5.8
Dubcova2	168	1.0	1.9	3.3	2.1	13.8	4.9	4.4	2.8
finan512	15	1.0	1.0	1.8	0.8	7.0	1.9	1.6	1.0
qa8fm	29	1.0	1.8	2.1	2.5	18.0	4.2	3.0	3.6
thermomech_dM	12	1.0	1.8	3.1	3.5	19.0	8.5	4.4	3.0
thermomech_TK	13226	1.0	1.9	3.4	3.5	10.7	6.2	4.3	2.7

TABLE 5.12: Speed-up of parallel sub-structuring P-CG (seconds) for CSR format

for the P-CG solver. For the presented set of test case matrices, the parallel sub-structuring P-CG gives satisfactory results. Nevertheless, we can remark that relative gains of some kinds of matrices increase when matrix bandwidths decrease and matrix sizes increase. As already mentioned, the gains of using GPU are the most visible for large-size matrices. Moreover, when the number of sub-domains increases, the GPU code underperforms. This is mainly due to the decreasing number of degrees of freedom within each sub-domain, and more particularly, the transfer between the GPU and the host when data exchanges are performed at the interfaces between the sub-domains. This degrades the overall execution time, as we can see in Table 5.12. In fact, MPI calls cannot be performed yet inside the GPU.

Jacobi The system is solved using a Jacobi splitting (see Section 4.4.3, Page 136). The benchmark consists in an analysis of the Jacobi method declined in 3 versions: *BANDROW (JB)*, *BANDROW-OP (JBO)* and *SUB-STRUCTURING (JSS)*, which correspond respectively to the Jacobi method with naive band-row partitioning, optimized band-row partitioning and sub-structuring partitioning (see Section 4.3, Page 112). We will prefix the synchronous version and the asynchronous version with an *S-* and *A-* respectively. For instance, *S-JSS* and *A-JSS* correspond to the synchronous and asynchronous Jacobi sub-structuring. The experiments have been performed on *Platform-6 of LISA cluster* (see Section 2.6, Page 35) with the set of matrices described in this chapter (see Table 5.6, Page 181). We have used the nodes $(C_1, C_2, C_3, C_4, C_5, C_6)$.

The stopping criterion used is the *simultaneous local convergence* of all the processors. The convergence criterion is the weighted norm defined by $\|D(u^{(k+1)} - u^{(k)})\|_\infty < \varepsilon$, where the residual threshold $\varepsilon = 10^{-8}$. The norm is weighted to the diagonal D .

Table 5.13, Table 5.14 and Table 5.15, Table 5.16 show the numerical results of the experiments. These tables are organized as follows: the top sub-table (e.g., Table 5.12(a)) reports the results of the *BANDROW (S-JB)*, *BANDROW-OP (S-JBO)* and *SUB-STRUCTURING (S-JSS)* synchronous Jacobi method and the bottom sub-table (e.g., Table 5.12(b)) collects the asynchronous sub-structuring method (*SUB-STRUCTURING (A-JSS)*). In the top sub-table (e.g., Table 5.12(a)), the number of processors is given in the first column, and the results of the *BANDROW (S-JB)*, *BANDROW-OP (S-JBO)* and *SUB-STRUCTURING (S-JSS)* synchronous

Jacobi method are respectively reported from column 2 to 4, from column 5 to 7, and from column 8 to 10. For each method, the first column gives the number of iterations, the second column collects respectively the communication (exchange) times and the solver execution times in seconds (s). The third column gives the efficiency (eff) of the synchronous algorithm upon the sequential code (one processor). In the bottom sub-table (e.g., Table 5.12(b)), we report the number of processors, the minimum, maximum and average number of iterations from column 1 to 4. The communication (exchange) times and the solver execution times in seconds (s) are respectively reported in column 5 and 6. The seventh and eighth columns collect the speed-up and efficiency of the asynchronous algorithm upon the sequential one. The two last columns give those based on the synchronous sub-structuring algorithm.

(a) Synchronous BANDROW, BANDROW-OP and SUB-STRUCTURING

#p (#n)	SYNCHRONOUS								
	S-J-BANDROW			S-J-BANDROW-OP			S-J-SUB-STRUCTURING		
	# iterations	time (s)	eff.	# iterations	time (s)	eff.	# iterations	time (s)	eff.
	comm – total			comm – total			comm – total		
1	746	0.0 – 10.9	100%	746	0.0 – 10.9	100%	746	0.0 – 10.9	100%
8	746	34.5 – 35.8	3.81%	746	1.5 – 3.1	43.81%	699	1.4 – 3.2	42.09%
16	746	67.0 – 67.7	1.01%	746	0.3 – 2.5	26.94%	678	1.5 – 2.6	25.88%
24	746	98.5 – 98.9	0.46%	746	0.5 – 2.9	15.96%	680	1.9 – 2.9	15.52%
32	746	132.6 – 132.9	0.26%	746	0.6 – 2.3	14.84%	646	1.7 – 2.4	14.46%
40	746	169.2 – 169.4	0.16%	746	0.4 – 2.4	11.38%	617	1.6 – 2.5	10.93%
48	746	547.5 – 547.6	0.04%	746	0.6 – 2.7	8.43%	582	1.9 – 2.6	8.70%
56	746	650.6 – 650.8	0.03%	746	0.8 – 2.3	8.48%	568	1.7 – 2.5	7.68%
64	746	734.6 – 734.7	0.02%	746	1.1 – 2.1	8.13%	525	1.8 – 2.4	7.02%

(b) Asynchronous SUB-STRUCTURING

#p (#n)	ASYNCHRONOUS J-SUB-STRUCTURING									
	# iterations			time (s)		upon sequential		upon parallel (Sync(SS)/Async)		
	min	max	mean	comm	total	ratio	efficiency (eff.)	ratio	efficiency (eff.)	
1	746	746	746	0.0	10.9	1.00	100%	1.00	100%	
8	1,293	1,625	1,441	0.3	3.1	3.58	44.75%	1.06	13.29%	
16	1,439	2,277	1,714	0.4	2.2	5.08	31.74%	1.23	7.67%	
24	1,741	2,589	2,122	0.2	1.8	5.98	24.90%	1.60	6.69%	
32	1,255	3,372	2,427	0.5	1.7	6.39	19.96%	1.38	4.31%	
40	958	3,945	2,913	0.4	1.9	5.86	14.65%	1.34	3.35%	
48	1,127	4,520	3,116	0.3	1.8	5.98	12.47%	1.43	2.98%	
56	1,338	4,537	3,060	0.5	1.7	6.57	11.74%	1.53	2.73%	
64	1,066	4,843	3,105	0.4	1.6	6.81	10.64%	1.52	2.37%	

TABLE 5.13: *luf_cube-35937*: Numerical results for the Synchronous (BANDROW, BANDROW-OP and SUB-STRUCTURING) and Asynchronous (SUB-STRUCTURING) Jacobi method

Normally, the number of iterations of all synchronous algorithms should be identical independently from the number of processors for all algorithms. The band-row versions give the same number of iterations for all numbers of processors. In contrast, the number of iterations of the synchronous sub-structuring algorithm is a little different. The effects on the number of iterations can be explained by the use of simultaneous local convergence as a stopping criterion. In fact, the global convergence depends on the local convergence, where the local sub-domain depends on the number of processors.

In parallel synchronous algorithms, the sub-structuring is always fast, in particular with a large number of processors. The naive band-row algorithm is the slowest. This version is slow because of a growing number of communications. We can see in all tables that with the optimized version we considerably decrease the number of exchanges and therefore improve the execution time of the solver. These results are even more important when the problem is large in size, as we can observe in Table 5.15(a).

(a) Synchronous BANDROW, BANDROW-OP and SUB-STRUCTURING

#p (#n)	SYNCHRONOUS								
	<i>S-J-BANDROW</i>			<i>S-J-BANDROW-OP</i>			<i>S-J-SUB-STRUCTURING</i>		
	# iterations	time (s)	eff.	# iterations	time (s)	eff.	# iterations	time (s)	eff.
	comm – total			comm – total			comm – total		
1	2,216	0.0 – 238.4	100%	2,216	0.0 – 238.4	100%	2,216	0.0 – 238.4	100%
8	2,216	848.7 – 882.7	3.38%	2,216	14.9 – 53.9	55.27%	2,132	13.7 – 48.0	62.02%
16	2,216	1,617.8 – 1,634.2	0.91%	2,216	2.3 – 37.1	40.21%	2,098	14.5 – 32.1	46.35%
24	2,216	2,675.6 – 2,686.6	0.37%	2,216	3.3 – 33.8	29.40%	2,067	13.7 – 27.8	35.74%
32	2,216	3,222.1 – 3,229.7	0.23%	2,216	2.9 – 28.7	25.98%	2,041	14.6 – 24.6	30.25%
40	2,216	3,725.8 – 3,731.7	0.16%	2,216	6.7 – 34.5	17.29%	1,995	17.2 – 27.1	21.95%
48	2,216	4,341.1 – 4,345.9	0.11%	2,216	6.6 – 33.6	14.79%	1,980	17.7 – 25.8	19.27%
56	2,216	4,957.1 – 4,961.2	0.09%	2,216	6.2 – 31.2	13.64%	1,907	16.5 – 24.7	17.26%
64	2,216	7,055.0 – 7,058.6	0.05%	2,216	5.7 – 28.0	13.31%	1,889	16.2 – 23.0	16.18%

(b) Asynchronous SUB-STRUCTURING

#p (#n)	ASYNCHRONOUS J-SUB-STRUCTURING								
	# iterations			time (s)		upon sequential		upon parallel (Sync(SS)/Async)	
	min	max	mean	comm	total	ratio	efficiency (eff.)	ratio	efficiency (eff.)
1	2,216	2,216	2,216	0.0	238.4	1.00	100%	1.00	100%
8	4,808	6,262	5,213	2.0	77.0	3.10	38.70%	0.62	7.80%
16	5,687	8,328	6,595	3.1	51.5	4.63	28.91%	0.62	3.90%
24	15,635	22,712	19,574	2.8	104.4	2.28	9.52%	0.27	1.11%
32	4,723	9,490	7,524	1.6	31.2	7.63	23.84%	0.79	2.46%
40	10,913	17,278	14,922	3.1	52.4	4.55	11.37%	0.52	1.30%
48	9,064	17,235	14,096	3.2	43.7	5.46	11.37%	0.59	1.23%
56	6,784	14,984	11,627	2.7	32.2	7.40	13.22%	0.77	1.37%
64	4,093	9,607	7,174	1.3	18.0	13.22	20.66%	1.28	1.99%

TABLE 5.14: *luf_cube-274625*: Numerical results for the Synchronous (BANDROW, BANDROW-OP and SUB-STRUCTURING) and Asynchronous (SUB-STRUCTURING) Jacobi method

The asynchronous versions of the algorithm are faster than the synchronous ones, and especially with a large number of processes. Let us note that the asynchronous algorithm is less efficient for small problems, as shown in Table 5.12(b).

(a) Synchronous BANDROW, BANDROW-OP and SUB-STRUCTURING

#p (#n)	SYNCHRONOUS								
	<i>S-J-BANDROW</i>			<i>S-J-BANDROW-OP</i>			<i>S-J-SUB-STRUCTURING</i>		
	# iterations	time (s)	eff.	# iterations	time (s)	eff.	# iterations	time (s)	eff.
	comm – total			comm – total			comm – total		
1	6,234	0.0 – 159.5	100%	6,234	0.0 – 159.5	100%	6,234	0.0 – 159.5	100%
8	6,234	650.1 – 668.2	2.98%	6,234	121.8 – 142.7	13.97%	6,234	8.4 – 33.8	59.00%
16	6,234	1,099.4 – 1,108.5	0.90%	6,234	105.2 – 133.1	7.49%	6,184	9.1 – 22.9	43.58%
24	6,234	1,644.8 – 1,650.9	0.40%	6,234	107.3 – 137.4	4.83%	6,130	12.0 – 25.7	25.82%
32	6,234	2,284.2 – 2,288.7	0.22%	6,234	105.3 – 136.6	3.65%	6,163	14.7 – 23.7	21.02%
40	6,234	5,655.7 – 5,659.4	0.07%	6,234	103.6 – 152.2	2.62%	5,940	12.7 – 22.8	17.50%
48	6,234	6,889.4 – 6,892.5	0.05%	6,234	110.7 – 151.7	2.19%	6,201	10.4 – 22.9	14.48%
56	6,234	7,422.6 – 7,425.3	0.04%	6,234	108.2 – 150.7	1.89%	6,025	11.8 – 21.7	13.11%
64	6,234	8,435.8 – 8,438.5	0.03%	6,234	98.1 – 149.2	1.67%	6,011	14.5 – 22.6	11.02%

(b) Asynchronous SUB-STRUCTURING

#p (#n)	ASYNCHRONOUS J-SUB-STRUCTURING								
	# iterations			time (s)		upon sequential		upon parallel (Sync(SS)/Async)	
	min	max	mean	comm	total	ratio	efficiency (eff.)	ratio	efficiency (eff.)
1	6,234	6,234	6,234	0.0	159.5	1.00	100%	1.00	100%
8	11,647	19,393	16,447	2.3	60.4	2.64	33.01%	0.56	6.99%
16	14,978	21,843	17,629	1.6	32.7	4.88	30.52%	0.70	4.38%
24	25,827	36,483	29,701	3.3	38.7	4.12	17.18%	0.67	2.77%
32	16,180	24,687	20,508	2.5	20.6	7.73	24.17%	1.15	3.59%
40	14,600	24,870	21,514	1.0	18.9	8.46	21.14%	1.21	3.02%
48	19,973	37,422	28,437	1.2	21.9	7.29	15.19%	1.05	2.18%
56	16,147	30,865	24,132	2.3	16.8	9.48	16.92%	1.29	2.30%
64	12,803	35,543	26,530	2.6	16.8	9.47	14.80%	1.34	2.10%

TABLE 5.15: *gravi_hexas100x100x6_0*: Numerical results for the Synchronous (BANDROW, BANDROW-OP and SUB-STRUCTURING) and Asynchronous (SUB-STRUCTURING) Jacobi method

(a) Synchronous BANDROW, BANDROW-OP and SUB-STRUCTURING

#p (#n)	SYNCHRONOUS											
	S-J-BANDROW			S-J-BANDROW-OP			S-J-SUB-STRUCTURING					
	# iterations	time (s)		eff.	# iterations	time (s)		eff.	# iterations	time (s)		eff.
		comm – total				comm – total				comm – total		
1	13,791	0.0 – 2,713.0		100%	13,791	0.0 – 2,713.0		100%	13,791	0.0 – 2,713.0		100%
8	13,791	9,508.5 – 9,907.6		3.42%	13,791	5,994.3 – 6,891.7		4.92%	13,791	66.0 – 469.7		72.20%
16	13,791	19,564.6 – 19,760.2		0.86%	13,791	6,869.0 – 7,399.3		2.29%	13,637	66.0 – 273.7		61.95%
24	13,791	27,329.7 – 27,453.9		0.41%	13,791	7,002.2 – 7,689.7		1.47%	13,791	71.9 – 222.8		50.73%
32	13,791	34,939.7 – 35,030.2		0.24%	13,791	7,309.4 – 8,429.6		1.01%	13,456	97.7 – 228.3		37.13%
40	13,791	42,060.2 – 42,132.9		0.16%	13,791	7,475.9 – 9,768.5		0.69%	13,734	116.4 – 210.4		32.24%
48	13,791	49,218.6 – 49,278.7		0.11%	13,791	6,513.5 – 10,769.9		0.52%	13,674	104.9 – 191.9		29.46%
56	13,791	56,170.8 – 56,223.9		0.09%	13,791	6,370.3 – 10,940.9		0.44%	13,590	105.5 – 187.8		25.80%
64	13,791	62,913.8 – 62,965.6		0.07%	13,791	7,770.8 – 11,921.9		0.36%	13,069	86.6 – 164.6		25.76%

(b) Asynchronous SUB-STRUCTURING

#p (#n)	ASYNCHRONOUS J-SUB-STRUCTURING									
	# iterations			time (s)		upon sequential		upon parallel (Sync(SS)/Async)		
	min	max	mean	comm	total	ratio	efficiency (eff.)	ratio	efficiency (eff.)	
1	13,791	13,791	13,791	0.0	2,713.0	1.00	100%	1.00	100%	
8	13,955	19,654	14,264	4.0	382.4	7.09	88.68%	1.23	15.35%	
16	16,057	22,056	18,124	5.8	243.8	11.13	69.55%	1.12	7.02%	
24	19,643	26,065	21,607	4.3	205.4	13.21	55.04%	1.09	4.52%	
32	42,963	54,869	46,466	8.5	350.3	7.74	24.20%	0.65	2.04%	
40	36,773	50,323	42,383	14.4	273.1	9.93	24.83%	0.77	1.93%	
48	19,577	31,430	26,074	10.3	148.1	18.32	38.16%	1.30	2.70%	
56	22,647	28,134	24,203	13.1	107.6	25.21	45.02%	1.75	3.12%	
64	17,645	38,945	29,456	14.1	100.9	26.89	42.01%	1.63	2.55%	

TABLE 5.16: *gravi_hexas100x100x6_1*: Numerical results for the Synchronous (BANDROW, BANDROW-OP and SUB-STRUCTURING) and Asynchronous (SUB-STRUCTURING) Jacobi method

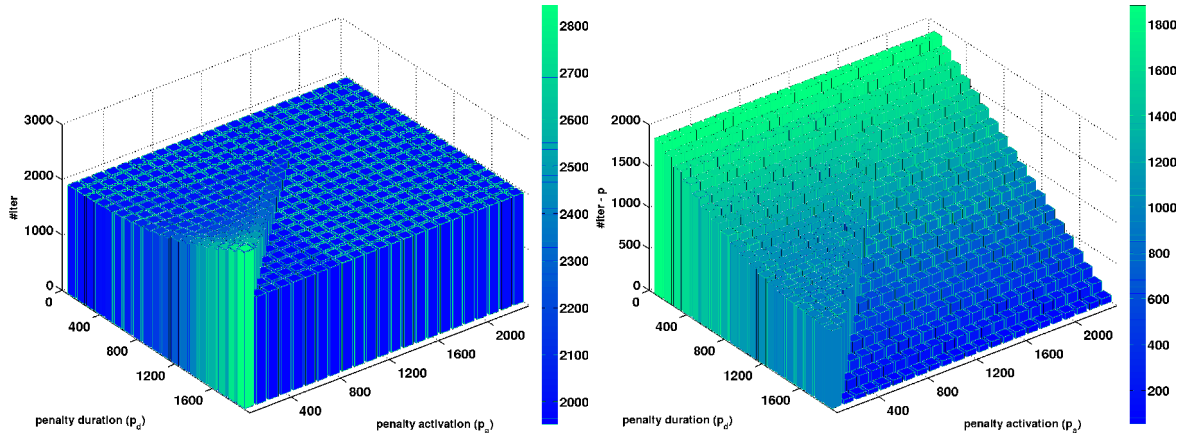
The results highlight the robustness of the theoretical speed-up analysis described previously. From these tables, the following observations can be made: the speed-up decreases with the increasing ratio $\delta = \frac{N^a}{N^s}$ (where N^a is the average number of asynchronous iterations) and increases with an increasing number of processors. These observations satisfy the theoretical speed-up theorems.

5.4.6 Fault-tolerance and iteration penalty behavior

The last set of results consists in numerical experiments that simulate the penalization of iterations, by stopping the solution update of a processor in idle state, and fault-tolerance by breaking down a given processor and resetting the solution at a given time. We analyze these behaviors with two test cases: *luf_cube-274625* and *gravi_hexas100x100x6_1*. In these experiments we have used 48 processors.

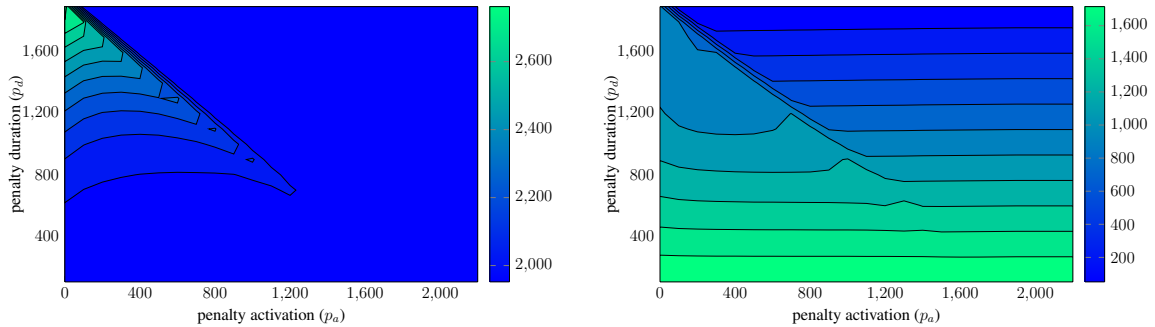
Theorem of delay (iteration penalization) In Figure 5.28 and Figure 5.29, we report the number of iterations of the sub-structuring Jacobi method of the *luf_cube-274625* test case with 48 processors where the iterations of processor #7 are penalized from the p_a^{th} iteration during p_d iterations. Figure 5.30 and Figure 5.31 give those of the *gravi_hexas100x100x6_1* test case. Figure 5.28(a) and Figure 5.29(a) collect the number of iterations of the non-penalized processors and Figure 5.28(b) and Figure 5.29(b) report those of prenalized processor according to the penalty activation, p_a , and the penalty duration p_d .

As we can see in Figure 5.28 and Figure 5.29, the penalized processor carries out fewer iterations than the non-penalized ones. After its idle time, the penalized processor takes advantage of the solution advance from the other processors. We can see in Figure 5.28(a) and Figure 5.29(a) that the number of iterations of the algorithm increases with an increasing



(a) Number of iterations of non-penalized processors (b) Number of iterations of the penalized processor (#7)

FIGURE 5.28: Simulation of chaotic iterations with the sub-structuring Jacobi method for *luf_cube-274625* with 48 processors



(a) Number of iterations of non-penalized processors (b) Number of iterations of the penalized processor (#7)

FIGURE 5.29: Number of iterations of the sub-structuring Jacobi method of the *luf_cube-274625* test case with 48 processors where the iterations of processor #7 are penalized from the p_a^{th} iteration during p_d iterations

penalty duration time. In contrast, the number of iterations of the penalized processor decreases with an increasing penalty duration time, as described in Figure 5.28(b) and Figure 5.29(b). Moreover, from these figures, we observe that when the activation is done tardily, *i.e.*, at a point when the solution is relatively advanced, near the convergence point, the number of iterations of the algorithm is close to that of a pure synchronous version. In addition, if the penalty duration time is relatively long, then the algorithm converges slowly and can diverge if the penalty duration exceeds a certain limit. These observations highlight the *delay theorem* (see Theorem 5.8, Page 160) of a simple chaotic algorithm.

We have performed the same experiments with the *gravi_hexas100x100x6_1* test case in order to analyze these observations for a bigger problem size wise, with a larger number of iterations in the pure synchronous case. Numerical results are summarized in Figure 5.30 and Figure 5.31. The experiment results corroborate and confirm our previous analysis. However, the limits of the penalty activation point and time of the idle state are larger. Indeed, the limits depend on the convergence of the pure synchronous case. The small number of iterations of the penalized processor is obtained when we disable the processor early and remains longer in pause. Let us note that the execution time is proportional to the number of iterations.

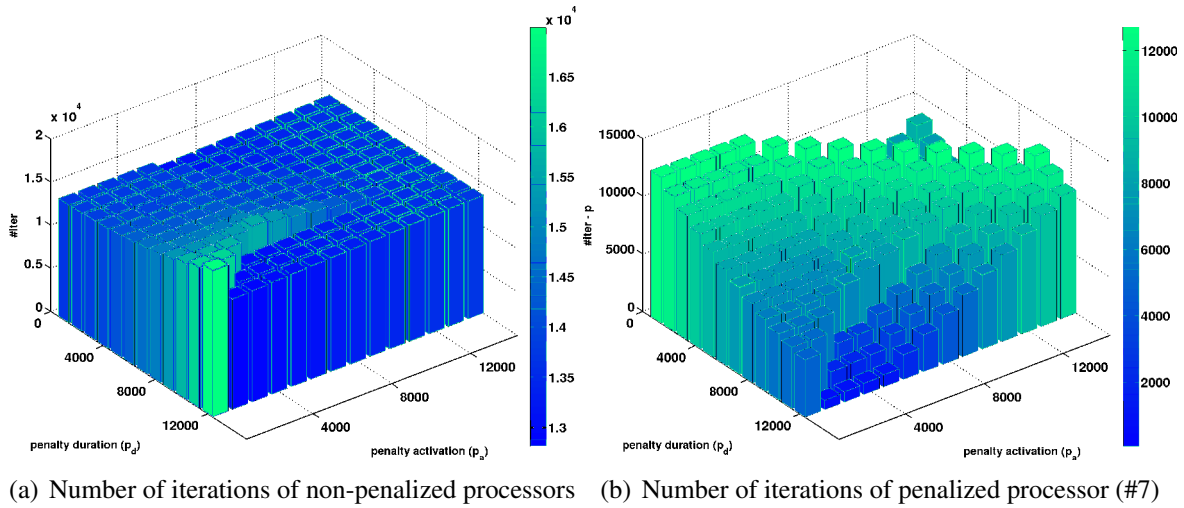


FIGURE 5.30: Simulation of chaotic iterations with the sub-structuring Jacobi method for *gravi_hexas100x100x6_1* with 48 processors

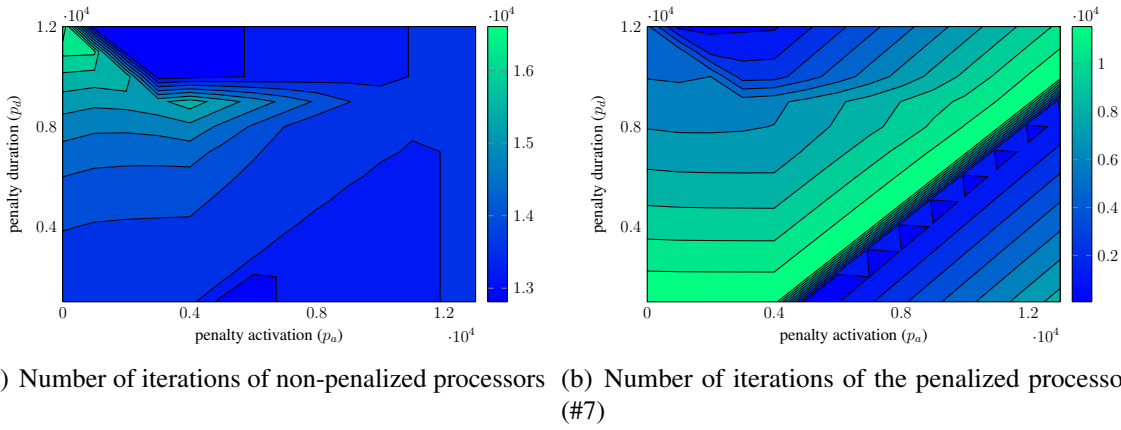


FIGURE 5.31: Number of iterations of the sub-structuring Jacobi method of the *gravi_hexas100x100x6_1* test case with 48 processors where the iterations of processor #7 are penalized from the p_a^{th} iteration during p_d iterations

Fault-tolerance In this experiment, we have solved both test cases with the Jacobi synchronous sub-structuring method, in so far as the k^{th} iteration the solution of the broken processor, $s \in \{0, \dots, p\}$, is restarted to the corresponding initial guess. Let us note \mathcal{E} the execution of this algorithm. At this iteration, the sending and receiving from the broken processor to and from the cooperating processors are disabled. Let $N_{[s]}$ be the number of iterations obtained from the execution of the pure synchronous algorithm. Let $N_{[c]}^{\{s\}}$ be the number of iterations obtained by the version with breakdown. In the following, we consider $s = 1$, the processor which will break down. We have carried out 10 simulations which consist of tests with $k = n \times \frac{N_{[s]}}{10}$, where $n = \{1, \dots, 10\}$. At iteration $k = n \times \frac{N_{[s]}}{10}$, $n = \{1, \dots, 10\}$, the solution is reset to 0, and the processor $s = 1$ does not send nor receive messages. Therefore, the processor s performs $N_{[c]}^{\{s\}} = N_{[c]}^{\{q\}} - n \times \frac{N_{[s]}}{10}$, $\forall q \in \{1, \dots, p\} \setminus \{s\}$. Let us note that the other processors that cooperate with the broken one do not either receive from it. This mechanism allows us to simulate the breakdown of a process, and simulates a model of *fault-tolerance*. Figure 5.32(a) and Figure 5.32(b) respectively report the number of iterations obtained from the execution of the sub-structuring Jacobi method of the *luf_cube-274625* and

gravi_hexas100x100x6_1 tests cases with 48 processors, where the solution of the processor $s = 1$ is restarted to 0 at the iteration $k = n \times \frac{N_{[S]}}{10}$, $n = \{1, \dots, 10\}$. From Figure 5.32(a)

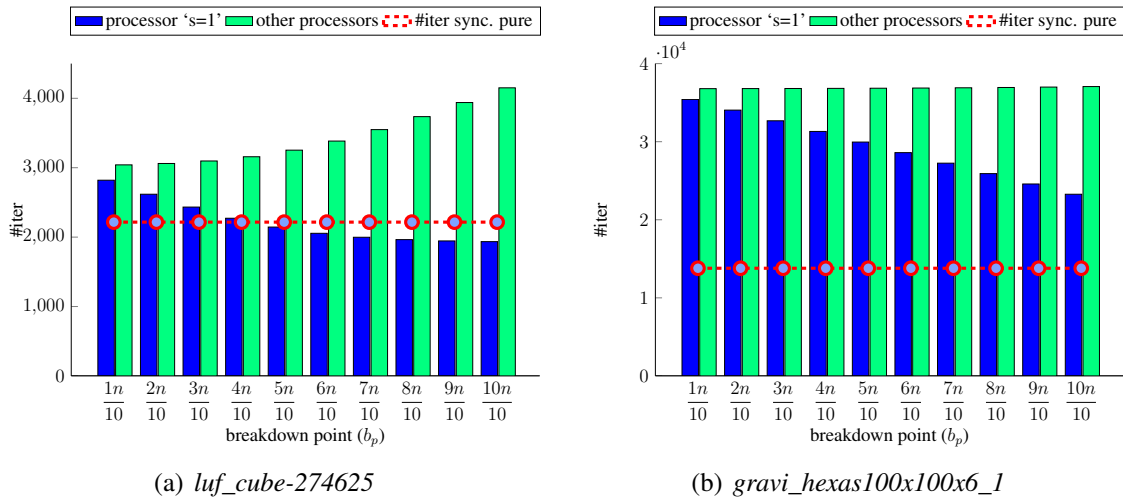


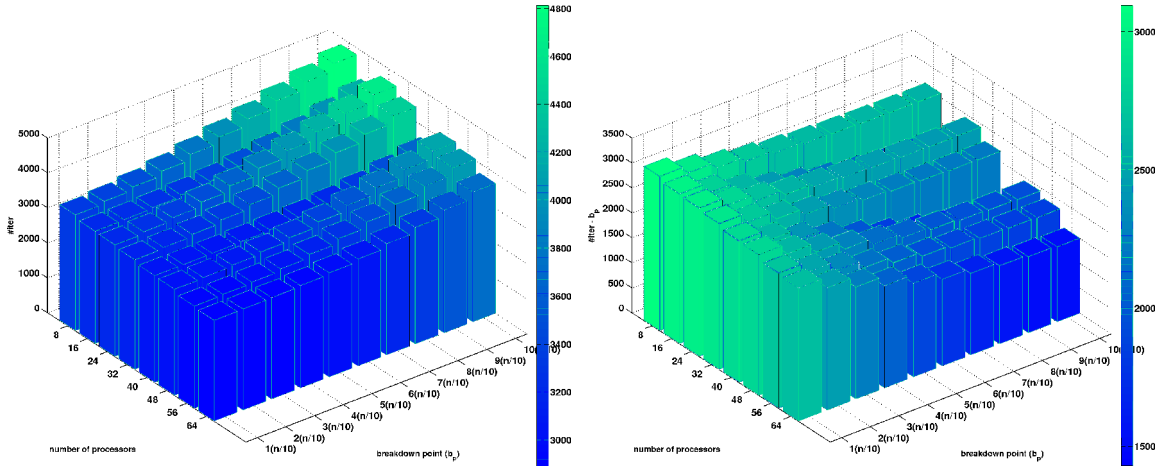
FIGURE 5.32: Number of iterations of the sub-structuring Jacobi method of the *luf_cube-274625* and *gravi_hexas100x100x6_1* tests cases with 48 processors with $s = 1$

and Figure 5.32(b), the following observations can be made: the number of iterations of the processor $s = 1$ decreases with an increasing breakdown point, and in contrast, that of the non-penalized processors increases with an increasing breakdown point. As we can see in Figure 5.32, before the breakdown point $k = 5 \times \frac{N_{[S]}}{10}$, the number of iterations of the processor $s = 1$ is higher than the number of iterations of the pure synchronous case. This result means that from a certain limit breakdown point n_l , the number of iterations of the processor s will be smaller than the pure synchronous one. As we can remark in Figure 5.32(b), this limit increases for a problem that requires a large number of iterations to converge synchronously. Otherwise formulated, there exists $n_l \in \mathbb{R}$, $0 < n_l \leq 1$ so that if a processor s breaks down at $k = n_l \times N_{[S]}$, the number of iterations of the processor s will be smaller than the synchronous one, $N_{[S]}$. To confirm this observation we now analyze the asymptotic behavior.

We have performed the same algorithm as in Figure 5.32 for 8, 16, 24, 32, 48, 56, 64 processors. The numerical results of these experiments are collected in Figure 5.33 and Figure 5.34. Figure 5.33(a) and Figure 5.33(b) report respectively the numerical results of the *luf_cube-274625* test case for processor the $s = 1$ and other non-penalized processors. Those of the *gravi_hexas100x100x6_1* are described respectively in Figure 5.34(a) and Figure 5.34(b).

Figure 5.33 and Figure 5.34, with a different number of processors, corroborate the results obtained in Figure 5.32 with 48 processors. Moreover, from these figures, we remark that the $k = n_l \times N_{[S]}$ limit moves back with an increasing number of processors. This analysis attempted to illustrate the fault-tolerance theorem (see Theorem 5.9, Page 161) and (see Proposition 5.10, Page 161).

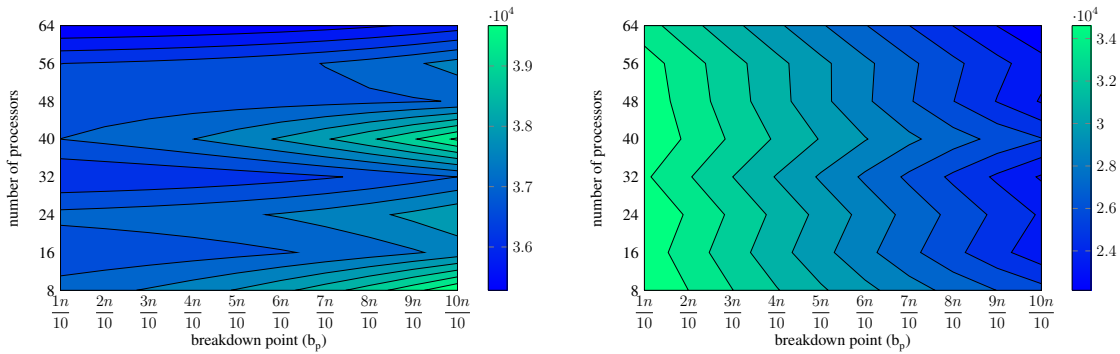
Conclusion This analysis was an attempt to simulate chaotic iterations. This analysis is interesting for heterogeneous systems. Indeed, when we solve an algorithm, one can voluntarily stop (or breakdown) a slow processor over a given time so that it does not slow down general calculations. According to our proposition, the processor can take advantage from the advance of the solution in the other processors.



(a) Number of iterations of non-penalized processors (b) Number of iterations of processor $s = 1$

FIGURE 5.33: Number of iterations of the sub-structuring Jacobi method with *luf_cube-274625* regarding the number of processors (8, 16, 24, 32, 48, 56, 64) and breakdown points

$$(k = n \times \frac{N[S]}{10}, n = \{1, \dots, 10\})$$



(a) Number of iterations of the non-penalized processors (b) Number of iterations of the processor $s = 1$

FIGURE 5.34: Number of iterations of the sub-structuring Jacobi method with *gravi_hexas100x100x6_1* regarding the number of processors and breakdown points

5.5 Conclusion

In this chapter, we have presented and evaluated parallel algorithms for solving large sparse linear systems. We have proposed and analyzed hybrid sub-structuring methods that should pave the way to exascale hybrid methods. Given the large-size of the problems to be solved, we focused particularly on asynchronous parallel iterative algorithms. We have presented the proof of the convergence of sub-structuring methods with asynchronous iterations. The numerical results clearly show that the asynchronous versions of the algorithm are generally faster than the synchronous versions, in particular with a large number of processes. The asynchronous algorithm scales much more with the number of processors compared to synchronous algorithm. This is due to the slow interconnection and the use of synchronous communications for the synchronous version. The sub-structuring algorithms are more efficient than the classical splitting algorithms. They also show that the integration of GPUs helps algorithms accelerate their resolution. However, when the number of sub-domains increases, the GPU code underperforms. This is mainly due to the decreasing number of

degrees of freedom within each sub-domain, and especially to the transfer between the GPU and the host when data exchanges are performed at the interfaces between sub-domains. This degrades the overall execution time. Indeed, MPI calls cannot be performed inside the GPU.

To better understand the behavior of both synchronous and asynchronous iterations, compared to sequential iterations, we have proposed some theorems concerning the speed-up of fully parallelizable algorithms such as the Jacobi method. We have also studied the behavior of parallel algorithms in terms of fault-tolerance and when an iteration is penalized. The scalability and performance of the methods are tested on diverse scientific problems, together with numerous numerical experiments which clearly illustrate the robustness, competitiveness and efficiency of parallel algorithms, and much more so when combined with GPU Computing.

In the next chapter, we present parallel algorithms by an effective application of parallel processing based on the hybrid CPU/GPU domain decomposition method (DDM). The purpose is to analyze the behavior of DDM on a cluster of GPUs.

Implementation of the Optimized Schwarz method without overlap for the gravitational potential equation on a cluster of GPUs

” *Computer science is no more about computers than astronomy is about telescopes.*

— Edsger Dijkstra
(Scientist)

6.1 Introduction

MANY engineering and scientific problems need to solve boundary value problems for partial differential equations or their systems. For most cases, to obtain the solution with desired precision and in acceptable time, the only practical way is to harness the power of parallel processing. In this chapter, we present some effective applications of parallel processing based on the hybrid CPU/GPU domain decomposition method. Within the family of domain decomposition methods, the so-called optimized Schwarz methods have proven to have good convergence behavior compared to the classical Schwarz methods. The price for this feature is the need to transfer more physical information between sub-domain interfaces. For solving large systems of linear algebraic equations resulting from the finite element discretization of the sub-problem for each sub-domain, the Krylov method is often a good choice.

Since the overall efficiency of such methods depends on the effective solver, approaches that use GPU instead of CPU for such tasks look very promising. In this chapter, we use the effective implementation of algebraic operations for the iterative Krylov methods on GPU (see Chapter 3, Page 39). In order to ensure a good performance of the non-overlapping Schwarz method, we propose to use optimized conditions obtained by a stochastic technique based on the Covariance Matrix Adaptation Evolution Strategy (CMA-ES). The performance, robustness, and accuracy of the proposed approach are demonstrated for the solution of the gravitational potential equation for the data acquired from the geological survey of the Chicxulub crater.

My contribution in this chapter is to modify the high-order finite element solver to be run on the GPU. We use optimized conditions obtained by a stochastic technique based on the CMA-ES algorithm.

I started with an existing code of optimized Schwarz domain decomposition developed by Frederic Magoulès *et al.* [22] I modify the sub-domain solvers in order to be run on GPUs. The

implementation is based on the acceleration of the local solutions of the linear sub-systems associated with each sub-domain using GPUs.

Geophysical exploration by seismic imaging can often benefit from a complementary analysis of the gravitational potential equation by the finite element method. Our approach to the analysis of the gravitational potential equation is based on partitioning the computational domain into a number of sub-domains, with each sub-domain assigned to one processor. The next step is to carry on the iterations of the Schwarz method with optimal conditions obtained by the CMA-ES algorithm. The solution of the independent sub-problems in each sub-domain required for each iteration step of the Schwarz algorithm is done in parallel. Central to our proposal is the application of CUDA/GPU computing for solving local problems in each sub-domain. We evaluate the gravimetry field in the Chicxulub crater area located between the Yucatan region and the Gulf of Mexico, which shows strong gravimetry and magnetic anomalies. The gravimetry problem is described in Chapter 3 (see Section 3.5.3, Page 73).

The plan of the chapter is the following. Section 6.2 describes the optimized Schwarz method, followed in Section 6.3 by an overview of the stochastic-based technique to determine the optimized transmission conditions. Section 6.4 reports numerical results performed on a realistic test case of our Schwarz methods with optimized stochastic conditions. Finally, Section 6.5 concludes the chapter.

Keywords — Domain Decomposition Method, Optimized Schwarz method, CMA-ES algorithm, Krylov methods, GPU, CUDA, Gravitational potential equation

6.2 Optimized Schwarz method

In order to solve gravity problems arising from the finite element analysis, domain decomposition methods [16] [275] [276] [277] are considered. As said in the previous section, preconditioning techniques, such as domain decomposition methods, guarantee fast convergence. Domain decomposition methods [17] [18] [19] [98] [278] rely on splitting the global domain into several sub-domains, and in independently solving, in parallel, the sub-problems formulated for each sub-domain. In order to set up boundary conditions for the sub-problems, the neighboring sub-systems must exchange information about the solution along the interfaces. The performance of the algorithms strongly depends on the interface conditions [99], which can be adjusted either with a discrete approach [105] [170] [103] [172] or a continuous approach [165] [279] [167] [101] [168] [280] [281]. Similar approaches like the Aitken-Schwarz methods have shown a strong efficiency too, as described in references [282] [283]. These transmission conditions [99] between adjacent sub-domains must be carefully defined to assure the convergence of the solution. The solution of independent sub-problems in parallel is required at each iteration. The interface problem is often solved with an iterative method, whereas the sub-problems can be solved either by using a direct method or an iterative one. In the presented case, we chose the iterative Krylov methods to resolve the sub-problems.

The origins of the classical Schwarz algorithms date back to over 100 years ago [15]. At the time, Schwarz had proposed a method, later called after him, to demonstrate the existence and uniqueness of solutions to the Laplace equation on irregular domains. At the core of his method is the decomposition of non-regular domains into regular sub-domains, with some overlap. The solution for the whole domain was obtained by a special iterative scheme,

special in the sense that each iteration step requires solving a problem only for the regular sub-domain. The suitably selected boundary conditions are called transmission conditions, as they propagate the solution in one sub-domain onto the boundary of the overlapping sub-domain. The transmission conditions between sub-domains are crucial to obtain a satisfactory speed of convergence of the Schwarz domain decomposition methods [284]. The rate of convergence of the classical Schwarz method depends on both the size of the overlapping layers and the spatial frequency k of the Fourier transform of the solution. In the case of the Schwarz domain decomposition methods without overlap between sub-domains [161] [162] [163] [285] [100], the algorithm can be formulated by changing the transmission conditions from Dirichlet to Robin [164] [165]. The absorbing boundary transmission conditions located on the interface between the non-overlapping sub-domains are the crucial factors to ensure a rapid convergence of the iterative Schwarz algorithm.

The difficulty in the parallel implementation of an algorithm that produces the “optimal” interface conditions lies in the non-local properties of the operators by which the problem is formulated. One of the approaches to deal with this problem is based on approximating the associated discrete non-local operators with algebraic ones [105] [103] [170] [172], or on approximating these continuous non-local operators with partial differential operators [167] [286] [287]. In this chapter, we study an approximation based on a new stochastic optimization process.

Without loss of generality and for the sake of clarity, the gravitational potential equation is defined in the domain Ω with homogeneous Dirichlet conditions. We consider this domain partitioned into two non-overlapping sub-domains Ω_1 and Ω_2 with an interface Γ as shown in Figure 6.1. We want to solve the *gravitational potential equation*, a particular case of the *Poisson equation*:

$$\begin{cases} -\Delta\Phi = 4\pi G\delta\rho = f & \text{on } \Omega \\ \mathcal{C}(\Phi) = b & \text{on } \partial\Omega \end{cases}$$

with $\delta\rho$ the density anomaly and G the gravitational constant ($6.67428 \times 10^{-11} \text{ m}^3 \cdot \text{kg}^{-1} \cdot \text{s}^{-2}$). Let (Φ_1^n, Φ_2^n) be an approximation to $(\Phi|_{\Omega_1}, \Phi|_{\Omega_2})$ at iteration n of the Schwarz algorithm,

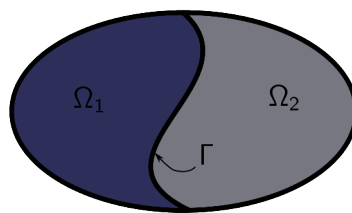


FIGURE 6.1: Non-overlapping domain decomposition, $\Omega = \Omega_1 \cup \Omega_2$

$(\Phi_1^{n+1}, \Phi_2^{n+1})$ expressed as:

$$\begin{aligned} -\Delta\Phi_1^{n+1} &= f, \text{ in } \Omega_1 \\ (\partial_\nu\Phi_1^{n+1} + \mathcal{A}^{(1)}\Phi_1^{n+1}) &= (\partial_\nu\Phi_2^n + \mathcal{A}^{(1)}\Phi_2^n), \text{ on } \Gamma \\ -\Delta\Phi_2^{n+1} &= f, \text{ in } \Omega_2 \\ (\partial_\nu\Phi_2^{n+1} - \mathcal{A}^{(2)}\Phi_2^{n+1}) &= (\partial_\nu\Phi_1^n - \mathcal{A}^{(2)}\Phi_1^n), \text{ on } \Gamma \end{aligned}$$

where ν is the unit normal vector along Γ . For the algorithm, to achieve the best performance, the operators $\mathcal{A}^{(1)}$ and $\mathcal{A}^{(2)}$ have to be suitably selected. An application of the Fourier trans-

form in $\Omega = \mathbb{R}^2$ for the homogeneous problem $f = 0$, leads to the expression of the Fourier convergence rate that depends on the quantities $\Lambda^{(1)}$ and $\Lambda^{(2)}$, which are the Fourier transforms of operators $\mathcal{A}^{(1)}$ and $\mathcal{A}^{(2)}$. Over the years, researches have proposed several ways to approximate these non-local operators with partial differential operators. One of such methods relies on looking for a partial differential operators enforcing a tangential derivative on the interface such as: $\mathcal{A}^{(s)} := p^{(s)} + q^{(s)} \partial_{\tau^2}^2$, where s denotes the sub-domain number, $p^{(s)}$, $q^{(s)}$ are two coefficients, and τ is the unit tangent vector. From references [101] [280] [168], one can clearly see that both the coefficients $p^{(s)}$, $q^{(s)}$ of the sub-domain s largely contribute to the speed-up the convergence when they are chosen optimally. The results shown in [165] [166] use a zero order Taylor expansion of the non-local operators to find $p^{(s)}$ and $q^{(s)}$. A minimization procedure is considered in [287] for the Helmholtz equation, in [167] for the Maxwell equation, in [288] for convection diffusion equations, and in [281] for heterogeneous media. The minimization function or cost function is defined as the maximum of the Fourier convergence rate for the considered frequency ranges. This approach consists in calculating the free parameters $p^{(s)}$ and $q^{(s)}$ through an optimization problem. The functions to minimize, $\tau \mapsto \tau(k, L, \mathbf{p})$, for zeroth and second order approximation for the one-sided (symmetric) formulation are respectively formulated as follows:

$$\max_{k_{min} < k < k_{max}} \frac{(|k| - p)^2}{(|k| + p)^2} e^{-2|k|L}; \quad \max_{k_{min} < k < k_{max}} \frac{(|k| - p - qk^2)^2}{(|k| + p + qk^2)^2} e^{-2|k|L} \quad (6.1)$$

where $p > 0$, $q > 0$, $L \geq 0$ is the size of the overlap and k is the spatial frequency of the Fourier transform. For the two-sided (non-symmetric) formulation:

$$\max_{k_{min} < k < k_{max}} \frac{(-|k| + p_1)(-|k| + p_2)}{(|k| + p_1)(|k| + p_2)} e^{-2|k|L}; \quad \max_{k_{min} < k < k_{max}} \frac{(|k| - p_1 - q_1k^2)(|k| - p_2 - q_2k^2)}{(|k| + p_1 + q_1k^2)(|k| + p_2 + q_2k^2)} e^{-2|k|L} \quad (6.2)$$

where $p_1 > 0$, $q_1 > 0$, $p_2 > 0$ and $q_2 > 0$ and L is the overlap size. Since the evaluation of the minimization function is quick enough and the dimension of the search space reasonable, a more effective and solid minimization method could be considered, as introduced in the next section.

6.3 Stochastic-based optimization

We consider the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) in order to obtain good results rapidly. The CMA-ES algorithm investigates the whole space of solutions and selects the absolute minima. This recent algorithm clearly demonstrates its robustness in [289] with good global search ability. Its advantage is that it does not need the calculation of the derivatives of the cost function. The main idea behind the CMA-ES strategy consists in finding the minimum of the cost function by refining a search distribution iteratively. The distribution is expressed as a general multivariate normal distribution $d(m, C)$. The initial distribution is taken to be the uniform one. The population size parameter reflects the trade-off between algorithm speed and its ability to find the global minimum. The algorithm is faster for smaller populations but we have a greater chance of finding a local minimum. When the size of the populations is large, the algorithm is able to avoid local minima but it needs a larger number of cost function evaluations. In this chapter, a population size of 26 has been enough to find the global minimum in a few seconds (<4 sec). Knowing the distribution, λ samples are randomly chosen in this distribution at each iteration and the evaluation of the cost function at

those points is performed in order to compute a new distribution. When the variance of the distribution is small enough, the center of the distribution m is considered to be the sought solution. After analyzing the cost function for the new population, the samples are sorted and we select the best sample μ . Then we recombine them to find a new distribution center by considering a weighted mean. The step size σ is an ingredient which serves to scale the standard deviation of the distribution. The variance of the search distribution is proportional to the square of the step size. The most complex task of the algorithm is to update the covariance matrix. While this could be done using only the current population, it would be unreliable, especially with a small population size; thus the population of the previous iteration should also be taken into account. According to the theory of CMA-ES, we have to fix a size of population that is a trade-off between speed and global search of the minimum. In our case, a population size of 25 has been enough to find the global minimum in a few seconds or less. The maximum number of iterations, depending on the population size, is computed as follows: $1000 \times \frac{(N + 5)^2}{\sqrt{p_o}}$, where N and p respectively present the problem dimension and the population size. In this chapter, with $N = 1$ and $p_o = 25$, the following stopping criteria for the CMA-ES algorithm are considered: a maximum number of iterations is equal to 7 200 and a residual threshold to 5×10^{-11} .

Figure 6.2(a) and Figure 6.2(b) collect the isolines of the convergence rate in the Fourier space of the Schwarz algorithm for the symmetric zeroth order and non-symmetric zeroth order arising from the CMA-ES strategy. Those of the symmetric and non-symmetric second order are respectively given in Figure 6.3(a) and Figure 6.3(b). Figure 6.2(b) and Figure 6.3(b) display the profiles corresponding to the non-symmetric zeroth ($p^{(1)}(i) = p(i)$ and $p^{(2)}(i) = p(N - i)$), $N = \text{card}(p)$) and second order ($p^{(1)}(i) = p(i), p^{(2)}(i) = p(N - i), q^{(1)}(i) = q(i)$ and $q^{(2)}(i) = q(N - i)$).

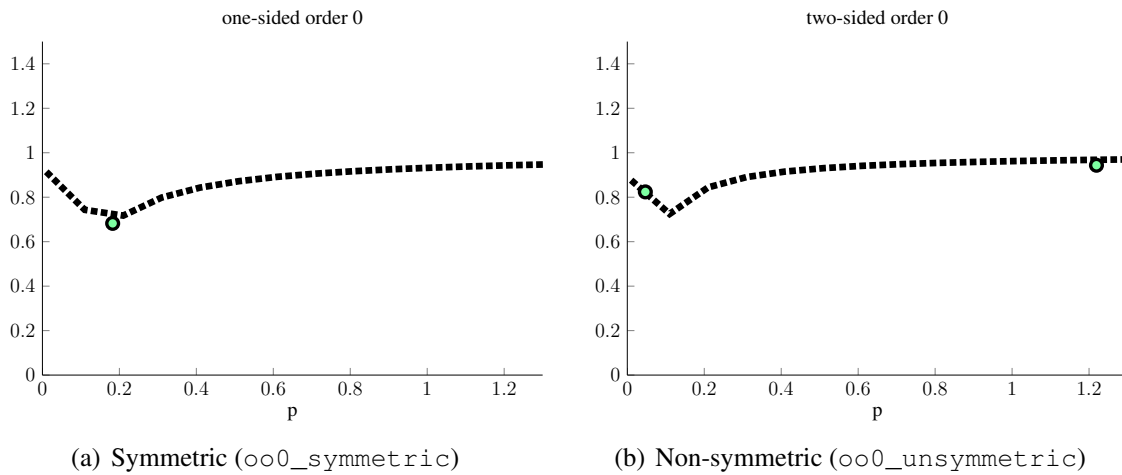


FIGURE 6.2: Convergence rate isolines of the Schwarz algorithm with zeroth order optimized transmission conditions

Figure 6.4(a) and Figure 6.4(b) give the convergence rate of the Schwarz algorithm in the Fourier space for the symmetric and non-symmetric zeroth order arising from the CMA-ES strategy. Those of the symmetric and non-symmetric second order are respectively given in Figure 6.5(a) and Figure 6.5(b). Table 6.1 reports the exact values of the coefficients obtained by the CMA-ES minimization procedure. These values are used to define the local operators previously mentioned in order to optimize our Schwarz domain decomposition method.

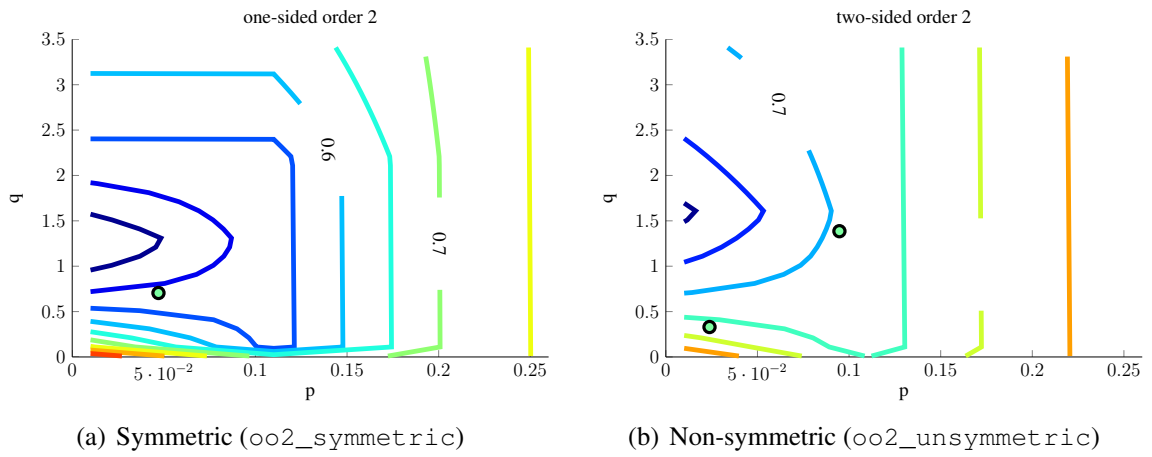


FIGURE 6.3: Convergence rate isolines of the Schwarz algorithm with second order optimized transmission conditions

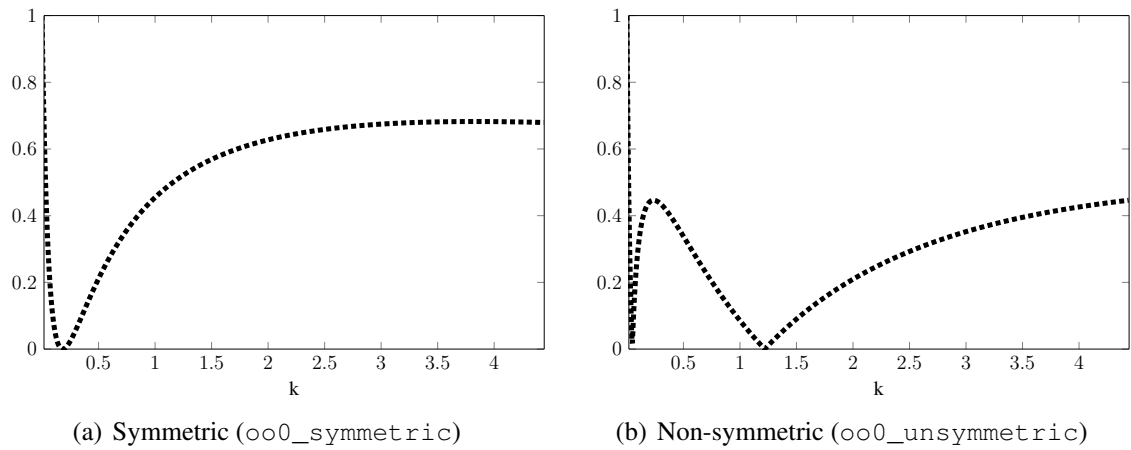


FIGURE 6.4: Fourier convergence rate of the Schwarz algorithm with zeroth order optimized transmission conditions

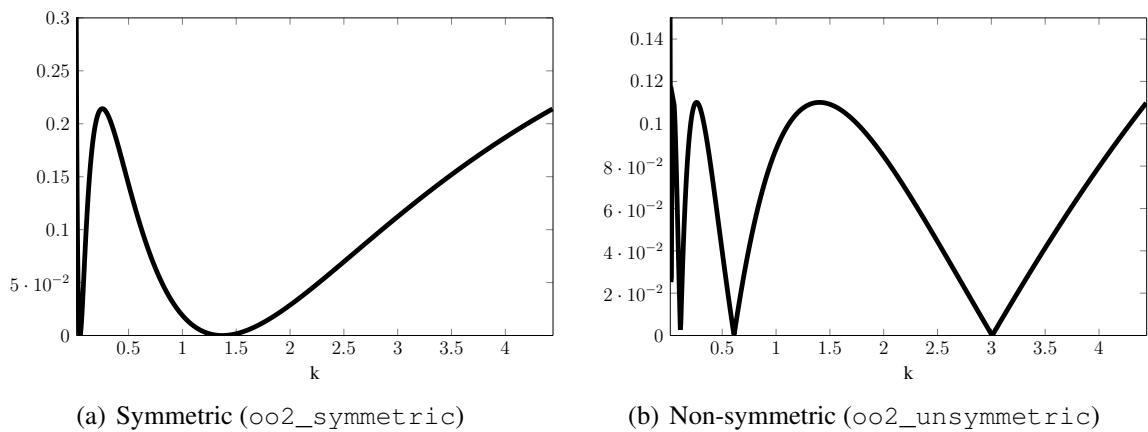


FIGURE 6.5: Fourier convergence rate of the Schwarz algorithm with second order optimized transmission conditions

	$p^{(1)}$	$q^{(1)}$	$p^{(2)}$	$q^{(2)}$	τ_{max}
oo0_symmetric	0.183	0.000	0.183	0.000	0.682
oo0_unsymmetric	1.219	0.000	0.047	0.000	0.446
oo2_symmetric	0.047	0.705	0.047	0.705	0.214
oo2_unsymmetric	0.108	0.321	0.023	1.579	0.110

TABLE 6.1: Optimized coefficients obtained from the *CMA-ES* algorithm

6.4 Numerical experiments

In this section, we summarize the results of the experiments performed to evaluate the speed-up of our optimized Schwarz implementation with the optimized coefficients obtained from the *CMA-ES* algorithm. The matrices are stored in CSR format.

All test cases are related to the study of the Chicxulub impact crater. The solution of the gravitational potential equation, $\nabla\Phi = -4\pi\delta\rho$ where $\delta\rho$ is the variation of an arbitrary density distribution, is evaluated with a finite element method. The Q_1 Lagrange finite elements are considered for the discretization. The numerical solution of the gravitational potential equation required for this study was done for a parallelepiped, a geometric domain of dimensions $250\text{ km} \times 250\text{ km} \times 15\text{ km}$. The finite element discretization of the domain (see Figure 3.26, Page 73) resulted in a total of 19,933,056 degrees of freedom. The mesh (see Figure 3.26, Page 73) is split in the x -direction. Each sub-domain is assigned to one single processor and each iteration of the optimized Schwarz method implying the solution of the equations inside each sub-domain is assigned to one single accelerator (GPU). Figure 6.6(a) and Figure 6.6(b) show respectively the perspective view of the main structural feature and the gravity measure of the Chicxulub impact crater, obtained from our simulation. The experiments

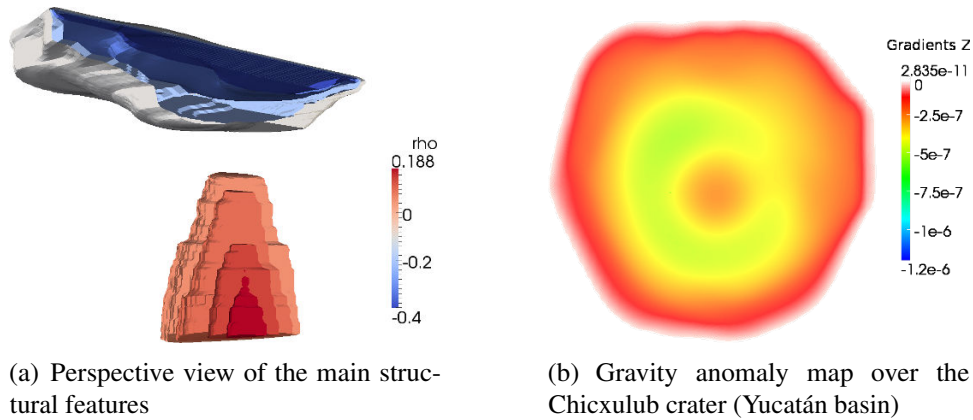


FIGURE 6.6: Perspective view of the main structural features and gravity anomaly map of the Chicxulub impact crater

have been performed on *Platform-5*, which consists of an hybrid CPU/GPU cluster, as per the schematic diagram (see Figure 2.21, Page 37). The cluster consists of 1,068 compute nodes and 24 nodes dedicated to IO and administration. Each node has 2 Intel Xeon 5570 Nehalem quad cores (2.93 GHz) and 24 GB of memory (3Go per cores). *Titane* includes 48 Tesla S1070 servers, with 4 processors of 4 GB memory each. Every server is attached to two compute nodes via the PCI-Express bus. The compute nodes are interconnected by a *Voltaire* network, based on the InfiniBand DDR technology.

For each iteration step, GPU is involved only in solving the sub-problems inside the sub-domains. The algebraic solver selected for this task is the preconditioned conjugate gradient (P-CG) with a diagonal preconditioner. We fix a residual tolerance threshold of $\varepsilon_{P-CG} = 10^{-10}$ for P-CG for the sub-domain solution, and a residual tolerance threshold of $\varepsilon_{Schwarz} = 10^{-6}$ for the Schwarz algorithm. The distribution of processors on our experiment workstation is computed as follows: number of processors = $2 \times$ number of nodes, where 2 corresponds to the number of GPUs per node available on our workstation. As a consequence, only two processors will share the bandwidth, which strongly enhance the communication, especially the inter-sub-domain communications. The number of computing units depends both on the size of the sub-domain problem and the gridification that use 256 threads per block and 8 threads per warp as mentioned in Chapter 2 and proved in Chapter 3. In the experiments, each sub-domain is handled by one classical processor (CPU) and one graphic processor (GPU). The data are double precision arithmetic. Table 6.2 collects the numerical results obtained when the preconditioned CG algorithm is used to solve the local sub-domain problem with $\varepsilon = 10^{-6}$, and the Jacobi method or CG for solving the problem at the interface. In Table 6.2, the results are organized as follows: the first column reports the number of sub-domains (*# subdom*), the second column gives the name of the interface solver (*itf. solver*), the next column gives the number of iterations of the Schwarz algorithm, the fourth and fifth columns, respectively, give the CPU and GPU times, in seconds (s), of the whole program, and lastly the speed-up (CPU/GPU) is reported in the last column.

# subdom	itf. solver	#iter. Schwarz	CPU time (s)	GPU time (s)	Speed-up
32	Jacobi	41	11,240	1,600	7.03
32	CG	24	7,560	1,031	7.33
64	Jacobi	45	5,360	860	6.23
64	CG	32	4,730	678	6.97
96	Jacobi	62	4,550	925	4.92
96	CG	42	3,200	645	4.96
128	Jacobi	92	6,535	960	6.81
128	CG	61	3,050	671	4.55

TABLE 6.2: Comparison of stochastic-based optimized Schwarz domain decomposition method on CPU and GPU

Increasing the number of sub-domains without increasing the number of degrees of freedom of the sub-meshes results in increasing the number of communications between the neighboring sub-domains, and this handicaps the GPU version. Each P-CG involves a copy from the host to the device for the right-hand side, and a copy back from the device to the host to update the sub-domain solutions. However, the GPU-based implementation remains clearly superior to the pure CPU implementation.

6.5 Conclusion

In this chapter, we have used an efficient double precision implementation of the iterative Krylov methods for GPU (see previous chapters), that leads to the effective implementation of the optimized Schwarz domain decomposition method. The main advantage of the classical Schwarz methods is the simplicity of their implementation.

In order to test the performance of the implemented solvers on realistic data we have solved the gravitational potential equation with input data collected from geological surveys

of the Chicxulub crater in Yucatán, southwest of Mexico. The approach taken to solve that equation is the optimized Schwarz domain decomposition method. Numerical optimization by the stochastic-based algorithm (CMA-ES) is used to find the optimal coefficients of the approximate transmission conditions. This algorithm is robust and has good global search properties. It is faster and more precise than a brute force approach.

In our experiments, the number of (mesh) sub-domains is ranged from 32 up to 128. Those experiments have clearly showed the benefits of applying the GPU-based implementation of domain decomposition method for large-size problems, and confirmed the robustness, performance and effectiveness of the Schwarz method with stochastic-based optimized transmission conditions. With a carefully tuned CPU/GPU code, and sufficient balance between the number of sub-domains and the size of the sub-problems, we were able to calculate the solution 7 times faster with respect to the calculation times for equivalent CPU-based only, reference implementations.

Conclusions and Perspectives

” *Man is still the most extraordinary computer of all.*

— **John F. Kennedy**
(President of USA)

7.1 Summary

We have presented different ways to efficiently solve large sparse linear systems. We have investigated the best way to effectively compute linear algebra operations in an heterogeneous multi-core-GPU environment in order to make solvers, especially iterative methods, more robust. In the first chapter, we have introduced the challenge of GPU computing in the field of computational science. We have given different programming techniques that offer access to the power of graphics cards. In the second chapter, we have demonstrated how to capitalize on the use of GPU computing and how we proceed to optimize CPU and GPU operations, data transfers and memory management. In order to ensure even better efficiency, we have proposed gridification strategies to auto-tune the grid upon the GPU architecture, which strongly impacts the performance of algorithms. I have also proposed a way to easily handle complex number arithmetics on GPU for all precisions, using advanced C++ template structures. The performance was benchmarked on modern GPUs such as Tesla K20c and GeForce GTX 570. Numerical experiments performed on a set of large-size sparse matrices arising from diverse engineering and scientific problems, have clearly shown the value of using GPU technology to solve large sparse systems of linear equations, and its robustness and accuracy compared to existing libraries such as Cusp. The implementations of all the techniques we proposed are compiled in the Alinea library, effective for computing advanced linear algebra and solving linear systems on both CPU and GPU.

Knowing how to better access the power of graphics cards in terms of computational time for solving large sparse linear systems, we have attempted to answer another important question related to energy issues, *i.e.*, “How much energy is consumed?” by the powerful GPU. To answer this question, we have established an experimental protocol to accurately the energy consumption of a GPU for fundamental linear algebra operations. This methodology could suggest a “new vision of high-performance computing” and answer some of the questions outlined in green computing when using GPUs.

The remainder of the manuscript was devoted to synchronous and asynchronous iterative algorithms for solving linear systems in the context of a multi-core-GPU system. We have implemented and analyzed these algorithms using iterative methods based on sub-structuring techniques. The sub-structuring method is a decomposition method without overlap that allows to reduce a global problem to a problem on the interface between sub-domains. In the fourth chapter, we have presented the proof and implementations of the synchronous and asynchronous sub-structuring methods. I have explored them in the context of linear systems only. The demonstration is based on fixed point iterative algorithms. However, we

have also analyzed the conjugate gradient sub-structuring method with synchronous iterations. Numerical results clearly show that asynchronous versions of the algorithm are generally faster than synchronous versions, in particular with a large number of processes. The asynchronous algorithm scales much more with the number of processors compared to the synchronous algorithm. The sub-structuring algorithms are more efficient than classical splitting algorithms. They have also showed that the integration of GPUs helps algorithms accelerate their resolution. However, when the number of sub-domains increases, the GPU code's performance decreases. To better understand the behavior of both synchronous and asynchronous iterations compared to sequential iterations, we have proposed some theorems concerning the speed-up of fully parallelizable algorithms such as the Jacobi method. We have also studied the behavior of parallel algorithms in terms of fault-tolerance and when an iteration is penalized. The scalability and performance of these methods are tested on diverse scientific problems, together with numerous numerical experiments which clearly illustrate the robustness, competitiveness and efficiency of parallel algorithms, much more so when combined with GPU computing.

Finally, in the last chapter, we have modified the non-overlapping Schwarz method to accelerate its use of GPUs. The implementation is based on the acceleration of the local solutions of the linear sub-systems associated with each sub-domain using GPUs. To ensure good performance, optimized conditions obtained by a stochastic technique based on the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) are used. In order to test the performance of the implemented solvers on real data, we have solved the gravitational potential equation with input data collected from geological surveys of the Chicxulub crater in Yucatán, southwest of Mexico. Those experiments have clearly showed the benefits of applying a GPU-based implementation of the domain decomposition method for large-size problems, and confirmed the robustness, performance and effectiveness of the Schwarz method with stochastic-based optimized transmission conditions. With a carefully tuned CPU/GPU code, and sufficient balance between the number of sub-domains and the size of the sub-problems, we were able to calculate the solution 7 times faster, with respect to the calculation times for an equivalent, CPU-based only, reference implementation.

7.2 Perspectives and future works

In this thesis, the experimental tests were carried out on a workstation or on a small cluster. It would thus be interesting to test the performance of our implementations on a big cluster, *e.g.*, an exascale machine. Moreover, most implementations of our iterative algorithms were evaluated on local hybrid clusters. In the context of a large number of processors, the parallel algorithms will perform a large number of communications. Let us recall that the performance strongly depends on the hardware characteristics and also on the network configuration (bandwidth, etc.). The comparative study with the Cusp library is old, whereas Cusp was clearly improved. In the hybrid CPU/GPU synchronous and asynchronous algorithms, at each iteration we perform copies between CPU and GPU, which degrade the overall execution time. The optimization of this task will considerably improve the performance of the algorithm. Concerning the acceleration of the non-overlapping Schwarz method, it would be worthwhile to also solve the interface problem on GPU. The analysis of the theoretical speed-up of synchronous and asynchronous algorithms presents some limits concerning the distribution of data, *i.e.*, the sub-domains and the interfaces. It would be interesting, to analyze this

distribution carefully in order to better evaluate the cost of data transfers and computation. In the analysis of the behavior of chaotic iterations, we have assumed that only one processor can be in idle state or breakdown. Future investigations may focus on analyzing the behavior with a more complex chaotic scheme and then propose clear mathematical models. The future investigations must also concentrate on the study of the parallel algorithms with a real fault-tolerance, allowing the breakdown of certain physical processors or even the reboot of a machine.

Bibliography

- [1] Michael J. Flynn. “Some Computer Organizations and Their Effectiveness”. In: *IEEE Transactions on Computers* C-21.9 (Sept. 1972), pp. 948–960 (cit. on p. 2).
- [2] Nvidia Corporation. *CUDA Toolkit Reference MANUAL*. 2011 (cit. on pp. 3, 11).
- [3] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*. Dec. 8, 2008 (cit. on pp. 3, 12).
- [4] C.J. Thompson, Sahngyun Hahn, and M. Oskin. “Using modern graphics architectures for general-purpose computing: a framework and analysis”. In: Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 306–317 (cit. on pp. 4, 11, 12).
- [5] Jeff Bolz et al. “The GPU as Numerical Simulation Engine”. In: *Caltech Multi-Res Modeling Group* (2003) (cit. on pp. 4, 11, 12).
- [6] Jens Krüger and Rüdiger Westermann. “Linear algebra operators for GPU implementation of numerical algorithms”. In: *ACM Transactions on Graphics* 22.3 (July 1, 2003), pp. 908–916 (cit. on pp. 4, 11, 12).
- [7] Sasko Ristov et al. “Performance Impact of Reconfigurable L1 Cache on GPU Devices.” In: Federated Conference on Computer Science and Information Systems (FEDCSIS 2013). Ed. by Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki. Krakow, Poland: IEEE Computer Society, 2013, pp. 507–510 (cit. on pp. 4, 11).
- [8] top500.org. *Top500 List*. 2015. URL: <http://www.top500.org> (cit. on pp. 4, 25, 105).
- [9] green500.org. *Green500 List*. 2015. URL: <http://www.green500.org> (cit. on pp. 4, 25).
- [10] S. Sharma, Chung-Hsing Hsu, and Wu-chun Feng. “Making a case for a Green500 list”. In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006), Workshop on High Performance - Power Aware Computing, Rhodes Island, 25-29 April 2006*. IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006), Workshop on High Performance - Power Aware Computing. IEEE, 2006, 8 pp. (Cit. on pp. 4, 25).
- [11] Balaji Subramaniam et al. “Trends in energy-efficient computing: A perspective from the Green500”. In: *4th International Green Computing Conference, Arlington, VA, 27-29 June 2013*. International Green Computing Conference. Arlington, VA, USA: IEEE, June 2013, pp. 1–8 (cit. on pp. 4, 25, 26).
- [12] Wenjie Liu et al. “A Waterfall Model to Achieve Energy Efficient Tasks Mapping for Large Scale GPU Clusters”. In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), Shanghai, China, 16-20 May 2011*. 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW). Shanghai, China: IEEE, May 2011, pp. 82–92 (cit. on pp. 4, 26).
- [13] Hongpeng Huo et al. “An energy efficient task scheduling scheme for heterogeneous GPU-enhanced clusters”. In: *2012 International Conference on Systems and Informatics (ICSAI), Yantai, May 19-20, 2012*. International Conference on Systems and Informatics (ICSAI). Yantai, China: IEEE, May 2012, pp. 623–627 (cit. on pp. 4, 26).

- [14] Juan M. Cebri'n, Gines D. Guerrero, and Jose M. Garcia. "Energy Efficiency Analysis of GPUs". In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW), Shanghai, China, 21-25 May 2012*. International Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW). Shanghai, China: IEEE, May 2012, pp. 1014–1022 (cit. on pp. 4, 26).
- [15] Hermann Amandus Schwarz. "Ueber einen Grenzübergang durch alternirendes Verfahren". In: *Verfahren Vierteljahrsschrift der Naturforschenden Gesellschaft, Zürich* 15 (1870), pp. 272–286 (cit. on pp. 5, 202).
- [16] Patrick Le Tallec. "Domain decomposition methods in computational mechanics". In: *Computational Mechanics Advances*. Ed. by J. Tinsley Oden. Vol. 1 (2). North-Holland, 1994, pp. 121–220 (cit. on pp. 5, 142, 165, 202).
- [17] Barry F. Smith, Petter E. Bjorstad, and William D. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. New York, NY, USA: Cambridge University Press, 1996 (cit. on pp. 5, 93, 142, 202).
- [18] A. Quarteroni and A. Valli. *Domain Decomposition Methods for Partial Differential Equations*. Oxford, UK: Oxford University Press, 1999 (cit. on pp. 5, 52, 86, 93, 142, 165, 202).
- [19] Andrea Toselli and Olof Widlund. *Domain Decomposition Methods - Algorithms and Theory*. Vol. 34. Springer Series in Computational Mathematics. Berlin/Heidelberg: Springer-Verlag, 2005 (cit. on pp. 5, 142, 165, 202).
- [20] OpenGPU. *OpenGPU*. 2010–. URL: <http://opengpu.net> (cit. on p. 5).
- [21] CRESTA. *CRESTA Project*. 2012–2014. URL: <http://www.cresta-project.eu> (cit. on pp. 5, 6).
- [22] F Magoulès and Cedric Venet. *Asynchronous parallel algorithms for for petaflop and exaflop computation*. Boca Raton, Fla.; London: Chapman & Hall/CRC, 2012 (cit. on pp. 8, 201).
- [23] J.D. Owens et al. "GPU Computing". In: *Proceedings of the IEEE 96.5* (May 2008), pp. 879–899 (cit. on p. 11).
- [24] Jeremy Meredith et al. *The GAIA Project: Evaluation of GPU-Based Programming Environments for Knowledge Discovery*. Lawrence Livermore National Labs, 2004 (cit. on pp. 11, 12).
- [25] John D. Owens et al. "A Survey of General-Purpose Computation on Graphics Hardware". In: *EUROGRAPHICS* (2005) (cit. on p. 11).
- [26] Nvidia Corporation. *CUDA Programming Guide*. 2011 (cit. on p. 13).
- [27] Ali Bakhoda et al. "Analyzing CUDA workloads using a detailed GPU simulator". In: *IEEE International Symposium on Performance Analysis of Systems and Software*. Boston, MA, USA: IEEE, Apr. 2009, pp. 163–174 (cit. on p. 13).
- [28] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. pub-IEEE-STD:adr: pub-IEEE-STD, Aug. 2008 (cit. on p. 13).
- [29] LM Itu et al. "Comparison of single and double floating point precision performance for Tesla architecture GPUs". In: *Bulletin of the Transilvania University of Br ov Series I: Engineering Sciences* 4.53 (2011) (cit. on p. 13).
- [30] Nathan Whitehead and Alex Fit-Florea. "Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs". In: *rn (A+ B)* 21 (2011), pp. 1–1874919424 (cit. on p. 13).
- [31] Nathan Bell and Michael Garland. *Efficient Sparse Matrix-Vector Multiplication on CUDA*. Nvidia Technical Report NVR-2008-004. Nvidia Corporation, 2008 (cit. on pp. 13, 42, 44, 65, 66).
- [32] Nathan Bell and Michael Garland. "Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors". In: *Proceedings of the Conference on High Performance*

- Computing Networking, Storage and Analysis, Portland, OR, 14-20 Nov. 2009*. Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. SC '09. Portland, OR, USA: ACM, 2009, 18:1–18:11 (cit. on pp. 13, 44, 65).
- [33] G. A. Gravvanis, C. K. Filelis-Papadopoulos, and K. M. Giannoutakis. “Solving finite difference linear systems on GPUs: CUDA based Parallel Explicit Preconditioned Biconjugate Conjugate Gradient type Methods”. In: *The Journal of Supercomputing* 61.3 (Sept. 2012), pp. 590–604 (cit. on p. 13).
- [34] Dana A. Jacobsen, Julien C. Thibault, and Inanc Senocak. “An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters”. In: *48th AIAA Aerospace Sciences Meeting and Exhibit*. Vol. 16. 2010 (cit. on p. 13).
- [35] Dawei Mu, Po Chen, and Liqiang Wang. “Accelerating the discontinuous Galerkin method for seismic wave propagation simulations using multiple GPUs with CUDA and MPI”. In: *Earthquake Science* 26.6 (Dec. 2013), pp. 377–393 (cit. on p. 13).
- [36] John E. Stone, David Gohara, and Guochun Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *Computing in Science & Engineering* 12.3 (May 2010), pp. 66–73 (cit. on p. 14).
- [37] Philippe Tillet, Karl Rupp, and Siegfried Selberherr. “An automatic OpenCL compute kernel generator for basic linear algebra operations”. In: *Proceedings of the 2012 Symposium on High Performance Computing*. Society for Computer Simulation International, 2012, p. 4 (cit. on p. 14).
- [38] Jambhlekhar Pushkar Arun, Manoj Mishra, and Sheshasayee V. Subramaniam. “Parallel Implementation of MOPSO on GPU Using OpenCL and CUDA”. In: *Proceedings of the 2011 18th International Conference on High Performance Computing*. HIPC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1–10 (cit. on p. 14).
- [39] Peng Du et al. “From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming”. In: *Parallel Computing* 38.8 (Aug. 2012), pp. 391–407 (cit. on p. 14).
- [40] Sebastien Noury, Samuel Boivin, and Olivier Le Maître. *A fast Poisson solver for OpenCL using multigrid methods*. Ed. by W. Engel and Ed. A K Peters. W. Engel, Ed. A K Peters. 2011 (cit. on p. 14).
- [41] PCI-SIG PCI-SIG. *PCI Express Architecture Frequently Asked Questions*. Available on line at: http://kavi.pcisig.com/news_room/faqs/PCI_Express_FAQ.pdf (accessed on October 6, 2015). 2005 (cit. on p. 16).
- [42] PCI-SIG PCI-SIG. *PCI Express 3.0 Frequently Asked Questions PCI-SIG*. Available on line at: http://www.pcisig.com/news_room/faqs/pcie3.0_faq/PCI-SIG_PCIE_3_0_FAQ_Final_07102012.pdf/ (accessed on October 6, 2015). 2012 (cit. on p. 16).
- [43] Ping Guo and Liqiang Wang. “Auto-Tuning CUDA Parameters for Sparse Matrix-Vector Multiplication on GPUs”. In: *2010 International Conference on Computational and Information Sciences (ICCIS), Chengdu, China, 17-19 Dec. 2010*. International Conference on Computational and Information Sciences (ICCIS). IEEE, Dec. 2010, pp. 1154–1157 (cit. on pp. 17, 63).
- [44] Andrew Davidson, Yao Zhang, and John D. Owens. “An Auto-tuned Method for Solving Large Tridiagonal Systems on the GPU”. In: *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, Anchorage, AK, 16-20 May 2011*. International Parallel & Distributed Processing Symposium. IPDPS '11. Anchorage, AK, USA: IEEE Computer Society, 2011, pp. 956–965 (cit. on pp. 17, 63).

- [45] Abal-Kassim Cheik Ahamed and Frédéric Magoulès. “Fast Sparse Matrix-Vector Multiplication on Graphics Processing Unit for Finite Element Analysis”. In: *14th IEEE International Conference on High Performance Computing and Communications (HPCC 2012), Liverpool, UK, June 25-27, 2012*. International Conference on High Performance Computing and Communications (HPCC). Liverpool, UK: IEEE, June 2012, pp. 1307–1314 (cit. on pp. 17, 26, 63).
- [46] E. Lezar and D.B. Davidson. “GPU-based LU decomposition for large method of moments problems”. In: *Electronics Letters* 46.17 (2010), p. 1194 (cit. on p. 22).
- [47] A. Dziekonski, A. Lamecki, and M. Mrozowski. “Tuning a Hybrid GPU-CPU V-Cycle Multilevel Preconditioner for Solving Large Real and Complex Systems of FEM Equations”. In: *IEEE Antennas and Wireless Propagation Letters* 10 (2011), pp. 619–622 (cit. on p. 22).
- [48] Kenneth A Hawick and Daniel P Playne. “Numerical Simulation of the Complex Ginzburg-Landau Equation on GPUs with CUDA”. In: *Proc. IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN)*. 2011, pp. 39–45 (cit. on p. 22).
- [49] Tokyo Tech. *TSUBAME-KFC*. 2015. URL: <http://www.titech.ac.jp/english/news/2013/024456.html> (cit. on p. 25).
- [50] Dominik Göttsche. “Fast and accurate finite-element multigrid solvers for PDE simulations on GPU clusters”. TU Dortmund, 2010 (cit. on p. 26).
- [51] Cris Cecka, Adrian J. Lew, and E. Darve. “Assembly of finite element methods on graphics processors”. In: *International Journal for Numerical Methods in Engineering* 85.5 (Feb. 4, 2011), pp. 640–669 (cit. on p. 26).
- [52] Dan Zou and Yong Dou. “Implementation of parallel sparse Cholesky factorization on GPU”. In: *2012 2nd International Conference on Computer Science and Network Technology (ICCSNT)*. IEEE, Dec. 2012, pp. 2228–2232 (cit. on p. 26).
- [53] Ling Ren et al. “Sparse LU Factorization for Parallel Circuit Simulation on GPU”. In: *Proceedings of the 49th Annual Design Automation Conference*. DAC ’12. New York, NY, USA: ACM, 2012, pp. 1125–1130 (cit. on pp. 26, 42, 46).
- [54] M. Creel and M. Zubair. “High Performance Implementation of an Econometrics and Financial Application on GPUs”. In: *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion*: Nov. 2012, pp. 1147–1153 (cit. on p. 26).
- [55] L. Susan Blackford et al. “An updated set of basic linear algebra subprograms (BLAS)”. In: *ACM Transactions on Mathematical Software* 28.2 (June 1, 2002), pp. 135–151 (cit. on p. 39).
- [56] Kazushige Goto and Robert van de Geijn. “High-performance implementation of the level-3 BLAS”. In: *ACM Transactions on Mathematical Software* 35.1 (2008), 4:1–4:14 (cit. on p. 39).
- [57] Jack J. Dongarra, Piotr Luszczek, and Antoine Petit. “The LINPACK Benchmark: past, present and future”. In: *Concurrency and Computation: Practice and Experience* 15.9 (2003), pp. 803–820 (cit. on pp. 39, 52).
- [58] Na Li et al. *ITSOL*. Nov. 2010 (cit. on pp. 39, 52, 53).
- [59] Wieb Bosma, John Cannon, and Catherine Playoust. “The Magma Algebra System I: The User Language”. In: *Journal of Symbolic Computation* 24.3-4 (Sept. 1997), pp. 235–265 (cit. on p. 39).
- [60] nVidia nVidia. *CUBLAS Library User Guide*. nVidia, Oct. 2012 (cit. on pp. 39, 53).
- [61] nVidia. *CUDA CUSPARSE Library*. NVIDIA, Aug. 2010 (cit. on pp. 39, 53).
- [62] Nathan Bell and Michael Garland. *Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations*. 2012. URL: <http://cusp-library.googlecode.com> (cit. on pp. 39, 52, 53).

- [63] Jeff Bolz et al. “Sparse matrix solvers on the GPU: conjugate gradients and multigrid”. In: *ACM Transactions on Graphics* 22.3 (July 1, 2003), p. 917 (cit. on p. 42).
- [64] Shiming Xu, Hai Xiang Lin, and Wei Xue. “Sparse Matrix-Vector Multiplication Optimizations based on Matrix Bandwidth Reduction using NVIDIA CUDA”. In: IEEE, Aug. 2010, pp. 609–614 (cit. on p. 42).
- [65] Jacques M. Bahi, Raphaël Couturier, and Lilia Ziane Khodja. “Parallel GMRES implementation for solving sparse linear systems on GPU clusters”. In: San Diego, CA, USA: Society for Computer Simulation International, 2011, pp. 12–19 (cit. on p. 42).
- [66] Moritz Kreutzer et al. “Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation”. In: *CoRR* abs/1112.5588 (2011) (cit. on p. 42).
- [67] Brad Suchoski et al. “Adapting Sparse Triangular Solution to GPUs”. In: 2012 41st International Conference on Parallel Processing Workshops (ICPPW). IEEE, Sept. 2012, pp. 140–148 (cit. on p. 42).
- [68] Rashid Hassani et al. “Analysis of Sparse Matrix-Vector Multiplication Using Iterative Method in CUDA”. In: 2013 IEEE Eighth International Conference on Networking, Architecture and Storage (NAS). IEEE, July 2013, pp. 262–266 (cit. on p. 42).
- [69] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Second. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, Jan. 2003 (cit. on pp. 42, 50, 63, 83, 84, 86, 130).
- [70] David R. Kincaid, Thomas C. Oppe, and David M. Young. *ITPACKV 2D User’s Guide*. Report CNA-232. University of Texas at Austin Department of Mathematics Austin, TX USA, May 1989 (cit. on p. 43).
- [71] Iain S Duff, Albert M Erisman, and John K Reid. *Direct Methods for Sparse Matrices*. New York, NY, USA: Oxford University Press, Inc., 1986 (cit. on p. 45).
- [72] Timothy A. Davis and Iain S. Duff. “An Unsymmetric-Pattern Multifrontal Method for Sparse LU Factorization”. In: *SIAM Journal on Matrix Analysis and Applications* 18.1 (Jan. 1997), pp. 140–158 (cit. on p. 45).
- [73] P. R. Amestoy, I. S. Duff, and J.-Y. L’Excellent. “MUMPS MULTifrontal Massively Parallel Solver Version 2.0”. In: (1998) (cit. on p. 46).
- [74] James W. Demmel et al. “A Supernodal Approach to Sparse Partial Pivoting”. In: *SIAM Journal on Matrix Analysis and Applications* 20.3 (Jan. 1999), pp. 720–755 (cit. on p. 46).
- [75] Timothy A. Davis and Iain S. Duff. “A combined unifrontal/multifrontal method for unsymmetric sparse matrices”. In: *ACM Transactions on Mathematical Software* 25.1 (Mar. 1, 1999), pp. 1–20 (cit. on p. 46).
- [76] V. V S Prakash and R. Mittra. “Multi-frontal preconditioners for iterative solvers”. In: IEEE Antennas and Propagation Society International Symposium, 2001. Vol. 3. July 2001, 12–15 vol.3 (cit. on p. 50).
- [77] L Giraud and S Gratton. “Accelerating Krylov solvers with low rank correction”. In: *ECCOMAS CFD 2006: Proceedings of the European Conference on Computational Fluid Dynamics, Egmond aan Zee, The Netherlands, September 5-8, 2006*. European Conference on Computational Fluid Dynamics. Delft University of Technology; European Community on Computational Methods in Applied Sciences (ECCOMAS), 2006, pp. 1–18 (cit. on p. 50).
- [78] David S Watkins. *The matrix eigenvalue problem: GR and Krylov subspace methods*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2007 (cit. on p. 50).
- [79] Hartwig Anzt, Vincent Heuveline, and Björn Rucker. “Mixed Precision Iterative Refinement Methods for Linear Systems: Convergence Analysis Based on Krylov Subspace Methods.” In: International Conference PARA. Ed. by Kristján Jónasson. Vol. 7134. Lecture NOTES in

- Computer Science. Reykjavík, Iceland: Springer Berlin Heidelberg, June 6–9, 2010, pp. 237–247 (cit. on pp. 50, 80).
- [80] Abhijeet Gaikwad and Ioane Muni Toke. “Parallel Iterative Linear Solvers on GPU: A Financial Engineering Case”. In: 2010 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). IEEE, Feb. 2010, pp. 607–614 (cit. on p. 50).
- [81] H. Knibbe, C.W. Oosterlee, and C. Vuik. “GPU implementation of a Helmholtz Krylov solver preconditioned by a shifted Laplace multigrid method”. In: *Journal of Computational and Applied Mathematics* 236.3 (Sept. 2011), pp. 281–293 (cit. on p. 50).
- [82] Raphaël Couturier and Stéphane Domas. “Sparse systems solving on GPUs with GMRES”. In: *The Journal of Supercomputing* 59.3 (Mar. 2012), pp. 1504–1516 (cit. on p. 50).
- [83] Emmanuel Agullo et al. “Task-based Conjugate-Gradient for multi-GPUs platforms”. In: (RR-8192 2012), p. 28 (cit. on p. 50).
- [84] Ruipeng Li and Yousef Saad. “GPU-accelerated preconditioned iterative linear solvers”. In: *The Journal of Supercomputing, Springer US* 63.2 (Feb. 2013), pp. 443–466 (cit. on pp. 50, 80).
- [85] Youcef Saad and Martin H. Schultz. “GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems”. In: *SIAM Journal on Scientific and Statistical Computing* 7.3 (July 1986), pp. 856–869 (cit. on pp. 51, 82).
- [86] W. E. Arnoldi. “The principle of minimized iterations in the solution of the matrix eigenvalue problem”. In: *Q. Appl. Math* 9.17 (1951), pp. 17–29 (cit. on p. 51).
- [87] Achiya Dax. “A modified Gram-Schmidt algorithm with iterative orthogonalization and column pivoting”. In: *Linear Algebra and its Applications* 310.1-3 (May 2000), pp. 25–42 (cit. on p. 51).
- [88] Christopher C. Paige, Miroslav Rozložnik, and Zdenek Strakos. “MODIFIED GRAM-SCHMIDT (MGS), LEAST SQUARES, AND BACKWARD STABILITY OF MGS-GMRES”. In: (2006) (cit. on p. 51).
- [89] Howard C Elman. “Iterative methods for large, sparse, nonsymmetric systems of linear equations”. Yale University New Haven, Conn, 1982 (cit. on p. 51).
- [90] Andreas Stathopoulos, Yousef Saad, and Charlotte F. Fischer. “Robust preconditioning of large, sparse, symmetric eigenvalue problems”. In: *Journal of Computational and Applied Mathematics* 64.3 (Dec. 1995), pp. 197–215 (cit. on p. 51).
- [91] R.C Mittal and A.H Al-Kurdi. “An efficient method for constructing an ILU preconditioner for solving large sparse nonsymmetric linear systems by the GMRES method”. In: *Computers & Mathematics with Applications* 45.10-11 (May 2003), pp. 1757–1772 (cit. on p. 51).
- [92] Daniel Osei-Kuffuor. *Robust preconditioning for indefinite and ill-conditioned sparse linear systems*. University of Minnesota, 2011 (cit. on p. 51).
- [93] R.E Plessix and W.A Mulder. “Separation-of-variables as a preconditioner for an iterative Helmholtz solver”. In: *Applied Numerical Mathematics* 44.3 (Feb. 2003), pp. 385–400 (cit. on p. 52).
- [94] Martin J. Gander and FrÉdÉric Nataf. “AILU FOR HELMHOLTZ PROBLEMS: A NEW PRECONDITIONER BASED ON THE ANALYTIC PARABOLIC FACTORIZATION”. In: *Journal of Computational Acoustics* 09.04 (Dec. 2001), pp. 1499–1506 (cit. on p. 52).
- [95] José-Ignacio Aliaga et al. “Parallelization of multilevel ILU preconditioners on distributed-memory multiprocessors”. In: vol. 7133. *Lecture NOTES in Computer Science*. Springer Berlin Heidelberg, 2010, pp. 162–172 (cit. on pp. 52, 86, 87).

- [96] Carlo Janna, Massimiliano Ferronato, and Giuseppe Gambolati. “A Block FSAI-ILU Parallel Preconditioner for Symmetric Positive Definite Linear Systems.” In: *SIAM J. Scientific Computing* 32.5 (2010), pp. 2468–2484 (cit. on pp. 52, 86).
- [97] Andrea Toselli and Olof Widlund. “Domain Decomposition Methods”. In: *Computational Mathematics* 34 (2004) (cit. on pp. 52, 86, 93).
- [98] Frédéric Magoulès and François-Xavier Roux. “Lagrangian formulation of domain decomposition methods: A unified theory”. In: *Applied Mathematical Modelling* 30.7 (July 2006), pp. 593–615 (cit. on pp. 52, 86, 93, 202).
- [99] Yvon Maday and Frédéric Magoulès. “Absorbing interface conditions for domain decomposition methods: A general presentation”. In: *Computer Methods in Applied Mechanics and Engineering* 195.29-32 (June 2006), pp. 3880–3900 (cit. on pp. 52, 86, 93, 202).
- [100] Martin J. Gander. “Optimized Schwarz Methods”. In: *SIAM Journal on Numerical Analysis* 44.2 (Jan. 2006), pp. 699–731 (cit. on pp. 52, 93, 165, 203).
- [101] F Magoulès, P Iványi, and B.H.V Topping. “Convergence analysis of Schwarz methods without overlap for the Helmholtz equation”. In: *Computers & Structures* 82.22 (Sept. 2004), pp. 1835–1847 (cit. on pp. 52, 86, 202, 204).
- [102] F. Magoulès, P. Iványi, and B.H.V. Topping. “Non-overlapping Schwarz methods with optimized transmission conditions for the Helmholtz equation”. In: *Computer Methods in Applied Mechanics and Engineering* 193.45-47 (Nov. 2004), pp. 4797–4818 (cit. on pp. 52, 86).
- [103] F. Magoulès, F.-X. Roux, and L. Series. “Algebraic approach to absorbing boundary conditions for the Helmholtz equation”. In: *International Journal of Computer Mathematics* 84.2 (Feb. 2007), pp. 231–240 (cit. on pp. 52, 86, 93, 202, 203).
- [104] Y. Maday and F. Magoulès. “Optimal convergence properties of the FETI domain decomposition method”. In: *International Journal for Numerical Methods in Fluids* 55.1 (Sept. 10, 2007), pp. 1–14 (cit. on pp. 52, 86, 93).
- [105] F.-X. Roux et al. “Approximation of optimal interface boundary conditions for two-Lagrange multiplier FETI method”. In: *Proceedings of the 15th International Conference on Domain Decomposition Methods, Berlin, Germany, July 21-15, 2003*. Ed. by R. Kornhuber et al. Springer-Verlag, Haidelberg, 2005, pp. 283–290 (cit. on pp. 52, 86, 93, 202, 203).
- [106] J. Dongarra et al. “A sparse matrix library in C++ for high performance architectures”. In: (1994) (cit. on p. 52).
- [107] R Pozo, K Remington, and A Lumsdaine. “SparseLib++ Sparse Matrix Class Library”. In: *User’s Guide* (1996) (cit. on p. 52).
- [108] Satish Balay et al. *PETSc Users Manual*. ANL-95/11 - Revision 3.4. Argonne, IL, USA: Argonne National Laboratory, 2013 (cit. on p. 52).
- [109] R. J. Hanson, F. T. Krogh, and C. L. Lawson. “A Proposal for Standard Linear Algebra Subprograms”. In: *ACM Signum Newsletter* 8 (1973), p. 146 (cit. on p. 52).
- [110] C. L. Lawson et al. “Algorithm 539: Basic Linear Algebra Subprograms for Fortran Usage [F1]”. In: *ACM Transactions on Mathematical Software* 5.3 (Sept. 1979), pp. 324–325 (cit. on p. 52).
- [111] C. L. Lawson et al. “Basic Linear Algebra Subprograms for Fortran Usage”. In: *ACM Trans. Math. Softw.* 5.3 (Sept. 1979), pp. 308–323 (cit. on p. 52).
- [112] Chris Phillips. “The performance of the BLAS and LAPACK on a shared memory scalar multiprocessor”. In: *Parallel Computing* 17.6-7 (Sept. 1991), pp. 751–761 (cit. on p. 52).
- [113] O. Coulaud, P. Fortin, and J. Roman. “High performance BLAS formulation of the multipole-to-local operator in the fast multipole method”. In: *Journal of Computational Physics* 227.3 (Jan. 2008), pp. 1836–1862 (cit. on p. 52).

- [114] O. Coulaud, P. Fortin, and J. Roman. “High performance BLAS formulation of the adaptive Fast Multipole Method”. In: *Mathematical and Computer Modelling* 51.3-4 (Feb. 2010), pp. 177–188 (cit. on p. 52).
- [115] Edoardo Di Napoli et al. “Towards an efficient use of the BLAS library for multilinear tensor contractions”. In: *Applied Mathematics and Computation* 235 (May 2014), pp. 454–468 (cit. on p. 52).
- [116] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. pub-IEEE-STD:adr: pub-IEEE-STD, Aug. 1985 (cit. on p. 52).
- [117] Jack J. Dongarra et al. “An extended set of FORTRAN basic linear algebra subprograms”. In: *ACM Transactions on Mathematical Software* 14.1 (Mar. 1, 1988), pp. 1–17 (cit. on p. 52).
- [118] J. J. Dongarra et al. “A set of level 3 basic linear algebra subprograms”. In: *ACM Transactions on Mathematical Software* 16.1 (Mar. 1, 1990), pp. 1–17 (cit. on p. 53).
- [119] The Numerical Algorithms Group Ltd. *The NAG C Library, Mark 1*. Oxford, 1990 (cit. on p. 53).
- [120] Abal Kassim Cheik Ahamed and Frederic Magoules. “Iterative Methods for Sparse Linear Systems on Graphics Processing Unit”. In: *14th IEEE International Conference on High Performance Computing and Communications (HPCC 2012), Liverpool, UK, June 25-27, 2012*. International Conference on High Performance Computing and Communications (HPCC). Liverpool, UK: IEEE, June 2012, pp. 836–842 (cit. on p. 63).
- [121] Frédéric Magoulès, Abal-Kassim Cheik Ahamed, and Roman Putanowicz. “Fast Iterative Solvers for large compressed-sparse row linear systems on Graphics Processing Unit”. In: *Pollack Periodica, An International Journal for Engineering and Information Sciences, Akadémiai Kiadó* 10.1 (Apr. 2015), pp. 3–18 (cit. on p. 63).
- [122] Frédéric Magoulès, Abal-Kassim Cheik Ahamed, and Roman Putanowicz. “Auto-tuned Krylov methods on cluster of graphics processing unit”. In: *International Journal of Computer Mathematics* 92.6 (June 3, 2014), pp. 1222–1250 (cit. on p. 63).
- [123] Hans Henrik Brandenborg Sørensen. “Auto-tuning of level 1 and level 2 BLAS for GPUs.” In: *Concurrency and Computation: Practice and Experience* 25.8 (2013), pp. 1183–1198 (cit. on p. 63).
- [124] Timothy A. Davis and Yifan Hu. “The University of Florida sparse matrix collection”. In: *ACM Trans. Math. Softw.* 38.1 (2011), pp. 1–25 (cit. on p. 67).
- [125] T J R Hughes. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Vol. 682. 2. Dover Publications, 2000. 682 pp. (cit. on p. 69).
- [126] Philippe G. Ciarlet. *Finite Element Method for Elliptic Problems*. Philadelphia, PA, USA: SIAM, 2002 (cit. on p. 69).
- [127] Isaac Harari and Thomas J. R. Hughes. “Finite element methods for the Helmholtz equation in an exterior domain: model problems”. In: *Comput. Methods Appl. Mech. Eng.* 87.1 (1991), pp. 59–96 (cit. on p. 70).
- [128] Isaac Harari and Frédéric Magoulès. “Numerical investigations of stabilized finite element computations for acoustics”. In: *Wave Motion* 39.4 (Apr. 2004), pp. 339–349 (cit. on p. 70).
- [129] Michael W. Berry, Bruce Hendrickson, and Padma Raghavan. “Sparse matrix reordering schemes for browsing hypertext”. In: ed. by James et al. *Renegar*. Vol. 32. The mathematics of numerical analysis. 1995 AMS-SIAM Summer Seminar in Applied Mathematics, July 17–August 11, 1995, Park City, UT, USA. Providence. 1996, pp. 99–123 (cit. on p. 70).
- [130] Takeo Taniguchi and Naruhito Shiraishi. “New renumbering algorithm for minimizing the bandwidth of sparse matrices”. In: *Advances in Engineering Software (1978)* 2.4 (1980), pp. 173–179 (cit. on p. 70).

- [131] J. Sulaiman, M. Othman, and M. K. Hasan. “Red-Black EDGSOR Iterative Method Using Triangle Element Approximation for 2D Poisson Equations”. In: *Computational Science and Its Applications – ICCSA 2007*. Ed. by Osvaldo Gervasi and Marina L. Gavrilova. Vol. 4707. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 298–308 (cit. on p. 74).
- [132] FrÉdÉric MagoulÈs, Karl Meerbergen, and Jean-Pierre Coyette. “APPLICATION OF A DOMAIN DECOMPOSITION METHOD WITH LAGRANGE MULTIPLIERS TO ACOUSTIC PROBLEMS ARISING FROM THE AUTOMOTIVE INDUSTRY”. In: *Journal of Computational Acoustics* 08.03 (Sept. 2000), pp. 503–521 (cit. on p. 77).
- [133] Magnus R. Hestenes and Eduard Stiefel. “Methods of Conjugate Gradients for Solving Linear Systems”. In: *Journal of Research of the National Bureau of Standards* 49.6 (Dec. 1952), pp. 409–436 (cit. on pp. 81, 84).
- [134] B. D. Craven. “Complex symmetric matrices”. In: *Journal of the Australian Mathematical Society* 10.3-4 (Nov. 1969), p. 341 (cit. on p. 82).
- [135] David AH Jacobs. “A generalization of the conjugate-gradient method to solve complex systems”. In: *IMA journal of numerical analysis* 6.4 (1986), pp. 447–452 (cit. on p. 84).
- [136] Kapil Ahuja et al. “Recycling BiCG with an Application to Model Reduction”. In: *SIAM Journal on Scientific Computing* 34.4 (Jan. 2012), A1925–A1949 (cit. on p. 84).
- [137] Peter Sonneveld. “CGS, A Fast Lanczos-Type Solver for Nonsymmetric Linear systems”. In: *SIAM Journal on Scientific and Statistical Computing* 10.1 (Jan. 1989), pp. 36–52 (cit. on p. 85).
- [138] H. A. Van der Vorst. “The convergence behaviour of preconditioned CG and CG-S in the presence of rounding errors”. In: *Preconditioned Conjugate Gradient Methods*. Ed. by Owe Axelsson and Lily Yu. Kolotilina. Red. by A. Dold, B. Eckmann, and F. Takens. Vol. 1457. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 126–136 (cit. on p. 85).
- [139] H. A. van der Vorst. “Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems”. In: *SIAM Journal on Scientific and Statistical Computing* 13.2 (Mar. 1992), pp. 631–644 (cit. on p. 85).
- [140] Henk A Van der Vorst, Diederik R Fokkema, and Gerard L Sleijpen. “Further improvements in nonsymmetric hybrid CG methods”. In: (1994) (cit. on p. 85).
- [141] Andreas Meister and Christof Vömel. “Preconditioned krylov subspace methods for Hyperbolic conservation laws”. In: *Hyperbolic Problems: Theory, Numerics, Applications*. Ed. by Heinrich Freistühler and Gerald Warnecke. Basel: Birkhäuser Basel, 2001, pp. 703–712 (cit. on p. 85).
- [142] L.T. Yang and R.P. Brent. “The improved BiCGStab method for large and sparse unsymmetric linear systems on parallel distributed memory architectures”. In: *IEEE Comput. Soc*, 2002, pp. 324–328 (cit. on p. 85).
- [143] Gianna M. Del Corso, Antonio Gullí, and Francesco Romani. “Comparison of Krylov subspace methods on the PageRank problem”. In: *Journal of Computational and Applied Mathematics* 210.1-2 (Dec. 2007), pp. 159–166 (cit. on p. 85).
- [144] Jie Chen, LOIS CURFMAN McInnes, and HONG Zhang. “Analysis and practical use of flexible BICGSTAB”. In: *Preprint ANL/MCS-P3039-0912, Argonne National Laboratory* (2012) (cit. on p. 85).
- [145] Ning Zhao and Xuben Wang. “A Parallel Preconditioned Bi-Conjugate Gradient Stabilized Solver for the Poisson Problem”. In: *Journal of Computers* 7.12 (Dec. 1, 2012) (cit. on p. 85).
- [146] Gerard L. G. Sleijpen and Diederik R. Fokkema. “BiCGstab (l) for linear equations involving unsymmetric matrices with complex spectrum”. In: *Electronic Transactions on Numerical Analysis, Kent State University* 1 (Sept. 1993), pp. 11–32 (cit. on p. 85).

- [147] G. L. G. Sleijpen, H. A. van der Vorst, and D. R. Fokkema. “BiCGstab(1) and other hybrid Bi-CG methods”. In: *Numerical Algorithms* 7.1 (Mar. 1994), pp. 75–109 (cit. on p. 85).
- [148] Martin H. Gutknecht. *Variants of BICGSTAB for Matrices with Complex Spectrum*. IPS Research Report 91-14. Aug. 1991 (cit. on p. 85).
- [149] Roland W. Freund and Noël M. Nachtigal. “QMR: a quasi-minimal residual method for non-Hermitian linear systems”. In: *Numerische Mathematik* 60.1 (Dec. 1991), pp. 315–339 (cit. on p. 85).
- [150] Roland W. Freund. “A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems”. In: *SIAM Journal on Scientific Computing* 14.2 (Mar. 1993), pp. 470–482 (cit. on pp. 85, 86).
- [151] Roland W. Freund. “Quasi-Kernel Polynomials and Convergence Results for Quasi-Minimal Residual Iterations”. In: *Numerical Methods in Approximation Theory, Vol. 9*. Ed. by Dietrich Braess and Larry L. Schumaker. Basel: Birkhäuser Basel, 1992, pp. 77–95 (cit. on p. 86).
- [152] Roland W. Freund, Martin H. Gutknecht, and Noël M. Nachtigal. “An Implementation of the Look-Ahead Lanczos Algorithm for Non-Hermitian Matrices”. In: *SIAM Journal on Scientific Computing* 14.1 (Jan. 1993), pp. 137–158 (cit. on p. 86).
- [153] Markus Clemens and Thomas Weiland. “Iterative Methods for the Solution of Very Large Complex Symmetric Linear Systems of Equations in Electrodynamics”. In: (1996) (cit. on p. 86).
- [154] Markus Clemens et al. “Modern Krylov subspace methods in electromagnetic field computation using the finite integration theory”. In: *Applied Computational Electromagnetics Society Journal* 11.1 (1996), pp. 70–84 (cit. on p. 86).
- [155] Cornelius Lanczos. “Solution of systems of linear equations by minimized iterations”. In: *J. Res. Nat. Bur. Standards* 49.1 (1952), pp. 33–53 (cit. on p. 86).
- [156] Olof Widlund. “A Lanczos Method for a Class of Nonsymmetric Systems of Linear Equations”. In: *SIAM Journal on Numerical Analysis* 15.4 (Aug. 1978), pp. 801–812 (cit. on p. 86).
- [157] Y. Saad. “The Lanczos Biorthogonalization Algorithm and Other Oblique Projection Methods for Solving Large Unsymmetric Systems”. In: *SIAM Journal on Numerical Analysis* 19.3 (June 1982), pp. 485–506 (cit. on p. 86).
- [158] Martin J. Gander. “Optimized Schwarz Methods for Helmholtz Problems”. In: Lyon, France, 2002, pp. 245–252 (cit. on p. 86).
- [159] Charbel Farhat and Francois-Xavier Roux. “A method of finite element tearing and interconnecting and its parallel solution algorithm”. In: *International Journal for Numerical Methods in Engineering* 32.6 (Oct. 25, 1991), pp. 1205–1227 (cit. on p. 93).
- [160] Xiao-Chuan Cai et al. “Overlapping Schwarz algorithms for solving Helmholtz’s equation”. In: *Domain decomposition methods, 10 (Boulder, CO, 1997)*. Providence, RI: Amer. Math. Soc., 1998, pp. 391–399 (cit. on p. 93).
- [161] Pierre-Louis Lions. “On the Schwarz alternating method. I.” In: *First International Symposium on Domain Decomposition Methods for Partial Differential Equations, Paris, 1987*. SIAM, 1988, pp. 1–42 (cit. on pp. 93, 203).
- [162] Pierre-Louis Lions and T. F. Chan. “On the Schwarz alternating method. II. Stochastic interpretation and order properties”. In: *Domain decomposition methods, Los Angeles, CA, 1988*. Domain Decomposition Methods. Ed. by R. Glowinski, J. Périaux, and Olof Widlund. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1989, pp. 47–70 (cit. on pp. 93, 203).

- [163] Pierre-Louis Lions. “On the Schwarz alternating method. III. A variant for nonoverlapping subdomains”. In: *Third International Symposium on Domain Decomposition Methods for Partial Differential Equations, Houston, TX, 1989*. Domain Decomposition Methods. Philadelphia, PA: SIAM, T.F Chan, R. Glowinski, J. Périaux, and O. Widlund, eds., 1990, pp. 202–223 (cit. on pp. [93](#), [203](#)).
- [164] Bruno Després, Patrick Joly, and Jean E. Roberts. “A domain decomposition method for the harmonic Maxwell equations”. In: *Iterative methods in linear algebra (Brussels, 1991)*. Ed. by R. Beauwens and P. de Groen. North-Holland, Amsterdam: Elsevier Science Publishers B. V., 1992, pp. 475–484 (cit. on pp. [93](#), [203](#)).
- [165] B. Després. “Domain decomposition method and the Helmholtz problem II”. In: Philadelphia, PA: SIAM, 1993, pp. 197–206 (cit. on pp. [93](#), [202–204](#)).
- [166] Jean-David Benamou and Bruno Després. “A Domain Decomposition Method for the Helmholtz Equation and Related Optimal Control Problems”. In: *Journal of Computational Physics* 136.1 (Sept. 1997), pp. 68–82 (cit. on pp. [93](#), [204](#)).
- [167] P. Chevalier and F. Nataf. “Symmetrized method with optimized second-order conditions for the Helmholtz equation”. In: *Domain decomposition methods, 10 (Boulder, CO, 1997)*. Providence, RI: Amer. Math. Soc., 1998, pp. 400–407 (cit. on pp. [93](#), [202–204](#)).
- [168] Yvon Maday and Frédéric Magoulès. “Optimized Schwarz methods without overlap for highly heterogeneous media”. In: *Computer Methods in Applied Mechanics and Engineering* 196.8 (Jan. 2007), pp. 1541–1553 (cit. on pp. [93](#), [202](#), [204](#)).
- [169] Frédéric Magoulès, François-Xavier Roux, and Laurent Series. “Algebraic Dirichlet-to-Neumann mapping for linear elasticity problems with extreme contrasts in the coefficients”. In: *Applied Mathematical Modelling* 30.8 (Aug. 2006), pp. 702–713 (cit. on p. [93](#)).
- [170] Frédéric Magoulès, François-Xavier Roux, and Laurent Series. “Algebraic approximation of Dirichlet-to-Neumann maps for the equations of linear elasticity”. In: *Computer Methods in Applied Mechanics and Engineering* 195.29-32 (June 2006), pp. 3742–3759 (cit. on pp. [93](#), [202](#), [203](#)).
- [171] Frédéric Magoulès, François-Xavier Roux, and Laurent Series. “Algebraic way to derive absorbing boundary conditions for the Helmholtz equation”. In: *Journal of Computational Acoustics* 13.03 (Sept. 2005), pp. 433–454 (cit. on p. [93](#)).
- [172] Martin Gander et al. “Analysis of Patch Substructuring Methods”. In: *International Journal of Applied Mathematics and Computer Science* 17.3 (Jan. 1, 2007) (cit. on pp. [93](#), [165](#), [202](#), [203](#)).
- [173] Ronald F. Boisvert et al. “Matrix Market: A Web Resource for Test Matrix Collections”. In: *The Quality of Numerical Software: Assessment and Enhancement, 1977, London, UK*. The Quality of Numerical Software: Assessment and Enhancement. London, UK, UK: Chapman & Hall, Ltd., 1997, pp. 125–137 (cit. on pp. [98](#), [112](#)).
- [174] John N. Tsitsiklis, Dimitri P. Bertsekas, and M. Athans. “Distributed asynchronous deterministic and stochastic gradient optimization algorithms”. In: *IEEE Transactions on Automatic Control* 31.9 (Sept. 1986), pp. 803–812 (cit. on pp. [107](#), [109](#)).
- [175] Chengchang Huang and Philip K McKinley. “Communication issues in parallel computing across ATM networks”. In: *IEEE Concurrency* 2.4 (1994), pp. 73–86 (cit. on p. [107](#)).
- [176] Ted Nesson. “Randomized, Oblivious, Minimal Routing Algorithms for Multicomputers”. 1995 (cit. on p. [107](#)).
- [177] D. Chazan and W. Miranker. “Chaotic relaxation”. In: *Linear Algebra and its Applications* 2.2 (Apr. 1969), pp. 199–222 (cit. on pp. [109](#), [110](#), [131](#), [133](#), [136](#)).
- [178] J.D.P. Donnelly. “Periodic chaotic relaxation”. In: *Linear Algebra and its Applications* 4.2 (Apr. 1971), pp. 117–128 (cit. on pp. [109](#), [110](#)).

- [179] Dimitri P Bertsekas and John N Tsitsiklis. *Parallel and distributed computation: numerical methods*. Vol. 23. Prentice hall Englewood Cliffs, NJ, 1989 (cit. on pp. [109](#), [110](#), [112](#), [131](#), [134](#)).
- [180] John N. Tsitsiklis. “On the stability of asynchronous iterative processes”. In: *Mathematical Systems Theory* 20.1 (Dec. 1987), pp. 137–153 (cit. on p. [109](#)).
- [181] R. de Leone. “Partially and totally asynchronous algorithms for linear complementarity problems”. In: *Journal of Optimization Theory and Applications* 69.2 (May 1991), pp. 235–249 (cit. on p. [109](#)).
- [182] J Miellou, Didier El Baz, and Pierre Spiteri. “A new class of asynchronous iterative algorithms with order intervals”. In: *Mathematics of Computation of the American Mathematical Society* 67.221 (1998), pp. 237–255 (cit. on p. [110](#)).
- [183] Didier El Baz et al. “Asynchronous Iterative Algorithms with Flexible Communication for Nonlinear Network Flow Problems”. In: *Journal of Parallel and Distributed Computing* 38.1 (Oct. 1996), pp. 1–15 (cit. on p. [110](#)).
- [184] Didier El Baz et al. “Flexible communication for parallel asynchronous methods with application to a nonlinear optimization problem”. In: *Advances in Parallel Computing*. Vol. 12. Elsevier, 1998, pp. 429–436 (cit. on p. [110](#)).
- [185] Andreas Frommer and Daniel B Szyld. “Asynchronous iterations with flexible communication for linear systems”. In: *Calculateurs Paralleles*. Citeseer, 1998 (cit. on pp. [110](#), [111](#)).
- [186] Jean Claude Miellou. “Algorithmes de relaxation chaotique à retards”. In: *ESAIM: Mathematical Modelling and Numerical Analysis-Modélisation Mathématique et Analyse Numérique* 9 (R1 1975), pp. 55–82 (cit. on pp. [110](#), [131](#), [133](#)).
- [187] Gérard M. Baudet. “Asynchronous Iterative Methods for Multiprocessors”. In: *Journal of the ACM* 25.2 (Apr. 1, 1978), pp. 226–244 (cit. on pp. [110](#), [131](#), [133](#)).
- [188] Michel Charnay. “Itérations chaotiques sur un produit d’espaces métriques”. 1975 (cit. on p. [110](#)).
- [189] P Comte. “Itérations chaotiques à retards, étude de la convergence dans le cas d’un espace produit d’espaces vectoriellement normés”. In: *CR Acad. Sci. Paris* 281 (1975) (cit. on p. [110](#)).
- [190] Claude Jacquemard. “Contribution à l’étude d’algorithmes de relaxation à convergence monotone”. 1977 (cit. on p. [110](#)).
- [191] Mouhamed Nabih El Tarazi. “Contraction et ordre partiel pour l’étude d’algorithmes synchrones et asynchrones en analyse numérique”. 1981 (cit. on pp. [110](#), [133](#)).
- [192] Mouhamed Nabih El Tarazi. “Some convergence results for asynchronous algorithms”. In: *Numerische Mathematik* 39.3 (Oct. 1982), pp. 325–340 (cit. on pp. [110](#), [133](#)).
- [193] Mouhamed Nabih El Tarazi. “Algorithmes mixtes asynchrones. Etude de convergence monotone”. In: *Numerische Mathematik* 44.3 (Oct. 1984), pp. 363–369 (cit. on p. [110](#)).
- [194] Didier El Baz. “ M -Functions and Parallel Asynchronous Algorithms”. In: *SIAM Journal on Numerical Analysis* 27.1 (Feb. 1990), pp. 136–140 (cit. on p. [110](#)).
- [195] J. Bahi, E. Griepentrog, and J. C. Miellou. “Parallel Treatment of a Class of Differential-Algebraic Systems”. In: *SIAM Journal on Numerical Analysis* 33.5 (Oct. 1996), pp. 1969–1980 (cit. on p. [110](#)).
- [196] Didier El Baz, Andreas Frommer, and Pierre Spiteri. “Asynchronous iterations with flexible communication: contracting operators”. In: *Journal of Computational and Applied Mathematics* 176.1 (Apr. 2005), pp. 91–103 (cit. on p. [110](#)).
- [197] D.J. Evans and Wang Deren. “An asynchronous parallel algorithm for solving a class of nonlinear simultaneous equations”. In: *Parallel Computing* 17.2-3 (June 1991), pp. 165–180 (cit. on p. [110](#)).

- [198] Rafael Bru et al. “Parallel, synchronous and asynchronous two-stage multisplitting methods”. In: *Electronic Transactions on Numerical Analysis* 3 (1995), pp. 24–38 (cit. on p. 110).
- [199] P Spitéri. “Contribution a l’etude de grands systemes non lineaires”. Faculté des Sciences et des Techniques. Université de Franche-Comté, 1984 (cit. on p. 110).
- [200] ZZ Bai, DR Wang, and DJ Evans. “Models of asynchronous parallel nonlinear multisplitting relaxed iterations”. In: *JOURNAL OF COMPUTATIONAL MATHEMATICS-INTERNATIONAL EDITION*- 13 (1995), pp. 369–386 (cit. on p. 110).
- [201] Didier El Baz. “A METHOD OF TERMINATING ASYNCHRONOUS ITERATIVE ALGORITHMS ON MESSAGE PASSING SYSTEMS”. In: *Parallel Algorithms and Applications* 9.1-2 (Jan. 1996), pp. 153–158 (cit. on pp. 110, 112).
- [202] Jacques Bahi, Jean-Claude Miellou, and Karim Rhofir. “Asynchronous multisplitting methods for nonlinear fixed point problems”. In: *Numerical Algorithms* 15.3-4 (1997), pp. 315–345 (cit. on p. 110).
- [203] Josep Arnal, Violeta Migallón, and José Penadés. “Parallel Newton Two-Stage Multisplitting Iterative Methods for Nonlinear Systems”. In: *BIT Numerical Mathematics* 43.5 (2003), pp. 849–861 (cit. on p. 110).
- [204] Dimitri P Bertsekas and Didier El Baz. “Distributed asynchronous relaxation methods for convex network flow problems”. In: *SIAM Journal on Control and Optimization* 25.1 (1987), pp. 74–85 (cit. on p. 110).
- [205] Emmanuel D. Chajakis and Stavros A. Zenios. “Synchronous and asynchronous implementations of relaxation algorithms for nonlinear network optimization”. In: *Parallel Computing* 17.8 (Oct. 1991), pp. 873–894 (cit. on p. 110).
- [206] Didier El Baz. “Asynchronous implementation of relaxation and gradient algorithms for convex network flow problems”. In: *Parallel Computing* 19.9 (Sept. 1993), pp. 1019–1028 (cit. on p. 110).
- [207] J Bernussou, F Le Gall, and G Authie. “About some iterative synchronous and asynchronous methods for Markov chain distribution computation.” In: *10. Triennial World Congress of the International Federation of Automatic Control*. Vol. 7. 1987, pp. 39–44 (cit. on p. 111).
- [208] Didier El Baz. “Parallel iterative algorithms for the solution of Markov systems”. In: vol. 3. IEEE, 1994, pp. 2524–2527 (cit. on p. 111).
- [209] Jacques M. Bahi. “Asynchronous Iterative Algorithms for Nonexpansive Linear Systems”. In: *Journal of Parallel and Distributed Computing* 60.1 (Jan. 2000), pp. 92–112 (cit. on p. 111).
- [210] P Spiteri, JC Miellou, and D El Baz. “Asynchronous Schwarz alternating methods with flexible communication for the obstacle problem”. In: *CALCULATEURS PARALLELES RESEAUX ET SYSTEMES REPARTIS* 13.1 (2001), pp. 47–66 (cit. on p. 111).
- [211] Pierre Spiteri, Jean-Claude Miellou, and Didier El Baz. “Parallel asynchronous Schwarz and multisplitting methods for a nonlinear diffusion problem”. In: *Numerical Algorithms* 33.1-4 (2003), pp. 461–474 (cit. on p. 111).
- [212] Li Lei. “Convergence of asynchronous iteration with arbitrary splitting form”. In: *Linear Algebra and its Applications* 113 (Feb. 1989), pp. 119–127 (cit. on p. 111).
- [213] J.M. Bahi, K. Rhofir, and J.-C. Miellou. “Parallel solution of linear DAEs by multisplitting waveform relaxation methods”. In: *Linear Algebra and its Applications* 332-334 (Aug. 2001), pp. 181–196 (cit. on p. 111).
- [214] Bert Pohl. “On the convergence of the discretized multi-splitting waveform relaxation algorithm”. In: *Applied numerical mathematics* 11.1 (1993), pp. 251–258 (cit. on p. 111).
- [215] Daniel B Szyld. “Different models of parallel asynchronous iterations with overlapping blocks”. In: *Computational and applied mathematics* 17 (1998), pp. 101–115 (cit. on p. 111).

- [216] Jacques Mohcine Bahi, Sylvain Contassot-Vivier, and Raphael Couturier. *Parallel iterative algorithms: from sequential to grid computing*. CRC Press, 2007 (cit. on pp. [111](#), [112](#), [130](#), [131](#), [136](#)).
- [217] M El-Kyal and A Machmoum. “Superlinear convergence of asynchronous multi-splitting waveform relaxation methods applied to a system of nonlinear ordinary differential equations”. In: *Mathematics and Computers in Simulation* 77.2 (2008), pp. 179–188 (cit. on p. [111](#)).
- [218] D. O’Leary and R. White. “Multi-Splittings of Matrices and Parallel Solution of Linear Systems”. In: *SIAM Journal on Algebraic Discrete Methods* 6.4 (Oct. 1, 1985), pp. 630–640 (cit. on pp. [111](#), [153](#)).
- [219] Behtash Babadi and Vahid Tarokh. “A distributed asynchronous algorithm for spectrum sharing in wireless ad hoc networks”. In: IEEE, Mar. 2008, pp. 831–835 (cit. on p. [111](#)).
- [220] Benoît Patra. “Convergence of Distributed Asynchronous Learning Vector Quantization Algorithms”. In: *J. Mach. Learn. Res.* 12 (Dec. 2011), pp. 3431–3466 (cit. on p. [111](#)).
- [221] H. Brendan McMahan and Matthew J. Streeter. “Delay-Tolerant Algorithms for Asynchronous Distributed Online Learning”. In: *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*. 2014, pp. 2915–2923 (cit. on p. [111](#)).
- [222] J. Bahi, S. Domas, and K. Mazouzi. “Jace: a Java environment for distributed asynchronous iterative computations”. In: IEEE, 2004, pp. 350–357 (cit. on pp. [111](#), [158](#)).
- [223] Jacques M. Bahi et al. “Java and asynchronous iterative applications: large scale experiments”. In: IEEE, 2007, pp. 1–7 (cit. on pp. [111](#), [158](#)).
- [224] Jean-Claude Charr, Raphaël Couturier, and David Laiymani. “JACEP2P-V2: A fully decentralized and fault tolerant environment for executing parallel iterative asynchronous applications on volatile distributed architectures”. In: *Future Generation Computer Systems* 27.5 (May 2011), pp. 606–613 (cit. on p. [111](#)).
- [225] Raphael Couturier and Stephane Domas. “CRAC: a Grid Environment to Solve Scientific Applications with Asynchronous Iterative Algorithms”. In: IEEE, 2007, pp. 1–8 (cit. on pp. [111](#), [158](#)).
- [226] Kostas Blathras, Daniel B. Szyld, and Yuan Shi. “Timing Models and Local Stopping Criteria for Asynchronous Iterative Algorithms”. In: *Journal of Parallel and Distributed Computing* 58.3 (Sept. 1999), pp. 446–465 (cit. on p. [112](#)).
- [227] J.M. Bahi et al. “A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms”. In: *IEEE Transactions on Parallel and Distributed Systems* 16.1 (Jan. 2005), pp. 4–13 (cit. on pp. [112](#), [159](#)).
- [228] Jacques M. Bahi, Sylvain Contassot-Vivier, and Raphaël Couturier. “An Efficient and Robust Decentralized Algorithm for Detecting the Global Convergence in Asynchronous Iterative Algorithms”. In: *High Performance Computing for Computational Science - VECPAR 2008*. Ed. by José M. Laginha M. Palma et al. Vol. 5336. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 240–254 (cit. on pp. [112](#), [158](#), [159](#)).
- [229] Dimitri P. Bertsekas and John N. Tsitsiklis. “Parallel and distributed iterative algorithms : a selective survey”. In: (Nov. 1988) (cit. on p. [112](#)).
- [230] Bassem F. Beidas and George P. Papavassilopoulos. “Distributed asynchronous algorithms with stochastic delays for constrained optimization problems with conditions of time drift”. In: *Parallel Computing* 21.9 (Sept. 1995), pp. 1431–1450 (cit. on p. [112](#)).
- [231] Didier El Baz. “An efficient termination method for asynchronous iterative algorithms on message passing architectures”. In: *Proc. of the International Conference on Parallel and Distributed Computing Systems, Dijon*. Vol. 1. 1996, pp. 1–7 (cit. on p. [112](#)).

- [232] S.A. Savari and D.P. Bertsekas. “Finite termination of asynchronous iterative algorithms”. In: *Parallel Computing* 22.1 (Jan. 1996), pp. 39–56 (cit. on p. 112).
- [233] Dimitri P. Bertsekas. “Distributed asynchronous computation of fixed points”. In: *Mathematical Programming* 27.1 (Sept. 1983), pp. 107–120 (cit. on pp. 112, 133, 155).
- [234] D El Baz et al. “Mathematical study of perturbed asynchronous iterations designed for distributed termination”. In: 1 (2002), pp. 491–503 (cit. on p. 112).
- [235] JC Miellou, P Spiteri, and D El Baz. “Stopping criteria, forward and backward errors for perturbed asynchronous linear fixed point methods in finite precision”. In: *IMA journal of numerical analysis* 25.3 (2005), pp. 429–442 (cit. on p. 112).
- [236] J.C. Miellou, P. Spiteri, and D. El Baz. “A new stopping criterion for linear perturbed asynchronous iterations”. In: *Journal of Computational and Applied Mathematics* 219.2 (Oct. 2008), pp. 471–483 (cit. on p. 112).
- [237] Shu Li and T. Basar. “Asymptotic agreement and convergence of asynchronous stochastic algorithms”. In: *IEEE Transactions on Automatic Control* 32.7 (July 1987), pp. 612–618 (cit. on p. 112).
- [238] Vivek S. Borkar. “Asynchronous Stochastic Approximations”. In: *SIAM Journal on Control and Optimization* 36.3 (May 1998), pp. 840–851 (cit. on p. 112).
- [239] Haitao Fang and Hanfu Chen. “Asymptotic behavior of asynchronous stochastic approximation”. In: *Science in China Series: Information Sciences* 44.4 (Aug. 2001) (cit. on p. 112).
- [240] K. Georgiev and Z. Zlatev. “Numerical experiments with applying approximate LU-factorizations as preconditioners for solving SLAEs with coefficient matrices from the "Sparse Matrix Market"”. In: 2012, pp. 104–111 (cit. on p. 112).
- [241] George Karypis and Vipin Kumar. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices. Version 5.1.0.* 1998 (cit. on pp. 112, 117, 120).
- [242] Vipin Kumar et al. *Introduction to parallel computing: design and analysis of algorithms.* Benjamin/Cummings Publishing Company Redwood City, CA, 1994 (cit. on p. 117).
- [243] G Karypis and V Kumar. “Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0”. In: (1995) (cit. on p. 117).
- [244] Stephen T. Barnard and Horst D. Simon. “Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems”. In: *Concurrency: Practice and Experience* 6.2 (Apr. 1994), pp. 101–117 (cit. on p. 117).
- [245] Bruce Hendrickson and Robert Leland. *The Chaco user’s guide: Version 2.0.* Technical Report SAND95-2344, Sandia National Laboratories, 1995 (cit. on p. 117).
- [246] Bruce Hendrickson and Robert W Leland. “A Multi-Level Algorithm For Partitioning Graphs.” In: *SC 95* (1995), p. 28 (cit. on p. 117).
- [247] Frank Harary. *Graph theory.* Addison-Wesley, Reading, MA, 1969 (cit. on p. 117).
- [248] John Adrian Bondy and Uppaluri Siva Ramachandra Murty. *Graph theory with applications.* Vol. 290. Macmillan London, 1976 (cit. on p. 117).
- [249] Béla Bollobás. *Graph theory.* Elsevier, 1982 (cit. on p. 117).
- [250] Béla Bollobás. *Modern graph theory.* Vol. 184. Springer Science & Business Media, 1998 (cit. on p. 117).
- [251] Douglas Brent West. *Introduction to graph theory.* Vol. 2. Prentice hall Upper Saddle River, 2001 (cit. on p. 117).
- [252] Bruce Hendrickson. “Graph partitioning and parallel solvers: Has the emperor no clothes?” In: *Solving Irregularly Structured Problems in Parallel.* Ed. by Alfonso Ferreira et al. Red. by

- G. Goos, J. Hartmanis, and J. van Leeuwen. Vol. 1457. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 218–225 (cit. on p. 121).
- [253] Bruce Hendrickson and Tamara G Kolda. “Graph partitioning models for parallel computing”. In: *Parallel Computing* 26.12 (Nov. 2000), pp. 1519–1534 (cit. on p. 121).
- [254] James B Orlin. “Line-digraphs, arborescences, and theorems of Tutte and Knuth”. In: *Journal of Combinatorial Theory, Series B* 25.2 (Oct. 1978), pp. 187–198 (cit. on pp. 122, 126).
- [255] Jonathan L Gross and Jay Yellen. *Graph theory and its applications*. CRC press, 2005 (cit. on pp. 122, 126).
- [256] D. J. A. Welsh and M. B. Powell. “An upper bound for the chromatic number of a graph and its application to timetabling problems”. In: *The Computer Journal* 10.1 (Jan. 1, 1967), pp. 85–86 (cit. on p. 126).
- [257] François Robert, Michel Charnay, and François Musy. “Itérations chaotiques série-parallèle pour des équations non-linéaires de point fixe”. In: *Aplikace Matematiky* 20.1 (1975), pp. 1–38 (cit. on p. 132).
- [258] C. Venet and F. Magoules. “Asynchronous substructuring methods”. In: *Substructuring Techniques and Domain Decomposition Methods*. Saxe-Coburg Publications, Stirlingshire, UK, 2010, pp. 71–104 (cit. on pp. 142, 153, 158, 159).
- [259] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface, 2nd edition*. Cambridge, MA: MIT Press, 1999 (cit. on p. 158).
- [260] L. Giraud, A. Marrocco, and J.-C. Rioual. “Iterative versus direct parallel substructuring methods in semiconductor device modelling”. In: *Numerical Linear Algebra with Applications* 12.1 (Feb. 2005), pp. 33–53 (cit. on p. 165).
- [261] N. Nupairoj and L.M. Ni. “Performance evaluation of some MPI implementations on workstation clusters”. In: IEEE Comput. Soc. Press, 1995, pp. 98–105 (cit. on pp. 169, 170, 186).
- [262] Zhixin Ba et al. “Performance evaluation of some MPI implementations on workstation clusters”. In: IEEE, 2000, 392–394 vol.1 (cit. on p. 169).
- [263] Khalid Al-Tawil and Csaba Andras Moritz. “Performance Modeling and Evaluation of MPI”. In: *Journal of Parallel and Distributed Computing* 61.2 (Feb. 2001), pp. 202–223 (cit. on p. 169).
- [264] Amith R. Mamidala et al. “MPI Collectives on Modern Multicore Clusters: Performance Optimizations and Communication Characteristics”. In: IEEE, May 2008, pp. 130–137 (cit. on p. 169).
- [265] J. M. Bull and T. L. Freeman. “Numerical performance of an asynchronous Jacobi iteration”. In: *Parallel Processing: CONPAR 92–VAPP V*. Ed. by Luc Bougé et al. Red. by Gerhard Goos and Juris Hartmanis. Vol. 634. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 361–366 (cit. on p. 171).
- [266] Patrick A Domenico and Franklin W Schwartz. *Physical and chemical hydrogeology*. Vol. 44. Wiley New York, 1998 (cit. on p. 174).
- [267] Timothy Theodore Eaton et al. “Hydraulic conductivity and specific storage of the Maquoketa shale”. In: *Final Report for the University of Wisconsin Water Resources Institute. Open-File Report 00-01. Madison, Wisconsin: Wisconsin Geological and Natural History Survey* (2000) (cit. on p. 174).
- [268] Barbara Suski. “Caractérisation et suivi des écoulements hydriques dans les milieux poreux par la méthode du Potentiel Spontané. Thèse”. Université Paris-Sud, 2006 (cit. on p. 174).

- [269] Frederic Magoules, Remi Cerise, and Patrick Callet. “A Beam-Tracing Domain Decomposition Method for Sound Holography in Church Acoustics”. In: IEEE, Sept. 2013, pp. 61–65 (cit. on pp. 179, 180).
- [270] Guillaume Gbikpi-Benissan, Patrick Callet, and Frederic Magoules. “Spectral Domain Decomposition Method for Physically-Based Rendering of Photochromic/Electrochromic Glass Windows”. In: IEEE, Nov. 2014, pp. 117–121 (cit. on p. 180).
- [271] Ian Foster. *Designing and building parallel programs: concepts and tools for parallel software engineering*. Reading, Mass: Addison-Wesley, 1995. 381 pp. (cit. on p. 186).
- [272] Rajeev Thakur and William Gropp. “Test Suite for Evaluating Performance of MPI Implementations That Support MPI_THREAD_MULTIPLE”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Franck Cappello, Thomas Herault, and Jack Dongarra. Vol. 4757. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 46–55 (cit. on p. 186).
- [273] R.P. Martin et al. “Effects Of Communication Latency, Overhead, And Bandwidth In A Cluster Architecture”. In: *Computer Architecture, 1997. Conference Proceedings. The 24th Annual International Symposium on*. June 1997, pp. 85–97 (cit. on p. 186).
- [274] Nico Mittenzwey. “Evaluating and Improving the Performance of MPI-Allreduce on QLogic HTX/PCIe InfiniBand HCA”. 2009 (cit. on p. 186).
- [275] Armel de La Bourdonnaye et al. “A non overlapping domain decomposition method for the exterior Helmholtz problem”. In: *Contemporary Mathematics* 218.2 (1998), pp. 42–66 (cit. on p. 202).
- [276] Martin J. Gander, Frédéric Magoulès, and Frédéric Nataf. “Optimized Schwarz Methods without Overlap for the Helmholtz Equation”. In: *SIAM Journal on Scientific Computing* 24.1 (Jan. 2002), pp. 38–60 (cit. on p. 202).
- [277] Abal-Kassim Cheik Ahamed and Frédéric Magoulès. “A Stochastic-Based Optimized Schwarz Method for the Gravimetry Equations on GPU Clusters”. In: *Domain Decomposition Methods in Science and Engineering XXI*. Ed. by Jocelyne Erhel et al. Vol. 98. Cham: Springer International Publishing, 2014, pp. 687–695 (cit. on p. 202).
- [278] Jaroslav Kruis. *Domain Decomposition Methods for Distributed Computing*. Saxe-Coburg Publications, 2007 (cit. on p. 202).
- [279] Souad Ghanemi. “A domain decomposition method for Helmholtz scattering problems”. In: ed. by P. E. Bjørstad, M. Espedal, and D. Keyes. ddm.org, 1997, pp. 105–112 (cit. on p. 202).
- [280] Yvon Maday and Frédéric Magoulès. “Improved ad hoc interface conditions for Schwarz solution procedure tuned to highly heterogeneous media”. In: *Applied Mathematical Modelling* 30.8 (Aug. 2006), pp. 731–743 (cit. on pp. 202, 204).
- [281] Yvon Maday and Frédéric Magoulès. “Non-overlapping additive Schwarz methods tuned to highly heterogeneous media”. In: *Comptes Rendus Mathématique* 341.11 (Dec. 2005), pp. 701–705 (cit. on pp. 202, 204).
- [282] M. Garbey and D. Tromeur-Dervout. “On some Aitken-like acceleration of the Schwarz method”. In: *International Journal for Numerical Methods in Fluids* 40.12 (Dec. 30, 2002), pp. 1493–1513 (cit. on p. 202).
- [283] Thomas Dufaud and Damien Tromeur-Dervout. “Efficient parallel implementation of the fully algebraic multiplicative Aitken-RAS preconditioning technique”. In: *Advances in Engineering Software* 53 (Nov. 2012), pp. 33–44 (cit. on p. 202).
- [284] L. Tourrette and L. Halpern. *Absorbing Boundaries and Layers, Domain Decomposition Methods: Applications to Large Scale Computers*. Nova Science publishers, 2001 (cit. on p. 203).

- [285] TRAN Minh-Binh. “Convergence Properties of Overlapping Schwarz Domain Decomposition Algorithms”. In: (2011) (cit. on p. [203](#)).
- [286] C. Japhet, F. Nataf, and F. Rogier. “The optimized order 2 method Application to convection-diffusion problems”. In: *Future Generation Computer Systems* 18.1 (Sept. 2001), pp. 17–30 (cit. on p. [203](#)).
- [287] M. J. Gander, L. Halpern, and F. Magoulès. “An optimized Schwarz method with two-sided Robin transmission conditions for the Helmholtz equation”. In: *International Journal for Numerical Methods in Fluids* 55.2 (Sept. 20, 2007), pp. 163–175 (cit. on pp. [203](#), [204](#)).
- [288] Caroline Japhet and Frédéric Nataf. “The Best Interface Conditions for Domain Decomposition Methods: Absorbing Boundary Conditions”. In: *Absorbing Boundaries and Layers, Domain Decomposition Methods. Applications to Large Scale Computations*. Ed. by L. Tournette and L. Halpern. New York: Nova Science publishers, Inc., 2001, pp. 348–373 (cit. on p. [204](#)).
- [289] Anne Auger and Nikolaus Hansen. “Tutorial CMA-ES: evolution strategies and covariance matrix adaptation”. In: ACM Press, 2012, p. 827 (cit. on p. [204](#)).